src/ - luajit-2.0-src

- Makefile
- host/
- jit/
- lauxlib.h
- lib_aux.c
- lib_base.c
- lib_bit.c
- lib_debug.c
- lib_ffi.c
- lib_init.c
- lib_io.c
- lib_jit.c
- lib_math.c
- lib_os.c
- lib_package.c
- lib_string.c
- lib_table.c
- lj_alloc.c
- lj_alloc.h
- lj_api.c
- lj_arch.h
- lj_asm.c
- lj_asm.h
- lj_asm_arm.h
- lj_asm_mips.h
- lj_asm_ppc.h
- lj_asm_x86.h
- lj_bc.c
- lj_bc.h
- lj_bcdef.h
- lj_bcdump.h
- lj_bcread.c

- lj_bcwrite.c
- lj_buf.c
- lj_buf.h
- lj_carith.c
- lj_carith.h
- lj_ccall.c
- lj_ccall.h
- lj_ccallback.c
- lj_ccallback.h
- lj_cconv.c
- lj_cconv.h
- lj_cdata.c
- lj_cdata.h
- lj_char.c
- lj_char.h
- lj_clib.c
- lj_clib.h
- lj_cparse.c
- lj_cparse.h
- lj_crecord.c
- lj_crecord.h
- lj_ctype.c
- lj_ctype.h
- lj_debug.c
- lj_debug.h
- lj_def.h
- lj_dispatch.c
- lj_dispatch.h
- lj_emit_arm.h
- lj_emit_mips.h
- lj_emit_ppc.h
- lj_emit_x86.h
- lj_err.c

- lj_err.h
- lj_errmsg.h
- lj_ff.h
- lj_ffdef.h
- lj_ffrecord.c
- lj_ffrecord.h
- lj_folddef.h
- lj_frame.h
- lj_func.c
- lj_func.h
- lj_gc.c
- lj_gc.h
- lj_gdbjit.c
- lj_gdbjit.h
- lj_ir.c
- lj_ir.h
- lj_ircall.h
- lj_iropt.h
- lj_jit.h
- lj_lex.c
- lj_lex.h
- lj_lib.c
- lj_lib.h
- lj_libdef.h
- lj_load.c
- lj_mcode.c
- lj_mcode.h
- lj_meta.c
- lj_meta.h
- lj_obj.c
- lj_obj.h
- lj_opt_dce.c
- lj_opt_fold.c

- lj_opt_loop.c
- lj_opt_mem.c
- lj_opt_narrow.c
- lj_opt_sink.c
- lj_opt_split.c
- lj_parse.c
- lj_parse.h
- lj_profile.c
- lj_profile.h
- lj_recdef.h
- lj_record.c
- lj_record.h
- lj_snap.c
- lj_snap.h
- lj_state.c
- lj_state.h
- lj_str.c
- lj_str.h
- lj_strfmt.c
- lj_strfmt.h
- lj_strscan.c
- lj_strscan.h
- lj_tab.c
- lj_tab.h
- lj_target.h
- lj_target_arm.h
- lj_target_arm64.h
- lj_target_mips.h
- lj_target_ppc.h
- lj_target_x86.h
- lj_trace.c
- lj_trace.h
- lj_traceerr.h

- lj_udata.c
- lj_udata.h
- lj_vm.S
- lj_vm.h
- lj_vm.s
- lj_vmevent.c
- lj_vmevent.h
- lj_vmmath.c
- ljamalg.c
- lua.h
- lua.hpp
- luaconf.h
- luajit.c
- luajit.h
- lualib.h
- msvcbuild.bat
- ps4build.bat
- psvitabuild.bat
- xedkbuild.bat

src/Makefile - luajit-2.0-src

```
1  ##############################################################################
2  # LuaJIT Makefile. Requires GNU Make.
3  #
4  # Please read doc/install.html before changing any variables!
5  #
6  # Suitable for POSIX platforms (Linux, *BSD, OSX etc.).
7  # Also works with MinGW and Cygwin on Windows.
8  # Please check msvcbuild.bat for building with MSVC on Windows.
9  #
10  # Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
11  ##############################################################################
12
13  MAJVER=  2
14  MINVER=  1
15  RELVER=  0
16  ABIVER=  5.1
17  NODOTABIVER= 51
18
19  ##############################################################################
20  ############################  COMPILER OPTIONS  ##############################
21  ##############################################################################
22  # These options mainly affect the speed of the JIT compiler itself, not the
23  # speed of the JIT-compiled code. Turn any of the optional settings on by
24  # removing the '#' in front of them. Make sure you force a full recompile
25  # with "make clean", followed by "make" if you change any options.
26  #
27  # LuaJIT builds as a native 32 or 64 bit binary by default.
28  CC= gcc
29  #
30  # Use this if you want to force a 32 bit build on a 64 bit multilib OS.
31  #CC= gcc -m32
32  #
33  # Since the assembler part does NOT maintain a frame pointer, it's pointless
34  # to slow down the C part by not omitting it. Debugging, tracebacks and
35  # unwinding are not affected -- the assembler part has frame unwind
36  # information and GCC emits it where needed (x64) or with -g (see CCDEBUG).
37  CCOPT= -O2 -fomit-frame-pointer
38  # Use this if you want to generate a smaller binary (but it's slower):
39  #CCOPT= -Os -fomit-frame-pointer
40  # Note: it's no longer recommended to use -O3 with GCC 4.x.
41  # The I-Cache bloat usually outweighs the benefits from aggressive inlining.
42  #
43  # Target-specific compiler options:
44  #
45  # x86/x64 only: For GCC 4.2 or higher and if you don't intend to distribute
46  # the binaries to a different machine you could also use: -march=native
47  #
48  CCOPT_x86= -march=i686 -msse -msse2 -mfpmath=sse
49  CCOPT_x64=
50  CCOPT_arm=
51  CCOPT_arm64=
52  CCOPT_ppc=
53  CCOPT_mips=
54  #
55  CCDEBUG=
56  # Uncomment the next line to generate debug information:
57  #CCDEBUG= -g
58  #
59  CCWARN= -Wall
60  # Uncomment the next line to enable more warnings:
61  #CCWARN+= -Wextra -Wdeclaration-after-statement -Wredundant-decls -Wshadow -Wpointer-arith
62  #
63  ##############################################################################
64
65  ##############################################################################
66  ################################  BUILD MODE  ################################
67  ##############################################################################
68  # The default build mode is mixed mode on POSIX. On Windows this is the same
69  # as dynamic mode.
70  #
71  # Mixed mode creates a static + dynamic library and a statically linked luajit.
```

```
BUILDMODE= mixed
#
# Static mode creates a static library and a statically linked luajit.
#BUILDMODE= static
#
# Dynamic mode creates a dynamic library and a dynamically linked luajit.
# Note: this executable will only run when the library is installed!
#BUILDMODE= dynamic
#
##############################################################################

##############################################################################
#################################  FEATURES  #################################
##############################################################################
# Enable/disable these features as needed, but make sure you force a full
# recompile with "make clean", followed by "make".
XCFLAGS=
#
# Permanently disable the FFI extension to reduce the size of the LuaJIT
# executable. But please consider that the FFI library is compiled-in,
# but NOT loaded by default. It only allocates any memory, if you actually
# make use of it.
#XCFLAGS+= -DLUAJIT_DISABLE_FFI
#
# Features from Lua 5.2 that are unlikely to break existing code are
# enabled by default. Some other features that *might* break some existing
# code (e.g. __pairs or os.execute() return values) can be enabled here.
# Note: this does not provide full compatibility with Lua 5.2 at this time.
#XCFLAGS+= -DLUAJIT_ENABLE_LUA52COMPAT
#
# Disable the JIT compiler, i.e. turn LuaJIT into a pure interpreter.
#XCFLAGS+= -DLUAJIT_DISABLE_JIT
#
# Some architectures (e.g. PPC) can use either single-number (1) or
# dual-number (2) mode. Uncomment one of these lines to override the
# default mode. Please see LJ_ARCH_NUMMODE in lj_arch.h for details.
#XCFLAGS+= -DLUAJIT_NUMMODE=1
#XCFLAGS+= -DLUAJIT_NUMMODE=2
#
##############################################################################

##############################################################################
############################  DEBUGGING SUPPORT  #############################
##############################################################################
# Enable these options as needed, but make sure you force a full recompile
# with "make clean", followed by "make".
# Note that most of these are NOT suitable for benchmarking or release mode!
#
# Use the system provided memory allocator (realloc) instead of the
# bundled memory allocator. This is slower, but sometimes helpful for
# debugging. This option cannot be enabled on x64, since realloc usually
# doesn't return addresses in the right address range.
# OTOH this option is mandatory for Valgrind's memcheck tool on x64 and
# the only way to get useful results from it for all other architectures.
#XCFLAGS+= -DLUAJIT_USE_SYSMALLOC
#
# This define is required to run LuaJIT under Valgrind. The Valgrind
# header files must be installed. You should enable debug information, too.
# Use --suppressions=lj.supp to avoid some false positives.
#XCFLAGS+= -DLUAJIT_USE_VALGRIND
#
# This is the client for the GDB JIT API. GDB 7.0 or higher is required
# to make use of it. See lj_gdbjit.c for details. Enabling this causes
# a non-negligible overhead, even when not running under GDB.
#XCFLAGS+= -DLUAJIT_USE_GDBJIT
#
# Turn on assertions for the Lua/C API to debug problems with lua_* calls.
# This is rather slow -- use only while developing C libraries/embeddings.
#XCFLAGS+= -DLUA_USE_APICHECK
#
# Turn on assertions for the whole LuaJIT VM. This significantly slows down
# everything. Use only if you suspect a problem with LuaJIT itself.
#XCFLAGS+= -DLUA_USE_ASSERT
#
##############################################################################
# You probably don't need to change anything below this line!
```

```
148 ##############################################################################
149
150 ##############################################################################
151 # Flags and options for host and target.
152 ##############################################################################
153
154 # You can override the following variables at the make command line:
155 #   CC       HOST_CC       STATIC_CC       DYNAMIC_CC
156 #   CFLAGS   HOST_CFLAGS   TARGET_CFLAGS
157 #   LDFLAGS  HOST_LDFLAGS  TARGET_LDFLAGS  TARGET_SHLDFLAGS
158 #   LIBS     HOST_LIBS     TARGET_LIBS
159 #   CROSS    HOST_SYS      TARGET_SYS      TARGET_FLAGS
160 #
161 # Cross-compilation examples:
162 #   make HOST_CC="gcc -m32" CROSS=i586-mingw32msvc- TARGET_SYS=Windows
163 #   make HOST_CC="gcc -m32" CROSS=powerpc-linux-gnu-
164
165 ASOPTIONS= $(CCOPT) $(CCWARN) $(XCFLAGS) $(CFLAGS)
166 CCOPTIONS= $(CCDEBUG) $(ASOPTIONS)
167 LDOPTIONS= $(CCDEBUG) $(LDFLAGS)
168
169 HOST_CC= $(CC)
170 HOST_RM= rm -f
171 # If left blank, minilua is built and used. You can supply an installed
172 # copy of (plain) Lua 5.1 or 5.2, plus Lua BitOp. E.g. with: HOST_LUA=lua
173 HOST_LUA=
174
175 HOST_XCFLAGS= -I.
176 HOST_XLDFLAGS=
177 HOST_XLIBS=
178 HOST_ACFLAGS= $(CCOPTIONS) $(HOST_XCFLAGS) $(TARGET_ARCH) $(HOST_CFLAGS)
179 HOST_ALDFLAGS= $(LDOPTIONS) $(HOST_XLDFLAGS) $(HOST_LDFLAGS)
180 HOST_ALIBS= $(HOST_XLIBS) $(LIBS) $(HOST_LIBS)
181
182 STATIC_CC = $(CROSS)$(CC)
183 DYNAMIC_CC = $(CROSS)$(CC) -fPIC
184 TARGET_CC= $(STATIC_CC)
185 TARGET_STCC= $(STATIC_CC)
186 TARGET_DYNCC= $(DYNAMIC_CC)
187 TARGET_LD= $(CROSS)$(CC)
188 TARGET_AR= $(CROSS)ar rcus
189 TARGET_STRIP= $(CROSS)strip
190
191 TARGET_LIBPATH= $(or $(PREFIX),/usr/local)/$(or $(MULTILIB),lib)
192 TARGET_SONAME= libluajit-$(ABIVER).so.$(MAJVER)
193 TARGET_DYLIBNAME= libluajit-$(ABIVER).$(MAJVER).dylib
194 TARGET_DYLIBPATH= $(TARGET_LIBPATH)/$(TARGET_DYLIBNAME)
195 TARGET_DLLNAME= lua$(NODOTABIVER).dll
196 TARGET_XSHLDFLAGS= -shared -fPIC -Wl,-soname,$(TARGET_SONAME)
197 TARGET_DYNXLDOPTS=
198
199 TARGET_LFSFLAGS= -D_FILE_OFFSET_BITS=64 -D_LARGEFILE_SOURCE
200 TARGET_XCFLAGS= $(TARGET_LFSFLAGS) -U_FORTIFY_SOURCE
201 TARGET_XLDFLAGS=
202 TARGET_XLIBS= -lm
203 TARGET_TCFLAGS= $(CCOPTIONS) $(TARGET_XCFLAGS) $(TARGET_FLAGS) $(TARGET_CFLAGS)
204 TARGET_ACFLAGS= $(CCOPTIONS) $(TARGET_XCFLAGS) $(TARGET_FLAGS) $(TARGET_CFLAGS)
205 TARGET_ASFLAGS= $(ASOPTIONS) $(TARGET_XCFLAGS) $(TARGET_FLAGS) $(TARGET_CFLAGS)
206 TARGET_ALDFLAGS= $(LDOPTIONS) $(TARGET_XLDFLAGS) $(TARGET_FLAGS) $(TARGET_LDFLAGS)
207 TARGET_ASHLDFLAGS= $(LDOPTIONS) $(TARGET_XSHLDFLAGS) $(TARGET_FLAGS) $(TARGET_SHLDFLAGS)
208 TARGET_ALIBS= $(TARGET_XLIBS) $(LIBS) $(TARGET_LIBS)
209
210 TARGET_TESTARCH=$(shell $(TARGET_CC) $(TARGET_TCFLAGS) -E lj_arch.h -dM)
211 ifneq (,$(findstring LJ_TARGET_X64 ,$(TARGET_TESTARCH)))
212   TARGET_LJARCH= x64
213 else
214 ifneq (,$(findstring LJ_TARGET_X86 ,$(TARGET_TESTARCH)))
215   TARGET_LJARCH= x86
216 else
217 ifneq (,$(findstring LJ_TARGET_ARM ,$(TARGET_TESTARCH)))
218   TARGET_LJARCH= arm
219 else
220 ifneq (,$(findstring LJ_TARGET_ARM64 ,$(TARGET_TESTARCH)))
221   TARGET_LJARCH= arm64
222 else
223 ifneq (,$(findstring LJ_TARGET_PPC ,$(TARGET_TESTARCH)))
```

```
  TARGET_LJARCH= ppc
else
ifneq (,$(findstring LJ_TARGET_MIPS ,$(TARGET_TESTARCH)))
  ifneq (,$(findstring MIPSEL ,$(TARGET_TESTARCH)))
    TARGET_ARCH= -D__MIPSEL__=1
  endif
  TARGET_LJARCH= mips
else
  $(error Unsupported target architecture)
endif
endif
endif
endif
endif
endif

ifneq (,$(findstring LJ_TARGET_PS3 1,$(TARGET_TESTARCH)))
  TARGET_SYS= PS3
  TARGET_ARCH+= -D__CELLOS_LV2__
  TARGET_XCFLAGS+= -DLUAJIT_USE_SYSMALLOC
  TARGET_XLIBS+= -lpthread
endif
ifneq (,$(findstring LJ_NO_UNWIND 1,$(TARGET_TESTARCH)))
  TARGET_ARCH+= -DLUAJIT_NO_UNWIND
endif

TARGET_XCFLAGS+= $(CCOPT_$(TARGET_LJARCH))
TARGET_ARCH+= $(patsubst %,-DLUAJIT_TARGET=LUAJIT_ARCH_%,$(TARGET_LJARCH))

ifneq (,$(PREFIX))
ifneq (/usr/local,$(PREFIX))
  TARGET_XCFLAGS+= -DLUA_ROOT=\"$(PREFIX)\"
  ifneq (/usr,$(PREFIX))
    TARGET_DYNXLDOPTS= -Wl,-rpath,$(TARGET_LIBPATH)
  endif
endif
endif
ifneq (,$(MULTILIB))
  TARGET_XCFLAGS+= -DLUA_MULTILIB=\"$(MULTILIB)\"
endif
ifneq (,$(LMULTILIB))
  TARGET_XCFLAGS+= -DLUA_LMULTILIB=\"$(LMULTILIB)\"
endif

##############################################################################
# System detection.
##############################################################################

ifeq (Windows,$(findstring Windows,$(OS))$(MSYSTEM)$(TERM))
  HOST_SYS= Windows
  HOST_RM= del
else
  HOST_SYS:= $(shell uname -s)
  ifneq (,$(findstring MINGW,$(HOST_SYS)))
    HOST_SYS= Windows
    HOST_MSYS= mingw
  endif
  ifneq (,$(findstring CYGWIN,$(HOST_SYS)))
    HOST_SYS= Windows
    HOST_MSYS= cygwin
  endif
endif

TARGET_SYS?= $(HOST_SYS)
ifeq (Windows,$(TARGET_SYS))
  TARGET_STRIP+= --strip-unneeded
  TARGET_XSHLDFLAGS= -shared
  TARGET_DYNXLDOPTS=
else
ifeq (,$(shell $(TARGET_CC) -o /dev/null -c -x c /dev/null -fno-stack-protector 2>/dev/null || echo 1))
  TARGET_XCFLAGS+= -fno-stack-protector
endif
ifeq (Darwin,$(TARGET_SYS))
  ifeq (,$(MACOSX_DEPLOYMENT_TARGET))
    export MACOSX_DEPLOYMENT_TARGET=10.4
  endif
```

```
  TARGET_STRIP+= -x
  TARGET_AR+= 2>/dev/null
  TARGET_XSHLDFLAGS= -dynamiclib -single_module -undefined dynamic_lookup -fPIC
  TARGET_DYNXLDOPTS=
  TARGET_XSHLDFLAGS+= -install_name $(TARGET_DYLIBPATH) -compatibility_version $(MAJVER).$(MINVER) -
current_version $(MAJVER).$(MINVER).$(RELVER)
  ifeq (x64,$(TARGET_LJARCH))
    TARGET_XLDFLAGS+= -pagezero_size 10000 -image_base 100000000
    TARGET_XSHLDFLAGS+= -image_base 7fff04c4a000
  endif
else
ifeq (iOS,$(TARGET_SYS))
  TARGET_STRIP+= -x
  TARGET_AR+= 2>/dev/null
  TARGET_XSHLDFLAGS= -dynamiclib -single_module -undefined dynamic_lookup -fPIC
  TARGET_DYNXLDOPTS=
  TARGET_XSHLDFLAGS+= -install_name $(TARGET_DYLIBPATH) -compatibility_version $(MAJVER).$(MINVER) -
current_version $(MAJVER).$(MINVER).$(RELVER)
  ifeq (arm64,$(TARGET_LJARCH))
    TARGET_XCFLAGS+= -fno-omit-frame-pointer
  endif
else
  ifneq (SunOS,$(TARGET_SYS))
    ifneq (PS3,$(TARGET_SYS))
      TARGET_XLDFLAGS+= -Wl,-E
    endif
  endif
  ifeq (Linux,$(TARGET_SYS))
    TARGET_XLIBS+= -ldl
  endif
  ifeq (GNU/kFreeBSD,$(TARGET_SYS))
    TARGET_XLIBS+= -ldl
  endif
endif
endif
endif

ifneq ($(HOST_SYS),$(TARGET_SYS))
  ifeq (Windows,$(TARGET_SYS))
    HOST_XCFLAGS+= -malign-double -DLUAJIT_OS=LUAJIT_OS_WINDOWS
  else
  ifeq (Linux,$(TARGET_SYS))
    HOST_XCFLAGS+= -DLUAJIT_OS=LUAJIT_OS_LINUX
  else
  ifeq (Darwin,$(TARGET_SYS))
    HOST_XCFLAGS+= -DLUAJIT_OS=LUAJIT_OS_OSX
  else
  ifeq (iOS,$(TARGET_SYS))
    HOST_XCFLAGS+= -DLUAJIT_OS=LUAJIT_OS_OSX
  else
    HOST_XCFLAGS+= -DLUAJIT_OS=LUAJIT_OS_OTHER
  endif
  endif
  endif
  endif
endif

ifneq (,$(CCDEBUG))
  TARGET_STRIP= @:
endif

##############################################################################
# Files and pathnames.
##############################################################################

MINILUA_O= host/minilua.o
MINILUA_LIBS= -lm
MINILUA_T= host/minilua
MINILUA_X= $(MINILUA_T)

ifeq (,$(HOST_LUA))
  HOST_LUA= $(MINILUA_X)
  DASM_DEP= $(MINILUA_T)
endif

DASM_DIR= ../dynasm
```

```
DASM= $(HOST_LUA) $(DASM_DIR)/dynasm.lua
DASM_XFLAGS=
DASM_AFLAGS=
DASM_ARCH= $(TARGET_LJARCH)

ifneq (,$(findstring LJ_ARCH_BITS 64,$(TARGET_TESTARCH)))
  DASM_AFLAGS+= -D P64
endif
ifneq (,$(findstring LJ_HASJIT 1,$(TARGET_TESTARCH)))
  DASM_AFLAGS+= -D JIT
endif
ifneq (,$(findstring LJ_HASFFI 1,$(TARGET_TESTARCH)))
  DASM_AFLAGS+= -D FFI
endif
ifneq (,$(findstring LJ_DUALNUM 1,$(TARGET_TESTARCH)))
  DASM_AFLAGS+= -D DUALNUM
endif
ifneq (,$(findstring LJ_ARCH_HASFPU 1,$(TARGET_TESTARCH)))
  DASM_AFLAGS+= -D FPU
  TARGET_ARCH+= -DLJ_ARCH_HASFPU=1
else
  TARGET_ARCH+= -DLJ_ARCH_HASFPU=0
endif
ifeq (,$(findstring LJ_ABI_SOFTFP 1,$(TARGET_TESTARCH)))
  DASM_AFLAGS+= -D HFABI
  TARGET_ARCH+= -DLJ_ABI_SOFTFP=0
else
  TARGET_ARCH+= -DLJ_ABI_SOFTFP=1
endif
DASM_AFLAGS+= -D VER=$(subst LJ_ARCH_VERSION_,,$(filter LJ_ARCH_VERSION_%,$(subst LJ_ARCH_VERSION
,LJ_ARCH_VERSION_,$(TARGET_TESTARCH))))
ifeq (Windows,$(TARGET_SYS))
  DASM_AFLAGS+= -D WIN
endif
ifeq (x64,$(TARGET_LJARCH))
  ifeq (,$(findstring LJ_FR2 1,$(TARGET_TESTARCH)))
    DASM_ARCH= x86
  endif
else
ifeq (arm,$(TARGET_LJARCH))
  ifeq (iOS,$(TARGET_SYS))
    DASM_AFLAGS+= -D IOS
  endif
else
ifeq (ppc,$(TARGET_LJARCH))
  ifneq (,$(findstring LJ_ARCH_SQRT 1,$(TARGET_TESTARCH)))
    DASM_AFLAGS+= -D SQRT
  endif
  ifneq (,$(findstring LJ_ARCH_ROUND 1,$(TARGET_TESTARCH)))
    DASM_AFLAGS+= -D ROUND
  endif
  ifneq (,$(findstring LJ_ARCH_PPC32ON64 1,$(TARGET_TESTARCH)))
    DASM_AFLAGS+= -D GPR64
  endif
  ifeq (PS3,$(TARGET_SYS))
    DASM_AFLAGS+= -D PPE -D TOC
  endif
  ifneq (,$(findstring LJ_ARCH_PPC64 ,$(TARGET_TESTARCH)))
    DASM_ARCH= ppc64
  endif
endif
endif
endif

DASM_FLAGS= $(DASM_XFLAGS) $(DASM_AFLAGS)
DASM_DASC= vm_$(DASM_ARCH).dasc

BUILDVM_O= host/buildvm.o host/buildvm_asm.o host/buildvm_peobj.o \
           host/buildvm_lib.o host/buildvm_fold.o
BUILDVM_T= host/buildvm
BUILDVM_X= $(BUILDVM_T)

HOST_O= $(MINILUA_O) $(BUILDVM_O)
HOST_T= $(MINILUA_T) $(BUILDVM_T)

LJVM_S= lj_vm.S
```

```
LJVM_O= lj_vm.o
LJVM_BOUT= $(LJVM_S)
LJVM_MODE= elfasm

LJLIB_O= lib_base.o lib_math.o lib_bit.o lib_string.o lib_table.o \
         lib_io.o lib_os.o lib_package.o lib_debug.o lib_jit.o lib_ffi.o
LJLIB_C= $(LJLIB_O:.o=.c)

LJCORE_O= lj_gc.o lj_err.o lj_char.o lj_bc.o lj_obj.o lj_buf.o \
          lj_str.o lj_tab.o lj_func.o lj_udata.o lj_meta.o lj_debug.o \
          lj_state.o lj_dispatch.o lj_vmevent.o lj_vmmath.o lj_strscan.o \
          lj_strfmt.o lj_api.o lj_profile.o \
          lj_lex.o lj_parse.o lj_bcread.o lj_bcwrite.o lj_load.o \
          lj_ir.o lj_opt_mem.o lj_opt_fold.o lj_opt_narrow.o \
          lj_opt_dce.o lj_opt_loop.o lj_opt_split.o lj_opt_sink.o \
          lj_mcode.o lj_snap.o lj_record.o lj_crecord.o lj_ffrecord.o \
          lj_asm.o lj_trace.o lj_gdbjit.o \
          lj_ctype.o lj_cdata.o lj_cconv.o lj_ccall.o lj_ccallback.o \
          lj_carith.o lj_clib.o lj_cparse.o \
          lj_lib.o lj_alloc.o lib_aux.o \
          $(LJLIB_O) lib_init.o

LJVMCORE_O= $(LJVM_O) $(LJCORE_O)
LJVMCORE_DYNO= $(LJVMCORE_O:.o=_dyn.o)

LIB_VMDEF= jit/vmdef.lua
LIB_VMDEFP= $(LIB_VMDEF)

LUAJIT_O= luajit.o
LUAJIT_A= libluajit.a
LUAJIT_SO= libluajit.so
LUAJIT_T= luajit

ALL_T= $(LUAJIT_T) $(LUAJIT_A) $(LUAJIT_SO) $(HOST_T)
ALL_HDRGEN= lj_bcdef.h lj_ffdef.h lj_libdef.h lj_recdef.h lj_folddef.h \
            host/buildvm_arch.h
ALL_GEN= $(LJVM_S) $(ALL_HDRGEN) $(LIB_VMDEFP)
WIN_RM= *.obj *.lib *.exp *.dll *.exe *.manifest *.pdb *.ilk
ALL_RM= $(ALL_T) $(ALL_GEN) *.o host/*.o $(WIN_RM)

##############################################################################
# Build mode handling.
##############################################################################

# Mixed mode defaults.
TARGET_O= $(LUAJIT_A)
TARGET_T= $(LUAJIT_T) $(LUAJIT_SO)
TARGET_DEP= $(LIB_VMDEF) $(LUAJIT_SO)

ifeq (Windows,$(TARGET_SYS))
  TARGET_DYNCC= $(STATIC_CC)
  LJVM_MODE= peobj
  LJVM_BOUT= $(LJVM_O)
  LUAJIT_T= luajit.exe
  ifeq (cygwin,$(HOST_MSYS))
    LUAJIT_SO= cyg$(TARGET_DLLNAME)
  else
    LUAJIT_SO= $(TARGET_DLLNAME)
  endif
  # Mixed mode is not supported on Windows. And static mode doesn't work well.
  # C modules cannot be loaded, because they bind to lua51.dll.
  ifneq (static,$(BUILDMODE))
    BUILDMODE= dynamic
    TARGET_XCFLAGS+= -DLUA_BUILD_AS_DLL
  endif
endif
ifeq (Darwin,$(TARGET_SYS))
  LJVM_MODE= machasm
endif
ifeq (iOS,$(TARGET_SYS))
  LJVM_MODE= machasm
endif
ifeq (SunOS,$(TARGET_SYS))
  BUILDMODE= static
endif
ifeq (PS3,$(TARGET_SYS))
```

```
  BUILDMODE= static
endif

ifeq (Windows,$(HOST_SYS))
  MINILUA_T= host/minilua.exe
  BUILDVM_T= host/buildvm.exe
  ifeq (,$(HOST_MSYS))
    MINILUA_X= host\minilua
    BUILDVM_X= host\buildvm
    ALL_RM:= $(subst /,\,$(ALL_RM))
  endif
endif

ifeq (static,$(BUILDMODE))
  TARGET_DYNCC= @:
  TARGET_T= $(LUAJIT_T)
  TARGET_DEP= $(LIB_VMDEF)
else
ifeq (dynamic,$(BUILDMODE))
  ifneq (Windows,$(TARGET_SYS))
    TARGET_CC= $(DYNAMIC_CC)
  endif
  TARGET_DYNCC= @:
  LJVMCORE_DYNO= $(LJVMCORE_O)
  TARGET_O= $(LUAJIT_SO)
  TARGET_XLDFLAGS+= $(TARGET_DYNXLDOPTS)
else
ifeq (Darwin,$(TARGET_SYS))
  TARGET_DYNCC= @:
  LJVMCORE_DYNO= $(LJVMCORE_O)
endif
ifeq (iOS,$(TARGET_SYS))
  TARGET_DYNCC= @:
  LJVMCORE_DYNO= $(LJVMCORE_O)
endif
endif
endif

Q= @
E= @echo
#Q=
#E= @:

##############################################################################
# Make targets.
##############################################################################

default all:        $(TARGET_T)

amalg:
        @grep "^[+|]" ljamalg.c
        $(MAKE) all "LJCORE_O=ljamalg.o"

clean:
        $(HOST_RM) $(ALL_RM)

libbc:
        ./$(LUAJIT_T) host/genlibbc.lua -o host/buildvm_libbc.h $(LJLIB_C)
        $(MAKE) all

depend:
        @for file in $(ALL_HDRGEN); do \
          test -f $$file || touch $$file; \
          done
        @$(HOST_CC) $(HOST_ACFLAGS) -MM *.c host/*.c | \
          sed -e "s| [^ ]*/dasm_\S*\.h||g" \
              -e "s|^\([^l ]\)|host/\1|" \
              -e "s| lj_target_\S*\.h| lj_target_*.h|g" \
              -e "s| lj_emit_\S*\.h| lj_emit_*.h|g" \
              -e "s| lj_asm_\S*\.h| lj_asm_*.h|g" >Makefile.dep
        @for file in $(ALL_HDRGEN); do \
          test -s $$file || $(HOST_RM) $$file; \
          done

.PHONY: default all amalg clean libbc depend

```

```
601 ##############################################################################
602 # Rules for generated files.
603 ##############################################################################
604
605 $(MINILUA_T): $(MINILUA_O)
606         $(E) "HOSTLINK  $@"
607         $(Q)$(HOST_CC) $(HOST_ALDFLAGS) -o $@ $(MINILUA_O) $(MINILUA_LIBS) $(HOST_ALIBS)
608
609 host/buildvm_arch.h: $(DASM_DASC) $(DASM_DEP)
610         $(E) "DYNASM    $@"
611         $(Q)$(DASM) $(DASM_FLAGS) -o $@ $(DASM_DASC)
612
613 host/buildvm.o: $(DASM_DIR)/dasm_*.h
614
615 $(BUILDVM_T): $(BUILDVM_O)
616         $(E) "HOSTLINK  $@"
617         $(Q)$(HOST_CC) $(HOST_ALDFLAGS) -o $@ $(BUILDVM_O) $(HOST_ALIBS)
618
619 $(LJVM_BOUT): $(BUILDVM_T)
620         $(E) "BUILDVM   $@"
621         $(Q)$(BUILDVM_X) -m $(LJVM_MODE) -o $@
622
623 lj_bcdef.h: $(BUILDVM_T) $(LJLIB_C)
624         $(E) "BUILDVM   $@"
625         $(Q)$(BUILDVM_X) -m bcdef -o $@ $(LJLIB_C)
626
627 lj_ffdef.h: $(BUILDVM_T) $(LJLIB_C)
628         $(E) "BUILDVM   $@"
629         $(Q)$(BUILDVM_X) -m ffdef -o $@ $(LJLIB_C)
630
631 lj_libdef.h: $(BUILDVM_T) $(LJLIB_C)
632         $(E) "BUILDVM   $@"
633         $(Q)$(BUILDVM_X) -m libdef -o $@ $(LJLIB_C)
634
635 lj_recdef.h: $(BUILDVM_T) $(LJLIB_C)
636         $(E) "BUILDVM   $@"
637         $(Q)$(BUILDVM_X) -m recdef -o $@ $(LJLIB_C)
638
639 $(LIB_VMDEF): $(BUILDVM_T) $(LJLIB_C)
640         $(E) "BUILDVM   $@"
641         $(Q)$(BUILDVM_X) -m vmdef -o $(LIB_VMDEFP) $(LJLIB_C)
642
643 lj_folddef.h: $(BUILDVM_T) lj_opt_fold.c
644         $(E) "BUILDVM   $@"
645         $(Q)$(BUILDVM_X) -m folddef -o $@ lj_opt_fold.c
646
647 ##############################################################################
648 # Object file rules.
649 ##############################################################################
650
651 %.o: %.c
652         $(E) "CC        $@"
653         $(Q)$(TARGET_DYNCC) $(TARGET_ACFLAGS) -c -o $(@:.o=_dyn.o) $<
654         $(Q)$(TARGET_CC) $(TARGET_ACFLAGS) -c -o $@ $<
655
656 %.o: %.S
657         $(E) "ASM       $@"
658         $(Q)$(TARGET_DYNCC) $(TARGET_ASFLAGS) -c -o $(@:.o=_dyn.o) $<
659         $(Q)$(TARGET_CC) $(TARGET_ASFLAGS) -c -o $@ $<
660
661 $(LUAJIT_O):
662         $(E) "CC        $@"
663         $(Q)$(TARGET_STCC) $(TARGET_ACFLAGS) -c -o $@ $<
664
665 $(HOST_O): %.o: %.c
666         $(E) "HOSTCC    $@"
667         $(Q)$(HOST_CC) $(HOST_ACFLAGS) -c -o $@ $<
668
669 include Makefile.dep
670
671 ##############################################################################
672 # Target file rules.
673 ##############################################################################
674
675 $(LUAJIT_A): $(LJVMCORE_O)
676         $(E) "AR        $@"
```

```
677         $(Q)$(TARGET_AR) $@ $(LJVMCORE_O)
678
679 # The dependency on _O, but linking with _DYNO is intentional.
680 $(LUAJIT_SO): $(LJVMCORE_O)
681         $(E) "DYNLINK   $@"
682         $(Q)$(TARGET_LD) $(TARGET_ASHLDFLAGS) -o $@ $(LJVMCORE_DYNO) $(TARGET_ALIBS)
683         $(Q)$(TARGET_STRIP) $@
684
685 $(LUAJIT_T): $(TARGET_O) $(LUAJIT_O) $(TARGET_DEP)
686         $(E) "LINK      $@"
687         $(Q)$(TARGET_LD) $(TARGET_ALDFLAGS) -o $@ $(LUAJIT_O) $(TARGET_O) $(TARGET_ALIBS)
688         $(Q)$(TARGET_STRIP) $@
689         $(E) "OK        Successfully built LuaJIT"
690
691 ##############################################################################
```

One Level Up Top Level

src/host/ - luajit-2.0-src

- buildvm.c
- buildvm.h
- buildvm_arch.h
- buildvm_asm.c
- buildvm_fold.c
- buildvm_lib.c
- buildvm_libbc.h
- buildvm_peobj.c
- genlibbc.lua

src/host/buildvm.c - luajit-2.0-src

Global variables defined

- bc_names
- ir_names
- ircall_names
- irfield_names
- irfpm_names
- irt_names
- modenames
- relocmap
- trace_errors

Functions defined

- build_code
- collect_reloc
- emit_bcdef
- emit_raw
- emit_vmdef
- lower
- main
- owrite
- parseargs
- parsemode
- sym_decorate
- sym_insert
- usage

Macros defined

- BCNAME
- BCNAME
- BUILDNAME
- BUILDNAME
- DASM_ALIGNED_WRITES

- DASM_CHECKS
- DASM_EXTERN
- Dst
- Dst_DECL
- Dst_REF
- FLNAME
- FLNAME
- FPMNAME
- FPMNAME
- IRCALLNAME
- IRCALLNAME
- IRNAME
- IRNAME
- IRTNAME
- IRTNAME
- NRELOCSYM
- TREDEF

Source code

```
/*
** LuaJIT VM builder.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** This is a tool to build the hand-tuned assembler code required for
** LuaJIT's bytecode interpreter. It supports a variety of output formats
** to feed different toolchains (see usage() below).
**
** This tool is not particularly optimized because it's only used while
** _building_ LuaJIT. There's no point in distributing or installing it.
** Only the object code generated by this tool is linked into LuaJIT.
**
** Caveat: some memory is not free'd, error handling is lazy.
** It's a one-shot tool -- any effort fixing this would be wasted.
*/

#include "buildvm.h"
#include "lj_obj.h"
#include "lj_gc.h"
#include "lj_bc.h"
#include "lj_ir.h"
#include "lj_ircall.h"
#include "lj_frame.h"
#include "lj_dispatch.h"
#if LJ_HASFFI
#include "lj_ctype.h"
#include "lj_ccall.h"
#endif
#include "luajit.h"

#if defined(_WIN32)
#include <fcntl.h>
#include <io.h>
#endif
```

```
35
36  /* ------------------------------------------------------------------------ */
37
38  /* DynASM glue definitions. */
39  #define Dst                ctx
40  #define Dst_DECL        BuildCtx *ctx
41  #define Dst_REF                (ctx->D)
42  #define DASM_CHECKS        1
43
44  #include "../dynasm/dasm_proto.h"
45
46  /* Glue macros for DynASM. */
47  static int collect_reloc(BuildCtx *ctx, uint8_t *addr, int idx, int type);
48
49  #define DASM_EXTERN(ctx, addr, idx, type) \
50    collect_reloc(ctx, addr, idx, type)
51
52  /* ------------------------------------------------------------------------ */
53
54  /* Avoid trouble if cross-compiling for an x86 target. Speed doesn't matter. */
55  #define DASM_ALIGNED_WRITES        1
56
57  /* Embed architecture-specific DynASM encoder. */
58  #if LJ_TARGET_X86ORX64
59  #include "../dynasm/dasm_x86.h"
60  #elif LJ_TARGET_ARM
61  #include "../dynasm/dasm_arm.h"
62  #elif LJ_TARGET_ARM64
63  #include "../dynasm/dasm_arm64.h"
64  #elif LJ_TARGET_PPC
65  #include "../dynasm/dasm_ppc.h"
66  #elif LJ_TARGET_MIPS
67  #include "../dynasm/dasm_mips.h"
68  #else
69  #error "No support for this architecture (yet)"
70  #endif
71
72  /* Embed generated architecture-specific backend. */
73  #include "buildvm_arch.h"
74
75  /* ------------------------------------------------------------------------ */
76
77  void owrite(BuildCtx *ctx, const void *ptr, size_t sz)
78  {
79    if (fwrite(ptr, 1, sz, ctx->fp) != sz) {
80      fprintf(stderr, "Error: cannot write to output file: %s\n",
81              strerror(errno));
82      exit(1);
83    }
84  }
85
86  /* ------------------------------------------------------------------------ */
87
88  /* Emit code as raw bytes. Only used for DynASM debugging. */
89  static void emit_raw(BuildCtx *ctx)
90  {
91    owrite(ctx, ctx->code, ctx->codesz);
92  }
93
94  /* -- Build machine code ------------------------------------------------- */
95
96  static const char *sym_decorate(BuildCtx *ctx,
97                                  const char *prefix, const char *suffix)
98  {
99    char name[256];
100    char *p;
101   #if LJ_64
102    const char *symprefix = ctx->mode == BUILD_machasm ? "_" : "";
103   #elif LJ_TARGET_XBOX360
104    const char *symprefix = "";
105   #else
106    const char *symprefix = ctx->mode != BUILD_elfasm ? "_" : "";
107   #endif
108    sprintf(name, "%s%s%s", symprefix, prefix, suffix);
109    p = strchr(name, '@');
110    if (p) {
```

```
#if LJ_TARGET_X86ORX64
    if (!LJ_64 && (ctx->mode == BUILD_coffasm || ctx->mode == BUILD_peobj))
      name[0] = '@';
    else
      *p = '\0';
#elif LJ_TARGET_PPC && !LJ_TARGET_CONSOLE
    /* Keep @plt etc. */
#else
    *p = '\0';
#endif
  }
  p = (char *)malloc(strlen(name)+1);  /* MSVC doesn't like strdup. */
  strcpy(p, name);
  return p;
}

#define NRELOCSYM        (sizeof(extnames)/sizeof(extnames[0])-1)

static int relocmap[NRELOCSYM];

/* Collect external relocations. */
static int collect_reloc(BuildCtx *ctx, uint8_t *addr, int idx, int type)
{
  if (ctx->nreloc >= BUILD_MAX_RELOC) {
    fprintf(stderr, "Error: too many relocations, increase BUILD_MAX_RELOC.\n");
    exit(1);
  }
  if (relocmap[idx] < 0) {
    relocmap[idx] = ctx->nrelocsym;
    ctx->relocsym[ctx->nrelocsym] = sym_decorate(ctx, "", extnames[idx]);
    ctx->nrelocsym++;
  }
  ctx->reloc[ctx->nreloc].ofs = (int32_t)(addr - ctx->code);
  ctx->reloc[ctx->nreloc].sym = relocmap[idx];
  ctx->reloc[ctx->nreloc].type = type;
  ctx->nreloc++;
#if LJ_TARGET_XBOX360
  return (int)(ctx->code - addr) + 4;  /* Encode symbol offset of .text. */
#else
  return 0;  /* Encode symbol offset of 0. */
#endif
}

/* Naive insertion sort. Performance doesn't matter here. */
static void sym_insert(BuildCtx *ctx, int32_t ofs,
                       const char *prefix, const char *suffix)
{
  ptrdiff_t i = ctx->nsym++;
  while (i > 0) {
    if (ctx->sym[i-1].ofs <= ofs)
      break;
    ctx->sym[i] = ctx->sym[i-1];
    i--;
  }
  ctx->sym[i].ofs = ofs;
  ctx->sym[i].name = sym_decorate(ctx, prefix, suffix);
}

/* Build the machine code. */
static int build_code(BuildCtx *ctx)
{
  int status;
  int i;

  /* Initialize DynASM structures. */
  ctx->nglob = GLOB__MAX;
  ctx->glob = (void **)malloc(ctx->nglob*sizeof(void *));
  memset(ctx->glob, 0, ctx->nglob*sizeof(void *));
  ctx->nreloc = 0;

  ctx->globnames = globnames;
  ctx->extnames = extnames;
  ctx->relocsym = (const char **)malloc(NRELOCSYM*sizeof(const char *));
  ctx->nrelocsym = 0;
  for (i = 0; i < (int)NRELOCSYM; i++) relocmap[i] = -1;

```

```
  ctx->dasm_ident = DASM_IDENT;
  ctx->dasm_arch = DASM_ARCH;

  dasm_init(Dst, DASM_MAXSECTION);
  dasm_setupglobal(Dst, ctx->glob, ctx->nglob);
  dasm_setup(Dst, build_actionlist);

  /* Call arch-specific backend to emit the code. */
  ctx->npc = build_backend(ctx);

  /* Finalize the code. */
  (void)dasm_checkstep(Dst, -1);
  if ((status = dasm_link(Dst, &ctx->codesz))) return status;
  ctx->code = (uint8_t *)malloc(ctx->codesz);
  if ((status = dasm_encode(Dst, (void *)ctx->code))) return status;

  /* Allocate symbol table and bytecode offsets. */
  ctx->beginsym = sym_decorate(ctx, "", LABEL_PREFIX "vm_asm_begin");
  ctx->sym = (BuildSym *)malloc((ctx->npc+ctx->nglob+1)*sizeof(BuildSym));
  ctx->nsym = 0;
  ctx->bc_ofs = (int32_t *)malloc(ctx->npc*sizeof(int32_t));

  /* Collect the opcodes (PC labels). */
  for (i = 0; i < ctx->npc; i++) {
    int32_t ofs = dasm_getpclabel(Dst, i);
    if (ofs < 0) return 0x22000000|i;
    ctx->bc_ofs[i] = ofs;
    if ((LJ_HASJIT ||
         !(i == BC_JFORI || i == BC_JFORL || i == BC_JITERL || i == BC_JLOOP ||
           i == BC_IFORL || i == BC_IITERL || i == BC_ILOOP)) &&
        (LJ_HASFFI || i != BC_KCDATA))
      sym_insert(ctx, ofs, LABEL_PREFIX_BC, bc_names[i]);
  }

  /* Collect the globals (named labels). */
  for (i = 0; i < ctx->nglob; i++) {
    const char *gl = globnames[i];
    int len = (int)strlen(gl);
    if (!ctx->glob[i]) {
      fprintf(stderr, "Error: undefined global %s\n", gl);
      exit(2);
    }
    /* Skip the _Z symbols. */
    if (!(len >= 2 && gl[len-2] == '_' && gl[len-1] == 'Z'))
      sym_insert(ctx, (int32_t)((uint8_t *)(ctx->glob[i]) - ctx->code),
                 LABEL_PREFIX, globnames[i]);
  }

  /* Close the address range. */
  sym_insert(ctx, (int32_t)ctx->codesz, "", "");
  ctx->nsym--;

  dasm_free(Dst);

  return 0;
}

/* -- Generate VM enums ------------------------------------------------- */

const char *const bc_names[] = {
#define BCNAME(name, ma, mb, mc, mt)       #name,
BCDEF(BCNAME)
#undef BCNAME
  NULL
};

const char *const ir_names[] = {
#define IRNAME(name, m, m1, m2)        #name,
IRDEF(IRNAME)
#undef IRNAME
  NULL
};

const char *const irt_names[] = {
#define IRTNAME(name, size)        #name,
IRTDEF(IRTNAME)
```

```
#undef IRTNAME
  NULL
};

const char *const irfpm_names[] = {
#define FPMNAME(name)               #name,
IRFPMDEF(FPMNAME)
#undef FPMNAME
  NULL
};

const char *const irfield_names[] = {
#define FLNAME(name, ofs)        #name,
IRFLDEF(FLNAME)
#undef FLNAME
  NULL
};

const char *const ircall_names[] = {
#define IRCALLNAME(cond, name, nargs, kind, type, flags)        #name,
IRCALLDEF(IRCALLNAME)
#undef IRCALLNAME
  NULL
};

static const char *const trace_errors[] = {
#define TREDEF(name, msg)        msg,
#include "lj_traceerr.h"
  NULL
};

static const char *lower(char *buf, const char *s)
{
  char *p = buf;
  while (*s) {
    *p++ = (*s >= 'A' && *s <= 'Z') ? *s+0x20 : *s;
    s++;
  }
  *p = '\0';
  return buf;
}

/* Emit C source code for bytecode-related definitions. */
static void emit_bcdef(BuildCtx *ctx)
{
  int i;
  fprintf(ctx->fp, "/* This is a generated file. DO NOT EDIT! */\n\n");
  fprintf(ctx->fp, "LJ_DATADEF const uint16_t lj_bc_ofs[] = {\n");
  for (i = 0; i < ctx->npc; i++) {
    if (i != 0)
      fprintf(ctx->fp, ",\n");
    fprintf(ctx->fp, "%d", ctx->bc_ofs[i]);
  }
}

/* Emit VM definitions as Lua code for debug modules. */
static void emit_vmdef(BuildCtx *ctx)
{
  char buf[80];
  int i;
  fprintf(ctx->fp, "-- This is a generated file. DO NOT EDIT!\n\n");
  fprintf(ctx->fp, "return {\n\n");

  fprintf(ctx->fp, "bcnames = \"");
  for (i = 0; bc_names[i]; i++) fprintf(ctx->fp, "%-6s", bc_names[i]);
  fprintf(ctx->fp, "\",\n\n");

  fprintf(ctx->fp, "irnames = \"");
  for (i = 0; ir_names[i]; i++) fprintf(ctx->fp, "%-6s", ir_names[i]);
  fprintf(ctx->fp, "\",\n\n");

  fprintf(ctx->fp, "irfpm = { [0]=");
  for (i = 0; irfpm_names[i]; i++)
    fprintf(ctx->fp, "\"%s\", ", lower(buf, irfpm_names[i]));
  fprintf(ctx->fp, "},\n\n");

```

```
  fprintf(ctx->fp, "irfield = { [0]=");
  for (i = 0; irfield_names[i]; i++) {
    char *p;
    lower(buf, irfield_names[i]);
    p = strchr(buf, '_');
    if (p) *p = '.';
    fprintf(ctx->fp, "\"%s\", ", buf);
  }
  fprintf(ctx->fp, "},\n\n");

  fprintf(ctx->fp, "ircall = {\n[0]=");
  for (i = 0; ircall_names[i]; i++)
    fprintf(ctx->fp, "\"%s\",\n", ircall_names[i]);
  fprintf(ctx->fp, "},\n\n");

  fprintf(ctx->fp, "traceerr = {\n[0]=");
  for (i = 0; trace_errors[i]; i++)
    fprintf(ctx->fp, "\"%s\",\n", trace_errors[i]);
  fprintf(ctx->fp, "},\n\n");
}

/* -- Argument parsing ---------------------------------------------------- */

/* Build mode names. */
static const char *const modenames[] = {
#define BUILDNAME(name)               #name,
BUILDDEF(BUILDNAME)
#undef BUILDNAME
  NULL
};

/* Print usage information and exit. */
static void usage(void)
{
  int i;
  fprintf(stderr, LUAJIT_VERSION " VM builder.\n");
  fprintf(stderr, LUAJIT_COPYRIGHT ", " LUAJIT_URL "\n");
  fprintf(stderr, "Target architecture: " LJ_ARCH_NAME "\n\n");
  fprintf(stderr, "Usage: buildvm -m mode [-o outfile] [infiles...]\n\n");
  fprintf(stderr, "Available modes:\n");
  for (i = 0; i < BUILD__MAX; i++)
    fprintf(stderr, "  %s\n", modenames[i]);
  exit(1);
}

/* Parse the output mode name. */
static BuildMode parsemode(const char *mode)
{
  int i;
  for (i = 0; modenames[i]; i++)
    if (!strcmp(mode, modenames[i]))
      return (BuildMode)i;
  usage();
  return (BuildMode)-1;
}

/* Parse arguments. */
static void parseargs(BuildCtx *ctx, char **argv)
{
  const char *a;
  int i;
  ctx->mode = (BuildMode)-1;
  ctx->outname = "-";
  for (i = 1; (a = argv[i]) != NULL; i++) {
    if (a[0] != '-')
      break;
    switch (a[1]) {
    case '-':
      if (a[2]) goto err;
      i++;
      goto ok;
    case '\0':
      goto ok;
    case 'm':
      i++;
      if (a[2] || argv[i] == NULL) goto err;
```

```
      ctx->mode = parsemode(argv[i]);
      break;
    case 'o':
      i++;
      if (a[2] || argv[i] == NULL) goto err;
      ctx->outname = argv[i];
      break;
    default: err:
      usage();
      break;
    }
  }
ok:
  ctx->args = argv+i;
  if (ctx->mode == (BuildMode)-1) goto err;
}

int main(int argc, char **argv)
{
  BuildCtx ctx_;
  BuildCtx *ctx = &ctx_;
  int status, binmode;

  if (sizeof(void *) != 4*LJ_32+8*LJ_64) {
    fprintf(stderr,"Error: pointer size mismatch in cross-build.\n");
    fprintf(stderr,"Try: make HOST_CC=\"gcc -m32\" CROSS=...\n\n");
    return 1;
  }

  UNUSED(argc);
  parseargs(ctx, argv);

  if ((status = build_code(ctx))) {
    fprintf(stderr,"Error: DASM error %08x\n", status);
    return 1;
  }

  switch (ctx->mode) {
  case BUILD_peobj:
  case BUILD_raw:
    binmode = 1;
    break;
  default:
    binmode = 0;
    break;
  }

  if (ctx->outname[0] == '-' && ctx->outname[1] == '\0') {
    ctx->fp = stdout;
#if defined(_WIN32)
    if (binmode)
      _setmode(_fileno(stdout), _O_BINARY);  /* Yuck. */
#endif
  } else if (!(ctx->fp = fopen(ctx->outname, binmode ? "wb" : "w"))) {
    fprintf(stderr, "Error: cannot open output file '%s': %s\n",
            ctx->outname, strerror(errno));
    exit(1);
  }

  switch (ctx->mode) {
  case BUILD_elfasm:
  case BUILD_coffasm:
  case BUILD_machasm:
    emit_asm(ctx);
    emit_asm_debug(ctx);
    break;
  case BUILD_peobj:
    emit_peobj(ctx);
    break;
  case BUILD_raw:
    emit_raw(ctx);
    break;
  case BUILD_bcdef:
    emit_bcdef(ctx);
    emit_lib(ctx);
    break;
```

```
491   case BUILD_vmdef:
492     emit_vmdef(ctx);
493     emit_lib(ctx);
494     fprintf(ctx->fp, "}\n\n");
495     break;
496   case BUILD_ffdef:
497   case BUILD_libdef:
498   case BUILD_recdef:
499     emit_lib(ctx);
500     break;
501   case BUILD_folddef:
502     emit_fold(ctx);
503     break;
504   default:
505     break;
506   }
507
508   fflush(ctx->fp);
509   if (ferror(ctx->fp)) {
510     fprintf(stderr, "Error: cannot write to output file: %s\n",
511             strerror(errno));
512     exit(1);
513   }
514   fclose(ctx->fp);
515
516   return 0;
517 }
518
```

One Level Up Top Level

src/lj_arch.h - luajit-2.0-src

Macros defined

- LJ_32
- LJ_32
- LJ_52
- LJ_52
- LJ_64
- LJ_64
- LJ_ABI_EABI
- LJ_ABI_SOFTFP
- LJ_ABI_SOFTFP
- LJ_ABI_WIN
- LJ_ABI_WIN
- LJ_ABI_WIN
- LJ_ARCH_BITS
- LJ_ARCH_BITS
- LJ_ARCH_BITS
- LJ_ARCH_BITS
- LJ_ARCH_BITS
- LJ_ARCH_BITS
- LJ_ARCH_BITS
- LJ_ARCH_CELL
- LJ_ARCH_ENDIAN
- LJ_ARCH_ENDIAN
- LJ_ARCH_ENDIAN
- LJ_ARCH_ENDIAN
- LJ_ARCH_ENDIAN
- LJ_ARCH_ENDIAN
- LJ_ARCH_ENDIAN
- LJ_ARCH_ENDIAN
- LJ_ARCH_HASFPU
- LJ_ARCH_HASFPU

- LJ_ARCH_NAME
- LJ_ARCH_NAME
- LJ_ARCH_NAME
- LJ_ARCH_NAME
- LJ_ARCH_NAME
- LJ_ARCH_NAME
- LJ_ARCH_NAME
- LJ_ARCH_NAME
- LJ_ARCH_NAME
- LJ_ARCH_NOFFI
- LJ_ARCH_NOJIT
- LJ_ARCH_NOJIT
- LJ_ARCH_NUMMODE
- LJ_ARCH_NUMMODE
- LJ_ARCH_NUMMODE
- LJ_ARCH_NUMMODE
- LJ_ARCH_NUMMODE
- LJ_ARCH_NUMMODE
- LJ_ARCH_PPC32ON64
- LJ_ARCH_PPC64
- LJ_ARCH_ROUND
- LJ_ARCH_SQRT
- LJ_ARCH_VERSION
- LJ_ARCH_VERSION
- LJ_ARCH_VERSION
- LJ_ARCH_VERSION
- LJ_ARCH_VERSION
- LJ_ARCH_VERSION
- LJ_ARCH_VERSION
- LJ_ARCH_VERSION
- LJ_ARCH_VERSION
- LJ_ARCH_VERSION
- LJ_ARCH_VERSION

- LJ_ARCH_VERSION
- LJ_ARCH_VERSION
- LJ_ARCH_VERSION
- LJ_ARCH_XENON
- LJ_BE
- LJ_BE
- LJ_DUALNUM
- LJ_DUALNUM
- LJ_ENDIAN_LOHI
- LJ_ENDIAN_LOHI
- LJ_ENDIAN_SELECT
- LJ_ENDIAN_SELECT
- LJ_FR2
- LJ_FR2
- LJ_GC64
- LJ_GC64
- LJ_HASFFI
- LJ_HASFFI
- LJ_HASJIT
- LJ_HASJIT
- LJ_HASPROFILE
- LJ_HASPROFILE
- LJ_HASPROFILE
- LJ_HASPROFILE
- LJ_HASPROFILE
- LJ_LE
- LJ_LE
- LJ_NO_UNWIND
- LJ_NUMMODE_DUAL
- LJ_NUMMODE_DUAL_SINGLE
- LJ_NUMMODE_SINGLE
- LJ_NUMMODE_SINGLE_DUAL
- LJ_OS_NAME

- LJ_OS_NAME
- LJ_OS_NAME
- LJ_OS_NAME
- LJ_OS_NAME
- LJ_OS_NAME
- LJ_OS_NOJIT
- LJ_PAGESIZE
- LJ_PROFILE_PTHREAD
- LJ_PROFILE_SIGPROF
- LJ_PROFILE_WTHREAD
- LJ_SOFTFP
- LJ_TARGET_ARM
- LJ_TARGET_ARM64
- LJ_TARGET_CONSOLE
- LJ_TARGET_CONSOLE
- LJ_TARGET_CONSOLE
- LJ_TARGET_CONSOLE
- LJ_TARGET_DLOPEN
- LJ_TARGET_EHRETREG
- LJ_TARGET_EHRETREG
- LJ_TARGET_EHRETREG
- LJ_TARGET_EHRETREG
- LJ_TARGET_EHRETREG
- LJ_TARGET_EHRETREG
- LJ_TARGET_GC64
- LJ_TARGET_GC64
- LJ_TARGET_IOS
- LJ_TARGET_JUMPRANGE
- LJ_TARGET_JUMPRANGE
- LJ_TARGET_JUMPRANGE
- LJ_TARGET_JUMPRANGE
- LJ_TARGET_JUMPRANGE
- LJ_TARGET_LINUX

- LJ_TARGET_MASKROT
- LJ_TARGET_MASKROT
- LJ_TARGET_MASKROT
- LJ_TARGET_MASKROT
- LJ_TARGET_MASKROT
- LJ_TARGET_MASKROT
- LJ_TARGET_MASKSHIFT
- LJ_TARGET_MASKSHIFT
- LJ_TARGET_MASKSHIFT
- LJ_TARGET_MASKSHIFT
- LJ_TARGET_MASKSHIFT
- LJ_TARGET_MASKSHIFT
- LJ_TARGET_MIPS
- LJ_TARGET_OSX
- LJ_TARGET_POSIX
- LJ_TARGET_PPC
- LJ_TARGET_PS3
- LJ_TARGET_PS4
- LJ_TARGET_PSVITA
- LJ_TARGET_UNALIGNED
- LJ_TARGET_UNALIGNED
- LJ_TARGET_UNALIGNED
- LJ_TARGET_UNIFYROT
- LJ_TARGET_UNIFYROT
- LJ_TARGET_UNIFYROT
- LJ_TARGET_UNIFYROT
- LJ_TARGET_WINDOWS
- LJ_TARGET_X64
- LJ_TARGET_X86
- LJ_TARGET_X86ORX64
- LJ_TARGET_X86ORX64
- LJ_TARGET_XBOX360
- LUAJIT_ARCH_ARM

- LUAJIT_ARCH_ARM64
- LUAJIT_ARCH_MIPS
- LUAJIT_ARCH_PPC
- LUAJIT_ARCH_X64
- LUAJIT_ARCH_X86
- LUAJIT_ARCH_arm
- LUAJIT_ARCH_arm64
- LUAJIT_ARCH_mips
- LUAJIT_ARCH_ppc
- LUAJIT_ARCH_x64
- LUAJIT_ARCH_x86
- LUAJIT_BE
- LUAJIT_LE
- LUAJIT_NO_EXP2
- LUAJIT_NO_LOG2
- LUAJIT_OS
- LUAJIT_OS
- LUAJIT_OS
- LUAJIT_OS
- LUAJIT_OS
- LUAJIT_OS
- LUAJIT_OS_BSD
- LUAJIT_OS_LINUX
- LUAJIT_OS_OSX
- LUAJIT_OS_OTHER
- LUAJIT_OS_POSIX
- LUAJIT_OS_WINDOWS
- LUAJIT_TARGET
- LUAJIT_TARGET
- LUAJIT_TARGET
- LUAJIT_TARGET
- LUAJIT_TARGET
- LUAJIT_TARGET

- NULL
- NULL
- _LJ_ARCH_H

Source code

```
1  /*
2  ** Target architecture selection.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_ARCH_H
7  #define _LJ_ARCH_H
8
9  #include "lua.h"
10
11  /* Target endianess. */
12  #define LUAJIT_LE        0
13  #define LUAJIT_BE        1
14
15  /* Target architectures. */
16  #define LUAJIT_ARCH_X86               1
17  #define LUAJIT_ARCH_x86               1
18  #define LUAJIT_ARCH_X64               2
19  #define LUAJIT_ARCH_x64               2
20  #define LUAJIT_ARCH_ARM               3
21  #define LUAJIT_ARCH_arm               3
22  #define LUAJIT_ARCH_ARM64        4
23  #define LUAJIT_ARCH_arm64        4
24  #define LUAJIT_ARCH_PPC               5
25  #define LUAJIT_ARCH_ppc               5
26  #define LUAJIT_ARCH_MIPS        6
27  #define LUAJIT_ARCH_mips        6
28
29  /* Target OS. */
30  #define LUAJIT_OS_OTHER               0
31  #define LUAJIT_OS_WINDOWS        1
32  #define LUAJIT_OS_LINUX               2
33  #define LUAJIT_OS_OSX               3
34  #define LUAJIT_OS_BSD               4
35  #define LUAJIT_OS_POSIX               5
36
37  /* Select native target if no target defined. */
38  #ifndef LUAJIT_TARGET
39
40  #if defined(__i386) || defined(__i386__) || defined(_M_IX86)
41  #define LUAJIT_TARGET        LUAJIT_ARCH_X86
42  #elif defined(__x86_64__) || defined(__x86_64) || defined(_M_X64) || defined(_M_AMD64)
43  #define LUAJIT_TARGET        LUAJIT_ARCH_X64
44  #elif defined(__arm__) || defined(__arm) || defined(__ARM__) || defined(__ARM)
45  #define LUAJIT_TARGET        LUAJIT_ARCH_ARM
46  #elif defined(__aarch64__)
47  #define LUAJIT_TARGET        LUAJIT_ARCH_ARM64
48  #elif defined(__ppc__) || defined(__ppc) || defined(__PPC__) || defined(__PPC) || defined(__powerpc__) ||
defined(__powerpc) || defined(__POWERPC__) || defined(__POWERPC) || defined(_M_PPC)
49  #define LUAJIT_TARGET        LUAJIT_ARCH_PPC
50  #elif defined(__mips__) || defined(__mips) || defined(__MIPS__) || defined(__MIPS)
51  #define LUAJIT_TARGET        LUAJIT_ARCH_MIPS
52  #else
53  #error "No support for this architecture (yet)"
54  #endif
55
56  #endif
57
58  /* Select native OS if no target OS defined. */
59  #ifndef LUAJIT_OS
60
61  #if defined(_WIN32) && !defined(_XBOX_VER)
62  #define LUAJIT_OS        LUAJIT_OS_WINDOWS
63  #elif defined(__linux__)
64  #define LUAJIT_OS        LUAJIT_OS_LINUX
```

```
#elif defined(__MACH__) && defined(__APPLE__)
#define LUAJIT_OS        LUAJIT_OS_OSX
#elif (defined(__FreeBSD__) || defined(__FreeBSD_kernel__) || \
       defined(__NetBSD__) || defined(__OpenBSD__) || \
       defined(__DragonFly__)) && !defined(__ORBIS__)
#define LUAJIT_OS        LUAJIT_OS_BSD
#elif (defined(__sun__) && defined(__svr4__)) || defined(__CYGWIN__)
#define LUAJIT_OS        LUAJIT_OS_POSIX
#else
#define LUAJIT_OS        LUAJIT_OS_OTHER
#endif

#endif

/* Set target OS properties. */
#if LUAJIT_OS == LUAJIT_OS_WINDOWS
#define LJ_OS_NAME        "Windows"
#elif LUAJIT_OS == LUAJIT_OS_LINUX
#define LJ_OS_NAME        "Linux"
#elif LUAJIT_OS == LUAJIT_OS_OSX
#define LJ_OS_NAME        "OSX"
#elif LUAJIT_OS == LUAJIT_OS_BSD
#define LJ_OS_NAME        "BSD"
#elif LUAJIT_OS == LUAJIT_OS_POSIX
#define LJ_OS_NAME        "POSIX"
#else
#define LJ_OS_NAME        "Other"
#endif

#define LJ_TARGET_WINDOWS        (LUAJIT_OS == LUAJIT_OS_WINDOWS)
#define LJ_TARGET_LINUX             (LUAJIT_OS == LUAJIT_OS_LINUX)
#define LJ_TARGET_OSX               (LUAJIT_OS == LUAJIT_OS_OSX)
#define LJ_TARGET_IOS               (LJ_TARGET_OSX && (LUAJIT_TARGET == LUAJIT_ARCH_ARM || LUAJIT_TARGET == LUAJIT_ARCH_ARM64))
#define LJ_TARGET_POSIX              (LUAJIT_OS > LUAJIT_OS_WINDOWS)
#define LJ_TARGET_DLOPEN        LJ_TARGET_POSIX

#ifdef __CELLOS_LV2__
#define LJ_TARGET_PS3               1
#define LJ_TARGET_CONSOLE        1
#endif

#ifdef __ORBIS__
#define LJ_TARGET_PS4               1
#define LJ_TARGET_CONSOLE        1
#undef NULL
#define NULL ((void*)0)
#endif

#ifdef __psp2__
#define LJ_TARGET_PSVITA        1
#define LJ_TARGET_CONSOLE        1
#endif

#if _XBOX_VER >= 200
#define LJ_TARGET_XBOX360        1
#define LJ_TARGET_CONSOLE        1
#endif

#define LJ_NUMMODE_SINGLE        0        /* Single-number mode only. */
#define LJ_NUMMODE_SINGLE_DUAL        1        /* Default to single-number mode. */
#define LJ_NUMMODE_DUAL                2        /* Dual-number mode only. */
#define LJ_NUMMODE_DUAL_SINGLE        3        /* Default to dual-number mode. */

/* Set target architecture properties. */
#if LUAJIT_TARGET == LUAJIT_ARCH_X86

#define LJ_ARCH_NAME               "x86"
#define LJ_ARCH_BITS               32
#define LJ_ARCH_ENDIAN               LUAJIT_LE
#if LJ_TARGET_WINDOWS || __CYGWIN__
#define LJ_ABI_WIN               1
#else
#define LJ_ABI_WIN               0
#endif
#define LJ_TARGET_X86               1
```

```
#define LJ_TARGET_X86ORX64        1
#define LJ_TARGET_EHRETREG        0
#define LJ_TARGET_MASKSHIFT       1
#define LJ_TARGET_MASKROT        1
#define LJ_TARGET_UNALIGNED       1
#define LJ_ARCH_NUMMODE               LJ_NUMMODE_SINGLE_DUAL

#elif LUAJIT_TARGET == LUAJIT_ARCH_X64

#define LJ_ARCH_NAME              "x64"
#define LJ_ARCH_BITS              64
#define LJ_ARCH_ENDIAN              LUAJIT_LE
#define LJ_ABI_WIN                LJ_TARGET_WINDOWS
#define LJ_TARGET_X64               1
#define LJ_TARGET_X86ORX64        1
#define LJ_TARGET_EHRETREG        0
#define LJ_TARGET_JUMPRANGE       31        /* +-2^31 = +-2GB */
#define LJ_TARGET_MASKSHIFT       1
#define LJ_TARGET_MASKROT        1
#define LJ_TARGET_UNALIGNED       1
#define LJ_ARCH_NUMMODE               LJ_NUMMODE_SINGLE_DUAL

#elif LUAJIT_TARGET == LUAJIT_ARCH_ARM

#define LJ_ARCH_NAME              "arm"
#define LJ_ARCH_BITS              32
#define LJ_ARCH_ENDIAN              LUAJIT_LE
#if !defined(LJ_ARCH_HASFPU) && __SOFTFP__
#define LJ_ARCH_HASFPU              0
#endif
#if !defined(LJ_ABI_SOFTFP) && !__ARM_PCS_VFP
#define LJ_ABI_SOFTFP               1
#endif
#define LJ_ABI_EABI               1
#define LJ_TARGET_ARM               1
#define LJ_TARGET_EHRETREG        0
#define LJ_TARGET_JUMPRANGE       25        /* +-2^25 = +-32MB */
#define LJ_TARGET_MASKSHIFT       0
#define LJ_TARGET_MASKROT        1
#define LJ_TARGET_UNIFYROT        2       /* Want only IR_BROR. */
#define LJ_ARCH_NUMMODE               LJ_NUMMODE_DUAL

#if __ARM_ARCH___ARM_ARCH_8__ || __ARM_ARCH_8A__
#define LJ_ARCH_VERSION               80
#elif __ARM_ARCH_7__ || __ARM_ARCH_7A__ || __ARM_ARCH_7R__ || __ARM_ARCH_7S__ || __ARM_ARCH_7VE__
#define LJ_ARCH_VERSION               70
#elif __ARM_ARCH_6T2__
#define LJ_ARCH_VERSION               61
#elif __ARM_ARCH_6__ || __ARM_ARCH_6J__ || __ARM_ARCH_6K__ || __ARM_ARCH_6Z__ || __ARM_ARCH_6ZK__
#define LJ_ARCH_VERSION               60
#else
#define LJ_ARCH_VERSION               50
#endif

#elif LUAJIT_TARGET == LUAJIT_ARCH_ARM64

#define LJ_ARCH_NAME              "arm64"
#define LJ_ARCH_BITS              64
#define LJ_ARCH_ENDIAN              LUAJIT_LE
#define LJ_TARGET_ARM64             1
#define LJ_TARGET_EHRETREG        0
#define LJ_TARGET_JUMPRANGE       27        /* +-2^27 = +-128MB */
#define LJ_TARGET_MASKSHIFT       1
#define LJ_TARGET_MASKROT        1
#define LJ_TARGET_UNIFYROT        2       /* Want only IR_BROR. */
#define LJ_TARGET_GC64              1
#define LJ_ARCH_NUMMODE               LJ_NUMMODE_DUAL
#define LJ_ARCH_NOJIT             1       /* NYI */

#define LJ_ARCH_VERSION               80

#elif LUAJIT_TARGET == LUAJIT_ARCH_PPC

#if __BYTE_ORDER__ != __ORDER_BIG_ENDIAN__
#define LJ_ARCH_ENDIAN              LUAJIT_LE
#else
```

```
#define LJ_ARCH_ENDIAN		LUAJIT_BE
#endif

#if _LP64
#define LJ_ARCH_BITS		64
#if LJ_ARCH_ENDIAN == LUAJIT_LE
#define LJ_ARCH_NAME		"ppc64le"
#else
#define LJ_ARCH_NAME		"ppc64"
#endif
#else
#define LJ_ARCH_BITS		32
#define LJ_ARCH_NAME		"ppc"
#endif

#define LJ_TARGET_PPC		1
#define LJ_TARGET_EHRETREG	3
#define LJ_TARGET_JUMPRANGE	25	/* +-2^25 = +-32MB */
#define LJ_TARGET_MASKSHIFT	0
#define LJ_TARGET_MASKROT	1
#define LJ_TARGET_UNIFYROT	1	/* Want only IR_BROL. */
#define LJ_ARCH_NUMMODE		LJ_NUMMODE_DUAL_SINGLE

#if LJ_TARGET_CONSOLE
#define LJ_ARCH_PPC32ON64	1
#define LJ_ARCH_NOFFI		1
#elif LJ_ARCH_BITS == 64
#define LJ_ARCH_PPC64		1
#define LJ_TARGET_GC64		1
#define LJ_ARCH_NOJIT		1	/* NYI */
#endif

#if _ARCH_PWR7
#define LJ_ARCH_VERSION		70
#elif _ARCH_PWR6
#define LJ_ARCH_VERSION		60
#elif _ARCH_PWR5X
#define LJ_ARCH_VERSION		51
#elif _ARCH_PWR5
#define LJ_ARCH_VERSION		50
#elif _ARCH_PWR4
#define LJ_ARCH_VERSION		40
#else
#define LJ_ARCH_VERSION		0
#endif
#if _ARCH_PPCSQ
#define LJ_ARCH_SQRT		1
#endif
#if _ARCH_PWR5X
#define LJ_ARCH_ROUND		1
#endif
#if __PPU__
#define LJ_ARCH_CELL		1
#endif
#if LJ_TARGET_XBOX360
#define LJ_ARCH_XENON		1
#endif

#elif LUAJIT_TARGET == LUAJIT_ARCH_MIPS

#if defined(__MIPSEL__) || defined(__MIPSEL) || defined(_MIPSEL)
#define LJ_ARCH_NAME		"mipsel"
#define LJ_ARCH_ENDIAN		LUAJIT_LE
#else
#define LJ_ARCH_NAME		"mips"
#define LJ_ARCH_ENDIAN		LUAJIT_BE
#endif
#define LJ_ARCH_BITS		32
#define LJ_TARGET_MIPS		1
#define LJ_TARGET_EHRETREG	4
#define LJ_TARGET_JUMPRANGE	27	/* 2*2^27 = 256MB-aligned region */
#define LJ_TARGET_MASKSHIFT	1
#define LJ_TARGET_MASKROT	1
#define LJ_TARGET_UNIFYROT	2	/* Want only IR_BROR. */
#define LJ_ARCH_NUMMODE		LJ_NUMMODE_SINGLE

```

```
#if _MIPS_ARCH_MIPS32R2
#define LJ_ARCH_VERSION 20
#else
#define LJ_ARCH_VERSION 10
#endif

#else
#error "No target architecture defined"
#endif

#ifndef LJ_PAGESIZE
#define LJ_PAGESIZE 4096
#endif

/* Check for minimum required compiler versions. */
#if defined(__GNUC__)
#if LJ_TARGET_X86
#if (__GNUC__ < 3) || ((__GNUC__ == 3) && __GNUC_MINOR__ < 4)
#error "Need at least GCC 3.4 or newer"
#endif
#elif LJ_TARGET_X64
#if __GNUC__ < 4
#error "Need at least GCC 4.0 or newer"
#endif
#elif LJ_TARGET_ARM
#if (__GNUC__ < 4) || ((__GNUC__ == 4) && __GNUC_MINOR__ < 2)
#error "Need at least GCC 4.2 or newer"
#endif
#elif LJ_TARGET_ARM64
#if __clang__
#if (__clang_major__ < 3) || ((__clang_major__ == 3) && __clang_minor__ < 5)
#error "Need at least Clang 3.5 or newer"
#endif
#else
#if (__GNUC__ < 4) || ((__GNUC__ == 4) && __GNUC_MINOR__ < 8)
#error "Need at least GCC 4.8 or newer"
#endif
#endif
#elif !LJ_TARGET_PS3
#if (__GNUC__ < 4) || ((__GNUC__ == 4) && __GNUC_MINOR__ < 3)
#error "Need at least GCC 4.3 or newer"
#endif
#endif
#endif

/* Check target-specific constraints. */
#ifndef _BUILDVM_H
#if LJ_TARGET_X64
#if __USING_SJLJ_EXCEPTIONS__
#error "Need a C compiler with native exception handling on x64"
#endif
#elif LJ_TARGET_ARM
#if defined(__ARMEB__)
#error "No support for big-endian ARM"
#endif
#if __ARM_ARCH_6M__ || __ARM_ARCH_7M__ || __ARM_ARCH_7EM__
#error "No support for Cortex-M CPUs"
#endif
#if !(__ARM_EABI__ || LJ_TARGET_IOS)
#error "Only ARM EABI or iOS 3.0+ ABI is supported"
#endif
#elif LJ_TARGET_ARM64
#if defined(__AARCH64EB__)
#error "No support for big-endian ARM64"
#endif
#if defined(_ILP32)
#error "No support for ILP32 model on ARM64"
#endif
#elif LJ_TARGET_PPC
#if defined(_SOFT_FLOAT) || defined(_SOFT_DOUBLE)
#error "No support for PowerPC CPUs without double-precision FPU"
#endif
#if !LJ_ARCH_PPC64 && LJ_ARCH_ENDIAN == LUAJIT_LE
#error "No support for little-endian PPC32"
#endif
#if LJ_ARCH_PPC64
```

```
#error "No support for PowerPC 64 bit mode (yet)"
#endif
#ifdef __NO_FPRS__
#error "No support for PPC/e500 anymore (use LuaJIT 2.0)"
#endif
#elif LJ_TARGET_MIPS
#if defined(__mips_soft_float)
#error "No support for MIPS CPUs without FPU"
#endif
#if defined(_LP64)
#error "No support for MIPS64"
#endif
#endif
#endif

/* Enable or disable the dual-number mode for the VM. */
#if (LJ_ARCH_NUMMODE == LJ_NUMMODE_SINGLE && LUAJIT_NUMMODE == 2) || \
    (LJ_ARCH_NUMMODE == LJ_NUMMODE_DUAL && LUAJIT_NUMMODE == 1)
#error "No support for this number mode on this architecture"
#endif
#if LJ_ARCH_NUMMODE == LJ_NUMMODE_DUAL || \
    (LJ_ARCH_NUMMODE == LJ_NUMMODE_DUAL_SINGLE && LUAJIT_NUMMODE != 1) || \
    (LJ_ARCH_NUMMODE == LJ_NUMMODE_SINGLE_DUAL && LUAJIT_NUMMODE == 2)
#define LJ_DUALNUM		1
#else
#define LJ_DUALNUM		0
#endif

#if LJ_TARGET_IOS || LJ_TARGET_CONSOLE
/* Runtime code generation is restricted on iOS. Complain to Apple, not me. */
/* Ditto for the consoles. Complain to Sony or MS, not me. */
#ifndef LUAJIT_ENABLE_JIT
#define LJ_OS_NOJIT		1
#endif
#endif

/* 64 bit GC references. */
#if LJ_TARGET_GC64
#define LJ_GC64			1
#else
#define LJ_GC64			0
#endif

/* 2-slot frame info. */
#if LJ_GC64
#define LJ_FR2			1
#else
#define LJ_FR2			0
#endif

/* Disable or enable the JIT compiler. */
#if defined(LUAJIT_DISABLE_JIT) || defined(LJ_ARCH_NOJIT) || defined(LJ_OS_NOJIT) || LJ_FR2 || LJ_GC64
#define LJ_HASJIT		0
#else
#define LJ_HASJIT		1
#endif

/* Disable or enable the FFI extension. */
#if defined(LUAJIT_DISABLE_FFI) || defined(LJ_ARCH_NOFFI)
#define LJ_HASFFI		0
#else
#define LJ_HASFFI		1
#endif

#if defined(LUAJIT_DISABLE_PROFILE)
#define LJ_HASPROFILE		0
#elif LJ_TARGET_POSIX
#define LJ_HASPROFILE		1
#define LJ_PROFILE_SIGPROF	1
#elif LJ_TARGET_PS3
#define LJ_HASPROFILE		1
#define LJ_PROFILE_PTHREAD	1
#elif LJ_TARGET_WINDOWS || LJ_TARGET_XBOX360
#define LJ_HASPROFILE		1
#define LJ_PROFILE_WTHREAD	1
#else
```

```
444 #define LJ_HASPROFILE                0
445 #endif
446
447 #ifndef LJ_ARCH_HASFPU
448 #define LJ_ARCH_HASFPU                1
449 #endif
450 #ifndef LJ_ABI_SOFTFP
451 #define LJ_ABI_SOFTFP                0
452 #endif
453 #define LJ_SOFTFP                (!LJ_ARCH_HASFPU)
454
455 #if LJ_ARCH_ENDIAN == LUAJIT_BE
456 #define LJ_LE                        0
457 #define LJ_BE                        1
458 #define LJ_ENDIAN_SELECT(le, be)        be
459 #define LJ_ENDIAN_LOHI(lo, hi)                hi lo
460 #else
461 #define LJ_LE                        1
462 #define LJ_BE                        0
463 #define LJ_ENDIAN_SELECT(le, be)        le
464 #define LJ_ENDIAN_LOHI(lo, hi)                lo hi
465 #endif
466
467 #if LJ_ARCH_BITS == 32
468 #define LJ_32                        1
469 #define LJ_64                        0
470 #else
471 #define LJ_32                        0
472 #define LJ_64                        1
473 #endif
474
475 #ifndef LJ_TARGET_UNALIGNED
476 #define LJ_TARGET_UNALIGNED        0
477 #endif
478
479 /* Various workarounds for embedded operating systems or weak C runtimes. */
480 #if (defined(__ANDROID__) && !defined(LJ_TARGET_X86ORX64)) || defined(__symbian__) || LJ_TARGET_XBOX360 ||
LJ_TARGET_WINDOWS
481 #define LUAJIT_NO_LOG2
482 #endif
483 #if defined(__symbian__) || LJ_TARGET_WINDOWS
484 #define LUAJIT_NO_EXP2
485 #endif
486
487 #if defined(LUAJIT_NO_UNWIND) || defined(__symbian__) || LJ_TARGET_IOS || LJ_TARGET_PS3
488 #define LJ_NO_UNWIND                1
489 #endif
490
491 /* Compatibility with Lua 5.1 vs. 5.2. */
492 #ifdef LUAJIT_ENABLE_LUA52COMPAT
493 #define LJ_52                        1
494 #else
495 #define LJ_52                        0
496 #endif
497
498 #endif
```

One Level Up Top Level

src/host/buildvm.h - luajit-2.0-src

Data types defined

- BuildCtx
- BuildCtx
- BuildMode
- BuildReloc
- BuildReloc
- BuildSym
- BuildSym

Macros defined

- BUILDDEF
- BUILDENUM
- BUILDENUM
- BUILD_MAX_FOLD
- BUILD_MAX_RELOC
- FOLDDEF_PREFIX
- LABEL_PREFIX
- LABEL_PREFIX_BC
- LABEL_PREFIX_CF
- LABEL_PREFIX_FF
- LABEL_PREFIX_FFH
- LABEL_PREFIX_LIBCF
- LABEL_PREFIX_LIBINIT
- LIBDEF_PREFIX
- _BUILDVM_H

Source code

```
1  /*
2  ** LuaJIT VM builder.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _BUILDVM_H
7  #define _BUILDVM_H
8
9  #include <sys/types.h>
10  #include <stdio.h>
11  #include <stdlib.h>
```

```c
#include <string.h>
#include <errno.h>

#include "lj_def.h"
#include "lj_arch.h"

/* Hardcoded limits. Increase as needed. */
#define BUILD_MAX_RELOC		200	/* Max. number of relocations. */
#define BUILD_MAX_FOLD		4096	/* Max. number of fold rules. */

/* Prefix for scanned library definitions. */
#define LIBDEF_PREFIX		"LJLIB_"

/* Prefix for scanned fold definitions. */
#define FOLDDEF_PREFIX		"LJFOLD"

/* Prefixes for generated labels. */
#define LABEL_PREFIX		"lj_"
#define LABEL_PREFIX_BC		LABEL_PREFIX "BC_"
#define LABEL_PREFIX_FF		LABEL_PREFIX "ff_"
#define LABEL_PREFIX_CF		LABEL_PREFIX "cf_"
#define LABEL_PREFIX_FFH	LABEL_PREFIX "ffh_"
#define LABEL_PREFIX_LIBCF	LABEL_PREFIX "lib_cf_"
#define LABEL_PREFIX_LIBINIT	LABEL_PREFIX "lib_init_"

/* Forward declaration. */
struct dasm_State;

/* Build modes. */
#define BUILDDEF(_) \
  _(elfasm) _(coffasm) _(machasm) _(peobj) _(raw) \
  _(bcdef) _(ffdef) _(libdef) _(recdef) _(vmdef) \
  _(folddef)

typedef enum {
#define BUILDENUM(name)		BUILD_##name,
BUILDDEF(BUILDENUM)
#undef BUILDENUM
  BUILD__MAX
} BuildMode;

/* Code relocation. */
typedef struct BuildReloc {
  int32_t ofs;
  int sym;
  int type;
} BuildReloc;

typedef struct BuildSym {
  const char *name;
  int32_t ofs;
} BuildSym;

/* Build context structure. */
typedef struct BuildCtx {
  /* DynASM state pointer. Should be first member. */
  struct dasm_State *D;
  /* Parsed command line. */
  BuildMode mode;
  FILE *fp;
  const char *outname;
  char **args;
  /* Code and symbols generated by DynASM. */
  uint8_t *code;
  size_t codesz;
  int npc, nglob, nsym, nreloc, nrelocsym;
  void **glob;
  BuildSym *sym;
  const char **relocsym;
  int32_t *bc_ofs;
  const char *beginsym;
  /* Strings generated by DynASM. */
  const char *const *globnames;
  const char *const *extnames;
  const char *dasm_ident;
  const char *dasm_arch;
```

```
  /* Relocations. */
  BuildReloc reloc[BUILD_MAX_RELOC];
} BuildCtx;

extern void owrite(BuildCtx *ctx, const void *ptr, size_t sz);
extern void emit_asm(BuildCtx *ctx);
extern void emit_peobj(BuildCtx *ctx);
extern void emit_lib(BuildCtx *ctx);
extern void emit_fold(BuildCtx *ctx);

extern const char *const bc_names[];
extern const char *const ir_names[];
 extern const char *const irt_names[];
 extern const char *const irfpm_names[];
 extern const char *const irfield_names[];
 extern const char *const ircall_names[];

 #endif
```

One Level Up Top Level

src/lj_opt_fold.c - luajit-2.0-src

Data types defined

- FoldFunc

Functions defined

- LJFOLDF(kfold_numarith)
- LJFOLDF(kfold_ldexp)
- LJFOLDF(kfold_fpmath)
- LJFOLDF(kfold_numpow)
- LJFOLDF(kfold_numcomp)
- LJFOLDF(kfold_intarith)
- LJFOLDF(kfold_intovarith)
- LJFOLDF(kfold_bnot)
- LJFOLDF(kfold_bswap)
- LJFOLDF(kfold_intcomp)
- LJFOLDF(kfold_intcomp0)
- LJFOLDF(kfold_int64arith)
- LJFOLDF(kfold_int64arith2)
- LJFOLDF(kfold_int64shift)
- LJFOLDF(kfold_bnot64)
- LJFOLDF(kfold_bswap64)
- LJFOLDF(kfold_int64comp)
- LJFOLDF(kfold_int64comp0)
- LJFOLDF(kfold_snew_kptr)
- LJFOLDF(kfold_snew_empty)
- LJFOLDF(kfold_strref)
- LJFOLDF(kfold_strref_snew)
- LJFOLDF(kfold_strcmp)
- LJFOLDF(bufput_append)
- LJFOLDF(bufput_kgc)
- LJFOLDF(bufstr_kfold_cse)
- LJFOLDF(bufput_kfold_op)

- LJFOLDF(bufput_kfold_rep)
- LJFOLDF(bufput_kfold_fmt)
- LJFOLDF(kfold_add_kgc)
- LJFOLDF(kfold_add_kptr)
- LJFOLDF(kfold_add_kright)
- LJFOLDF(kfold_tobit)
- LJFOLDF(kfold_conv_kint_num)
- LJFOLDF(kfold_conv_kintu32_num)
- LJFOLDF(kfold_conv_kint_ext)
- LJFOLDF(kfold_conv_kint_i64)
- LJFOLDF(kfold_conv_kint64_num_i64)
- LJFOLDF(kfold_conv_kint64_num_u64)
- LJFOLDF(kfold_conv_kint64_int_i64)
- LJFOLDF(kfold_conv_knum_int_num)
- LJFOLDF(kfold_conv_knum_u32_num)
- LJFOLDF(kfold_conv_knum_i64_num)
- LJFOLDF(kfold_conv_knum_u64_num)
- LJFOLDF(kfold_tostr_knum)
- LJFOLDF(kfold_tostr_kint)
- LJFOLDF(kfold_strto)
- LJFOLDF(shortcut_round)
- LJFOLDF(shortcut_left)
- LJFOLDF(shortcut_dropleft)
- LJFOLDF(shortcut_leftleft)
- LJFOLDF(simplify_numadd_negx)
- LJFOLDF(simplify_numadd_xneg)
- LJFOLDF(simplify_numsub_k)
- LJFOLDF(simplify_numsub_negk)
- LJFOLDF(simplify_numsub_xneg)
- LJFOLDF(simplify_nummuldiv_k)
- LJFOLDF(simplify_nummuldiv_negk)
- LJFOLDF(simplify_nummuldiv_negneg)
- LJFOLDF(simplify_numpow_xk)

- LJFOLDF(simplify_numpow_kx)
- LJFOLDF(shortcut_conv_num_int)
- LJFOLDF(simplify_conv_int_num)
- LJFOLDF(simplify_conv_i64_num)
- LJFOLDF(simplify_conv_int_i64)
- LJFOLDF(simplify_conv_flt_num)
- LJFOLDF(simplify_tobit_conv)
- LJFOLDF(simplify_floor_conv)
- LJFOLDF(simplify_conv_sext)
- LJFOLDF(simplify_conv_narrow)
- LJFOLDF(cse_conv)
- LJFOLDF(narrow_convert)
- LJFOLDF(simplify_intadd_k)
- LJFOLDF(simplify_intmul_k)
- LJFOLDF(simplify_intsub_k)
- LJFOLDF(simplify_intsub_kleft)
- LJFOLDF(simplify_intadd_k64)
- LJFOLDF(simplify_intsub_k64)
- LJFOLDF(simplify_intmul_k32)
- LJFOLDF(simplify_intmul_k64)
- LJFOLDF(simplify_intmod_k)
- LJFOLDF(simplify_intmod_kleft)
- LJFOLDF(simplify_intsub)
- LJFOLDF(simplify_intsubadd_leftcancel)
- LJFOLDF(simplify_intsubsub_leftcancel)
- LJFOLDF(simplify_intsubsub_rightcancel)
- LJFOLDF(simplify_intsubadd_rightcancel)
- LJFOLDF(simplify_intsubaddadd_cancel)
- LJFOLDF(simplify_band_k)
- LJFOLDF(simplify_bor_k)
- LJFOLDF(simplify_bxor_k)
- LJFOLDF(simplify_shift_ik)
- LJFOLDF(simplify_shift_andk)

- LJFOLDF(simplify_shift1_ki)
- LJFOLDF(simplify_shift2_ki)
- LJFOLDF(simplify_shiftk_andk)
- LJFOLDF(simplify_andk_shiftk)
- LJFOLDF(reassoc_intarith_k)
- LJFOLDF(reassoc_intarith_k64)
- LJFOLDF(reassoc_dup)
- LJFOLDF(reassoc_bxor)
- LJFOLDF(reassoc_shift)
- LJFOLDF(reassoc_minmax_k)
- LJFOLDF(reassoc_minmax_left)
- LJFOLDF(reassoc_minmax_right)
- LJFOLDF(abc_fwd)
- LJFOLDF(abc_k)
- LJFOLDF(abc_invar)
- LJFOLDF(comm_swap)
- LJFOLDF(comm_equal)
- LJFOLDF(comm_comp)
- LJFOLDF(comm_dup)
- LJFOLDF(comm_bxor)
- LJFOLDF(merge_eqne_snew_kgc)
- LJFOLDF(fwd_href_tdup)
- LJFOLDF(fload_tab_tnew_asize)
- LJFOLDF(fload_tab_tnew_hmask)
- LJFOLDF(fload_tab_tdup_asize)
- LJFOLDF(fload_tab_tdup_hmask)
- LJFOLDF(fload_tab_ah)
- LJFOLDF(fload_str_len_kgc)
- LJFOLDF(fload_str_len_snew)
- LJFOLDF(fload_str_len_tostr)
- LJFOLDF(fload_cdata_typeid_kgc)
- LJFOLDF(fload_cdata_int64_kgc)
- LJFOLDF(fload_cdata_typeid_cnew)

- LJFOLDF(fload_cdata_ptr_int64_cnew)
- LJFOLDF(xload_kptr)
- LJFOLDF(barrier_tnew_tdup)
- LJFOLDF(prof)
- LJFOLDX(lj_opt_cse)
- LJFOLDX(lj_opt_fwd_aload)
- LJFOLDX(lj_opt_fwd_hload)
- LJFOLDX(lj_opt_fwd_hrefk)
- LJFOLDX(lj_opt_cse)
- LJFOLDX(lj_opt_fwd_xload)
- LJFOLDX(lj_opt_dse_ahstore)
- kfold_int64arith
- kfold_intop
- kfold_xload
- lj_opt_cse
- lj_opt_cselim
- simplify_intmul_k

Macros defined

- FOLD_SNEW_MAX_LEN
- FOLD_SNEW_TYPE8
- IR
- IR
- LJFOLD
- LJFOLDF
- LJFOLDX
- LUA_CORE
- PHIBARRIER
- emitir
- emitir
- fins
- fins
- fleft
- fleft

- fright
- fright
- gcstep_barrier
- knumleft
- knumleft
- knumright
- knumright
- lj_opt_fold_c

Source code

```
/*
** FOLD: Constant Folding, Algebraic Simplifications and Reassociation.
** ABCelim: Array Bounds Check Elimination.
** CSE: Common-Subexpression Elimination.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_opt_fold_c
#define LUA_CORE

 #include <math.h>

 #include "lj_obj.h"

 #if LJ_HASJIT

 #include "lj_buf.h"
 #include "lj_str.h"
 #include "lj_tab.h"
 #include "lj_ir.h"
 #include "lj_jit.h"
 #include "lj_ircall.h"
 #include "lj_iropt.h"
 #include "lj_trace.h"
 #if LJ_HASFFI
 #include "lj_ctype.h"
 #include "lj_carith.h"
 #endif
 #include "lj_vm.h"
 #include "lj_strscan.h"
 #include "lj_strfmt.h"

 /* Here's a short description how the FOLD engine processes instructions:
 **
 ** The FOLD engine receives a single instruction stored in fins (J->fold.ins).
 ** The instruction and its operands are used to select matching fold rules.
 ** These are applied iteratively until a fixed point is reached.
 **
 ** The 8 bit opcode of the instruction itself plus the opcodes of the
 ** two instructions referenced by its operands form a 24 bit key
 ** 'ins left right' (unused operands -> 0, literals -> lowest 8 bits).
 **
 ** This key is used for partial matching against the fold rules. The
 ** left/right operand fields of the key are successively masked with
 ** the 'any' wildcard, from most specific to least specific:
 **
 **   ins left right
 **   ins any  right
 **   ins left any
 **   ins any  any
 **
 ** The masked key is used to lookup a matching fold rule in a semi-perfect
 ** hash table. If a matching rule is found, the related fold function is run.
 ** Multiple rules can share the same fold function. A fold rule may return
```

```
** one of several special values:
**
** - NEXTFOLD means no folding was applied, because an additional test
**   inside the fold function failed. Matching continues against less
**   specific fold rules. Finally the instruction is passed on to CSE.
**
** - RETRYFOLD means the instruction was modified in-place. Folding is
**   retried as if this instruction had just been received.
**
** All other return values are terminal actions -- no further folding is
** applied:
**
** - INTFOLD(i) returns a reference to the integer constant i.
**
** - LEFTFOLD and RIGHTFOLD return the left/right operand reference
**   without emitting an instruction.
**
** - CSEFOLD and EMITFOLD pass the instruction directly to CSE or emit
**   it without passing through any further optimizations.
**
** - FAILFOLD, DROPFOLD and CONDFOLD only apply to instructions which have
**   no result (e.g. guarded assertions): FAILFOLD means the guard would
**   always fail, i.e. the current trace is pointless. DROPFOLD means
**   the guard is always true and has been eliminated. CONDFOLD is a
**   shortcut for FAILFOLD + cond (i.e. drop if true, otherwise fail).
**
** - Any other return value is interpreted as an IRRef or TRef. This
**   can be a reference to an existing or a newly created instruction.
**   Only the least-significant 16 bits (IRRef1) are used to form a TRef
**   which is finally returned to the caller.
**
** The FOLD engine receives instructions both from the trace recorder and
** substituted instructions from LOOP unrolling. This means all types
** of instructions may end up here, even though the recorder bypasses
** FOLD in some cases. Thus all loads, stores and allocations must have
** an any/any rule to avoid being passed on to CSE.
**
** Carefully read the following requirements before adding or modifying
** any fold rules:
**
** Requirement #1: All fold rules must preserve their destination type.
**
** Consistently use INTFOLD() (KINT result) or lj_ir_knum() (KNUM result).
** Never use lj_ir_knumint() which can have either a KINT or KNUM result.
**
** Requirement #2: Fold rules should not create *new* instructions which
** reference operands *across* PHIs.
**
** E.g. a RETRYFOLD with 'fins->op1 = fleft->op1' is invalid if the
** left operand is a PHI. Then fleft->op1 would point across the PHI
** frontier to an invariant instruction. Adding a PHI for this instruction
** would be counterproductive. The solution is to add a barrier which
** prevents folding across PHIs, i.e. 'PHIBARRIER(fleft)' in this case.
** The only exception is for recurrences with high latencies like
** repeated int->num->int conversions.
**
** One could relax this condition a bit if the referenced instruction is
** a PHI, too. But this often leads to worse code due to excessive
** register shuffling.
**
** Note: returning *existing* instructions (e.g. LEFTFOLD) is ok, though.
** Even returning fleft->op1 would be ok, because a new PHI will added,
** if needed. But again, this leads to excessive register shuffling and
** should be avoided.
**
** Requirement #3: The set of all fold rules must be monotonic to guarantee
** termination.
**
** The goal is optimization, so one primarily wants to add strength-reducing
** rules. This means eliminating an instruction or replacing an instruction
** with one or more simpler instructions. Don't add fold rules which point
** into the other direction.
**
** Some rules (like commutativity) do not directly reduce the strength of
** an instruction, but enable other fold rules (e.g. by moving constants
** to the right operand). These rules must be made unidirectional to avoid
```

```
** cycles.
**
** Rule of thumb: the trace recorder expands the IR and FOLD shrinks it.
*/

/* Some local macros to save typing. Undef'd at the end. */
#define IR(ref)                 (&J->cur.ir[(ref)])
#define fins                (&J->fold.ins)
#define fleft                (&J->fold.left)
#define fright                (&J->fold.right)
#define knumleft        (ir_knum(fleft)->n)
#define knumright        (ir_knum(fright)->n)

/* Pass IR on to next optimization in chain (FOLD). */
#define emitir(ot, a, b)        (lj_ir_set(J, (ot), (a), (b)), lj_opt_fold(J))

/* Fold function type. Fastcall on x86 significantly reduces their size. */
typedef IRRef (LJ_FASTCALL *FoldFunc)(jit_State *J);

/* Macros for the fold specs, so buildvm can recognize them. */
#define LJFOLD(x)
#define LJFOLDX(x)
#define LJFOLDF(name)        static TRef LJ_FASTCALL fold_##name(jit_State *J)
/* Note: They must be at the start of a line or buildvm ignores them! */

/* Barrier to prevent using operands across PHIs. */
#define PHIBARRIER(ir)        if (irt_isphi((ir)->t)) return NEXTFOLD

/* Barrier to prevent folding across a GC step.
** GC steps can only happen at the head of a trace and at LOOP.
** And the GC is only driven forward if there's at least one allocation.
*/
#define gcstep_barrier(J, ref) \
  ((ref) < J->chain[IR_LOOP] && \
   (J->chain[IR_SNEW] || J->chain[IR_XSNEW] || \
    J->chain[IR_TNEW] || J->chain[IR_TDUP] || \
    J->chain[IR_CNEW] || J->chain[IR_CNEWI] || \
    J->chain[IR_BUFSTR] || J->chain[IR_TOSTR] || J->chain[IR_CALLA]))

/* -- Constant folding for FP numbers ------------------------------------ */

LJFOLD(ADD KNUM KNUM)
LJFOLD(SUB KNUM KNUM)
LJFOLD(MUL KNUM KNUM)
LJFOLD(DIV KNUM KNUM)
LJFOLD(NEG KNUM KNUM)
LJFOLD(ABS KNUM KNUM)
LJFOLD(ATAN2 KNUM KNUM)
LJFOLD(LDEXP KNUM KNUM)
LJFOLD(MIN KNUM KNUM)
LJFOLD(MAX KNUM KNUM)
LJFOLDF(kfold_numarith)
{
  lua_Number a = knumleft;
  lua_Number b = knumright;
  lua_Number y = lj_vm_foldarith(a, b, fins->o - IR_ADD);
  return lj_ir_knum(J, y);
}

LJFOLD(LDEXP KNUM KINT)
LJFOLDF(kfold_ldexp)
{
#if LJ_TARGET_X86ORX64
  UNUSED(J);
  return NEXTFOLD;
#else
  return lj_ir_knum(J, ldexp(knumleft, fright->i));
#endif
}

LJFOLD(FPMATH KNUM any)
LJFOLDF(kfold_fpmath)
{
  lua_Number a = knumleft;
  lua_Number y = lj_vm_foldfpm(a, fins->op2);
  return lj_ir_knum(J, y);
```

```
}

LJFOLD(POW KNUM KINT)
LJFOLDF(kfold_numpow)
{
  lua_Number a = knumleft;
  lua_Number b = (lua_Number)fright->i;
  lua_Number y = lj_vm_foldarith(a, b, IR_POW - IR_ADD);
  return lj_ir_knum(J, y);
}

/* Must not use kfold_kref for numbers (could be NaN). */
LJFOLD(EQ KNUM KNUM)
LJFOLD(NE KNUM KNUM)
LJFOLD(LT KNUM KNUM)
LJFOLD(GE KNUM KNUM)
LJFOLD(LE KNUM KNUM)
LJFOLD(GT KNUM KNUM)
LJFOLD(ULT KNUM KNUM)
LJFOLD(UGE KNUM KNUM)
LJFOLD(ULE KNUM KNUM)
LJFOLD(UGT KNUM KNUM)
LJFOLDF(kfold_numcomp)
{
  return CONDFOLD(lj_ir_numcmp(knumleft, knumright, (IROp)fins->o));
}

/* -- Constant folding for 32 bit integers ------------------------------ */

static int32_t kfold_intop(int32_t k1, int32_t k2, IROp op)
{
  switch (op) {
  case IR_ADD: k1 += k2; break;
  case IR_SUB: k1 -= k2; break;
  case IR_MUL: k1 *= k2; break;
  case IR_MOD: k1 = lj_vm_modi(k1, k2); break;
  case IR_NEG: k1 = -k1; break;
  case IR_BAND: k1 &= k2; break;
  case IR_BOR: k1 |= k2; break;
  case IR_BXOR: k1 ^= k2; break;
  case IR_BSHL: k1 <<= (k2 & 31); break;
  case IR_BSHR: k1 = (int32_t)((uint32_t)k1 >> (k2 & 31)); break;
  case IR_BSAR: k1 >>= (k2 & 31); break;
  case IR_BROL: k1 = (int32_t)lj_rol((uint32_t)k1, (k2 & 31)); break;
  case IR_BROR: k1 = (int32_t)lj_ror((uint32_t)k1, (k2 & 31)); break;
  case IR_MIN: k1 = k1 < k2 ? k1 : k2; break;
  case IR_MAX: k1 = k1 > k2 ? k1 : k2; break;
  default: lua_assert(0); break;
  }
  return k1;
}

LJFOLD(ADD KINT KINT)
LJFOLD(SUB KINT KINT)
LJFOLD(MUL KINT KINT)
LJFOLD(MOD KINT KINT)
LJFOLD(NEG KINT KINT)
LJFOLD(BAND KINT KINT)
LJFOLD(BOR KINT KINT)
LJFOLD(BXOR KINT KINT)
LJFOLD(BSHL KINT KINT)
LJFOLD(BSHR KINT KINT)
LJFOLD(BSAR KINT KINT)
LJFOLD(BROL KINT KINT)
LJFOLD(BROR KINT KINT)
LJFOLD(MIN KINT KINT)
LJFOLD(MAX KINT KINT)
LJFOLDF(kfold_intarith)
{
  return INTFOLD(kfold_intop(fleft->i, fright->i, (IROp)fins->o));
}

LJFOLD(ADDOV KINT KINT)
LJFOLD(SUBOV KINT KINT)
LJFOLD(MULOV KINT KINT)
LJFOLDF(kfold_intovarith)
```

```
{
  lua_Number n = lj_vm_foldarith((lua_Number)fleft->i, (lua_Number)fright->i,
                                 fins->o - IR_ADDOV);
  int32_t k = lj_num2int(n);
  if (n != (lua_Number)k)
    return FAILFOLD;
  return INTFOLD(k);
}

LJFOLD(BNOT KINT)
LJFOLDF(kfold_bnot)
{
  return INTFOLD(~fleft->i);
}

LJFOLD(BSWAP KINT)
LJFOLDF(kfold_bswap)
{
  return INTFOLD((int32_t)lj_bswap((uint32_t)fleft->i));
}

LJFOLD(LT KINT KINT)
LJFOLD(GE KINT KINT)
LJFOLD(LE KINT KINT)
LJFOLD(GT KINT KINT)
LJFOLD(ULT KINT KINT)
LJFOLD(UGE KINT KINT)
LJFOLD(ULE KINT KINT)
LJFOLD(UGT KINT KINT)
LJFOLD(ABC KINT KINT)
LJFOLDF(kfold_intcomp)
{
  int32_t a = fleft->i, b = fright->i;
  switch ((IROp)fins->o) {
  case IR_LT: return CONDFOLD(a < b);
  case IR_GE: return CONDFOLD(a >= b);
  case IR_LE: return CONDFOLD(a <= b);
  case IR_GT: return CONDFOLD(a > b);
  case IR_ULT: return CONDFOLD((uint32_t)a < (uint32_t)b);
  case IR_UGE: return CONDFOLD((uint32_t)a >= (uint32_t)b);
  case IR_ULE: return CONDFOLD((uint32_t)a <= (uint32_t)b);
  case IR_ABC:
  case IR_UGT: return CONDFOLD((uint32_t)a > (uint32_t)b);
  default: lua_assert(0); return FAILFOLD;
  }
}

LJFOLD(UGE any KINT)
LJFOLDF(kfold_intcomp0)
{
  if (fright->i == 0)
    return DROPFOLD;
  return NEXTFOLD;
}

/* -- Constant folding for 64 bit integers -------------------------------- */

static uint64_t kfold_int64arith(uint64_t k1, uint64_t k2, IROp op)
{
  switch (op) {
#if LJ_HASFFI
  case IR_ADD: k1 += k2; break;
  case IR_SUB: k1 -= k2; break;
  case IR_MUL: k1 *= k2; break;
  case IR_BAND: k1 &= k2; break;
  case IR_BOR: k1 |= k2; break;
  case IR_BXOR: k1 ^= k2; break;
#endif
  default: UNUSED(k2); lua_assert(0); break;
  }
  return k1;
}

LJFOLD(ADD KINT64 KINT64)
LJFOLD(SUB KINT64 KINT64)
LJFOLD(MUL KINT64 KINT64)
```

```
LJFOLD(BAND KINT64 KINT64)
LJFOLD(BOR KINT64 KINT64)
LJFOLD(BXOR KINT64 KINT64)
LJFOLDF(kfold_int64arith)
{
  return INT64FOLD(kfold_int64arith(ir_k64(fleft)->u64,
                                    ir_k64(fright)->u64, (IROp)fins->o));
}

LJFOLD(DIV KINT64 KINT64)
LJFOLD(MOD KINT64 KINT64)
LJFOLD(POW KINT64 KINT64)
LJFOLDF(kfold_int64arith2)
{
#if LJ_HASFFI
  uint64_t k1 = ir_k64(fleft)->u64, k2 = ir_k64(fright)->u64;
  if (irt_isi64(fins->t)) {
    k1 = fins->o == IR_DIV ? lj_carith_divi64((int64_t)k1, (int64_t)k2) :
         fins->o == IR_MOD ? lj_carith_modi64((int64_t)k1, (int64_t)k2) :
                             lj_carith_powi64((int64_t)k1, (int64_t)k2);
  } else {
    k1 = fins->o == IR_DIV ? lj_carith_divu64(k1, k2) :
         fins->o == IR_MOD ? lj_carith_modu64(k1, k2) :
                             lj_carith_powu64(k1, k2);
  }
  return INT64FOLD(k1);
#else
  UNUSED(J); lua_assert(0); return FAILFOLD;
#endif
}

LJFOLD(BSHL KINT64 KINT)
LJFOLD(BSHR KINT64 KINT)
LJFOLD(BSAR KINT64 KINT)
LJFOLD(BROL KINT64 KINT)
LJFOLD(BROR KINT64 KINT)
LJFOLDF(kfold_int64shift)
{
#if LJ_HASFFI
  uint64_t k = ir_k64(fleft)->u64;
  int32_t sh = (fright->i & 63);
  return INT64FOLD(lj_carith_shift64(k, sh, fins->o - IR_BSHL));
#else
  UNUSED(J); lua_assert(0); return FAILFOLD;
#endif
}

LJFOLD(BNOT KINT64)
LJFOLDF(kfold_bnot64)
{
#if LJ_HASFFI
  return INT64FOLD(~ir_k64(fleft)->u64);
#else
  UNUSED(J); lua_assert(0); return FAILFOLD;
#endif
}

LJFOLD(BSWAP KINT64)
LJFOLDF(kfold_bswap64)
{
#if LJ_HASFFI
  return INT64FOLD(lj_bswap64(ir_k64(fleft)->u64));
#else
  UNUSED(J); lua_assert(0); return FAILFOLD;
#endif
}

LJFOLD(LT KINT64 KINT64)
LJFOLD(GE KINT64 KINT64)
LJFOLD(LE KINT64 KINT64)
LJFOLD(GT KINT64 KINT64)
LJFOLD(ULT KINT64 KINT64)
LJFOLD(UGE KINT64 KINT64)
LJFOLD(ULE KINT64 KINT64)
LJFOLD(UGT KINT64 KINT64)
LJFOLDF(kfold_int64comp)
```

```
{
#if LJ_HASFFI
  uint64_t a = ir_k64(fleft)->u64, b = ir_k64(fright)->u64;
  switch ((IROp)fins->o) {
  case IR_LT: return CONDFOLD(a < b);
  case IR_GE: return CONDFOLD(a >= b);
  case IR_LE: return CONDFOLD(a <= b);
  case IR_GT: return CONDFOLD(a > b);
  case IR_ULT: return CONDFOLD((uint64_t)a < (uint64_t)b);
  case IR_UGE: return CONDFOLD((uint64_t)a >= (uint64_t)b);
  case IR_ULE: return CONDFOLD((uint64_t)a <= (uint64_t)b);
  case IR_UGT: return CONDFOLD((uint64_t)a > (uint64_t)b);
  default: lua_assert(0); return FAILFOLD;
  }
#else
  UNUSED(J); lua_assert(0); return FAILFOLD;
#endif
}

LJFOLD(UGE any KINT64)
LJFOLDF(kfold_int64comp0)
{
#if LJ_HASFFI
  if (ir_k64(fright)->u64 == 0)
    return DROPFOLD;
  return NEXTFOLD;
#else
  UNUSED(J); lua_assert(0); return FAILFOLD;
#endif
}

/* -- Constant folding for strings ---------------------------------------- */

LJFOLD(SNEW KKPTR KINT)
LJFOLDF(kfold_snew_kptr)
{
  GCstr *s = lj_str_new(J->L, (const char *)ir_kptr(fleft), (size_t)fright->i);
  return lj_ir_kstr(J, s);
}

LJFOLD(SNEW any KINT)
LJFOLDF(kfold_snew_empty)
{
  if (fright->i == 0)
    return lj_ir_kstr(J, &J2G(J)->strempty);
  return NEXTFOLD;
}

LJFOLD(STRREF KGC KINT)
LJFOLDF(kfold_strref)
{
  GCstr *str = ir_kstr(fleft);
  lua_assert((MSize)fright->i <= str->len);
  return lj_ir_kkptr(J, (char *)strdata(str) + fright->i);
}

LJFOLD(STRREF SNEW any)
LJFOLDF(kfold_strref_snew)
{
  PHIBARRIER(fleft);
  if (irref_isk(fins->op2) && fright->i == 0) {
    return fleft->op1;  /* strref(snew(ptr, len), 0) ==> ptr */
  } else {
    /* Reassociate: strref(snew(strref(str, a), len), b) ==> strref(str, a+b) */
    IRIns *ir = IR(fleft->op1);
    if (ir->o == IR_STRREF) {
      IRRef1 str = ir->op1;  /* IRIns * is not valid across emitir. */
      PHIBARRIER(ir);
      fins->op2 = emitir(IRTI(IR_ADD), ir->op2, fins->op2); /* Clobbers fins! */
      fins->op1 = str;
      fins->ot = IRT(IR_STRREF, IRT_P32);
      return RETRYFOLD;
    }
  }
  return NEXTFOLD;
}
```

```
511
512 LJFOLD(CALLN CARG IRCALL_lj_str_cmp)
513 LJFOLDF(kfold_strcmp)
514 {
515   if (irref_isk(fleft->op1) && irref_isk(fleft->op2)) {
516     GCstr *a = ir_kstr(IR(fleft->op1));
517     GCstr *b = ir_kstr(IR(fleft->op2));
518     return INTFOLD(lj_str_cmp(a, b));
519   }
520   return NEXTFOLD;
521 }
522
523 /* -- Constant folding and forwarding for buffers ----------------------- */
524
525 /*
526 ** Buffer ops perform stores, but their effect is limited to the buffer
527 ** itself. Also, buffer ops are chained: a use of an op implies a use of
528 ** all other ops up the chain. Conversely, if an op is unused, all ops
529 ** up the chain can go unsed. This largely eliminates the need to treat
530 ** them as stores.
531 **
532 ** Alas, treating them as normal (IRM_N) ops doesn't work, because they
533 ** cannot be CSEd in isolation. CSE for IRM_N is implicitly done in LOOP
534 ** or if FOLD is disabled.
535 **
536 ** The compromise is to declare them as loads, emit them like stores and
537 ** CSE whole chains manually when the BUFSTR is to be emitted. Any chain
538 ** fragments left over from CSE are eliminated by DCE.
539 */
540
541 /* BUFHDR is emitted like a store, see below. */
542
543 LJFOLD(BUFPUT BUFHDR BUFSTR)
544 LJFOLDF(bufput_append)
545 {
546   /* New buffer, no other buffer op inbetween and same buffer? */
547   if ((J->flags & JIT_F_OPT_FWD) &&
548       !(fleft->op2 & IRBUFHDR_APPEND) &&
549       fleft->prev == fright->op2 &&
550       fleft->op1 == IR(fright->op2)->op1) {
551     IRRef ref = fins->op1;
552     IR(ref)->op2 = (fleft->op2 | IRBUFHDR_APPEND);  /* Modify BUFHDR. */
553     IR(ref)->op1 = fright->op1;
554     return ref;
555   }
556   return EMITFOLD;  /* Always emit, CSE later. */
557 }
558
559 LJFOLD(BUFPUT any any)
560 LJFOLDF(bufput_kgc)
561 {
562   if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD) && fright->o == IR_KGC) {
563     GCstr *s2 = ir_kstr(fright);
564     if (s2->len == 0) {  /* Empty string? */
565       return LEFTFOLD;
566     } else {
567       if (fleft->o == IR_BUFPUT && irref_isk(fleft->op2) &&
568           !irt_isphi(fleft->t)) {  /* Join two constant string puts in a row. */
569         GCstr *s1 = ir_kstr(IR(fleft->op2));
570         IRRef kref = lj_ir_kstr(J, lj_buf_cat2str(J->L, s1, s2));
571         /* lj_ir_kstr() may realloc the IR and invalidates any IRIns *. */
572         IR(fins->op1)->op2 = kref;  /* Modify previous BUFPUT. */
573         return fins->op1;
574       }
575     }
576   }
577   return EMITFOLD;  /* Always emit, CSE later. */
578 }
579
580 LJFOLD(BUFSTR any any)
581 LJFOLDF(bufstr_kfold_cse)
582 {
583   lua_assert(fleft->o == IR_BUFHDR || fleft->o == IR_BUFPUT ||
584              fleft->o == IR_CALLL);
585   if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD)) {
586     if (fleft->o == IR_BUFHDR) {  /* No put operations? */
```

```
      if (!(fleft->op2 & IRBUFHDR_APPEND))  /* Empty buffer? */
        return lj_ir_kstr(J, &J2G(J)->strempty);
      fins->op1 = fleft->op1;
      fins->op2 = fleft->prev;  /* Relies on checks in bufput_append. */
      return CSEFOLD;
    } else if (fleft->o == IR_BUFPUT) {
      IRIns *irb = IR(fleft->op1);
      if (irb->o == IR_BUFHDR && !(irb->op2 & IRBUFHDR_APPEND))
        return fleft->op2;  /* Shortcut for a single put operation. */
    }
  }
  /* Try to CSE the whole chain. */
  if (LJ_LIKELY(J->flags & JIT_F_OPT_CSE)) {
    IRRef ref = J->chain[IR_BUFSTR];
    while (ref) {
      IRIns *irs = IR(ref), *ira = fleft, *irb = IR(irs->op1);
      while (ira->o == irb->o && ira->op2 == irb->op2) {
        lua_assert(ira->o == IR_BUFHDR || ira->o == IR_BUFPUT ||
                   ira->o == IR_CALLL || ira->o == IR_CARG);
        if (ira->o == IR_BUFHDR && !(ira->op2 & IRBUFHDR_APPEND))
          return ref;  /* CSE succeeded. */
        if (ira->o == IR_CALLL && ira->op2 == IRCALL_lj_buf_puttab)
          break;
        ira = IR(ira->op1);
        irb = IR(irb->op1);
      }
      ref = irs->prev;
    }
  }
  return EMITFOLD;  /* No CSE possible. */
}

LJFOLD(CALLL CARG IRCALL_lj_buf_putstr_reverse)
LJFOLD(CALLL CARG IRCALL_lj_buf_putstr_upper)
LJFOLD(CALLL CARG IRCALL_lj_buf_putstr_lower)
LJFOLD(CALLL CARG IRCALL_lj_strfmt_putquoted)
LJFOLDF(bufput_kfold_op)
{
  if (irref_isk(fleft->op2)) {
    const CCallInfo *ci = &lj_ir_callinfo[fins->op2];
    SBuf *sb = lj_buf_tmp_(J->L);
    sb = ((SBuf * (LJ_FASTCALL *)(SBuf *, GCstr *))ci->func)(sb,
                                                    ir_kstr(IR(fleft->op2)));
    fins->o = IR_BUFPUT;
    fins->op1 = fleft->op1;
    fins->op2 = lj_ir_kstr(J, lj_buf_tostr(sb));
    return RETRYFOLD;
  }
  return EMITFOLD;  /* Always emit, CSE later. */
}

LJFOLD(CALLL CARG IRCALL_lj_buf_putstr_rep)
LJFOLDF(bufput_kfold_rep)
{
  if (irref_isk(fleft->op2)) {
    IRIns *irc = IR(fleft->op1);
    if (irref_isk(irc->op2)) {
      SBuf *sb = lj_buf_tmp_(J->L);
      sb = lj_buf_putstr_rep(sb, ir_kstr(IR(irc->op2)), IR(fleft->op2)->i);
      fins->o = IR_BUFPUT;
      fins->op1 = irc->op1;
      fins->op2 = lj_ir_kstr(J, lj_buf_tostr(sb));
      return RETRYFOLD;
    }
  }
  return EMITFOLD;  /* Always emit, CSE later. */
}

LJFOLD(CALLL CARG IRCALL_lj_strfmt_putfxint)
LJFOLD(CALLL CARG IRCALL_lj_strfmt_putfnum_int)
LJFOLD(CALLL CARG IRCALL_lj_strfmt_putfnum_uint)
LJFOLD(CALLL CARG IRCALL_lj_strfmt_putfnum)
LJFOLD(CALLL CARG IRCALL_lj_strfmt_putfstr)
LJFOLD(CALLL CARG IRCALL_lj_strfmt_putfchar)
LJFOLDF(bufput_kfold_fmt)
{
```

```
  IRIns *irc = IR(fleft->op1);
  lua_assert(irref_isk(irc->op2));  /* SFormat must be const. */
  if (irref_isk(fleft->op2)) {
    SFormat sf = (SFormat)IR(irc->op2)->i;
    IRIns *ira = IR(fleft->op2);
    SBuf *sb = lj_buf_tmp_(J->L);
    switch (fins->op2) {
    case IRCALL_lj_strfmt_putfxint:
      sb = lj_strfmt_putfxint(sb, sf, ir_k64(ira)->u64);
      break;
    case IRCALL_lj_strfmt_putfstr:
      sb = lj_strfmt_putfstr(sb, sf, ir_kstr(ira));
      break;
    case IRCALL_lj_strfmt_putfchar:
      sb = lj_strfmt_putfchar(sb, sf, ira->i);
      break;
    case IRCALL_lj_strfmt_putfnum_int:
    case IRCALL_lj_strfmt_putfnum_uint:
    case IRCALL_lj_strfmt_putfnum:
    default: {
      const CCallInfo *ci = &lj_ir_callinfo[fins->op2];
      sb = ((SBuf * (*)(SBuf *, SFormat, lua_Number))ci->func)(sb, sf,
                                                               ir_knum(ira)->n);
      break;
      }
    }
    fins->o = IR_BUFPUT;
    fins->op1 = irc->op1;
    fins->op2 = lj_ir_kstr(J, lj_buf_tostr(sb));
    return RETRYFOLD;
  }
  return EMITFOLD;  /* Always emit, CSE later. */
}

/* -- Constant folding of pointer arithmetic ------------------------------ */

LJFOLD(ADD KGC KINT)
LJFOLD(ADD KGC KINT64)
LJFOLDF(kfold_add_kgc)
{
  GCobj *o = ir_kgc(fleft);
#if LJ_64
  ptrdiff_t ofs = (ptrdiff_t)ir_kint64(fright)->u64;
#else
  ptrdiff_t ofs = fright->i;
#endif
#if LJ_HASFFI
  if (irt_iscdata(fleft->t)) {
    CType *ct = ctype_raw(ctype_ctsG(J2G(J)), gco2cd(o)->ctypeid);
    if (ctype_isnum(ct->info) || ctype_isenum(ct->info) ||
        ctype_isptr(ct->info) || ctype_isfunc(ct->info) ||
        ctype_iscomplex(ct->info) || ctype_isvector(ct->info))
      return lj_ir_kkptr(J, (char *)o + ofs);
  }
#endif
  return lj_ir_kptr(J, (char *)o + ofs);
}

LJFOLD(ADD KPTR KINT)
LJFOLD(ADD KPTR KINT64)
LJFOLD(ADD KKPTR KINT)
LJFOLD(ADD KKPTR KINT64)
LJFOLDF(kfold_add_kptr)
{
  void *p = ir_kptr(fleft);
#if LJ_64
  ptrdiff_t ofs = (ptrdiff_t)ir_kint64(fright)->u64;
#else
  ptrdiff_t ofs = fright->i;
#endif
  return lj_ir_kptr_(J, fleft->o, (char *)p + ofs);
}

LJFOLD(ADD any KGC)
LJFOLD(ADD any KPTR)
LJFOLD(ADD any KKPTR)
```

```
LJFOLDF(kfold_add_kright)
{
  if (fleft->o == IR_KINT || fleft->o == IR_KINT64) {
    IRRef1 tmp = fins->op1; fins->op1 = fins->op2; fins->op2 = tmp;
    return RETRYFOLD;
  }
  return NEXTFOLD;
}

/* -- Constant folding of conversions ------------------------------------ */

LJFOLD(TOBIT KNUM KNUM)
LJFOLDF(kfold_tobit)
{
  return INTFOLD(lj_num2bit(knumleft));
}

LJFOLD(CONV KINT IRCONV_NUM_INT)
LJFOLDF(kfold_conv_kint_num)
{
  return lj_ir_knum(J, (lua_Number)fleft->i);
}

LJFOLD(CONV KINT IRCONV_NUM_U32)
LJFOLDF(kfold_conv_kintu32_num)
{
  return lj_ir_knum(J, (lua_Number)(uint32_t)fleft->i);
}

LJFOLD(CONV KINT IRCONV_INT_I8)
LJFOLD(CONV KINT IRCONV_INT_U8)
LJFOLD(CONV KINT IRCONV_INT_I16)
LJFOLD(CONV KINT IRCONV_INT_U16)
LJFOLDF(kfold_conv_kint_ext)
{
  int32_t k = fleft->i;
  if ((fins->op2 & IRCONV_SRCMASK) == IRT_I8) k = (int8_t)k;
  else if ((fins->op2 & IRCONV_SRCMASK) == IRT_U8) k = (uint8_t)k;
  else if ((fins->op2 & IRCONV_SRCMASK) == IRT_I16) k = (int16_t)k;
  else k = (uint16_t)k;
  return INTFOLD(k);
}

LJFOLD(CONV KINT IRCONV_I64_INT)
LJFOLD(CONV KINT IRCONV_U64_INT)
LJFOLD(CONV KINT IRCONV_I64_U32)
LJFOLD(CONV KINT IRCONV_U64_U32)
LJFOLDF(kfold_conv_kint_i64)
{
  if ((fins->op2 & IRCONV_SEXT))
    return INT64FOLD((uint64_t)(int64_t)fleft->i);
  else
    return INT64FOLD((uint64_t)(int64_t)(uint32_t)fleft->i);
}

LJFOLD(CONV KINT64 IRCONV_NUM_I64)
LJFOLDF(kfold_conv_kint64_num_i64)
{
  return lj_ir_knum(J, (lua_Number)(int64_t)ir_kint64(fleft)->u64);
}

LJFOLD(CONV KINT64 IRCONV_NUM_U64)
LJFOLDF(kfold_conv_kint64_num_u64)
{
  return lj_ir_knum(J, (lua_Number)ir_kint64(fleft)->u64);
}

LJFOLD(CONV KINT64 IRCONV_INT_I64)
LJFOLD(CONV KINT64 IRCONV_U32_I64)
LJFOLDF(kfold_conv_kint64_int_i64)
{
  return INTFOLD((int32_t)ir_kint64(fleft)->u64);
}

LJFOLD(CONV KNUM IRCONV_INT_NUM)
LJFOLDF(kfold_conv_knum_int_num)
```

```
{
  lua_Number n = knumleft;
  int32_t k = lj_num2int(n);
  if (irt_isguard(fins->t) && n != (lua_Number)k) {
    /* We're about to create a guard which always fails, like CONV +1.5.
    ** Some pathological loops cause this during LICM, e.g.:
    **   local x,k,t = 0,1.5,{1,[1.5]=2}
    **   for i=1,200 do x = x+ t[k]; k = k == 1 and 1.5 or 1 end
    **   assert(x == 300)
    */
    return FAILFOLD;
  }
  return INTFOLD(k);
}

LJFOLD(CONV KNUM IRCONV_U32_NUM)
LJFOLDF(kfold_conv_knum_u32_num)
{
#ifdef _MSC_VER
  {  /* Workaround for MSVC bug. */
    volatile uint32_t u = (uint32_t)knumleft;
    return INTFOLD((int32_t)u);
  }
#else
  return INTFOLD((int32_t)(uint32_t)knumleft);
#endif
}

LJFOLD(CONV KNUM IRCONV_I64_NUM)
LJFOLDF(kfold_conv_knum_i64_num)
{
  return INT64FOLD((uint64_t)(int64_t)knumleft);
}

LJFOLD(CONV KNUM IRCONV_U64_NUM)
LJFOLDF(kfold_conv_knum_u64_num)
{
  return INT64FOLD(lj_num2u64(knumleft));
}

LJFOLD(TOSTR KNUM any)
LJFOLDF(kfold_tostr_knum)
{
  return lj_ir_kstr(J, lj_strfmt_num(J->L, ir_knum(fleft)));
}

LJFOLD(TOSTR KINT any)
LJFOLDF(kfold_tostr_kint)
{
  return lj_ir_kstr(J, fins->op2 == IRTOSTR_INT ?
                       lj_strfmt_int(J->L, fleft->i) :
                       lj_strfmt_char(J->L, fleft->i));
}

LJFOLD(STRTO KGC)
LJFOLDF(kfold_strto)
{
  TValue n;
  if (lj_strscan_num(ir_kstr(fleft), &n))
    return lj_ir_knum(J, numV(&n));
  return FAILFOLD;
}

/* -- Constant folding of equality checks --------------------------------- */

/* Don't constant-fold away FLOAD checks against KNULL. */
LJFOLD(EQ FLOAD KNULL)
LJFOLD(NE FLOAD KNULL)
LJFOLDX(lj_opt_cse)

/* But fold all other KNULL compares, since only KNULL is equal to KNULL. */
LJFOLD(EQ any KNULL)
LJFOLD(NE any KNULL)
LJFOLD(EQ KNULL any)
LJFOLD(NE KNULL any)
LJFOLD(EQ KINT KINT)  /* Constants are unique, so same refs <==> same value. */
```

```
LJFOLD(NE KINT KINT)
LJFOLD(EQ KINT64 KINT64)
LJFOLD(NE KINT64 KINT64)
LJFOLD(EQ KGC KGC)
LJFOLD(NE KGC KGC)
LJFOLDF(kfold_kref)
{
  return CONDFOLD((fins->op1 == fins->op2) ^ (fins->o == IR_NE));
}

/* -- Algebraic shortcuts ------------------------------------------------ */

LJFOLD(FPMATH FPMATH IRFPM_FLOOR)
LJFOLD(FPMATH FPMATH IRFPM_CEIL)
LJFOLD(FPMATH FPMATH IRFPM_TRUNC)
LJFOLDF(shortcut_round)
{
  IRFPMathOp op = (IRFPMathOp)fleft->op2;
  if (op == IRFPM_FLOOR || op == IRFPM_CEIL || op == IRFPM_TRUNC)
    return LEFTFOLD;  /* round(round_left(x)) = round_left(x) */
  return NEXTFOLD;
}

LJFOLD(ABS ABS KNUM)
LJFOLDF(shortcut_left)
{
  return LEFTFOLD;  /* f(g(x)) ==> g(x) */
}

LJFOLD(ABS NEG KNUM)
LJFOLDF(shortcut_dropleft)
{
  PHIBARRIER(fleft);
  fins->op1 = fleft->op1;  /* abs(neg(x)) ==> abs(x) */
  return RETRYFOLD;
}

/* Note: no safe shortcuts with STRTO and TOSTR ("1e2" ==> +100 ==> "100"). */
LJFOLD(NEG NEG any)
LJFOLD(BNOT BNOT)
LJFOLD(BSWAP BSWAP)
LJFOLDF(shortcut_leftleft)
{
  PHIBARRIER(fleft);  /* See above. Fold would be ok, but not beneficial. */
  return fleft->op1;  /* f(g(x)) ==> x */
}

/* -- FP algebraic simplifications ------------------------------------- */

/* FP arithmetic is tricky -- there's not much to simplify.
** Please note the following common pitfalls before sending "improvements":
**   x+0 ==> x  is INVALID for x=-0
**   0-x ==> -x is INVALID for x=+0
**   x*0 ==> 0  is INVALID for x=-0, x=+-Inf or x=NaN
*/

LJFOLD(ADD NEG any)
LJFOLDF(simplify_numadd_negx)
{
  PHIBARRIER(fleft);
  fins->o = IR_SUB;  /* (-a) + b ==> b - a */
  fins->op1 = fins->op2;
  fins->op2 = fleft->op1;
  return RETRYFOLD;
}

LJFOLD(ADD any NEG)
LJFOLDF(simplify_numadd_xneg)
{
  PHIBARRIER(fright);
  fins->o = IR_SUB;  /* a + (-b) ==> a - b */
  fins->op2 = fright->op1;
  return RETRYFOLD;
}

LJFOLD(SUB any KNUM)
```

```
967  LJFOLDF(simplify_numsub_k)
968  {
969    lua_Number n = knumright;
970    if (n == 0.0)  /* x - (+-0) ==> x */
971      return LEFTFOLD;
972    return NEXTFOLD;
973  }
974
975  LJFOLD(SUB NEG KNUM)
976  LJFOLDF(simplify_numsub_negk)
977  {
978    PHIBARRIER(fleft);
979    fins->op2 = fleft->op1;  /* (-x) - k ==> (-k) - x */
980    fins->op1 = (IRRef1)lj_ir_knum(J, -knumright);
981    return RETRYFOLD;
982  }
983
984  LJFOLD(SUB any NEG)
985  LJFOLDF(simplify_numsub_xneg)
986  {
987    PHIBARRIER(fright);
988    fins->o = IR_ADD;  /* a - (-b) ==> a + b */
989    fins->op2 = fright->op1;
990    return RETRYFOLD;
991  }
992
993  LJFOLD(MUL any KNUM)
994  LJFOLD(DIV any KNUM)
995  LJFOLDF(simplify_nummuldiv_k)
996  {
997    lua_Number n = knumright;
998    if (n == 1.0) {  /* x o 1 ==> x */
999      return LEFTFOLD;
1000   } else if (n == -1.0) {  /* x o -1 ==> -x */
1001     fins->o = IR_NEG;
1002     fins->op2 = (IRRef1)lj_ir_knum_neg(J);
1003     return RETRYFOLD;
1004   } else if (fins->o == IR_MUL && n == 2.0) {  /* x * 2 ==> x + x */
1005     fins->o = IR_ADD;
1006     fins->op2 = fins->op1;
1007     return RETRYFOLD;
1008   } else if (fins->o == IR_DIV) {  /* x / 2^k ==> x * 2^-k */
1009     uint64_t u = ir_knum(fright)->u64;
1010     uint32_t ex = ((uint32_t)(u >> 52) & 0x7ff);
1011     if ((u & U64x(000fffff,ffffffff)) == 0 && ex - 1 < 0x7fd) {
1012       u = (u & ((uint64_t)1 << 63)) | ((uint64_t)(0x7fe - ex) << 52);
1013       fins->o = IR_MUL;  /* Multiply by exact reciprocal. */
1014       fins->op2 = lj_ir_knum_u64(J, u);
1015       return RETRYFOLD;
1016     }
1017   }
1018   return NEXTFOLD;
1019 }
1020
1021 LJFOLD(MUL NEG KNUM)
1022 LJFOLD(DIV NEG KNUM)
1023 LJFOLDF(simplify_nummuldiv_negk)
1024 {
1025   PHIBARRIER(fleft);
1026   fins->op1 = fleft->op1;  /* (-a) o k ==> a o (-k) */
1027   fins->op2 = (IRRef1)lj_ir_knum(J, -knumright);
1028   return RETRYFOLD;
1029 }
1030
1031 LJFOLD(MUL NEG NEG)
1032 LJFOLD(DIV NEG NEG)
1033 LJFOLDF(simplify_nummuldiv_negneg)
1034 {
1035   PHIBARRIER(fleft);
1036   PHIBARRIER(fright);
1037   fins->op1 = fleft->op1;  /* (-a) o (-b) ==> a o b */
1038   fins->op2 = fright->op1;
1039   return RETRYFOLD;
1040 }
1041
1042 LJFOLD(POW any KINT)
```

```
LJFOLDF(simplify_numpow_xk)
{
  int32_t k = fright->i;
  TRef ref = fins->op1;
  if (k == 0)  /* x ^ 0 ==> 1 */
    return lj_ir_knum_one(J);  /* Result must be a number, not an int. */
  if (k == 1)  /* x ^ 1 ==> x */
    return LEFTFOLD;
  if ((uint32_t)(k+65536) > 2*65536u)  /* Limit code explosion. */
    return NEXTFOLD;
  if (k < 0) {  /* x ^ (-k) ==> (1/x) ^ k. */
    ref = emitir(IRTN(IR_DIV), lj_ir_knum_one(J), ref);
    k = -k;
  }
  /* Unroll x^k for 1 <= k <= 65536. */
  for (; (k & 1) == 0; k >>= 1)  /* Handle leading zeros. */
    ref = emitir(IRTN(IR_MUL), ref, ref);
  if ((k >>= 1) != 0) {  /* Handle trailing bits. */
    TRef tmp = emitir(IRTN(IR_MUL), ref, ref);
    for (; k != 1; k >>= 1) {
      if (k & 1)
        ref = emitir(IRTN(IR_MUL), ref, tmp);
      tmp = emitir(IRTN(IR_MUL), tmp, tmp);
    }
    ref = emitir(IRTN(IR_MUL), ref, tmp);
  }
  return ref;
}

LJFOLD(POW KNUM any)
LJFOLDF(simplify_numpow_kx)
{
  lua_Number n = knumleft;
  if (n == 2.0) {  /* 2.0 ^ i ==> ldexp(1.0, tonum(i)) */
    fins->o = IR_CONV;
#if LJ_TARGET_X86ORX64
    fins->op1 = fins->op2;
    fins->op2 = IRCONV_NUM_INT;
    fins->op2 = (IRRef1)lj_opt_fold(J);
#endif
    fins->op1 = (IRRef1)lj_ir_knum_one(J);
    fins->o = IR_LDEXP;
    return RETRYFOLD;
  }
  return NEXTFOLD;
}

/* -- Simplify conversions ------------------------------------------------ */

LJFOLD(CONV CONV IRCONV_NUM_INT)  /* _NUM */
LJFOLDF(shortcut_conv_num_int)
{
  PHIBARRIER(fleft);
  /* Only safe with a guarded conversion to int. */
  if ((fleft->op2 & IRCONV_SRCMASK) == IRT_NUM && irt_isguard(fleft->t))
    return fleft->op1;  /* f(g(x)) ==> x */
  return NEXTFOLD;
}

LJFOLD(CONV CONV IRCONV_INT_NUM)  /* _INT */
LJFOLD(CONV CONV IRCONV_U32_NUM)  /* _U32*/
LJFOLDF(simplify_conv_int_num)
{
  /* Fold even across PHI to avoid expensive num->int conversions in loop. */
  if ((fleft->op2 & IRCONV_SRCMASK) ==
      ((fins->op2 & IRCONV_DSTMASK) >> IRCONV_DSH))
    return fleft->op1;
  return NEXTFOLD;
}

LJFOLD(CONV CONV IRCONV_I64_NUM)  /* _INT or _U32 */
LJFOLD(CONV CONV IRCONV_U64_NUM)  /* _INT or _U32 */
LJFOLDF(simplify_conv_i64_num)
{
  PHIBARRIER(fleft);
  if ((fleft->op2 & IRCONV_SRCMASK) == IRT_INT) {
```

```
    /* Reduce to a sign-extension. */
    fins->op1 = fleft->op1;
    fins->op2 = ((IRT_I64<<5)|IRT_INT|IRCONV_SEXT);
    return RETRYFOLD;
  } else if ((fleft->op2 & IRCONV_SRCMASK) == IRT_U32) {
#if LJ_TARGET_X64
    return fleft->op1;
#else
    /* Reduce to a zero-extension. */
    fins->op1 = fleft->op1;
    fins->op2 = (IRT_I64<<5)|IRT_U32;
    return RETRYFOLD;
#endif
  }
  return NEXTFOLD;
}

LJFOLD(CONV CONV IRCONV_INT_I64)  /* _INT or _U32 */
LJFOLD(CONV CONV IRCONV_INT_U64)  /* _INT or _U32 */
LJFOLD(CONV CONV IRCONV_U32_I64)  /* _INT or _U32 */
LJFOLD(CONV CONV IRCONV_U32_U64)  /* _INT or _U32 */
LJFOLDF(simplify_conv_int_i64)
{
  int src;
  PHIBARRIER(fleft);
  src = (fleft->op2 & IRCONV_SRCMASK);
  if (src == IRT_INT || src == IRT_U32) {
    if (src == ((fins->op2 & IRCONV_DSTMASK) >> IRCONV_DSH)) {
      return fleft->op1;
    } else {
      fins->op2 = ((fins->op2 & IRCONV_DSTMASK) | src);
      fins->op1 = fleft->op1;
      return RETRYFOLD;
    }
  }
  return NEXTFOLD;
}

LJFOLD(CONV CONV IRCONV_FLOAT_NUM)  /* _FLOAT */
LJFOLDF(simplify_conv_flt_num)
{
  PHIBARRIER(fleft);
  if ((fleft->op2 & IRCONV_SRCMASK) == IRT_FLOAT)
    return fleft->op1;
  return NEXTFOLD;
}

/* Shortcut TOBIT + IRT_NUM <- IRT_INT/IRT_U32 conversion. */
LJFOLD(TOBIT CONV KNUM)
LJFOLDF(simplify_tobit_conv)
{
  /* Fold even across PHI to avoid expensive num->int conversions in loop. */
  if ((fleft->op2 & IRCONV_SRCMASK) == IRT_INT) {
    lua_assert(irt_isnum(fleft->t));
    return fleft->op1;
  } else if ((fleft->op2 & IRCONV_SRCMASK) == IRT_U32) {
    lua_assert(irt_isnum(fleft->t));
    fins->o = IR_CONV;
    fins->op1 = fleft->op1;
    fins->op2 = (IRT_INT<<5)|IRT_U32;
    return RETRYFOLD;
  }
  return NEXTFOLD;
}

/* Shortcut floor/ceil/round + IRT_NUM <- IRT_INT/IRT_U32 conversion. */
LJFOLD(FPMATH CONV IRFPM_FLOOR)
LJFOLD(FPMATH CONV IRFPM_CEIL)
LJFOLD(FPMATH CONV IRFPM_TRUNC)
LJFOLDF(simplify_floor_conv)
{
  if ((fleft->op2 & IRCONV_SRCMASK) == IRT_INT ||
      (fleft->op2 & IRCONV_SRCMASK) == IRT_U32)
    return LEFTFOLD;
  return NEXTFOLD;
}
```

```
1195
1196  /* Strength reduction of widening. */
1197  LJFOLD(CONV any IRCONV_I64_INT)
1198  LJFOLD(CONV any IRCONV_U64_INT)
1199  LJFOLDF(simplify_conv_sext)
1200  {
1201    IRRef ref = fins->op1;
1202    int64_t ofs = 0;
1203    if (!(fins->op2 & IRCONV_SEXT))
1204      return NEXTFOLD;
1205    PHIBARRIER(fleft);
1206    if (fleft->o == IR_XLOAD && (irt_isu8(fleft->t) || irt_isu16(fleft->t)))
1207      goto ok_reduce;
1208    if (fleft->o == IR_ADD && irref_isk(fleft->op2)) {
1209      ofs = (int64_t)IR(fleft->op2)->i;
1210      ref = fleft->op1;
1211    }
1212    /* Use scalar evolution analysis results to strength-reduce sign-extension. */
1213    if (ref == J->scev.idx) {
1214      IRRef lo = J->scev.dir ? J->scev.start : J->scev.stop;
1215      lua_assert(irt_isint(J->scev.t));
1216      if (lo && IR(lo)->i + ofs >= 0) {
1217      ok_reduce:
1218  #if LJ_TARGET_X64
1219        /* Eliminate widening. All 32 bit ops do an implicit zero-extension. */
1220        return LEFTFOLD;
1221  #else
1222        /* Reduce to a (cheaper) zero-extension. */
1223        fins->op2 &= ~IRCONV_SEXT;
1224        return RETRYFOLD;
1225  #endif
1226      }
1227    }
1228    return NEXTFOLD;
1229  }
1230
1231  /* Strength reduction of narrowing. */
1232  LJFOLD(CONV ADD IRCONV_INT_I64)
1233  LJFOLD(CONV SUB IRCONV_INT_I64)
1234  LJFOLD(CONV MUL IRCONV_INT_I64)
1235  LJFOLD(CONV ADD IRCONV_INT_U64)
1236  LJFOLD(CONV SUB IRCONV_INT_U64)
1237  LJFOLD(CONV MUL IRCONV_INT_U64)
1238  LJFOLD(CONV ADD IRCONV_U32_I64)
1239  LJFOLD(CONV SUB IRCONV_U32_I64)
1240  LJFOLD(CONV MUL IRCONV_U32_I64)
1241  LJFOLD(CONV ADD IRCONV_U32_U64)
1242  LJFOLD(CONV SUB IRCONV_U32_U64)
1243  LJFOLD(CONV MUL IRCONV_U32_U64)
1244  LJFOLDF(simplify_conv_narrow)
1245  {
1246    IROp op = (IROp)fleft->o;
1247    IRType t = irt_type(fins->t);
1248    IRRef op1 = fleft->op1, op2 = fleft->op2, mode = fins->op2;
1249    PHIBARRIER(fleft);
1250    op1 = emitir(IRTI(IR_CONV), op1, mode);
1251    op2 = emitir(IRTI(IR_CONV), op2, mode);
1252    fins->ot = IRT(op, t);
1253    fins->op1 = op1;
1254    fins->op2 = op2;
1255    return RETRYFOLD;
1256  }
1257
1258  /* Special CSE rule for CONV. */
1259  LJFOLD(CONV any any)
1260  LJFOLDF(cse_conv)
1261  {
1262    if (LJ_LIKELY(J->flags & JIT_F_OPT_CSE)) {
1263      IRRef op1 = fins->op1, op2 = (fins->op2 & IRCONV_MODEMASK);
1264      uint8_t guard = irt_isguard(fins->t);
1265      IRRef ref = J->chain[IR_CONV];
1266      while (ref > op1) {
1267        IRIns *ir = IR(ref);
1268        /* Commoning with stronger checks is ok. */
1269        if (ir->op1 == op1 && (ir->op2 & IRCONV_MODEMASK) == op2 &&
1270            irt_isguard(ir->t) >= guard)
```

```
1271          return ref;
1272        ref = ir->prev;
1273      }
1274    }
1275    return EMITFOLD;  /* No fallthrough to regular CSE. */
1276  }
1277
1278  /* FP conversion narrowing. */
1279  LJFOLD(TOBIT ADD KNUM)
1280  LJFOLD(TOBIT SUB KNUM)
1281  LJFOLD(CONV ADD IRCONV_INT_NUM)
1282  LJFOLD(CONV SUB IRCONV_INT_NUM)
1283  LJFOLD(CONV ADD IRCONV_I64_NUM)
1284  LJFOLD(CONV SUB IRCONV_I64_NUM)
1285  LJFOLDF(narrow_convert)
1286  {
1287    PHIBARRIER(fleft);
1288    /* Narrowing ignores PHIs and repeating it inside the loop is not useful. */
1289    if (J->chain[IR_LOOP])
1290      return NEXTFOLD;
1291    lua_assert(fins->o != IR_CONV || (fins->op2&IRCONV_CONVMASK) != IRCONV_TOBIT);
1292    return lj_opt_narrow_convert(J);
1293  }
1294
1295  /* -- Integer algebraic simplifications ---------------------------------- */
1296
1297  LJFOLD(ADD any KINT)
1298  LJFOLD(ADDOV any KINT)
1299  LJFOLD(SUBOV any KINT)
1300  LJFOLDF(simplify_intadd_k)
1301  {
1302    if (fright->i == 0)  /* i o 0 ==> i */
1303      return LEFTFOLD;
1304    return NEXTFOLD;
1305  }
1306
1307  LJFOLD(MULOV any KINT)
1308  LJFOLDF(simplify_intmul_k)
1309  {
1310    if (fright->i == 0)  /* i * 0 ==> 0 */
1311      return RIGHTFOLD;
1312    if (fright->i == 1)  /* i * 1 ==> i */
1313      return LEFTFOLD;
1314    if (fright->i == 2) {  /* i * 2 ==> i + i */
1315      fins->o = IR_ADDOV;
1316      fins->op2 = fins->op1;
1317      return RETRYFOLD;
1318    }
1319    return NEXTFOLD;
1320  }
1321
1322  LJFOLD(SUB any KINT)
1323  LJFOLDF(simplify_intsub_k)
1324  {
1325    if (fright->i == 0)  /* i - 0 ==> i */
1326      return LEFTFOLD;
1327    fins->o = IR_ADD;  /* i - k ==> i + (-k) */
1328    fins->op2 = (IRRef1)lj_ir_kint(J, -fright->i);  /* Overflow for -2^31 ok. */
1329    return RETRYFOLD;
1330  }
1331
1332  LJFOLD(SUB KINT any)
1333  LJFOLD(SUB KINT64 any)
1334  LJFOLDF(simplify_intsub_kleft)
1335  {
1336    if (fleft->o == IR_KINT ? (fleft->i == 0) : (ir_kint64(fleft)->u64 == 0)) {
1337      fins->o = IR_NEG;  /* 0 - i ==> -i */
1338      fins->op1 = fins->op2;
1339      return RETRYFOLD;
1340    }
1341    return NEXTFOLD;
1342  }
1343
1344  LJFOLD(ADD any KINT64)
1345  LJFOLDF(simplify_intadd_k64)
1346  {
```

```
  if (ir_kint64(fright)->u64 == 0)  /* i + 0 ==> i */
    return LEFTFOLD;
  return NEXTFOLD;
}

LJFOLD(SUB any KINT64)
LJFOLDF(simplify_intsub_k64)
{
  uint64_t k = ir_kint64(fright)->u64;
  if (k == 0)  /* i - 0 ==> i */
    return LEFTFOLD;
  fins->o = IR_ADD;  /* i - k ==> i + (-k) */
  fins->op2 = (IRRef1)lj_ir_kint64(J, (uint64_t)-(int64_t)k);
  return RETRYFOLD;
}

static TRef simplify_intmul_k(jit_State *J, int32_t k)
{
  /* Note: many more simplifications are possible, e.g. 2^k1 +- 2^k2.
  ** But this is mainly intended for simple address arithmetic.
  ** Also it's easier for the backend to optimize the original multiplies.
  */
  if (k == 0) {  /* i * 0 ==> 0 */
    return RIGHTFOLD;
  } else if (k == 1) {  /* i * 1 ==> i */
    return LEFTFOLD;
  } else if ((k & (k-1)) == 0) {  /* i * 2^k ==> i << k */
    fins->o = IR_BSHL;
    fins->op2 = lj_ir_kint(J, lj_fls((uint32_t)k));
    return RETRYFOLD;
  }
  return NEXTFOLD;
}

LJFOLD(MUL any KINT)
LJFOLDF(simplify_intmul_k32)
{
  if (fright->i >= 0)
    return simplify_intmul_k(J, fright->i);
  return NEXTFOLD;
}

LJFOLD(MUL any KINT64)
LJFOLDF(simplify_intmul_k64)
{
#if LJ_HASFFI
  if (ir_kint64(fright)->u64 < 0x80000000u)
    return simplify_intmul_k(J, (int32_t)ir_kint64(fright)->u64);
  return NEXTFOLD;
#else
  UNUSED(J); lua_assert(0); return FAILFOLD;
#endif
}

LJFOLD(MOD any KINT)
LJFOLDF(simplify_intmod_k)
{
  int32_t k = fright->i;
  lua_assert(k != 0);
  if (k > 0 && (k & (k-1)) == 0) {  /* i % (2^k) ==> i & (2^k-1) */
    fins->o = IR_BAND;
    fins->op2 = lj_ir_kint(J, k-1);
    return RETRYFOLD;
  }
  return NEXTFOLD;
}

LJFOLD(MOD KINT any)
LJFOLDF(simplify_intmod_kleft)
{
  if (fleft->i == 0)
    return INTFOLD(0);
  return NEXTFOLD;
}

LJFOLD(SUB any any)
```

```
LJFOLD(SUBOV any any)
LJFOLDF(simplify_intsub)
{
  if (fins->op1 == fins->op2 && !irt_isnum(fins->t))  /* i - i ==> 0 */
    return irt_is64(fins->t) ? INT64FOLD(0) : INTFOLD(0);
  return NEXTFOLD;
}

LJFOLD(SUB ADD any)
LJFOLDF(simplify_intsubadd_leftcancel)
{
  if (!irt_isnum(fins->t)) {
    PHIBARRIER(fleft);
    if (fins->op2 == fleft->op1)  /* (i + j) - i ==> j */
      return fleft->op2;
    if (fins->op2 == fleft->op2)  /* (i + j) - j ==> i */
      return fleft->op1;
  }
  return NEXTFOLD;
}

LJFOLD(SUB SUB any)
LJFOLDF(simplify_intsubsub_leftcancel)
{
  if (!irt_isnum(fins->t)) {
    PHIBARRIER(fleft);
    if (fins->op2 == fleft->op1) {  /* (i - j) - i ==> 0 - j */
      fins->op1 = (IRRef1)lj_ir_kint(J, 0);
      fins->op2 = fleft->op2;
      return RETRYFOLD;
    }
  }
  return NEXTFOLD;
}

LJFOLD(SUB any SUB)
LJFOLDF(simplify_intsubsub_rightcancel)
{
  if (!irt_isnum(fins->t)) {
    PHIBARRIER(fright);
    if (fins->op1 == fright->op1)  /* i - (i - j) ==> j */
      return fright->op2;
  }
  return NEXTFOLD;
}

LJFOLD(SUB any ADD)
LJFOLDF(simplify_intsubadd_rightcancel)
{
  if (!irt_isnum(fins->t)) {
    PHIBARRIER(fright);
    if (fins->op1 == fright->op1) {  /* i - (i + j) ==> 0 - j */
      fins->op2 = fright->op2;
      fins->op1 = (IRRef1)lj_ir_kint(J, 0);
      return RETRYFOLD;
    }
    if (fins->op1 == fright->op2) {  /* i - (j + i) ==> 0 - j */
      fins->op2 = fright->op1;
      fins->op1 = (IRRef1)lj_ir_kint(J, 0);
      return RETRYFOLD;
    }
  }
  return NEXTFOLD;
}

LJFOLD(SUB ADD ADD)
LJFOLDF(simplify_intsubaddadd_cancel)
{
  if (!irt_isnum(fins->t)) {
    PHIBARRIER(fleft);
    PHIBARRIER(fright);
    if (fleft->op1 == fright->op1) {  /* (i + j1) - (i + j2) ==> j1 - j2 */
      fins->op1 = fleft->op2;
      fins->op2 = fright->op2;
      return RETRYFOLD;
    }
```

```
    if (fleft->op1 == fright->op2) {  /* (i + j1) - (j2 + i) ==> j1 - j2 */
      fins->op1 = fleft->op2;
      fins->op2 = fright->op1;
      return RETRYFOLD;
    }
    if (fleft->op2 == fright->op1) {  /* (j1 + i) - (i + j2) ==> j1 - j2 */
      fins->op1 = fleft->op1;
      fins->op2 = fright->op2;
      return RETRYFOLD;
    }
    if (fleft->op2 == fright->op2) {  /* (j1 + i) - (j2 + i) ==> j1 - j2 */
      fins->op1 = fleft->op1;
      fins->op2 = fright->op1;
      return RETRYFOLD;
    }
  }
  return NEXTFOLD;
}

LJFOLD(BAND any KINT)
LJFOLD(BAND any KINT64)
LJFOLDF(simplify_band_k)
{
  int64_t k = fright->o == IR_KINT ? (int64_t)fright->i :
                                     (int64_t)ir_k64(fright)->u64;
  if (k == 0)  /* i & 0 ==> 0 */
    return RIGHTFOLD;
  if (k == -1)  /* i & -1 ==> i */
    return LEFTFOLD;
  return NEXTFOLD;
}

LJFOLD(BOR any KINT)
LJFOLD(BOR any KINT64)
LJFOLDF(simplify_bor_k)
{
  int64_t k = fright->o == IR_KINT ? (int64_t)fright->i :
                                     (int64_t)ir_k64(fright)->u64;
  if (k == 0)  /* i | 0 ==> i */
    return LEFTFOLD;
  if (k == -1)  /* i | -1 ==> -1 */
    return RIGHTFOLD;
  return NEXTFOLD;
}

LJFOLD(BXOR any KINT)
LJFOLD(BXOR any KINT64)
LJFOLDF(simplify_bxor_k)
{
  int64_t k = fright->o == IR_KINT ? (int64_t)fright->i :
                                     (int64_t)ir_k64(fright)->u64;
  if (k == 0)  /* i xor 0 ==> i */
    return LEFTFOLD;
  if (k == -1) {  /* i xor -1 ==> ~i */
    fins->o = IR_BNOT;
    fins->op2 = 0;
    return RETRYFOLD;
  }
  return NEXTFOLD;
}

LJFOLD(BSHL any KINT)
LJFOLD(BSHR any KINT)
LJFOLD(BSAR any KINT)
LJFOLD(BROL any KINT)
LJFOLD(BROR any KINT)
LJFOLDF(simplify_shift_ik)
{
  int32_t mask = irt_is64(fins->t) ? 63 : 31;
  int32_t k = (fright->i & mask);
  if (k == 0)  /* i o 0 ==> i */
    return LEFTFOLD;
  if (k == 1 && fins->o == IR_BSHL) {  /* i << 1 ==> i + i */
    fins->o = IR_ADD;
    fins->op2 = fins->op1;
    return RETRYFOLD;
```

```
  }
  if (k != fright->i) {  /* i o k ==> i o (k & mask) */
    fins->op2 = (IRRef1)lj_ir_kint(J, k);
    return RETRYFOLD;
  }
#ifndef LJ_TARGET_UNIFYROT
  if (fins->o == IR_BROR) {  /* bror(i, k) ==> brol(i, (-k)&mask) */
    fins->o = IR_BROL;
    fins->op2 = (IRRef1)lj_ir_kint(J, (-k)&mask);
    return RETRYFOLD;
  }
#endif
  return NEXTFOLD;
}

LJFOLD(BSHL any BAND)
LJFOLD(BSHR any BAND)
LJFOLD(BSAR any BAND)
LJFOLD(BROL any BAND)
LJFOLD(BROR any BAND)
LJFOLDF(simplify_shift_andk)
{
  IRIns *irk = IR(fright->op2);
  PHIBARRIER(fright);
  if ((fins->o < IR_BROL ? LJ_TARGET_MASKSHIFT : LJ_TARGET_MASKROT) &&
      irk->o == IR_KINT) {  /* i o (j & mask) ==> i o j */
    int32_t mask = irt_is64(fins->t) ? 63 : 31;
    int32_t k = irk->i & mask;
    if (k == mask) {
      fins->op2 = fright->op1;
      return RETRYFOLD;
    }
  }
  return NEXTFOLD;
}

LJFOLD(BSHL KINT any)
LJFOLD(BSHR KINT any)
LJFOLD(BSHL KINT64 any)
LJFOLD(BSHR KINT64 any)
LJFOLDF(simplify_shift1_ki)
{
  int64_t k = fleft->o == IR_KINT ? (int64_t)fleft->i :
                                    (int64_t)ir_k64(fleft)->u64;
  if (k == 0)  /* 0 o i ==> 0 */
    return LEFTFOLD;
  return NEXTFOLD;
}

LJFOLD(BSAR KINT any)
LJFOLD(BROL KINT any)
LJFOLD(BROR KINT any)
LJFOLD(BSAR KINT64 any)
LJFOLD(BROL KINT64 any)
LJFOLD(BROR KINT64 any)
LJFOLDF(simplify_shift2_ki)
{
  int64_t k = fleft->o == IR_KINT ? (int64_t)fleft->i :
                                    (int64_t)ir_k64(fleft)->u64;
  if (k == 0 || k == -1)  /* 0 o i ==> 0; -1 o i ==> -1 */
    return LEFTFOLD;
  return NEXTFOLD;
}

LJFOLD(BSHL BAND KINT)
LJFOLD(BSHR BAND KINT)
LJFOLD(BROL BAND KINT)
LJFOLD(BROR BAND KINT)
LJFOLDF(simplify_shiftk_andk)
{
  IRIns *irk = IR(fleft->op2);
  PHIBARRIER(fleft);
  if (irk->o == IR_KINT) {  /* (i & k1) o k2 ==> (i o k2) & (k1 o k2) */
    int32_t k = kfold_intop(irk->i, fright->i, (IROp)fins->o);
    fins->op1 = fleft->op1;
    fins->op1 = (IRRef1)lj_opt_fold(J);
```

```
1651      fins->op2 = (IRRef1)lj_ir_kint(J, k);
1652      fins->ot = IRTI(IR_BAND);
1653      return RETRYFOLD;
1654    }
1655    return NEXTFOLD;
1656  }
1657
1658  LJFOLD(BAND BSHL KINT)
1659  LJFOLD(BAND BSHR KINT)
1660  LJFOLDF(simplify_andk_shiftk)
1661  {
1662    IRIns *irk = IR(fleft->op2);
1663    if (irk->o == IR_KINT &&
1664        kfold_intop(-1, irk->i, (IROp)fleft->o) == fright->i)
1665      return LEFTFOLD;  /* (i o k1) & k2 ==> i, if (-1 o k1) == k2 */
1666    return NEXTFOLD;
1667  }
1668
1669  /* -- Reassociation ------------------------------------------------------ */
1670
1671  LJFOLD(ADD ADD KINT)
1672  LJFOLD(MUL MUL KINT)
1673  LJFOLD(BAND BAND KINT)
1674  LJFOLD(BOR BOR KINT)
1675  LJFOLD(BXOR BXOR KINT)
1676  LJFOLDF(reassoc_intarith_k)
1677  {
1678    IRIns *irk = IR(fleft->op2);
1679    if (irk->o == IR_KINT) {
1680      int32_t k = kfold_intop(irk->i, fright->i, (IROp)fins->o);
1681      if (k == irk->i)  /* (i o k1) o k2 ==> i o k1, if (k1 o k2) == k1. */
1682        return LEFTFOLD;
1683      PHIBARRIER(fleft);
1684      fins->op1 = fleft->op1;
1685      fins->op2 = (IRRef1)lj_ir_kint(J, k);
1686      return RETRYFOLD;  /* (i o k1) o k2 ==> i o (k1 o k2) */
1687    }
1688    return NEXTFOLD;
1689  }
1690
1691  LJFOLD(ADD ADD KINT64)
1692  LJFOLD(MUL MUL KINT64)
1693  LJFOLD(BAND BAND KINT64)
1694  LJFOLD(BOR BOR KINT64)
1695  LJFOLD(BXOR BXOR KINT64)
1696  LJFOLDF(reassoc_intarith_k64)
1697  {
1698  #if LJ_HASFFI
1699    IRIns *irk = IR(fleft->op2);
1700    if (irk->o == IR_KINT64) {
1701      uint64_t k = kfold_int64arith(ir_k64(irk)->u64,
1702                                    ir_k64(fright)->u64, (IROp)fins->o);
1703      PHIBARRIER(fleft);
1704      fins->op1 = fleft->op1;
1705      fins->op2 = (IRRef1)lj_ir_kint64(J, k);
1706      return RETRYFOLD;  /* (i o k1) o k2 ==> i o (k1 o k2) */
1707    }
1708    return NEXTFOLD;
1709  #else
1710    UNUSED(J); lua_assert(0); return FAILFOLD;
1711  #endif
1712  }
1713
1714  LJFOLD(MIN MIN any)
1715  LJFOLD(MAX MAX any)
1716  LJFOLD(BAND BAND any)
1717  LJFOLD(BOR BOR any)
1718  LJFOLDF(reassoc_dup)
1719  {
1720    if (fins->op2 == fleft->op1 || fins->op2 == fleft->op2)
1721      return LEFTFOLD;  /* (a o b) o a ==> a o b; (a o b) o b ==> a o b */
1722    return NEXTFOLD;
1723  }
1724
1725  LJFOLD(BXOR BXOR any)
1726  LJFOLDF(reassoc_bxor)
```

```
{
  PHIBARRIER(fleft);
  if (fins->op2 == fleft->op1)  /* (a xor b) xor a ==> b */
    return fleft->op2;
  if (fins->op2 == fleft->op2)  /* (a xor b) xor b ==> a */
    return fleft->op1;
  return NEXTFOLD;
}

LJFOLD(BSHL BSHL KINT)
LJFOLD(BSHR BSHR KINT)
LJFOLD(BSAR BSAR KINT)
LJFOLD(BROL BROL KINT)
LJFOLD(BROR BROR KINT)
LJFOLDF(reassoc_shift)
{
  IRIns *irk = IR(fleft->op2);
  PHIBARRIER(fleft);  /* The (shift any KINT) rule covers k2 == 0 and more. */
  if (irk->o == IR_KINT) {  /* (i o k1) o k2 ==> i o (k1 + k2) */
    int32_t mask = irt_is64(fins->t) ? 63 : 31;
    int32_t k = (irk->i & mask) + (fright->i & mask);
    if (k > mask) {  /* Combined shift too wide? */
      if (fins->o == IR_BSHL || fins->o == IR_BSHR)
        return mask == 31 ? INTFOLD(0) : INT64FOLD(0);
      else if (fins->o == IR_BSAR)
        k = mask;
      else
        k &= mask;
    }
    fins->op1 = fleft->op1;
    fins->op2 = (IRRef1)lj_ir_kint(J, k);
    return RETRYFOLD;
  }
  return NEXTFOLD;
}

LJFOLD(MIN MIN KNUM)
LJFOLD(MAX MAX KNUM)
LJFOLD(MIN MIN KINT)
LJFOLD(MAX MAX KINT)
LJFOLDF(reassoc_minmax_k)
{
  IRIns *irk = IR(fleft->op2);
  if (irk->o == IR_KNUM) {
    lua_Number a = ir_knum(irk)->n;
    lua_Number y = lj_vm_foldarith(a, knumright, fins->o - IR_ADD);
    if (a == y)  /* (x o k1) o k2 ==> x o k1, if (k1 o k2) == k1. */
      return LEFTFOLD;
    PHIBARRIER(fleft);
    fins->op1 = fleft->op1;
    fins->op2 = (IRRef1)lj_ir_knum(J, y);
    return RETRYFOLD;  /* (x o k1) o k2 ==> x o (k1 o k2) */
  } else if (irk->o == IR_KINT) {
    int32_t a = irk->i;
    int32_t y = kfold_intop(a, fright->i, fins->o);
    if (a == y)  /* (x o k1) o k2 ==> x o k1, if (k1 o k2) == k1. */
      return LEFTFOLD;
    PHIBARRIER(fleft);
    fins->op1 = fleft->op1;
    fins->op2 = (IRRef1)lj_ir_kint(J, y);
    return RETRYFOLD;  /* (x o k1) o k2 ==> x o (k1 o k2) */
  }
  return NEXTFOLD;
}

LJFOLD(MIN MAX any)
LJFOLD(MAX MIN any)
LJFOLDF(reassoc_minmax_left)
{
  if (fins->op2 == fleft->op1 || fins->op2 == fleft->op2)
    return RIGHTFOLD;  /* (b o1 a) o2 b ==> b; (a o1 b) o2 b ==> b */
  return NEXTFOLD;
}

LJFOLD(MIN any MAX)
LJFOLD(MAX any MIN)
```

```
LJFOLDF(reassoc_minmax_right)
{
  if (fins->op1 == fright->op1 || fins->op1 == fright->op2)
    return LEFTFOLD;  /* a o2 (a o1 b) ==> a; a o2 (b o1 a) ==> a */
  return NEXTFOLD;
}

/* -- Array bounds check elimination ------------------------------------ */

/* Eliminate ABC across PHIs to handle t[i-1] forwarding case.
** ABC(asize, (i+k)+(-k)) ==> ABC(asize, i), but only if it already exists.
** Could be generalized to (i+k1)+k2 ==> i+(k1+k2), but needs better disambig.
*/
LJFOLD(ABC any ADD)
LJFOLDF(abc_fwd)
{
  if (LJ_LIKELY(J->flags & JIT_F_OPT_ABC)) {
    if (irref_isk(fright->op2)) {
      IRIns *add2 = IR(fright->op1);
      if (add2->o == IR_ADD && irref_isk(add2->op2) &&
          IR(fright->op2)->i == -IR(add2->op2)->i) {
        IRRef ref = J->chain[IR_ABC];
        IRRef lim = add2->op1;
        if (fins->op1 > lim) lim = fins->op1;
        while (ref > lim) {
          IRIns *ir = IR(ref);
          if (ir->op1 == fins->op1 && ir->op2 == add2->op1)
            return DROPFOLD;
          ref = ir->prev;
        }
      }
    }
  }
  return NEXTFOLD;
}

/* Eliminate ABC for constants.
** ABC(asize, k1), ABC(asize k2) ==> ABC(asize, max(k1, k2))
** Drop second ABC if k2 is lower. Otherwise patch first ABC with k2.
*/
LJFOLD(ABC any KINT)
LJFOLDF(abc_k)
{
  if (LJ_LIKELY(J->flags & JIT_F_OPT_ABC)) {
    IRRef ref = J->chain[IR_ABC];
    IRRef asize = fins->op1;
    while (ref > asize) {
      IRIns *ir = IR(ref);
      if (ir->op1 == asize && irref_isk(ir->op2)) {
        int32_t k = IR(ir->op2)->i;
        if (fright->i > k)
          ir->op2 = fins->op2;
        return DROPFOLD;
      }
      ref = ir->prev;
    }
    return EMITFOLD;  /* Already performed CSE. */
  }
  return NEXTFOLD;
}

/* Eliminate invariant ABC inside loop. */
LJFOLD(ABC any any)
LJFOLDF(abc_invar)
{
  /* Invariant ABC marked as PTR. Drop if op1 is invariant, too. */
  if (!irt_isint(fins->t) && fins->op1 < J->chain[IR_LOOP] &&
      !irt_isphi(IR(fins->op1)->t))
    return DROPFOLD;
  return NEXTFOLD;
}

/* -- Commutativity ----------------------------------------------------- */

/* The refs of commutative ops are canonicalized. Lower refs go to the right.
** Rationale behind this:
```

```
** - It (also) moves constants to the right.
** - It reduces the number of FOLD rules (e.g. (BOR any KINT) suffices).
** - It helps CSE to find more matches.
** - The assembler generates better code with constants at the right.
*/

LJFOLD(ADD any any)
LJFOLD(MUL any any)
LJFOLD(ADDOV any any)
LJFOLD(MULOV any any)
LJFOLDF(comm_swap)
{
  if (fins->op1 < fins->op2) {  /* Move lower ref to the right. */
    IRRef1 tmp = fins->op1;
    fins->op1 = fins->op2;
    fins->op2 = tmp;
    return RETRYFOLD;
  }
  return NEXTFOLD;
}

LJFOLD(EQ any any)
LJFOLD(NE any any)
LJFOLDF(comm_equal)
{
  /* For non-numbers only: x == x ==> drop; x ~= x ==> fail */
  if (fins->op1 == fins->op2 && !irt_isnum(fins->t))
    return CONDFOLD(fins->o == IR_EQ);
  return fold_comm_swap(J);
}

LJFOLD(LT any any)
LJFOLD(GE any any)
LJFOLD(LE any any)
LJFOLD(GT any any)
LJFOLD(ULT any any)
LJFOLD(UGE any any)
LJFOLD(ULE any any)
LJFOLD(UGT any any)
LJFOLDF(comm_comp)
{
  /* For non-numbers only: x <=> x ==> drop; x <> x ==> fail */
  if (fins->op1 == fins->op2 && !irt_isnum(fins->t))
    return CONDFOLD((fins->o ^ (fins->o >> 1)) & 1);
  if (fins->op1 < fins->op2) {  /* Move lower ref to the right. */
    IRRef1 tmp = fins->op1;
    fins->op1 = fins->op2;
    fins->op2 = tmp;
    fins->o ^= 3; /* GT <-> LT, GE <-> LE, does not affect U */
    return RETRYFOLD;
  }
  return NEXTFOLD;
}

LJFOLD(BAND any any)
LJFOLD(BOR any any)
LJFOLD(MIN any any)
LJFOLD(MAX any any)
LJFOLDF(comm_dup)
{
  if (fins->op1 == fins->op2)  /* x o x ==> x */
    return LEFTFOLD;
  return fold_comm_swap(J);
}

LJFOLD(BXOR any any)
LJFOLDF(comm_bxor)
{
  if (fins->op1 == fins->op2)  /* i xor i ==> 0 */
    return irt_is64(fins->t) ? INT64FOLD(0) : INTFOLD(0);
  return fold_comm_swap(J);
}

/* -- Simplification of compound expressions ------------------------------ */

static TRef kfold_xload(jit_State *J, IRIns *ir, const void *p)
```

```
{
  int32_t k;
  switch (irt_type(ir->t)) {
  case IRT_NUM: return lj_ir_knum_u64(J, *(uint64_t *)p);
  case IRT_I8: k = (int32_t)*(int8_t *)p; break;
  case IRT_U8: k = (int32_t)*(uint8_t *)p; break;
  case IRT_I16: k = (int32_t)(int16_t)lj_getu16(p); break;
  case IRT_U16: k = (int32_t)(uint16_t)lj_getu16(p); break;
  case IRT_INT: case IRT_U32: k = (int32_t)lj_getu32(p); break;
  case IRT_I64: case IRT_U64: return lj_ir_kint64(J, *(uint64_t *)p);
  default: return 0;
  }
  return lj_ir_kint(J, k);
}

/* Turn: string.sub(str, a, b) == kstr
** into: string.byte(str, a) == string.byte(kstr, 1) etc.
** Note: this creates unaligned XLOADs on x86/x64.
*/
LJFOLD(EQ SNEW KGC)
LJFOLD(NE SNEW KGC)
LJFOLDF(merge_eqne_snew_kgc)
{
  GCstr *kstr = ir_kstr(fright);
  int32_t len = (int32_t)kstr->len;
  lua_assert(irt_isstr(fins->t));

#if LJ_TARGET_UNALIGNED
#define FOLD_SNEW_MAX_LEN	4  /* Handle string lengths 0, 1, 2, 3, 4. */
#define FOLD_SNEW_TYPE8		   IRT_I8	/* Creates shorter immediates. */
#else
#define FOLD_SNEW_MAX_LEN	1  /* Handle string lengths 0 or 1. */
#define FOLD_SNEW_TYPE8		   IRT_U8  /* Prefer unsigned loads. */
#endif

  PHIBARRIER(fleft);
  if (len <= FOLD_SNEW_MAX_LEN) {
    IROp op = (IROp)fins->o;
    IRRef strref = fleft->op1;
    if (IR(strref)->o != IR_STRREF)
      return NEXTFOLD;
    if (op == IR_EQ) {
      emitir(IRTGI(IR_EQ), fleft->op2, lj_ir_kint(J, len));
      /* Caveat: fins/fleft/fright is no longer valid after emitir. */
    } else {
      /* NE is not expanded since this would need an OR of two conds. */
      if (!irref_isk(fleft->op2))  /* Only handle the constant length case. */
        return NEXTFOLD;
      if (IR(fleft->op2)->i != len)
        return DROPFOLD;
    }
    if (len > 0) {
      /* A 4 byte load for length 3 is ok -- all strings have an extra NUL. */
      uint16_t ot = (uint16_t)(len == 1 ? IRT(IR_XLOAD, FOLD_SNEW_TYPE8) :
                               len == 2 ? IRT(IR_XLOAD, IRT_U16) :
                               IRTI(IR_XLOAD));
      TRef tmp = emitir(ot, strref,
                        IRXLOAD_READONLY | (len > 1 ? IRXLOAD_UNALIGNED : 0));
      TRef val = kfold_xload(J, IR(tref_ref(tmp)), strdata(kstr));
      if (len == 3)
        tmp = emitir(IRTI(IR_BAND), tmp,
                     lj_ir_kint(J, LJ_ENDIAN_SELECT(0x00ffffff, 0xffffff00)));
      fins->op1 = (IRRef1)tmp;
      fins->op2 = (IRRef1)val;
      fins->ot = (IROpT)IRTGI(op);
      return RETRYFOLD;
    } else {
      return DROPFOLD;
    }
  }
  return NEXTFOLD;
}

/* -- Loads ---------------------------------------------------------------- */

/* Loads cannot be folded or passed on to CSE in general.
```

```
2031 ** Alias analysis is needed to check for forwarding opportunities.
2032 **
2033 ** Caveat: *all* loads must be listed here or they end up at CSE!
2034 */
2035
2036 LJFOLD(ALOAD any)
2037 LJFOLDX(lj_opt_fwd_aload)
2038
2039 /* From HREF fwd (see below). Must eliminate, not supported by fwd/backend. */
2040 LJFOLD(HLOAD KKPTR)
2041 LJFOLDF(kfold_hload_kkptr)
2042 {
2043   UNUSED(J);
2044   lua_assert(ir_kptr(fleft) == niltvg(J2G(J)));
2045   return TREF_NIL;
2046 }
2047
2048 LJFOLD(HLOAD any)
2049 LJFOLDX(lj_opt_fwd_hload)
2050
2051 LJFOLD(ULOAD any)
2052 LJFOLDX(lj_opt_fwd_uload)
2053
2054 LJFOLD(CALLL any IRCALL_lj_tab_len)
2055 LJFOLDX(lj_opt_fwd_tab_len)
2056
2057 /* Upvalue refs are really loads, but there are no corresponding stores.
2058 ** So CSE is ok for them, except for UREFO across a GC step (see below).
2059 ** If the referenced function is const, its upvalue addresses are const, too.
2060 ** This can be used to improve CSE by looking for the same address,
2061 ** even if the upvalues originate from a different function.
2062 */
2063 LJFOLD(UREFO KGC any)
2064 LJFOLD(UREFC KGC any)
2065 LJFOLDF(cse_uref)
2066 {
2067   if (LJ_LIKELY(J->flags & JIT_F_OPT_CSE)) {
2068     IRRef ref = J->chain[fins->o];
2069     GCfunc *fn = ir_kfunc(fleft);
2070     GCupval *uv = gco2uv(gcref(fn->l.uvptr[(fins->op2 >> 8)]));
2071     while (ref > 0) {
2072       IRIns *ir = IR(ref);
2073       if (irref_isk(ir->op1)) {
2074         GCfunc *fn2 = ir_kfunc(IR(ir->op1));
2075         if (gco2uv(gcref(fn2->l.uvptr[(ir->op2 >> 8)])) == uv) {
2076           if (fins->o == IR_UREFO && gcstep_barrier(J, ref))
2077             break;
2078           return ref;
2079         }
2080       }
2081       ref = ir->prev;
2082     }
2083   }
2084   return EMITFOLD;
2085 }
2086
2087 LJFOLD(HREFK any any)
2088 LJFOLDX(lj_opt_fwd_hrefk)
2089
2090 LJFOLD(HREF TNEW any)
2091 LJFOLDF(fwd_href_tnew)
2092 {
2093   if (lj_opt_fwd_href_nokey(J))
2094     return lj_ir_kkptr(J, niltvg(J2G(J)));
2095   return NEXTFOLD;
2096 }
2097
2098 LJFOLD(HREF TDUP KPRI)
2099 LJFOLD(HREF TDUP KGC)
2100 LJFOLD(HREF TDUP KNUM)
2101 LJFOLDF(fwd_href_tdup)
2102 {
2103   TValue keyv;
2104   lj_ir_kvalue(J->L, &keyv, fright);
2105   if (lj_tab_get(J->L, ir_ktab(IR(fleft->op1)), &keyv) == niltvg(J2G(J)) &&
2106       lj_opt_fwd_href_nokey(J))
```

```
2107      return lj_ir_kkptr(J, niltvg(J2G(J)));
2108    return NEXTFOLD;
2109  }
2110
2111  /* We can safely FOLD/CSE array/hash refs and field loads, since there
2112  ** are no corresponding stores. But we need to check for any NEWREF with
2113  ** an aliased table, as it may invalidate all of the pointers and fields.
2114  ** Only HREF needs the NEWREF check -- AREF and HREFK already depend on
2115  ** FLOADs. And NEWREF itself is treated like a store (see below).
2116  ** LREF is constant (per trace) since coroutine switches are not inlined.
2117  */
2118  LJFOLD(FLOAD TNEW IRFL_TAB_ASIZE)
2119  LJFOLDF(fload_tab_tnew_asize)
2120  {
2121    if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD) && lj_opt_fwd_tptr(J, fins->op1))
2122      return INTFOLD(fleft->op1);
2123    return NEXTFOLD;
2124  }
2125
2126  LJFOLD(FLOAD TNEW IRFL_TAB_HMASK)
2127  LJFOLDF(fload_tab_tnew_hmask)
2128  {
2129    if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD) && lj_opt_fwd_tptr(J, fins->op1))
2130      return INTFOLD((1 << fleft->op2)-1);
2131    return NEXTFOLD;
2132  }
2133
2134  LJFOLD(FLOAD TDUP IRFL_TAB_ASIZE)
2135  LJFOLDF(fload_tab_tdup_asize)
2136  {
2137    if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD) && lj_opt_fwd_tptr(J, fins->op1))
2138      return INTFOLD((int32_t)ir_ktab(IR(fleft->op1))->asize);
2139    return NEXTFOLD;
2140  }
2141
2142  LJFOLD(FLOAD TDUP IRFL_TAB_HMASK)
2143  LJFOLDF(fload_tab_tdup_hmask)
2144  {
2145    if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD) && lj_opt_fwd_tptr(J, fins->op1))
2146      return INTFOLD((int32_t)ir_ktab(IR(fleft->op1))->hmask);
2147    return NEXTFOLD;
2148  }
2149
2150  LJFOLD(HREF any any)
2151  LJFOLD(FLOAD any IRFL_TAB_ARRAY)
2152  LJFOLD(FLOAD any IRFL_TAB_NODE)
2153  LJFOLD(FLOAD any IRFL_TAB_ASIZE)
2154  LJFOLD(FLOAD any IRFL_TAB_HMASK)
2155  LJFOLDF(fload_tab_ah)
2156  {
2157    TRef tr = lj_opt_cse(J);
2158    return lj_opt_fwd_tptr(J, tref_ref(tr)) ? tr : EMITFOLD;
2159  }
2160
2161  /* Strings are immutable, so we can safely FOLD/CSE the related FLOAD. */
2162  LJFOLD(FLOAD KGC IRFL_STR_LEN)
2163  LJFOLDF(fload_str_len_kgc)
2164  {
2165    if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD))
2166      return INTFOLD((int32_t)ir_kstr(fleft)->len);
2167    return NEXTFOLD;
2168  }
2169
2170  LJFOLD(FLOAD SNEW IRFL_STR_LEN)
2171  LJFOLDF(fload_str_len_snew)
2172  {
2173    if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD)) {
2174      PHIBARRIER(fleft);
2175      return fleft->op2;
2176    }
2177    return NEXTFOLD;
2178  }
2179
2180  LJFOLD(FLOAD TOSTR IRFL_STR_LEN)
2181  LJFOLDF(fload_str_len_tostr)
2182  {
```

```
  if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD) && fleft->op2 == IRTOSTR_CHAR)
    return INTFOLD(1);
  return NEXTFOLD;
}

/* The C type ID of cdata objects is immutable. */
LJFOLD(FLOAD KGC IRFL_CDATA_CTYPEID)
LJFOLDF(fload_cdata_typeid_kgc)
{
  if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD))
    return INTFOLD((int32_t)ir_kcdata(fleft)->ctypeid);
  return NEXTFOLD;
}

/* Get the contents of immutable cdata objects. */
LJFOLD(FLOAD KGC IRFL_CDATA_PTR)
LJFOLD(FLOAD KGC IRFL_CDATA_INT)
LJFOLD(FLOAD KGC IRFL_CDATA_INT64)
LJFOLDF(fload_cdata_int64_kgc)
{
  if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD)) {
    void *p = cdataptr(ir_kcdata(fleft));
    if (irt_is64(fins->t))
      return INT64FOLD(*(uint64_t *)p);
    else
      return INTFOLD(*(int32_t *)p);
  }
  return NEXTFOLD;
}

LJFOLD(FLOAD CNEW IRFL_CDATA_CTYPEID)
LJFOLD(FLOAD CNEWI IRFL_CDATA_CTYPEID)
LJFOLDF(fload_cdata_typeid_cnew)
{
  if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD))
    return fleft->op1;  /* No PHI barrier needed. CNEW/CNEWI op1 is const. */
  return NEXTFOLD;
}

/* Pointer, int and int64 cdata objects are immutable. */
LJFOLD(FLOAD CNEWI IRFL_CDATA_PTR)
LJFOLD(FLOAD CNEWI IRFL_CDATA_INT)
LJFOLD(FLOAD CNEWI IRFL_CDATA_INT64)
LJFOLDF(fload_cdata_ptr_int64_cnew)
{
  if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD))
    return fleft->op2;  /* Fold even across PHI to avoid allocations. */
  return NEXTFOLD;
}

LJFOLD(FLOAD any IRFL_STR_LEN)
LJFOLD(FLOAD any IRFL_FUNC_ENV)
LJFOLD(FLOAD any IRFL_THREAD_ENV)
LJFOLD(FLOAD any IRFL_CDATA_CTYPEID)
LJFOLD(FLOAD any IRFL_CDATA_PTR)
LJFOLD(FLOAD any IRFL_CDATA_INT)
LJFOLD(FLOAD any IRFL_CDATA_INT64)
LJFOLD(VLOAD any any)  /* Vararg loads have no corresponding stores. */
LJFOLDX(lj_opt_cse)

/* All other field loads need alias analysis. */
LJFOLD(FLOAD any any)
LJFOLDX(lj_opt_fwd_fload)

/* This is for LOOP only. Recording handles SLOADs internally. */
LJFOLD(SLOAD any any)
LJFOLDF(fwd_sload)
{
  if ((fins->op2 & IRSLOAD_FRAME)) {
    TRef tr = lj_opt_cse(J);
    return tref_ref(tr) < J->chain[IR_RETF] ? EMITFOLD : tr;
  } else {
    lua_assert(J->slot[fins->op1] != 0);
    return J->slot[fins->op1];
  }
}
```

```
2259
2260 /* Only fold for KKPTR. The pointer _and_ the contents must be const. */
2261 LJFOLD(XLOAD KKPTR any)
2262 LJFOLDF(xload_kptr)
2263 {
2264   TRef tr = kfold_xload(J, fins, ir_kptr(fleft));
2265   return tr ? tr : NEXTFOLD;
2266 }
2267
2268 LJFOLD(XLOAD any any)
2269 LJFOLDX(lj_opt_fwd_xload)
2270
2271 /* -- Write barriers ------------------------------------------------------ */
2272
2273 /* Write barriers are amenable to CSE, but not across any incremental
2274 ** GC steps.
2275 **
2276 ** The same logic applies to open upvalue references, because a stack
2277 ** may be resized during a GC step (not the current stack, but maybe that
2278 ** of a coroutine).
2279 */
2280 LJFOLD(TBAR any)
2281 LJFOLD(OBAR any any)
2282 LJFOLD(UREFO any any)
2283 LJFOLDF(barrier_tab)
2284 {
2285   TRef tr = lj_opt_cse(J);
2286   if (gcstep_barrier(J, tref_ref(tr)))  /* CSE across GC step? */
2287     return EMITFOLD;  /* Raw emit. Assumes fins is left intact by CSE. */
2288   return tr;
2289 }
2290
2291 LJFOLD(TBAR TNEW)
2292 LJFOLD(TBAR TDUP)
2293 LJFOLDF(barrier_tnew_tdup)
2294 {
2295   /* New tables are always white and never need a barrier. */
2296   if (fins->op1 < J->chain[IR_LOOP])  /* Except across a GC step. */
2297     return NEXTFOLD;
2298   return DROPFOLD;
2299 }
2300
2301 /* -- Profiling ----------------------------------------------------------- */
2302
2303 LJFOLD(PROF any any)
2304 LJFOLDF(prof)
2305 {
2306   IRRef ref = J->chain[IR_PROF];
2307   if (ref+1 == J->cur.nins)  /* Drop neighbouring IR_PROF. */
2308     return ref;
2309   return EMITFOLD;
2310 }
2311
2312 /* -- Stores and allocations ---------------------------------------------- */
2313
2314 /* Stores and allocations cannot be folded or passed on to CSE in general.
2315 ** But some stores can be eliminated with dead-store elimination (DSE).
2316 **
2317 ** Caveat: *all* stores and allocs must be listed here or they end up at CSE!
2318 */
2319
2320 LJFOLD(ASTORE any any)
2321 LJFOLD(HSTORE any any)
2322 LJFOLDX(lj_opt_dse_ahstore)
2323
2324 LJFOLD(USTORE any any)
2325 LJFOLDX(lj_opt_dse_ustore)
2326
2327 LJFOLD(FSTORE any any)
2328 LJFOLDX(lj_opt_dse_fstore)
2329
2330 LJFOLD(XSTORE any any)
2331 LJFOLDX(lj_opt_dse_xstore)
2332
2333 LJFOLD(NEWREF any any)  /* Treated like a store. */
2334 LJFOLD(CALLA any any)
```

```
LJFOLD(CALLL any any)  /* Safeguard fallback. */
LJFOLD(CALLS any any)
LJFOLD(CALLXS any any)
LJFOLD(XBAR)
LJFOLD(RETF any any)  /* Modifies BASE. */
LJFOLD(TNEW any any)
LJFOLD(TDUP any)
LJFOLD(CNEW any any)
LJFOLD(XSNEW any any)
LJFOLD(BUFHDR any any)
LJFOLDX(lj_ir_emit)

/* ------------------------------------------------------------------------ */

/* Every entry in the generated hash table is a 32 bit pattern:
**
** xxxxxxxx iiiiiii lllllll rrrrrrrrrr
**
**   xxxxxxxx = 8 bit index into fold function table
**    iiiiiii = 7 bit folded instruction opcode
**    lllllll = 7 bit left instruction opcode
** rrrrrrrrrr = 8 bit right instruction opcode or 10 bits from literal field
*/

#include "lj_folddef.h"

/* ------------------------------------------------------------------------ */

/* Fold IR instruction. */
TRef LJ_FASTCALL lj_opt_fold(jit_State *J)
{
  uint32_t key, any;
  IRRef ref;

  if (LJ_UNLIKELY((J->flags & JIT_F_OPT_MASK) != JIT_F_OPT_DEFAULT)) {
    lua_assert(((JIT_F_OPT_FOLD|JIT_F_OPT_FWD|JIT_F_OPT_CSE|JIT_F_OPT_DSE) |
               JIT_F_OPT_DEFAULT) == JIT_F_OPT_DEFAULT);
    /* Folding disabled? Chain to CSE, but not for loads/stores/allocs. */
    if (!(J->flags & JIT_F_OPT_FOLD) && irm_kind(lj_ir_mode[fins->o]) == IRM_N)
      return lj_opt_cse(J);

    /* No FOLD, forwarding or CSE? Emit raw IR for loads, except for SLOAD. */
    if ((J->flags & (JIT_F_OPT_FOLD|JIT_F_OPT_FWD|JIT_F_OPT_CSE)) !=
                    (JIT_F_OPT_FOLD|JIT_F_OPT_FWD|JIT_F_OPT_CSE) &&
        irm_kind(lj_ir_mode[fins->o]) == IRM_L && fins->o != IR_SLOAD)
      return lj_ir_emit(J);

    /* No FOLD or DSE? Emit raw IR for stores. */
    if ((J->flags & (JIT_F_OPT_FOLD|JIT_F_OPT_DSE)) !=
                    (JIT_F_OPT_FOLD|JIT_F_OPT_DSE) &&
        irm_kind(lj_ir_mode[fins->o]) == IRM_S)
      return lj_ir_emit(J);
  }

  /* Fold engine start/retry point. */
retry:
  /* Construct key from opcode and operand opcodes (unless literal/none). */
  key = ((uint32_t)fins->o << 17);
  if (fins->op1 >= J->cur.nk) {
    key += (uint32_t)IR(fins->op1)->o << 10;
    *fleft = *IR(fins->op1);
  }
  if (fins->op2 >= J->cur.nk) {
    key += (uint32_t)IR(fins->op2)->o;
    *fright = *IR(fins->op2);
  } else {
    key += (fins->op2 & 0x3ffu);  /* Literal mask. Must include IRCONV_*MASK. */
  }

  /* Check for a match in order from most specific to least specific. */
  any = 0;
  for (;;) {
    uint32_t k = key | (any & 0x1ffff);
    uint32_t h = fold_hashkey(k);
    uint32_t fh = fold_hash[h];  /* Lookup key in semi-perfect hash table. */
    if ((fh & 0xffffff) == k || (fh = fold_hash[h+1], (fh & 0xffffff) == k)) {
```

```
      ref = (IRRef)tref_ref(fold_func[fh >> 24](J));
      if (ref != NEXTFOLD)
        break;
    }
    if (any == 0xfffff)  /* Exhausted folding. Pass on to CSE. */
      return lj_opt_cse(J);
    any = (any | (any >> 10)) ^ 0xffc00;
  }

  /* Return value processing, ordered by frequency. */
  if (LJ_LIKELY(ref >= MAX_FOLD))
    return TREF(ref, irt_t(IR(ref)->t));
  if (ref == RETRYFOLD)
    goto retry;
  if (ref == KINTFOLD)
    return lj_ir_kint(J, fins->i);
  if (ref == FAILFOLD)
    lj_trace_err(J, LJ_TRERR_GFAIL);
  lua_assert(ref == DROPFOLD);
  return REF_DROP;
}

/* -- Common-Subexpression Elimination ------------------------------------ */

/* CSE an IR instruction. This is very fast due to the skip-list chains. */
TRef LJ_FASTCALL lj_opt_cse(jit_State *J)
{
  /* Avoid narrow to wide store-to-load forwarding stall */
  IRRef2 op12 = (IRRef2)fins->op1 + ((IRRef2)fins->op2 << 16);
  IROp op = fins->o;
  if (LJ_LIKELY(J->flags & JIT_F_OPT_CSE)) {
    /* Limited search for same operands in per-opcode chain. */
    IRRef ref = J->chain[op];
    IRRef lim = fins->op1;
    if (fins->op2 > lim) lim = fins->op2;  /* Relies on lit < REF_BIAS. */
    while (ref > lim) {
      if (IR(ref)->op12 == op12)
        return TREF(ref, irt_t(IR(ref)->t));  /* Common subexpression found. */
      ref = IR(ref)->prev;
    }
  }
  /* Otherwise emit IR (inlined for speed). */
  {
    IRRef ref = lj_ir_nextins(J);
    IRIns *ir = IR(ref);
    ir->prev = J->chain[op];
    ir->op12 = op12;
    J->chain[op] = (IRRef1)ref;
    ir->o = fins->o;
    J->guardemit.irt |= fins->t.irt;
    return TREF(ref, irt_t((ir->t = fins->t)));
  }
}

/* CSE with explicit search limit. */
TRef LJ_FASTCALL lj_opt_cselim(jit_State *J, IRRef lim)
{
  IRRef ref = J->chain[fins->o];
  IRRef2 op12 = (IRRef2)fins->op1 + ((IRRef2)fins->op2 << 16);
  while (ref > lim) {
    if (IR(ref)->op12 == op12)
      return ref;
    ref = IR(ref)->prev;
  }
  return lj_ir_emit(J);
}

/* ------------------------------------------------------------------------ */

#undef IR
#undef fins
#undef fleft
#undef fright
#undef knumleft
#undef knumright
#undef emitir
```

```
2487
2488 #endif
```

One Level Up Top Level

src/lj_opt_narrow.c - luajit-2.0-src

Data types defined

- NarrowConv
- NarrowConv
- NarrowIns

Functions defined

- lj_opt_narrow_arith
- lj_opt_narrow_cindex
- lj_opt_narrow_convert
- lj_opt_narrow_forl
- lj_opt_narrow_index
- lj_opt_narrow_mod
- lj_opt_narrow_pow
- lj_opt_narrow_tobit
- lj_opt_narrow_toint
- lj_opt_narrow_unm
- narrow_bpc_get
- narrow_bpc_set
- narrow_conv_backprop
- narrow_conv_emit
- narrow_forl
- narrow_stripov
- narrow_stripov_backprop
- numisint

Macros defined

- IR
- IR
- LUA_CORE
- NARROWINS
- NARROW_MAX_BACKPROP
- NARROW_MAX_STACK

- emitir
- emitir
- emitir_raw
- emitir_raw
- fins
- fins
- lj_opt_narrow_c
- narrow_op
- narrow_ref

Source code

```
/*
** NARROW: Narrowing of numbers to integers (double to int32_t).
** STRIPOV: Stripping of overflow checks.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_opt_narrow_c
#define LUA_CORE

 #include "lj_obj.h"

 #if LJ_HASJIT

 #include "lj_bc.h"
 #include "lj_ir.h"
 #include "lj_jit.h"
 #include "lj_iropt.h"
 #include "lj_trace.h"
 #include "lj_vm.h"
 #include "lj_strscan.h"

 /* Rationale for narrowing optimizations:
 **
 ** Lua has only a single number type and this is a FP double by default.
 ** Narrowing doubles to integers does not pay off for the interpreter on a
 ** current-generation x86/x64 machine. Most FP operations need the same
 ** amount of execution resources as their integer counterparts, except
 ** with slightly longer latencies. Longer latencies are a non-issue for
 ** the interpreter, since they are usually hidden by other overhead.
 **
 ** The total CPU execution bandwidth is the sum of the bandwidth of the FP
 ** and the integer units, because they execute in parallel. The FP units
 ** have an equal or higher bandwidth than the integer units. Not using
 ** them means losing execution bandwidth. Moving work away from them to
 ** the already quite busy integer units is a losing proposition.
 **
 ** The situation for JIT-compiled code is a bit different: the higher code
 ** density makes the extra latencies much more visible. Tight loops expose
 ** the latencies for updating the induction variables. Array indexing
 ** requires narrowing conversions with high latencies and additional
 ** guards (to check that the index is really an integer). And many common
 ** optimizations only work on integers.
 **
 ** One solution would be speculative, eager narrowing of all number loads.
 ** This causes many problems, like losing -0 or the need to resolve type
 ** mismatches between traces. It also effectively forces the integer type
 ** to have overflow-checking semantics. This impedes many basic
 ** optimizations and requires adding overflow checks to all integer
 ** arithmetic operations (whereas FP arithmetics can do without).
 **
 ** Always replacing an FP op with an integer op plus an overflow check is
 ** counter-productive on a current-generation super-scalar CPU. Although
```

```
** the overflow check branches are highly predictable, they will clog the
** execution port for the branch unit and tie up reorder buffers. This is
** turning a pure data-flow dependency into a different data-flow
** dependency (with slightly lower latency) *plus* a control dependency.
** In general, you don't want to do this since latencies due to data-flow
** dependencies can be well hidden by out-of-order execution.
**
** A better solution is to keep all numbers as FP values and only narrow
** when it's beneficial to do so. LuaJIT uses predictive narrowing for
** induction variables and demand-driven narrowing for index expressions,
** integer arguments and bit operations. Additionally it can eliminate or
** hoist most of the resulting overflow checks. Regular arithmetic
** computations are never narrowed to integers.
**
** The integer type in the IR has convenient wrap-around semantics and
** ignores overflow. Extra operations have been added for
** overflow-checking arithmetic (ADDOV/SUBOV) instead of an extra type.
** Apart from reducing overall complexity of the compiler, this also
** nicely solves the problem where you want to apply algebraic
** simplifications to ADD, but not to ADDOV. And the x86/x64 assembler can
** use lea instead of an add for integer ADD, but not for ADDOV (lea does
** not affect the flags, but it helps to avoid register moves).
**
**
** All of the above has to be reconsidered for architectures with slow FP
** operations or without a hardware FPU. The dual-number mode of LuaJIT
** addresses this issue. Arithmetic operations are performed on integers
** as far as possible and overflow checks are added as needed.
**
** This implies that narrowing for integer arguments and bit operations
** should also strip overflow checks, e.g. replace ADDOV with ADD. The
** original overflow guards are weak and can be eliminated by DCE, if
** there's no other use.
**
** A slight twist is that it's usually beneficial to use overflow-checked
** integer arithmetics if all inputs are already integers. This is the only
** change that affects the single-number mode, too.
*/

/* Some local macros to save typing. Undef'd at the end. */
#define IR(ref)                         (&J->cur.ir[(ref)])
#define fins                        (&J->fold.ins)

/* Pass IR on to next optimization in chain (FOLD). */
#define emitir(ot, a, b)        (lj_ir_set(J, (ot), (a), (b)), lj_opt_fold(J))

#define emitir_raw(ot, a, b)        (lj_ir_set(J, (ot), (a), (b)), lj_ir_emit(J))

 /* -- Elimination of narrowing type conversions ------------------------- */

 /* Narrowing of index expressions and bit operations is demand-driven. The
 ** trace recorder emits a narrowing type conversion (CONV.int.num or TOBIT)
 ** in all of these cases (e.g. array indexing or string indexing). FOLD
 ** already takes care of eliminating simple redundant conversions like
 ** CONV.int.num(CONV.num.int(x)) ==> x.
 **
 ** But the surrounding code is FP-heavy and arithmetic operations are
 ** performed on FP numbers (for the single-number mode). Consider a common
 ** example such as 'x=t[i+1]', with 'i' already an integer (due to induction
 ** variable narrowing). The index expression would be recorded as
 **   CONV.int.num(ADD(CONV.num.int(i), 1))
 ** which is clearly suboptimal.
 **
 ** One can do better by recursively backpropagating the narrowing type
 ** conversion across FP arithmetic operations. This turns FP ops into
 ** their corresponding integer counterparts. Depending on the semantics of
 ** the conversion they also need to check for overflow. Currently only ADD
 ** and SUB are supported.
 **
 ** The above example can be rewritten as
 **   ADDOV(CONV.int.num(CONV.num.int(i)), 1)
 ** and then into ADDOV(i, 1) after folding of the conversions. The original
 ** FP ops remain in the IR and are eliminated by DCE since all references to
 ** them are gone.
 **
 ** [In dual-number mode the trace recorder already emits ADDOV etc., but
```

```
** this can be further reduced. See below.]
**
** Special care has to be taken to avoid narrowing across an operation
** which is potentially operating on non-integral operands. One obvious
** case is when an expression contains a non-integral constant, but ends
** up as an integer index at runtime (like t[x+1.5] with x=0.5).
**
** Operations with two non-constant operands illustrate a similar problem
** (like t[a+b] with a=1.5 and b=2.5). Backpropagation has to stop there,
** unless it can be proven that either operand is integral (e.g. by CSEing
** a previous conversion). As a not-so-obvious corollary this logic also
** applies for a whole expression tree (e.g. t[(a+1)+(b+1)]).
**
** Correctness of the transformation is guaranteed by avoiding to expand
** the tree by adding more conversions than the one we would need to emit
** if not backpropagating. TOBIT employs a more optimistic rule, because
** the conversion has special semantics, designed to make the life of the
** compiler writer easier. ;-)
**
** Using on-the-fly backpropagation of an expression tree doesn't work
** because it's unknown whether the transform is correct until the end.
** This either requires IR rollback and cache invalidation for every
** subtree or a two-pass algorithm. The former didn't work out too well,
** so the code now combines a recursive collector with a stack-based
** emitter.
**
** [A recursive backpropagation algorithm with backtracking, employing
** skip-list lookup and round-robin caching, emitting stack operations
** on-the-fly for a stack-based interpreter -- and all of that in a meager
** kilobyte? Yep, compilers are a great treasure chest. Throw away your
** textbooks and read the codebase of a compiler today!]
**
** There's another optimization opportunity for array indexing: it's
** always accompanied by an array bounds-check. The outermost overflow
** check may be delegated to the ABC operation. This works because ABC is
** an unsigned comparison and wrap-around due to overflow creates negative
** numbers.
**
** But this optimization is only valid for constants that cannot overflow
** an int32_t into the range of valid array indexes [0..2^27+1). A check
** for +-2^30 is safe since -2^31 - 2^30 wraps to 2^30 and 2^31-1 + 2^30
** wraps to -2^30-1.
**
** It's also good enough in practice, since e.g. t[i+1] or t[i-10] are
** quite common. So the above example finally ends up as ADD(i, 1)!
**
** Later on, the assembler is able to fuse the whole array reference and
** the ADD into the memory operands of loads and other instructions. This
** is why LuaJIT is able to generate very pretty (and fast) machine code
** for array indexing. And that, my dear, concludes another story about
** one of the hidden secrets of LuaJIT ...
*/

/* Maximum backpropagation depth and maximum stack size. */
#define NARROW_MAX_BACKPROP        100
#define NARROW_MAX_STACK        256

/* The stack machine has a 32 bit instruction format: [IROpT | IRRef1]
** The lower 16 bits hold a reference (or 0). The upper 16 bits hold
** the IR opcode + type or one of the following special opcodes:
*/
enum {
  NARROW_REF,                /* Push ref. */
  NARROW_CONV,                /* Push conversion of ref. */
  NARROW_SEXT,                /* Push sign-extension of ref. */
  NARROW_INT                /* Push KINT ref. The next code holds an int32_t. */
};

typedef uint32_t NarrowIns;

#define NARROWINS(op, ref)        (((op) << 16) + (ref))
#define narrow_op(ins)                ((IROpT)((ins) >> 16))
#define narrow_ref(ins)                ((IRRef1)(ins))

/* Context used for narrowing of type conversions. */
typedef struct NarrowConv {
```

```
  jit_State *J;                /* JIT compiler state. */
  NarrowIns *sp;        /* Current stack pointer. */
  NarrowIns *maxsp;        /* Maximum stack pointer minus redzone. */
  int lim;                /* Limit on the number of emitted conversions. */
  IRRef mode;                /* Conversion mode (IRCONV_*). */
  IRType t;                /* Destination type: IRT_INT or IRT_I64. */
  NarrowIns stack[NARROW_MAX_STACK];  /* Stack holding stack-machine code. */
} NarrowConv;

/* Lookup a reference in the backpropagation cache. */
static BPropEntry *narrow_bpc_get(jit_State *J, IRRef1 key, IRRef mode)
{
  ptrdiff_t i;
  for (i = 0; i < BPROP_SLOTS; i++) {
    BPropEntry *bp = &J->bpropcache[i];
    /* Stronger checks are ok, too. */
    if (bp->key == key && bp->mode >= mode &&
        ((bp->mode ^ mode) & IRCONV_MODEMASK) == 0)
      return bp;
  }
  return NULL;
}

/* Add an entry to the backpropagation cache. */
static void narrow_bpc_set(jit_State *J, IRRef1 key, IRRef1 val, IRRef mode)
{
  uint32_t slot = J->bpropslot;
  BPropEntry *bp = &J->bpropcache[slot];
  J->bpropslot = (slot + 1) & (BPROP_SLOTS-1);
  bp->key = key;
  bp->val = val;
  bp->mode = mode;
}

/* Backpropagate overflow stripping. */
static void narrow_stripov_backprop(NarrowConv *nc, IRRef ref, int depth)
{
  jit_State *J = nc->J;
  IRIns *ir = IR(ref);
  if (ir->o == IR_ADDOV || ir->o == IR_SUBOV ||
      (ir->o == IR_MULOV && (nc->mode & IRCONV_CONVMASK) == IRCONV_ANY)) {
    BPropEntry *bp = narrow_bpc_get(nc->J, ref, IRCONV_TOBIT);
    if (bp) {
      ref = bp->val;
    } else if (++depth < NARROW_MAX_BACKPROP && nc->sp < nc->maxsp) {
      narrow_stripov_backprop(nc, ir->op1, depth);
      narrow_stripov_backprop(nc, ir->op2, depth);
      *nc->sp++ = NARROWINS(IRT(ir->o - IR_ADDOV + IR_ADD, IRT_INT), ref);
      return;
    }
  }
  *nc->sp++ = NARROWINS(NARROW_REF, ref);
}

/* Backpropagate narrowing conversion. Return number of needed conversions. */
static int narrow_conv_backprop(NarrowConv *nc, IRRef ref, int depth)
{
  jit_State *J = nc->J;
  IRIns *ir = IR(ref);
  IRRef cref;

  /* Check the easy cases first. */
  if (ir->o == IR_CONV && (ir->op2 & IRCONV_SRCMASK) == IRT_INT) {
    if ((nc->mode & IRCONV_CONVMASK) <= IRCONV_ANY)
      narrow_stripov_backprop(nc, ir->op1, depth+1);
    else
      *nc->sp++ = NARROWINS(NARROW_REF, ir->op1);  /* Undo conversion. */
    if (nc->t == IRT_I64)
      *nc->sp++ = NARROWINS(NARROW_SEXT, 0);  /* Sign-extend integer. */
    return 0;
  } else if (ir->o == IR_KNUM) {  /* Narrow FP constant. */
    lua_Number n = ir_knum(ir)->n;
    if ((nc->mode & IRCONV_CONVMASK) == IRCONV_TOBIT) {
      /* Allows a wider range of constants. */
      int64_t k64 = (int64_t)n;
      if (n == (lua_Number)k64) {  /* Only if const doesn't lose precision. */
```

```
        *nc->sp++ = NARROWINS(NARROW_INT, 0);
        *nc->sp++ = (NarrowIns)k64;  /* But always truncate to 32 bits. */
        return 0;
      }
    } else {
      int32_t k = lj_num2int(n);
      /* Only if constant is a small integer. */
      if (checki16(k) && n == (lua_Number)k) {
        *nc->sp++ = NARROWINS(NARROW_INT, 0);
        *nc->sp++ = (NarrowIns)k;
        return 0;
      }
    }
    return 10;  /* Never narrow other FP constants (this is rare). */
  }

  /* Try to CSE the conversion. Stronger checks are ok, too. */
  cref = J->chain[fins->o];
  while (cref > ref) {
    IRIns *cr = IR(cref);
    if (cr->op1 == ref &&
        (fins->o == IR_TOBIT ||
         ((cr->op2 & IRCONV_MODEMASK) == (nc->mode & IRCONV_MODEMASK) &&
          irt_isguard(cr->t) >= irt_isguard(fins->t)))) {
      *nc->sp++ = NARROWINS(NARROW_REF, cref);
      return 0;  /* Already there, no additional conversion needed. */
    }
    cref = cr->prev;
  }

  /* Backpropagate across ADD/SUB. */
  if (ir->o == IR_ADD || ir->o == IR_SUB) {
    /* Try cache lookup first. */
    IRRef mode = nc->mode;
    BPropEntry *bp;
    /* Inner conversions need a stronger check. */
    if ((mode & IRCONV_CONVMASK) == IRCONV_INDEX && depth > 0)
      mode += IRCONV_CHECK-IRCONV_INDEX;
    bp = narrow_bpc_get(nc->J, (IRRef1)ref, mode);
    if (bp) {
      *nc->sp++ = NARROWINS(NARROW_REF, bp->val);
      return 0;
    } else if (nc->t == IRT_I64) {
      /* Try sign-extending from an existing (checked) conversion to int. */
      mode = (IRT_INT<<5)|IRT_NUM|IRCONV_INDEX;
      bp = narrow_bpc_get(nc->J, (IRRef1)ref, mode);
      if (bp) {
        *nc->sp++ = NARROWINS(NARROW_REF, bp->val);
        *nc->sp++ = NARROWINS(NARROW_SEXT, 0);
        return 0;
      }
    }
    if (++depth < NARROW_MAX_BACKPROP && nc->sp < nc->maxsp) {
      NarrowIns *savesp = nc->sp;
      int count = narrow_conv_backprop(nc, ir->op1, depth);
      count += narrow_conv_backprop(nc, ir->op2, depth);
      if (count <= nc->lim) {  /* Limit total number of conversions. */
        *nc->sp++ = NARROWINS(IRT(ir->o, nc->t), ref);
        return count;
      }
      nc->sp = savesp;  /* Too many conversions, need to backtrack. */
    }
  }

  /* Otherwise add a conversion. */
  *nc->sp++ = NARROWINS(NARROW_CONV, ref);
  return 1;
}

/* Emit the conversions collected during backpropagation. */
static IRRef narrow_conv_emit(jit_State *J, NarrowConv *nc)
{
  /* The fins fields must be saved now -- emitir() overwrites them. */
  IROpT guardot = irt_isguard(fins->t) ? IRTG(IR_ADDOV-IR_ADD, 0) : 0;
  IROpT convot = fins->ot;
  IRRef1 convop2 = fins->op2;
```

```
  NarrowIns *next = nc->stack;  /* List of instructions from backpropagation. */
  NarrowIns *last = nc->sp;
  NarrowIns *sp = nc->stack;  /* Recycle the stack to store operands. */
  while (next < last) {  /* Simple stack machine to process the ins. list. */
    NarrowIns ref = *next++;
    IROpT op = narrow_op(ref);
    if (op == NARROW_REF) {
      *sp++ = ref;
    } else if (op == NARROW_CONV) {
      *sp++ = emitir_raw(convot, ref, convop2);  /* Raw emit avoids a loop. */
    } else if (op == NARROW_SEXT) {
      lua_assert(sp >= nc->stack+1);
      sp[-1] = emitir(IRT(IR_CONV, IRT_I64), sp[-1],
                      (IRT_I64<<5)|IRT_INT|IRCONV_SEXT);
    } else if (op == NARROW_INT) {
      lua_assert(next < last);
      *sp++ = nc->t == IRT_I64 ?
              lj_ir_kint64(J, (int64_t)(int32_t)*next++) :
              lj_ir_kint(J, *next++);
    } else {  /* Regular IROpT. Pops two operands and pushes one result. */
      IRRef mode = nc->mode;
      lua_assert(sp >= nc->stack+2);
      sp--;
      /* Omit some overflow checks for array indexing. See comments above. */
      if ((mode & IRCONV_CONVMASK) == IRCONV_INDEX) {
        if (next == last && irref_isk(narrow_ref(sp[0])) &&
          (uint32_t)IR(narrow_ref(sp[0]))->i + 0x40000000u < 0x80000000u)
          guardot = 0;
        else  /* Otherwise cache a stronger check. */
          mode += IRCONV_CHECK-IRCONV_INDEX;
      }
      sp[-1] = emitir(op+guardot, sp[-1], sp[0]);
      /* Add to cache. */
      if (narrow_ref(ref))
        narrow_bpc_set(J, narrow_ref(ref), narrow_ref(sp[-1]), mode);
    }
  }
  lua_assert(sp == nc->stack+1);
  return nc->stack[0];
}

/* Narrow a type conversion of an arithmetic operation. */
TRef LJ_FASTCALL lj_opt_narrow_convert(jit_State *J)
{
  if ((J->flags & JIT_F_OPT_NARROW)) {
    NarrowConv nc;
    nc.J = J;
    nc.sp = nc.stack;
    nc.maxsp = &nc.stack[NARROW_MAX_STACK-4];
    nc.t = irt_type(fins->t);
    if (fins->o == IR_TOBIT) {
      nc.mode = IRCONV_TOBIT;  /* Used only in the backpropagation cache. */
      nc.lim = 2;  /* TOBIT can use a more optimistic rule. */
    } else {
      nc.mode = fins->op2;
      nc.lim = 1;
    }
    if (narrow_conv_backprop(&nc, fins->op1, 0) <= nc.lim)
      return narrow_conv_emit(J, &nc);
  }
  return NEXTFOLD;
}

/* -- Narrowing of implicit conversions ----------------------------------- */

/* Recursively strip overflow checks. */
static TRef narrow_stripov(jit_State *J, TRef tr, int lastop, IRRef mode)
{
  IRRef ref = tref_ref(tr);
  IRIns *ir = IR(ref);
  int op = ir->o;
  if (op >= IR_ADDOV && op <= lastop) {
    BPropEntry *bp = narrow_bpc_get(J, ref, mode);
    if (bp) {
      return TREF(bp->val, irt_t(IR(bp->val)->t));
    } else {
```

```
      IRRef op1 = ir->op1, op2 = ir->op2;  /* The IR may be reallocated. */
      op1 = narrow_stripov(J, op1, lastop, mode);
      op2 = narrow_stripov(J, op2, lastop, mode);
      tr = emitir(IRT(op - IR_ADDOV + IR_ADD,
                      ((mode & IRCONV_DSTMASK) >> IRCONV_DSH)), op1, op2);
      narrow_bpc_set(J, ref, tref_ref(tr), mode);
    }
  } else if (LJ_64 && (mode & IRCONV_SEXT) && !irt_is64(ir->t)) {
    tr = emitir(IRT(IR_CONV, IRT_INTP), tr, mode);
  }
  return tr;
}

/* Narrow array index. */
TRef LJ_FASTCALL lj_opt_narrow_index(jit_State *J, TRef tr)
{
  IRIns *ir;
  lua_assert(tref_isnumber(tr));
  if (tref_isnum(tr))  /* Conversion may be narrowed, too. See above. */
    return emitir(IRTGI(IR_CONV), tr, IRCONV_INT_NUM|IRCONV_INDEX);
  /* Omit some overflow checks for array indexing. See comments above. */
  ir = IR(tref_ref(tr));
  if ((ir->o == IR_ADDOV || ir->o == IR_SUBOV) && irref_isk(ir->op2) &&
      (uint32_t)IR(ir->op2)->i + 0x40000000u < 0x80000000u)
    return emitir(IRTI(ir->o - IR_ADDOV + IR_ADD), ir->op1, ir->op2);
  return tr;
}

/* Narrow conversion to integer operand (overflow undefined). */
TRef LJ_FASTCALL lj_opt_narrow_toint(jit_State *J, TRef tr)
{
  if (tref_isstr(tr))
    tr = emitir(IRTG(IR_STRTO, IRT_NUM), tr, 0);
  if (tref_isnum(tr))  /* Conversion may be narrowed, too. See above. */
    return emitir(IRTI(IR_CONV), tr, IRCONV_INT_NUM|IRCONV_ANY);
  if (!tref_isinteger(tr))
    lj_trace_err(J, LJ_TRERR_BADTYPE);
  /*
  ** Undefined overflow semantics allow stripping of ADDOV, SUBOV and MULOV.
  ** Use IRCONV_TOBIT for the cache entries, since the semantics are the same.
  */
  return narrow_stripov(J, tr, IR_MULOV, (IRT_INT<<5)|IRT_INT|IRCONV_TOBIT);
}

/* Narrow conversion to bitop operand (overflow wrapped). */
TRef LJ_FASTCALL lj_opt_narrow_tobit(jit_State *J, TRef tr)
{
  if (tref_isstr(tr))
    tr = emitir(IRTG(IR_STRTO, IRT_NUM), tr, 0);
  if (tref_isnum(tr))  /* Conversion may be narrowed, too. See above. */
    return emitir(IRTI(IR_TOBIT), tr, lj_ir_knum_tobit(J));
  if (!tref_isinteger(tr))
    lj_trace_err(J, LJ_TRERR_BADTYPE);
  /*
  ** Wrapped overflow semantics allow stripping of ADDOV and SUBOV.
  ** MULOV cannot be stripped due to precision widening.
  */
  return narrow_stripov(J, tr, IR_SUBOV, (IRT_INT<<5)|IRT_INT|IRCONV_TOBIT);
}

#if LJ_HASFFI
/* Narrow C array index (overflow undefined). */
TRef LJ_FASTCALL lj_opt_narrow_cindex(jit_State *J, TRef tr)
{
  lua_assert(tref_isnumber(tr));
  if (tref_isnum(tr))
    return emitir(IRT(IR_CONV, IRT_INTP), tr, (IRT_INTP<<5)|IRT_NUM|IRCONV_ANY);
  /* Undefined overflow semantics allow stripping of ADDOV, SUBOV and MULOV. */
  return narrow_stripov(J, tr, IR_MULOV,
                        LJ_64 ? ((IRT_INTP<<5)|IRT_INT|IRCONV_SEXT) :
                                ((IRT_INTP<<5)|IRT_INT|IRCONV_TOBIT));
}
#endif

/* -- Narrowing of arithmetic operators ---------------------------------- */
```

```c
/* Check whether a number fits into an int32_t (-0 is ok, too). */
static int numisint(lua_Number n)
{
  return (n == (lua_Number)lj_num2int(n));
}

/* Narrowing of arithmetic operations. */
TRef lj_opt_narrow_arith(jit_State *J, TRef rb, TRef rc,
                         TValue *vb, TValue *vc, IROp op)
{
  if (tref_isstr(rb)) {
    rb = emitir(IRTG(IR_STRTO, IRT_NUM), rb, 0);
    lj_strscan_num(strV(vb), vb);
  }
  if (tref_isstr(rc)) {
    rc = emitir(IRTG(IR_STRTO, IRT_NUM), rc, 0);
    lj_strscan_num(strV(vc), vc);
  }
  /* Must not narrow MUL in non-DUALNUM variant, because it loses -0. */
  if ((op >= IR_ADD && op <= (LJ_DUALNUM ? IR_MUL : IR_SUB)) &&
      tref_isinteger(rb) && tref_isinteger(rc) &&
      numisint(lj_vm_foldarith(numberVnum(vb), numberVnum(vc),
                               (int)op - (int)IR_ADD)))
    return emitir(IRTGI((int)op - (int)IR_ADD + (int)IR_ADDOV), rb, rc);
  if (!tref_isnum(rb)) rb = emitir(IRTN(IR_CONV), rb, IRCONV_NUM_INT);
  if (!tref_isnum(rc)) rc = emitir(IRTN(IR_CONV), rc, IRCONV_NUM_INT);
  return emitir(IRTN(op), rb, rc);
}

/* Narrowing of unary minus operator. */
TRef lj_opt_narrow_unm(jit_State *J, TRef rc, TValue *vc)
{
  if (tref_isstr(rc)) {
    rc = emitir(IRTG(IR_STRTO, IRT_NUM), rc, 0);
    lj_strscan_num(strV(vc), vc);
  }
  if (tref_isinteger(rc)) {
    if ((uint32_t)numberVint(vc) != 0x80000000u)
      return emitir(IRTGI(IR_SUBOV), lj_ir_kint(J, 0), rc);
    rc = emitir(IRTN(IR_CONV), rc, IRCONV_NUM_INT);
  }
  return emitir(IRTN(IR_NEG), rc, lj_ir_knum_neg(J));
}

/* Narrowing of modulo operator. */
TRef lj_opt_narrow_mod(jit_State *J, TRef rb, TRef rc, TValue *vc)
{
  TRef tmp;
  if (tvisstr(vc) && !lj_strscan_num(strV(vc), vc))
    lj_trace_err(J, LJ_TRERR_BADTYPE);
  if ((LJ_DUALNUM || (J->flags & JIT_F_OPT_NARROW)) &&
      tref_isinteger(rb) && tref_isinteger(rc) &&
      (tvisint(vc) ? intV(vc) != 0 : !tviszero(vc))) {
    emitir(IRTGI(IR_NE), rc, lj_ir_kint(J, 0));
    return emitir(IRTI(IR_MOD), rb, rc);
  }
  /* b % c ==> b - floor(b/c)*c */
  rb = lj_ir_tonum(J, rb);
  rc = lj_ir_tonum(J, rc);
  tmp = emitir(IRTN(IR_DIV), rb, rc);
  tmp = emitir(IRTN(IR_FPMATH), tmp, IRFPM_FLOOR);
  tmp = emitir(IRTN(IR_MUL), tmp, rc);
  return emitir(IRTN(IR_SUB), rb, tmp);
}

/* Narrowing of power operator or math.pow. */
TRef lj_opt_narrow_pow(jit_State *J, TRef rb, TRef rc, TValue *vc)
{
  if (tvisstr(vc) && !lj_strscan_num(strV(vc), vc))
    lj_trace_err(J, LJ_TRERR_BADTYPE);
  /* Narrowing must be unconditional to preserve (-x)^i semantics. */
  if (tvisint(vc) || numisint(numV(vc))) {
    int checkrange = 0;
    /* Split pow is faster for bigger exponents. But do this only for (+k)^i. */
    if (tref_isk(rb) && (int32_t)ir_knum(IR(tref_ref(rb)))->u32.hi >= 0) {
      int32_t k = numberVint(vc);
```

```
      if (!(k >= -65536 && k <= 65536)) goto split_pow;
      checkrange = 1;
    }
    if (!tref_isinteger(rc)) {
      if (tref_isstr(rc))
        rc = emitir(IRTG(IR_STRTO, IRT_NUM), rc, 0);
      /* Guarded conversion to integer! */
      rc = emitir(IRTGI(IR_CONV), rc, IRCONV_INT_NUM|IRCONV_CHECK);
    }
    if (checkrange && !tref_isk(rc)) {  /* Range guard: -65536 <= i <= 65536 */
      TRef tmp = emitir(IRTI(IR_ADD), rc, lj_ir_kint(J, 65536));
      emitir(IRTGI(IR_ULE), tmp, lj_ir_kint(J, 2*65536));
    }
    return emitir(IRTN(IR_POW), rb, rc);
  }
split_pow:
  /* FOLD covers most cases, but some are easier to do here. */
  if (tref_isk(rb) && tvispone(ir_knum(IR(tref_ref(rb)))))
    return rb;  /* 1 ^ x ==> 1 */
  rc = lj_ir_tonum(J, rc);
  if (tref_isk(rc) && ir_knum(IR(tref_ref(rc)))->n == 0.5)
    return emitir(IRTN(IR_FPMATH), rb, IRFPM_SQRT);  /* x ^ 0.5 ==> sqrt(x) */
  /* Split up b^c into exp2(c*log2(b)). Assembler may rejoin later. */
  rb = emitir(IRTN(IR_FPMATH), rb, IRFPM_LOG2);
  rc = emitir(IRTN(IR_MUL), rb, rc);
  return emitir(IRTN(IR_FPMATH), rc, IRFPM_EXP2);
}

/* -- Predictive narrowing of induction variables ----------------------- */

/* Narrow a single runtime value. */
static int narrow_forl(jit_State *J, cTValue *o)
{
  if (tvisint(o)) return 1;
  if (LJ_DUALNUM || (J->flags & JIT_F_OPT_NARROW)) return numisint(numV(o));
  return 0;
}

/* Narrow the FORL index type by looking at the runtime values. */
IRType lj_opt_narrow_forl(jit_State *J, cTValue *tv)
{
  lua_assert(tvisnumber(&tv[FORL_IDX]) &&
             tvisnumber(&tv[FORL_STOP]) &&
             tvisnumber(&tv[FORL_STEP]));
  /* Narrow only if the runtime values of start/stop/step are all integers. */
  if (narrow_forl(J, &tv[FORL_IDX]) &&
      narrow_forl(J, &tv[FORL_STOP]) &&
      narrow_forl(J, &tv[FORL_STEP])) {
    /* And if the loop index can't possibly overflow. */
    lua_Number step = numberVnum(&tv[FORL_STEP]);
    lua_Number sum = numberVnum(&tv[FORL_STOP]) + step;
    if (0 <= step ? (sum <= 2147483647.0) : (sum >= -2147483648.0))
      return IRT_INT;
  }
  return IRT_NUM;
}

#undef IR
#undef fins
#undef emitir
#undef emitir_raw

#endif
```

One Level Up Top Level

src/lj_def.h - luajit-2.0-src

Data types defined

- BloomFilter
- Unaligned16
- Unaligned16
- Unaligned32
- Unaligned32
- int16_t
- int16_t
- int32_t
- int32_t
- int64_t
- int64_t
- int8_t
- int8_t
- intptr_t
- intptr_t
- intptr_t
- uint16_t
- uint16_t
- uint32_t
- uint32_t
- uint64_t
- uint64_t
- uint8_t
- uint8_t
- uintptr_t
- uintptr_t
- uintptr_t

Functions defined

- lj_bswap

- lj_bswap
- lj_bswap
- lj_bswap
- lj_bswap64
- lj_bswap64
- lj_bswap64
- lj_bswap64
- lj_bswap64
- lj_ffs
- lj_fls
- lj_fls
- lj_fls
- lj_getu16
- lj_getu16
- lj_getu32
- lj_getu32

Macros defined

- BLOOM_MASK
- LJ_AINLINE
- LJ_AINLINE
- LJ_ALIGN
- LJ_ALIGN
- LJ_ASMF
- LJ_ASMF_NORET
- LJ_ASSERT_NAME
- LJ_ASSERT_NAME2
- LJ_DATA
- LJ_DATADEF
- LJ_FASTCALL
- LJ_FASTCALL
- LJ_FASTCALL
- LJ_FUNC
- LJ_FUNC

- LJ_FUNCA
- LJ_FUNCA_NORET
- LJ_FUNC_NORET
- LJ_INLINE
- LJ_INLINE
- LJ_LIKELY
- LJ_LIKELY
- LJ_MAX_ABITS
- LJ_MAX_ALLOC
- LJ_MAX_ASIZE
- LJ_MAX_BCINS
- LJ_MAX_BUF
- LJ_MAX_COLOSIZE
- LJ_MAX_EXITSTUBGR
- LJ_MAX_HBITS
- LJ_MAX_IDXCHAIN
- LJ_MAX_JSLOTS
- LJ_MAX_LINE
- LJ_MAX_LOCVAR
- LJ_MAX_MEM
- LJ_MAX_MEM32
- LJ_MAX_MEM64
- LJ_MAX_PHI
- LJ_MAX_SLOTS
- LJ_MAX_STR
- LJ_MAX_STRTAB
- LJ_MAX_UDATA
- LJ_MAX_UPVAL
- LJ_MAX_XLEVEL
- LJ_MIN_GLOBAL
- LJ_MIN_IRSZ
- LJ_MIN_K64SZ
- LJ_MIN_REGISTRY

- LJ_MIN_SBUF
- LJ_MIN_STRTAB
- LJ_MIN_VECSZ
- LJ_NOAPI
- LJ_NOAPI
- LJ_NOINLINE
- LJ_NOINLINE
- LJ_NORET
- LJ_NORET
- LJ_NORET
- LJ_NUM_CBPAGE
- LJ_STACK_EXTRA
- LJ_STATIC_ASSERT
- LJ_STATIC_ASSERT
- LJ_UNLIKELY
- LJ_UNLIKELY
- U64x
- UNUSED
- _LJ_DEF_H
- api_check
- api_check
- bloombit
- bloomset
- bloomtest
- check_exp
- check_exp
- checki16
- checki32
- checki8
- checkptr32
- checkptr47
- checkptrGC
- checkptrGC

- checkptrGC
- checku16
- checku32
- checku8
- i32ptr
- lj_bswap
- lj_bswap64
- lj_ffs
- lj_fls
- lj_getu16
- lj_getu32
- lj_rol
- lj_ror
- lua_assert
- u32ptr

Source code

```
/*
** LuaJIT common internal definitions.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_DEF_H
#define _LJ_DEF_H

#include "lua.h"

 #if defined(_MSC_VER)
 /* MSVC is stuck in the last century and doesn't have C99's stdint.h. */
 typedef __int8 int8_t;
 typedef __int16 int16_t;
 typedef __int32 int32_t;
 typedef __int64 int64_t;
 typedef unsigned __int8 uint8_t;
 typedef unsigned __int16 uint16_t;
 typedef unsigned __int32 uint32_t;
 typedef unsigned __int64 uint64_t;
 #ifdef _WIN64
 typedef __int64 intptr_t;
 typedef unsigned __int64 uintptr_t;
 #else
 typedef __int32 intptr_t;
 typedef unsigned __int32 uintptr_t;
 #endif
 #elif defined(__symbian__)
 /* Cough. */
 typedef signed char int8_t;
 typedef short int int16_t;
 typedef int int32_t;
 typedef long long int64_t;
 typedef unsigned char uint8_t;
 typedef unsigned short int uint16_t;
 typedef unsigned int uint32_t;
 typedef unsigned long long uint64_t;
 typedef int intptr_t;
```

```
typedef unsigned int uintptr_t;
#else
#include <stdint.h>
#endif

/* Needed everywhere. */
#include <string.h>
#include <stdlib.h>

/* Various VM limits. */
#define LJ_MAX_MEM32        0x7fffff00        /* Max. 32 bit memory allocation. */
#define LJ_MAX_MEM64        ((uint64_t)1<<47)  /* Max. 64 bit memory allocation. */
/* Max. total memory allocation. */
#define LJ_MAX_MEM        (LJ_GC64 ? LJ_MAX_MEM64 : LJ_MAX_MEM32)
#define LJ_MAX_ALLOC        LJ_MAX_MEM        /* Max. individual allocation length. */
#define LJ_MAX_STR        LJ_MAX_MEM32        /* Max. string length. */
#define LJ_MAX_BUF        LJ_MAX_MEM32        /* Max. buffer length. */
#define LJ_MAX_UDATA        LJ_MAX_MEM32        /* Max. userdata length. */

#define LJ_MAX_STRTAB        (1<<26)                /* Max. string table size. */
#define LJ_MAX_HBITS        26                /* Max. hash bits. */
#define LJ_MAX_ABITS        28                /* Max. bits of array key. */
#define LJ_MAX_ASIZE        ((1<<(LJ_MAX_ABITS-1))+1)  /* Max. array part size. */
#define LJ_MAX_COLOSIZE        16                /* Max. elems for colocated array. */

#define LJ_MAX_LINE        LJ_MAX_MEM32        /* Max. source code line number. */
#define LJ_MAX_XLEVEL        200                /* Max. syntactic nesting level. */
#define LJ_MAX_BCINS        (1<<26)                /* Max. # of bytecode instructions. */
#define LJ_MAX_SLOTS        250                /* Max. # of slots in a Lua func. */
#define LJ_MAX_LOCVAR        200                /* Max. # of local variables. */
#define LJ_MAX_UPVAL        60                /* Max. # of upvalues. */

#define LJ_MAX_IDXCHAIN        100                /* __index/__newindex chain limit. */
#define LJ_STACK_EXTRA        (5+2*LJ_FR2)        /* Extra stack space (metamethods). */

#define LJ_NUM_CBPAGE        1                /* Number of FFI callback pages. */

/* Minimum table/buffer sizes. */
#define LJ_MIN_GLOBAL        6                /* Min. global table size (hbits). */
#define LJ_MIN_REGISTRY        2                /* Min. registry size (hbits). */
#define LJ_MIN_STRTAB        256                /* Min. string table size (pow2). */
#define LJ_MIN_SBUF        32                /* Min. string buffer length. */
#define LJ_MIN_VECSZ        8                /* Min. size for growable vectors. */
#define LJ_MIN_IRSZ        32                /* Min. size for growable IR. */
#define LJ_MIN_K64SZ        16                /* Min. size for chained K64Array. */

/* JIT compiler limits. */
#define LJ_MAX_JSLOTS        250                /* Max. # of stack slots for a trace. */
#define LJ_MAX_PHI        64                /* Max. # of PHIs for a loop. */
#define LJ_MAX_EXITSTUBGR        16        /* Max. # of exit stub groups. */

/* Various macros. */
#ifndef UNUSED
#define UNUSED(x)        ((void)(x))        /* to avoid warnings */
#endif

#define U64x(hi, lo)        (((uint64_t)0x##hi << 32) + (uint64_t)0x##lo)
#define i32ptr(p)        ((int32_t)(intptr_t)(void *)(p))
#define u32ptr(p)        ((uint32_t)(intptr_t)(void *)(p))

#define checki8(x)        ((x) == (int32_t)(int8_t)(x))
 #define checku8(x)        ((x) == (int32_t)(uint8_t)(x))
 #define checki16(x)        ((x) == (int32_t)(int16_t)(x))
 #define checku16(x)        ((x) == (int32_t)(uint16_t)(x))
 #define checki32(x)        ((x) == (int32_t)(x))
 #define checku32(x)        ((x) == (uint32_t)(x))
 #define checkptr32(x)        ((uintptr_t)(x) == (uint32_t)(uintptr_t)(x))
 #define checkptr47(x)        (((uint64_t)(x) >> 47) == 0)
 #if LJ_GC64
 #define checkptrGC(x)        (checkptr47((x)))
 #elif LJ_64
 #define checkptrGC(x)        (checkptr32((x)))
 #else
 #define checkptrGC(x)        1
 #endif
```

```c
/* Every half-decent C compiler transforms this into a rotate instruction. */
#define lj_rol(x, n)		(((x)<<(n)) | ((x)>>(-(int)(n)&(8*sizeof(x)-1))))
#define lj_ror(x, n)		(((x)<<(-(int)(n)&(8*sizeof(x)-1))) | ((x)>>(n)))

/* A really naive Bloom filter. But sufficient for our needs. */
typedef uintptr_t BloomFilter;
#define BLOOM_MASK		(8*sizeof(BloomFilter) - 1)
#define bloombit(x)		((uintptr_t)1 << ((x) & BLOOM_MASK))
#define bloomset(b, x)		((b) |= bloombit((x)))
#define bloomtest(b, x)		((b) & bloombit((x)))

#if defined(__GNUC__) || defined(__psp2__)

#define LJ_NORET		__attribute__((noreturn))
#define LJ_ALIGN(n)		__attribute__((aligned(n)))
#define LJ_INLINE		inline
#define LJ_AINLINE		inline __attribute__((always_inline))
#define LJ_NOINLINE		__attribute__((noinline))

#if defined(__ELF__) || defined(__MACH__) || defined(__psp2__)
#if !((defined(__sun__) && defined(__svr4__)) || defined(__CELLOS_LV2__))
#define LJ_NOAPI		extern __attribute__((visibility("hidden")))
#endif
#endif

/* Note: it's only beneficial to use fastcall on x86 and then only for up to
** two non-FP args. The amalgamated compile covers all LJ_FUNC cases. Only
** indirect calls and related tail-called C functions are marked as fastcall.
*/
#if defined(__i386__)
#define LJ_FASTCALL		__attribute__((fastcall))
#endif

#define LJ_LIKELY(x)		__builtin_expect(!!(x), 1)
#define LJ_UNLIKELY(x)		__builtin_expect(!!(x), 0)

#define lj_ffs(x)		((uint32_t)__builtin_ctz(x))
/* Don't ask ... */
#if defined(__INTEL_COMPILER) && (defined(__i386__) || defined(__x86_64__))
static LJ_AINLINE uint32_t lj_fls(uint32_t x)
{
  uint32_t r; __asm__("bsrl %1, %0" : "=r" (r) : "rm" (x) : "cc"); return r;
}
#else
#define lj_fls(x)		((uint32_t)(__builtin_clz(x)^31))
#endif

#if defined(__arm__)
static LJ_AINLINE uint32_t lj_bswap(uint32_t x)
{
#if defined(__psp2__)
  return __builtin_rev(x);
#else
  uint32_t r;
#if __ARM_ARCH_6__ || __ARM_ARCH_6J__ || __ARM_ARCH_6T2__ || __ARM_ARCH_6Z__ ||\
    __ARM_ARCH_6ZK__ || __ARM_ARCH_7__ || __ARM_ARCH_7A__ || __ARM_ARCH_7R__
  __asm__("rev %0, %1" : "=r" (r) : "r" (x));
  return r;
#else
#ifdef __thumb__
  r = x ^ lj_ror(x, 16);
#else
  __asm__("eor %0, %1, %1, ror #16" : "=r" (r) : "r" (x));
#endif
  return ((r & 0xff00ffffu) >> 8) ^ lj_ror(x, 8);
#endif
#endif
}

static LJ_AINLINE uint64_t lj_bswap64(uint64_t x)
{
  return ((uint64_t)lj_bswap((uint32_t)x)<<32) | lj_bswap((uint32_t)(x>>32));
}
#elif (__GNUC__ > 4) || (__GNUC__ == 4 && __GNUC_MINOR__ >= 3)
static LJ_AINLINE uint32_t lj_bswap(uint32_t x)
{
```

```
  return (uint32_t)__builtin_bswap32((int32_t)x);
}

static LJ_AINLINE uint64_t lj_bswap64(uint64_t x)
{
  return (uint64_t)__builtin_bswap64((int64_t)x);
}
#elif defined(__i386__) || defined(__x86_64__)
static LJ_AINLINE uint32_t lj_bswap(uint32_t x)
{
  uint32_t r; __asm__("bswap %0" : "=r" (r) : "0" (x)); return r;
}

#if defined(__i386__)
static LJ_AINLINE uint64_t lj_bswap64(uint64_t x)
{
  return ((uint64_t)lj_bswap((uint32_t)x)<<32) | lj_bswap((uint32_t)(x>>32));
}
#else
static LJ_AINLINE uint64_t lj_bswap64(uint64_t x)
{
  uint64_t r; __asm__("bswap %0" : "=r" (r) : "0" (x)); return r;
}
#endif
#else
static LJ_AINLINE uint32_t lj_bswap(uint32_t x)
{
  return (x << 24) | ((x & 0xff00) << 8) | ((x >> 8) & 0xff00) | (x >> 24);
}

static LJ_AINLINE uint64_t lj_bswap64(uint64_t x)
{
  return (uint64_t)lj_bswap((uint32_t)(x >> 32)) |
         ((uint64_t)lj_bswap((uint32_t)x) << 32);
}
#endif

typedef union __attribute__((packed)) Unaligned16 {
  uint16_t u;
  uint8_t b[2];
} Unaligned16;

typedef union __attribute__((packed)) Unaligned32 {
  uint32_t u;
  uint8_t b[4];
} Unaligned32;

/* Unaligned load of uint16_t. */
static LJ_AINLINE uint16_t lj_getu16(const void *p)
{
  return ((const Unaligned16 *)p)->u;
}

/* Unaligned load of uint32_t. */
static LJ_AINLINE uint32_t lj_getu32(const void *p)
{
  return ((const Unaligned32 *)p)->u;
}

#elif defined(_MSC_VER)

#define LJ_NORET        __declspec(noreturn)
#define LJ_ALIGN(n)       __declspec(align(n))
#define LJ_INLINE       __inline
#define LJ_AINLINE       __forceinline
#define LJ_NOINLINE       __declspec(noinline)
#if defined(_M_IX86)
#define LJ_FASTCALL       __fastcall
#endif

#ifdef _M_PPC
unsigned int _CountLeadingZeros(long);
#pragma intrinsic(_CountLeadingZeros)
static LJ_AINLINE uint32_t lj_fls(uint32_t x)
{
  return _CountLeadingZeros(x) ^ 31;
```

```
}
#else
unsigned char _BitScanForward(uint32_t *, unsigned long);
unsigned char _BitScanReverse(uint32_t *, unsigned long);
#pragma intrinsic(_BitScanForward)
#pragma intrinsic(_BitScanReverse)

static LJ_AINLINE uint32_t lj_ffs(uint32_t x)
{
  uint32_t r; _BitScanForward(&r, x); return r;
}

static LJ_AINLINE uint32_t lj_fls(uint32_t x)
{
  uint32_t r; _BitScanReverse(&r, x); return r;
}
#endif

unsigned long _byteswap_ulong(unsigned long);
uint64_t _byteswap_uint64(uint64_t);
#define lj_bswap(x)        (_byteswap_ulong((x)))
#define lj_bswap64(x)        (_byteswap_uint64((x)))

#if defined(_M_PPC) && defined(LUAJIT_NO_UNALIGNED)
/*
** Replacement for unaligned loads on Xbox 360. Disabled by default since it's
** usually more costly than the occasional stall when crossing a cache-line.
*/
static LJ_AINLINE uint16_t lj_getu16(const void *v)
{
  const uint8_t *p = (const uint8_t *)v;
  return (uint16_t)((p[0]<<8) | p[1]);
}
static LJ_AINLINE uint32_t lj_getu32(const void *v)
{
  const uint8_t *p = (const uint8_t *)v;
  return (uint32_t)((p[0]<<24) | (p[1]<<16) | (p[2]<<8) | p[3]);
}
#else
/* Unaligned loads are generally ok on x86/x64. */
#define lj_getu16(p)        (*(uint16_t *)(p))
#define lj_getu32(p)        (*(uint32_t *)(p))
#endif

#else
#error "missing defines for your compiler"
#endif

/* Optional defines. */
#ifndef LJ_FASTCALL
#define LJ_FASTCALL
#endif
#ifndef LJ_NORET
#define LJ_NORET
#endif
#ifndef LJ_NOAPI
#define LJ_NOAPI        extern
#endif
#ifndef LJ_LIKELY
#define LJ_LIKELY(x)        (x)
#define LJ_UNLIKELY(x)        (x)
#endif

/* Attributes for internal functions. */
#define LJ_DATA                LJ_NOAPI
#define LJ_DATADEF
#define LJ_ASMF                LJ_NOAPI
#define LJ_FUNCA        LJ_NOAPI
#if defined(ljamalg_c)
#define LJ_FUNC                static
#else
#define LJ_FUNC                LJ_NOAPI
#endif
#define LJ_FUNC_NORET        LJ_FUNC LJ_NORET
#define LJ_FUNCA_NORET        LJ_FUNCA LJ_NORET
#define LJ_ASMF_NORET        LJ_ASMF LJ_NORET
```

```
343
344 /* Runtime assertions. */
345 #ifdef lua_assert
346 #define check_exp(c, e)               (lua_assert(c), (e))
347 #define api_check(l, e)               lua_assert(e)
348 #else
349 #define lua_assert(c)               ((void)0)
350 #define check_exp(c, e)               (e)
351 #define api_check                luai_apicheck
352 #endif
353
354 /* Static assertions. */
355 #define LJ_ASSERT_NAME2(name, line)        name ## line
356 #define LJ_ASSERT_NAME(line)                LJ_ASSERT_NAME2(lj_assert_, line)
357 #ifdef __COUNTER__
358 #define LJ_STATIC_ASSERT(cond) \
359   extern void LJ_ASSERT_NAME(__COUNTER__)(int STATIC_ASSERTION_FAILED[(cond)?1:-1])
360 #else
361 #define LJ_STATIC_ASSERT(cond) \
362   extern void LJ_ASSERT_NAME(__LINE__)(int STATIC_ASSERTION_FAILED[(cond)?1:-1])
363 #endif
364
365 #endif
```

One Level Up Top Level

src/lj_obj.h - luajit-2.0-src

Global variables defined

- TValue
- lj_obj_itypename
- lj_obj_typename
- offsetof
- offsetof
- offsetof
- offsetof
- offsetof
- offsetof
- offsetof
- offsetof

Data types defined

- ASMFunction
- BCIns
- BCLine
- BCPos
- BCReg
- FrameLink
- GCRef
- GCRef
- GCRootID
- GCSize
- GCSize
- GCState
- GCState
- GCcdata
- GCcdata
- GCcdataVar
- GCcdataVar

- GCfunc
- GCfunc
- GCfuncC
- GCfuncC
- GCfuncL
- GCfuncL
- GChead
- GChead
- GCobj
- GCobj
- GCproto
- GCproto
- GCstr
- GCstr
- GCtab
- GCtab
- GCudata
- GCudata
- GCupval
- GCupval
- MMS
- MRef
- MRef
- MSize
- Node
- Node
- SBuf
- SBuf
- TValue
- cTValue
- global_State
- global_State
- lua_State

Functions defined

- copyTV
- lj_num2bit
- lj_num2u64
- numberVint
- numberVnum
- setgcV
- setgcVraw
- setint64V
- setintV
- setlightudV

Macros defined

- FF_C
- FF_LUA
- G
- GCHeader
- GCfuncHeader
- HOOK_ACTIVE
- HOOK_ACTIVE_SHIFT
- HOOK_EVENTMASK
- HOOK_GC
- HOOK_PROFILE
- HOOK_VMEVENT
- LAST_TT
- LJ_GCVMASK
- LJ_TCDATA
- LJ_TFALSE
- LJ_TFUNC
- LJ_TISGCV
- LJ_TISNUM
- LJ_TISNUM
- LJ_TISPRI

- LJ_TISTABUD
- LJ_TISTRUECOND
- LJ_TLIGHTUD
- LJ_TNIL
- LJ_TNUMX
- LJ_TPROTO
- LJ_TSTR
- LJ_TTAB
- LJ_TTHREAD
- LJ_TTRACE
- LJ_TTRUE
- LJ_TUDATA
- LJ_TUPVAL
- LUA_TCDATA
- LUA_TPROTO
- MMDEF
- MMDEF_FFI
- MMDEF_FFI
- MMDEF_PAIRS
- MMDEF_PAIRS
- MMENUM
- MMENUM
- MM_ipairs
- MM_pairs
- PROTO_CHILD
- PROTO_CLCOUNT
- PROTO_CLC_BITS
- PROTO_CLC_POLY
- PROTO_FFI
- PROTO_FIXUP_RETURN
- PROTO_HAS_RETURN
- PROTO_ILOOP
- PROTO_NOJIT

- PROTO_UV_IMMUTABLE
- PROTO_UV_LOCAL
- PROTO_VARARG
- SCALE_NUM_GCO
- _LJ_OBJ_H
- basemt_it
- basemt_obj
- boolV
- cdataV
- cdataisv
- cdataptr
- cdatav
- cdatavlen
- checklightudptr
- checklightudptr
- curr_func
- curr_func
- curr_func
- curr_funcisL
- curr_proto
- curr_top
- curr_topL
- define_setV
- funcV
- funcproto
- gcV
- gcnext
- gco2cd
- gco2func
- gco2pt
- gco2str
- gco2tab
- gco2th

- gco2ud
- gco2uv
- gcref
- gcref
- gcrefeq
- gcrefeq
- gcrefp
- gcrefp
- gcrefu
- gcrefu
- gcval
- gcval
- getfreetop
- getfreetop
- hook_active
- hook_enter
- hook_entergc
- hook_leave
- hook_restore
- hook_save
- hook_vmevent
- intV
- iscfunc
- isffunc
- isluafunc
- itype
- itype
- itypemap
- itypemap
- lightudV
- lightudV
- lj_num2int
- lj_typename

- mainthread
- memcdatav
- mmname_str
- mref
- mref
- nextnode
- niltv
- niltvg
- noderef
- numV
- obj2gco
- protoV
- proto_bc
- proto_bcpos
- proto_chunkname
- proto_chunknamestr
- proto_kgc
- proto_knumtv
- proto_lineinfo
- proto_uv
- proto_uvinfo
- proto_varinfo
- rawnumequal
- registry
- round_nkgc
- setboolV
- setboolV
- setcont
- setcont
- setcont
- setfreetop
- setfreetop
- setgcref

- setgcref
- setgcrefnull
- setgcrefnull
- setgcrefp
- setgcrefp
- setgcrefr
- setgcrefr
- setgcreft
- setintptrV
- setintptrV
- setitype
- setitype
- setminfV
- setmref
- setmref
- setmrefr
- setmrefr
- setnanV
- setnilV
- setnilV
- setnumV
- setpinfV
- setpriV
- setpriV
- setvmstate
- sizeCfunc
- sizeLfunc
- sizecdatav
- sizestring
- sizetabcolo
- sizeudata
- strV
- strVdata

- strdata
- strdatawr
- strref
- tabV
- tabref
- threadV
- tvchecklive
- tvisbool
- tviscdata
- tvisfalse
- tvisfunc
- tvisgcv
- tvisint
- tvislightud
- tvislightud
- tvismzero
- tvisnan
- tvisnil
- tvisnil
- tvisnum
- tvisnumber
- tvispone
- tvispri
- tvisproto
- tvispzero
- tvisstr
- tvistab
- tvistabud
- tvisthread
- tvistrue
- tvistruecond
- tvisudata
- tviszero

- tviszero
- tvref
- udataV
- uddata
- uvnext
- uvprev
- uvval

Source code

```
/*
** LuaJIT VM tags, values and objects.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Portions taken verbatim or adapted from the Lua interpreter.
** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
*/

#ifndef _LJ_OBJ_H
#define _LJ_OBJ_H

#include "lua.h"
#include "lj_def.h"
#include "lj_arch.h"

/* -- Memory references (32 bit address space) ---------------------------- */

/* Memory and GC object sizes. */
typedef uint32_t MSize;
#if LJ_GC64
typedef uint64_t GCSize;
#else
typedef uint32_t GCSize;
#endif

/* Memory reference */
typedef struct MRef {
#if LJ_GC64
  uint64_t ptr64;        /* True 64 bit pointer. */
#else
  uint32_t ptr32;        /* Pseudo 32 bit pointer. */
#endif
} MRef;

#if LJ_GC64
#define mref(r, t)        ((t *)(void *)(r).ptr64)

#define setmref(r, p)        ((r).ptr64 = (uint64_t)(void *)(p))
#define setmrefr(r, v)        ((r).ptr64 = (v).ptr64)
#else
#define mref(r, t)        ((t *)(void *)(uintptr_t)(r).ptr32)

#define setmref(r, p)        ((r).ptr32 = (uint32_t)(uintptr_t)(void *)(p))
#define setmrefr(r, v)        ((r).ptr32 = (v).ptr32)
#endif

/* -- GC object references (32 bit address space) ------------------------- */

/* GCobj reference */
typedef struct GCRef {
#if LJ_GC64
  uint64_t gcptr64;        /* True 64 bit pointer. */
#else
  uint32_t gcptr32;        /* Pseudo 32 bit pointer. */
#endif
} GCRef;
```

```
/* Common GC header for all collectable objects. */
#define GCHeader	GCRef nextgc; uint8_t marked; uint8_t gct
/* This occupies 6 bytes, so use the next 2 bytes for non-32 bit fields. */

#if LJ_GC64
#define gcref(r)	((GCobj *)(r).gcptr64)
#define gcrefp(r, t)	((t *)(void *)(r).gcptr64)
#define gcrefu(r)	((r).gcptr64)
#define gcrefeq(r1, r2)	((r1).gcptr64 == (r2).gcptr64)

#define setgcref(r, gc)	((r).gcptr64 = (uint64_t)&(gc)->gch)
#define setgcreft(r, gc, it) \
  (r).gcptr64 = (uint64_t)&(gc)->gch | (((uint64_t)(it)) << 47)
#define setgcrefp(r, p)	((r).gcptr64 = (uint64_t)(p))
#define setgcrefnull(r)	((r).gcptr64 = 0)
#define setgcrefr(r, v)	((r).gcptr64 = (v).gcptr64)
#else
#define gcref(r)	((GCobj *)(uintptr_t)(r).gcptr32)
#define gcrefp(r, t)	((t *)(void *)(uintptr_t)(r).gcptr32)
#define gcrefu(r)	((r).gcptr32)
#define gcrefeq(r1, r2)	((r1).gcptr32 == (r2).gcptr32)

#define setgcref(r, gc)	((r).gcptr32 = (uint32_t)(uintptr_t)&(gc)->gch)
#define setgcrefp(r, p)	((r).gcptr32 = (uint32_t)(uintptr_t)(p))
#define setgcrefnull(r)	((r).gcptr32 = 0)
#define setgcrefr(r, v)	((r).gcptr32 = (v).gcptr32)
#endif

#define gcnext(gc)	(gcref((gc)->gch.nextgc))

/* IMPORTANT NOTE:
**
** All uses of the setgcref* macros MUST be accompanied with a write barrier.
**
** This is to ensure the integrity of the incremental GC. The invariant
** to preserve is that a black object never points to a white object.
** I.e. never store a white object into a field of a black object.
**
** It's ok to LEAVE OUT the write barrier ONLY in the following cases:
** - The source is not a GC object (NULL).
** - The target is a GC root. I.e. everything in global_State.
** - The target is a lua_State field (threads are never black).
** - The target is a stack slot, see setgcV et al.
** - The target is an open upvalue, i.e. pointing to a stack slot.
** - The target is a newly created object (i.e. marked white). But make
**   sure nothing invokes the GC inbetween.
** - The target and the source are the same object (self-reference).
** - The target already contains the object (e.g. moving elements around).
**
** The most common case is a store to a stack slot. All other cases where
** a barrier has been omitted are annotated with a NOBARRIER comment.
**
** The same logic applies for stores to table slots (array part or hash
** part). ALL uses of lj_tab_set* require a barrier for the stored value
** *and* the stored key, based on the above rules. In practice this means
** a barrier is needed if *either* of the key or value are a GC object.
**
** It's ok to LEAVE OUT the write barrier in the following special cases:
** - The stored value is nil. The key doesn't matter because it's either
**   not resurrected or lj_tab_newkey() will take care of the key barrier.
** - The key doesn't matter if the *previously* stored value is guaranteed
**   to be non-nil (because the key is kept alive in the table).
** - The key doesn't matter if it's guaranteed not to be part of the table,
**   since lj_tab_newkey() takes care of the key barrier. This applies
**   trivially to new tables, but watch out for resurrected keys. Storing
**   a nil value leaves the key in the table!
**
** In case of doubt use lj_gc_anybarriert() as it's rather cheap. It's used
** by the interpreter for all table stores.
**
** Note: In contrast to Lua's GC, LuaJIT's GC does *not* specially mark
** dead keys in tables. The reference is left in, but it's guaranteed to
** be never dereferenced as long as the value is nil. It's ok if the key is
** freed or if any object subsequently gets the same address.
**
```

```
** Not destroying dead keys helps to keep key hash slots stable. This avoids
** specialization back-off for HREFK when a value flips between nil and
** non-nil and the GC gets in the way. It also allows safely hoisting
** HREF/HREFK across GC steps. Dead keys are only removed if a table is
** resized (i.e. by NEWREF) and xREF must not be CSEd across a resize.
**
** The trade-off is that a write barrier for tables must take the key into
** account, too. Implicitly resurrecting the key by storing a non-nil value
** may invalidate the incremental GC invariant.
*/

/* -- Common type definitions ---------------------------------------------- */

/* Types for handling bytecodes. Need this here, details in lj_bc.h. */
typedef uint32_t BCIns;  /* Bytecode instruction. */
typedef uint32_t BCPos;  /* Bytecode position. */
typedef uint32_t BCReg;  /* Bytecode register. */
typedef int32_t BCLine;  /* Bytecode line number. */

/* Internal assembler functions. Never call these directly from C. */
typedef void (*ASMFunction)(void);

/* Resizable string buffer. Need this here, details in lj_buf.h. */
typedef struct SBuf {
  MRef p;                /* String buffer pointer. */
  MRef e;                /* String buffer end pointer. */
  MRef b;                /* String buffer base. */
  MRef L;                /* lua_State, used for buffer resizing. */
} SBuf;

/* -- Tags and values ------------------------------------------------------ */

/* Frame link. */
typedef union {
  int32_t ftsz;               /* Frame type and size of previous frame. */
  MRef pcr;                /* Or PC for Lua frames. */
} FrameLink;

/* Tagged value. */
typedef LJ_ALIGN(8) union TValue {
  uint64_t u64;               /* 64 bit pattern overlaps number. */
  lua_Number n;               /* Number object overlaps split tag/value object. */
#if LJ_GC64
  GCRef gcr;               /* GCobj reference with tag. */
  int64_t it64;
  struct {
    LJ_ENDIAN_LOHI(
      int32_t i;        /* Integer value. */
    , uint32_t it;        /* Internal object tag. Must overlap MSW of number. */
    )
  };
#else
  struct {
    LJ_ENDIAN_LOHI(
      union {
        GCRef gcr;        /* GCobj reference (if any). */
        int32_t i;        /* Integer value. */
      };
    , uint32_t it;        /* Internal object tag. Must overlap MSW of number. */
    )
  };
#endif
#if LJ_FR2
  int64_t ftsz;               /* Frame type and size of previous frame, or PC. */
#else
  struct {
    LJ_ENDIAN_LOHI(
      GCRef func;        /* Function for next frame (or dummy L). */
    , FrameLink tp;        /* Link to previous frame. */
    )
  } fr;
#endif
  struct {
    LJ_ENDIAN_LOHI(
      uint32_t lo;        /* Lower 32 bits of number. */
    , uint32_t hi;        /* Upper 32 bits of number. */
```

```
    )
  } u32;
} TValue;

typedef const TValue cTValue;

#define tvref(r)        (mref(r, TValue))

/* More external and GCobj tags for internal objects. */
#define LAST_TT                 LUA_TTHREAD
#define LUA_TPROTO        (LAST_TT+1)
#define LUA_TCDATA        (LAST_TT+2)

/* Internal object tags.
**
** Format for 32 bit GC references (!LJ_GC64):
**
** Internal tags overlap the MSW of a number object (must be a double).
** Interpreted as a double these are special NaNs. The FPU only generates
** one type of NaN (0xfff8_0000_0000_0000). So MSWs > 0xfff80000 are available
** for use as internal tags. Small negative numbers are used to shorten the
** encoding of type comparisons (reg/mem against sign-ext. 8 bit immediate).
**
**                  ---MSW---.---LSW---
** primitive types |  itype  |         |
** lightuserdata   |  itype  |  void * |  (32 bit platforms)
** lightuserdata   |ffff|    void *    |  (64 bit platforms, 47 bit pointers)
** GC objects      |  itype  |  GCRef  |
** int (LJ_DUALNUM)|  itype  |   int   |
** number           -------double------
**
** Format for 64 bit GC references (LJ_GC64):
**
** The upper 13 bits must be 1 (0xfff8...) for a special NaN. The next
** 4 bits hold the internal tag. The lowest 47 bits either hold a pointer,
** a zero-extended 32 bit integer or all bits set to 1 for primitive types.
**
**                     ------MSW------.------LSW------
** primitive types    |1..1|itype|1...................1|
** GC objects/lightud |1..1|itype|-------GCRef--------|
** int (LJ_DUALNUM)   |1..1|itype|0..0|-----int-------|
** number              ------------double-------------
**
** ORDER LJ_T
** Primitive types nil/false/true must be first, lightuserdata next.
** GC objects are at the end, table/userdata must be lowest.
** Also check lj_ir.h for similar ordering constraints.
*/
#define LJ_TNIL                         (~0u)
#define LJ_TFALSE               (~1u)
#define LJ_TTRUE                (~2u)
#define LJ_TLIGHTUD               (~3u)
#define LJ_TSTR                         (~4u)
#define LJ_TUPVAL               (~5u)
#define LJ_TTHREAD               (~6u)
#define LJ_TPROTO               (~7u)
#define LJ_TFUNC                (~8u)
#define LJ_TTRACE               (~9u)
#define LJ_TCDATA               (~10u)
#define LJ_TTAB                         (~11u)
#define LJ_TUDATA               (~12u)
/* This is just the canonical number type used in some places. */
#define LJ_TNUMX                (~13u)

/* Integers have itype == LJ_TISNUM doubles have itype < LJ_TISNUM */
#if LJ_64 && !LJ_GC64
#define LJ_TISNUM               0xfffeffffu
#else
#define LJ_TISNUM               LJ_TNUMX
#endif
#define LJ_TISTRUECOND                LJ_TFALSE
#define LJ_TISPRI               LJ_TTRUE
#define LJ_TISGCV               (LJ_TSTR+1)
#define LJ_TISTABUD               LJ_TTAB

#if LJ_GC64
```

```
#define LJ_GCVMASK               (((uint64_t)1 << 47) - 1)
#endif

/* -- String object ------------------------------------------------------ */

/* String object header. String payload follows. */
typedef struct GCstr {
  GCHeader;
  uint8_t reserved;        /* Used by lexer for fast lookup of reserved words. */
  uint8_t unused;
  MSize hash;                /* Hash of string. */
  MSize len;                /* Size of string. */
} GCstr;

#define strref(r)        (&gcref((r))->str)
#define strdata(s)        ((const char *)((s)+1))
#define strdatawr(s)        ((char *)((s)+1))
#define strVdata(o)        strdata(strV(o))
#define sizestring(s)        (sizeof(struct GCstr)+(s)->len+1)

/* -- Userdata object ---------------------------------------------------- */

/* Userdata object. Payload follows. */
typedef struct GCudata {
  GCHeader;
  uint8_t udtype;        /* Userdata type. */
  uint8_t unused2;
  GCRef env;                /* Should be at same offset in GCfunc. */
  MSize len;                /* Size of payload. */
  GCRef metatable;        /* Must be at same offset in GCtab. */
  uint32_t align1;        /* To force 8 byte alignment of the payload. */
} GCudata;

/* Userdata types. */
enum {
  UDTYPE_USERDATA,        /* Regular userdata. */
  UDTYPE_IO_FILE,        /* I/O library FILE. */
  UDTYPE_FFI_CLIB,        /* FFI C library namespace. */
  UDTYPE__MAX
};

#define uddata(u)        ((void *)((u)+1))
#define sizeudata(u)        (sizeof(struct GCudata)+(u)->len)

/* -- C data object ------------------------------------------------------ */

/* C data object. Payload follows. */
typedef struct GCcdata {
  GCHeader;
  uint16_t ctypeid;        /* C type ID. */
} GCcdata;

/* Prepended to variable-sized or realigned C data objects. */
typedef struct GCcdataVar {
  uint16_t offset;        /* Offset to allocated memory (relative to GCcdata). */
  uint16_t extra;        /* Extra space allocated (incl. GCcdata + GCcdatav). */
  MSize len;                /* Size of payload. */
} GCcdataVar;

#define cdataptr(cd)        ((void *)((cd)+1))
#define cdataisv(cd)        ((cd)->marked & 0x80)
#define cdatav(cd)        ((GCcdataVar *)((char *)(cd) - sizeof(GCcdataVar)))
#define cdatavlen(cd)        check_exp(cdataisv(cd), cdatav(cd)->len)
#define sizecdatav(cd)        (cdatavlen(cd) + cdatav(cd)->extra)
#define memcdatav(cd)        ((void *)((char *)(cd) - cdatav(cd)->offset))

/* -- Prototype object --------------------------------------------------- */

#define SCALE_NUM_GCO        ((int32_t)sizeof(lua_Number)/sizeof(GCRef))
#define round_nkgc(n)        (((n) + SCALE_NUM_GCO-1) & ~(SCALE_NUM_GCO-1))

typedef struct GCproto {
  GCHeader;
  uint8_t numparams;        /* Number of parameters. */
  uint8_t framesize;        /* Fixed frame size. */
  MSize sizebc;                /* Number of bytecode instructions. */
```

```
#if LJ_GC64
  uint32_t unused_gc64;
#endif
  GCRef gclist;
  MRef k;                /* Split constant array (points to the middle). */
  MRef uv;               /* Upvalue list. local slot|0x8000 or parent uv idx. */
  MSize sizekgc;         /* Number of collectable constants. */
  MSize sizekn;          /* Number of lua_Number constants. */
  MSize sizept;          /* Total size including colocated arrays. */
  uint8_t sizeuv;        /* Number of upvalues. */
  uint8_t flags;         /* Miscellaneous flags (see below). */
  uint16_t trace;        /* Anchor for chain of root traces. */
  /* ------ The following fields are for debugging/tracebacks only ------ */
  GCRef chunkname;       /* Name of the chunk this function was defined in. */
  BCLine firstline;      /* First line of the function definition. */
  BCLine numline;        /* Number of lines for the function definition. */
  MRef lineinfo;         /* Compressed map from bytecode ins. to source line. */
  MRef uvinfo;           /* Upvalue names. */
  MRef varinfo;          /* Names and compressed extents of local variables. */
} GCproto;

/* Flags for prototype. */
#define PROTO_CHILD               0x01        /* Has child prototypes. */
#define PROTO_VARARG              0x02        /* Vararg function. */
#define PROTO_FFI                 0x04        /* Uses BC_KCDATA for FFI datatypes. */
#define PROTO_NOJIT               0x08        /* JIT disabled for this function. */
#define PROTO_ILOOP               0x10        /* Patched bytecode with ILOOP etc. */
/* Only used during parsing. */
#define PROTO_HAS_RETURN          0x20        /* Already emitted a return. */
#define PROTO_FIXUP_RETURN        0x40        /* Need to fixup emitted returns. */
/* Top bits used for counting created closures. */
#define PROTO_CLCOUNT             0x20        /* Base of saturating 3 bit counter. */
#define PROTO_CLC_BITS            3
#define PROTO_CLC_POLY            (3*PROTO_CLCOUNT)  /* Polymorphic threshold. */

#define PROTO_UV_LOCAL            0x8000        /* Upvalue for local slot. */
#define PROTO_UV_IMMUTABLE        0x4000        /* Immutable upvalue. */

#define proto_kgc(pt, idx) \
  check_exp((uintptr_t)(intptr_t)(idx) >= (uintptr_t)-(intptr_t)(pt)->sizekgc, \
            gcref(mref((pt)->k, GCRef)[(idx)]))
#define proto_knumtv(pt, idx) \
  check_exp((uintptr_t)(idx) < (pt)->sizekn, &mref((pt)->k, TValue)[(idx)])
#define proto_bc(pt)              ((BCIns *)((char *)(pt) + sizeof(GCproto)))
#define proto_bcpos(pt, pc)       ((BCPos)((pc) - proto_bc(pt)))
#define proto_uv(pt)              (mref((pt)->uv, uint16_t))

#define proto_chunkname(pt)       (strref((pt)->chunkname))
#define proto_chunknamestr(pt)       (strdata(proto_chunkname((pt))))
#define proto_lineinfo(pt)        (mref((pt)->lineinfo, const void))
#define proto_uvinfo(pt)        (mref((pt)->uvinfo, const uint8_t))
#define proto_varinfo(pt)        (mref((pt)->varinfo, const uint8_t))

/* -- Upvalue object ------------------------------------------------------ */

typedef struct GCupval {
  GCHeader;
  uint8_t closed;        /* Set if closed (i.e. uv->v == &uv->u.value). */
  uint8_t immutable;        /* Immutable value. */
  union {
    TValue tv;              /* If closed: the value itself. */
    struct {                /* If open: double linked list, anchored at thread. */
      GCRef prev;
      GCRef next;
    };
  };
  MRef v;                /* Points to stack slot (open) or above (closed). */
  uint32_t dhash;        /* Disambiguation hash: dh1 != dh2 => cannot alias. */
} GCupval;

#define uvprev(uv_)       (&gcref((uv_)->prev)->uv)
#define uvnext(uv_)       (&gcref((uv_)->next)->uv)
#define uvval(uv_)        (mref((uv_)->v, TValue))

/* -- Function object (closures) ------------------------------------------ */

```

```
/* Common header for functions. env should be at same offset in GCudata. */
#define GCfuncHeader \
  GCHeader; uint8_t ffid; uint8_t nupvalues; \
  GCRef env; GCRef gclist; MRef pc

typedef struct GCfuncC {
  GCfuncHeader;
  lua_CFunction f;        /* C function to be called. */
  TValue upvalue[1];        /* Array of upvalues (TValue). */
} GCfuncC;

typedef struct GCfuncL {
  GCfuncHeader;
  GCRef uvptr[1];        /* Array of _pointers_ to upvalue objects (GCupval). */
} GCfuncL;

typedef union GCfunc {
  GCfuncC c;
  GCfuncL l;
} GCfunc;

#define FF_LUA                0
#define FF_C                1
#define isluafunc(fn)        ((fn)->c.ffid == FF_LUA)
#define iscfunc(fn)        ((fn)->c.ffid == FF_C)
#define isffunc(fn)        ((fn)->c.ffid > FF_C)
#define funcproto(fn) \
  check_exp(isluafunc(fn), (GCproto *)(mref((fn)->l.pc, char)-sizeof(GCproto)))
#define sizeCfunc(n)        (sizeof(GCfuncC)-sizeof(TValue)+sizeof(TValue)*(n))
#define sizeLfunc(n)        (sizeof(GCfuncL)-sizeof(GCRef)+sizeof(GCRef)*(n))

/* -- Table object -------------------------------------------------------- */

/* Hash node. */
typedef struct Node {
  TValue val;                /* Value object. Must be first field. */
  TValue key;                /* Key object. */
  MRef next;                /* Hash chain. */
#if !LJ_GC64
  MRef freetop;                /* Top of free elements (stored in t->node[0]). */
#endif
} Node;

LJ_STATIC_ASSERT(offsetof(Node, val) == 0);

typedef struct GCtab {
  GCHeader;
  uint8_t nomm;                /* Negative cache for fast metamethods. */
  int8_t colo;                /* Array colocation. */
  MRef array;                /* Array part. */
  GCRef gclist;
  GCRef metatable;        /* Must be at same offset in GCudata. */
  MRef node;                /* Hash part. */
  uint32_t asize;        /* Size of array part (keys [0, asize-1]). */
  uint32_t hmask;        /* Hash part mask (size of hash part - 1). */
#if LJ_GC64
  MRef freetop;                /* Top of free elements. */
#endif
} GCtab;

#define sizetabcolo(n)        ((n)*sizeof(TValue) + sizeof(GCtab))
#define tabref(r)        (&gcref((r))->tab)
#define noderef(r)        (mref((r), Node))
#define nextnode(n)        (mref((n)->next, Node))
#if LJ_GC64
#define getfreetop(t, n)        (noderef((t)->freetop))
#define setfreetop(t, n, v)        (setmref((t)->freetop, (v)))
#else
#define getfreetop(t, n)        (noderef((n)->freetop))
#define setfreetop(t, n, v)        (setmref((n)->freetop, (v)))
#endif

/* -- State objects ------------------------------------------------------- */

/* VM states. */
enum {
```

```
  LJ_VMST_INTERP,        /* Interpreter. */
  LJ_VMST_C,                /* C function. */
  LJ_VMST_GC,                /* Garbage collector. */
  LJ_VMST_EXIT,                /* Trace exit handler. */
  LJ_VMST_RECORD,        /* Trace recorder. */
  LJ_VMST_OPT,                /* Optimizer. */
  LJ_VMST_ASM,                /* Assembler. */
  LJ_VMST__MAX
};

#define setvmstate(g, st)        ((g)->vmstate = ~LJ_VMST_##st)

/* Metamethods. ORDER MM */
#ifdef LJ_HASFFI
#define MMDEF_FFI(_) _(new)
#else
#define MMDEF_FFI(_)
#endif

#if LJ_52 || LJ_HASFFI
#define MMDEF_PAIRS(_) _(pairs) _(ipairs)
#else
#define MMDEF_PAIRS(_)
#define MM_pairs        255
#define MM_ipairs        255
#endif

#define MMDEF(_) \
  _(index) _(newindex) _(gc) _(mode) _(eq) _(len) \
  /* Only the above (fast) metamethods are negative cached (max. 8). */ \
  _(lt) _(le) _(concat) _(call) \
  /* The following must be in ORDER ARITH. */ \
  _(add) _(sub) _(mul) _(div) _(mod) _(pow) _(unm) \
  /* The following are used in the standard libraries. */ \
  _(metatable) _(tostring) MMDEF_FFI(_) MMDEF_PAIRS(_)

typedef enum {
#define MMENUM(name)        MM_##name,
MMDEF(MMENUM)
#undef MMENUM
  MM__MAX,
  MM____ = MM__MAX,
  MM_FAST = MM_len
} MMS;

/* GC root IDs. */
typedef enum {
  GCROOT_MMNAME,        /* Metamethod names. */
  GCROOT_MMNAME_LAST = GCROOT_MMNAME + MM__MAX-1,
  GCROOT_BASEMT,        /* Metatables for base types. */
  GCROOT_BASEMT_NUM = GCROOT_BASEMT + ~LJ_TNUMX,
  GCROOT_IO_INPUT,        /* Userdata for default I/O input file. */
  GCROOT_IO_OUTPUT,        /* Userdata for default I/O output file. */
  GCROOT_MAX
} GCRootID;

#define basemt_it(g, it)        ((g)->gcroot[GCROOT_BASEMT+~(it)])
#define basemt_obj(g, o)        ((g)->gcroot[GCROOT_BASEMT+itypemap(o)])
#define mmname_str(g, mm)        (strref((g)->gcroot[GCROOT_MMNAME+(mm)]))

typedef struct GCState {
  GCSize total;                /* Memory currently allocated. */
  GCSize threshold;        /* Memory threshold. */
  uint8_t currentwhite;        /* Current white color. */
  uint8_t state;        /* GC state. */
  uint8_t nocdatafin;        /* No cdata finalizer called. */
  uint8_t unused2;
  MSize sweepstr;        /* Sweep position in string table. */
  GCRef root;                /* List of all collectable objects. */
  MRef sweep;                /* Sweep position in root list. */
  GCRef gray;                /* List of gray objects. */
  GCRef grayagain;        /* List of objects for atomic traversal. */
  GCRef weak;                /* List of weak tables (to be cleared). */
  GCRef mmudata;        /* List of userdata (to be finalized). */
  GCSize debt;                /* Debt (how much GC is behind schedule). */
  GCSize estimate;        /* Estimate of memory actually in use. */
```

```
  MSize stepmul;        /* Incremental GC step granularity. */
  MSize pause;          /* Pause between successive GC cycles. */
} GCState;

/* Global state, shared by all threads of a Lua universe. */
typedef struct global_State {
  GCRef *strhash;       /* String hash table (hash chain anchors). */
  MSize strmask;        /* String hash mask (size of hash table - 1). */
  MSize strnum;         /* Number of strings in hash table. */
  lua_Alloc allocf;     /* Memory allocator. */
  void *allocd;         /* Memory allocator data. */
  GCState gc;           /* Garbage collector. */
  volatile int32_t vmstate;  /* VM state or current JIT code trace number. */
  SBuf tmpbuf;          /* Temporary string buffer. */
  GCstr strempty;       /* Empty string. */
  uint8_t stremptyz;    /* Zero terminator of empty string. */
  uint8_t hookmask;     /* Hook mask. */
  uint8_t dispatchmode;     /* Dispatch mode. */
  uint8_t vmevmask;     /* VM event mask. */
  GCRef mainthref;      /* Link to main thread. */
  TValue registrytv;    /* Anchor for registry. */
  TValue tmptv, tmptv2;     /* Temporary TValues. */
  Node nilnode;         /* Fallback 1-element hash part (nil key and value). */
  GCupval uvhead;       /* Head of double-linked list of all open upvalues. */
  int32_t hookcount;    /* Instruction hook countdown. */
  int32_t hookstart;    /* Start count for instruction hook counter. */
  lua_Hook hookf;       /* Hook function. */
  lua_CFunction wrapf;      /* Wrapper for C function calls. */
  lua_CFunction panic;      /* Called as a last resort for errors. */
  BCIns bc_cfunc_int;       /* Bytecode for internal C function calls. */
  BCIns bc_cfunc_ext;       /* Bytecode for external C function calls. */
  GCRef cur_L;          /* Currently executing lua_State. */
  MRef jit_base;        /* Current JIT code L->base or NULL. */
  MRef ctype_state;     /* Pointer to C type state. */
  GCRef gcroot[GCROOT_MAX];  /* GC roots. */
} global_State;

#define mainthread(g)       (&gcref(g->mainthref)->th)
#define niltv(L) \
  check_exp(tvisnil(&G(L)->nilnode.val), &G(L)->nilnode.val)
#define niltvg(g) \
  check_exp(tvisnil(&(g)->nilnode.val), &(g)->nilnode.val)

/* Hook management. Hook event masks are defined in lua.h. */
#define HOOK_EVENTMASK              0x0f
#define HOOK_ACTIVE               0x10
#define HOOK_ACTIVE_SHIFT       4
#define HOOK_VMEVENT              0x20
#define HOOK_GC                     0x40
#define HOOK_PROFILE              0x80
#define hook_active(g)              ((g)->hookmask & HOOK_ACTIVE)
#define hook_enter(g)               ((g)->hookmask |= HOOK_ACTIVE)
#define hook_entergc(g)               ((g)->hookmask |= (HOOK_ACTIVE|HOOK_GC))
#define hook_vmevent(g)               ((g)->hookmask |= (HOOK_ACTIVE|HOOK_VMEVENT))
#define hook_leave(g)               ((g)->hookmask &= ~HOOK_ACTIVE)
#define hook_save(g)              ((g)->hookmask & ~HOOK_EVENTMASK)
#define hook_restore(g, h) \
  ((g)->hookmask = ((g)->hookmask & HOOK_EVENTMASK) | (h))

/* Per-thread state object. */
struct lua_State {
  GCHeader;
  uint8_t dummy_ffid;       /* Fake FF_C for curr_funcisL() on dummy frames. */
  uint8_t status;       /* Thread status. */
  MRef glref;           /* Link to global state. */
  GCRef gclist;             /* GC chain. */
  TValue *base;             /* Base of currently executing function. */
  TValue *top;              /* First free slot in the stack. */
  MRef maxstack;        /* Last free slot in the stack. */
  MRef stack;           /* Stack base. */
  GCRef openupval;      /* List of open upvalues in the stack. */
  GCRef env;            /* Thread environment (table of globals). */
  void *cframe;             /* End of C stack frame chain. */
  MSize stacksize;      /* True stack size (incl. LJ_STACK_EXTRA). */
};
```

```c
#define G(L)                        (mref(L->glref, global_State))
#define registry(L)                (&G(L)->registrytv)

/* Macros to access the currently executing (Lua) function. */
#if LJ_GC64
#define curr_func(L)                (&gcval(L->base-2)->fn)
#elif LJ_FR2
#define curr_func(L)                (&gcref((L->base-2)->gcr)->fn)
#else
#define curr_func(L)                (&gcref((L->base-1)->fr.func)->fn)
#endif
#define curr_funcisL(L)               (isluafunc(curr_func(L)))
#define curr_proto(L)                (funcproto(curr_func(L)))
#define curr_topL(L)                (L->base + curr_proto(L)->framesize)
#define curr_top(L)                (curr_funcisL(L) ? curr_topL(L) : L->top)

/* -- GC object definition and conversions ----------------------------- */

/* GC header for generic access to common fields of GC objects. */
typedef struct GChead {
  GCHeader;
  uint8_t unused1;
  uint8_t unused2;
  GCRef env;
  GCRef gclist;
  GCRef metatable;
} GChead;

/* The env field SHOULD be at the same offset for all GC objects. */
LJ_STATIC_ASSERT(offsetof(GChead, env) == offsetof(GCfuncL, env));
LJ_STATIC_ASSERT(offsetof(GChead, env) == offsetof(GCudata, env));

/* The metatable field MUST be at the same offset for all GC objects. */
LJ_STATIC_ASSERT(offsetof(GChead, metatable) == offsetof(GCtab, metatable));
LJ_STATIC_ASSERT(offsetof(GChead, metatable) == offsetof(GCudata, metatable));

/* The gclist field MUST be at the same offset for all GC objects. */
LJ_STATIC_ASSERT(offsetof(GChead, gclist) == offsetof(lua_State, gclist));
LJ_STATIC_ASSERT(offsetof(GChead, gclist) == offsetof(GCproto, gclist));
LJ_STATIC_ASSERT(offsetof(GChead, gclist) == offsetof(GCfuncL, gclist));
LJ_STATIC_ASSERT(offsetof(GChead, gclist) == offsetof(GCtab, gclist));

typedef union GCobj {
  GChead gch;
  GCstr str;
  GCupval uv;
  lua_State th;
  GCproto pt;
  GCfunc fn;
  GCcdata cd;
  GCtab tab;
  GCudata ud;
} GCobj;

/* Macros to convert a GCobj pointer into a specific value. */
#define gco2str(o)        check_exp((o)->gch.gct == ~LJ_TSTR, &(o)->str)
#define gco2uv(o)        check_exp((o)->gch.gct == ~LJ_TUPVAL, &(o)->uv)
#define gco2th(o)        check_exp((o)->gch.gct == ~LJ_TTHREAD, &(o)->th)
#define gco2pt(o)        check_exp((o)->gch.gct == ~LJ_TPROTO, &(o)->pt)
#define gco2func(o)        check_exp((o)->gch.gct == ~LJ_TFUNC, &(o)->fn)
#define gco2cd(o)        check_exp((o)->gch.gct == ~LJ_TCDATA, &(o)->cd)
#define gco2tab(o)        check_exp((o)->gch.gct == ~LJ_TTAB, &(o)->tab)
#define gco2ud(o)        check_exp((o)->gch.gct == ~LJ_TUDATA, &(o)->ud)

/* Macro to convert any collectable object into a GCobj pointer. */
#define obj2gco(v)        ((GCobj *)(v))

/* -- TValue getters/setters ------------------------------------------- */

#ifdef LUA_USE_ASSERT
#include "lj_gc.h"
#endif

/* Macros to test types. */
#if LJ_GC64
#define itype(o)        ((uint32_t)((o)->it64 >> 47))
```

```
#define tvisnil(o)        ((o)->it64 == -1)
#else
#define itype(o)        ((o)->it)
#define tvisnil(o)        (itype(o) == LJ_TNIL)
#endif
#define tvisfalse(o)        (itype(o) == LJ_TFALSE)
#define tvistrue(o)        (itype(o) == LJ_TTRUE)
#define tvisbool(o)        (tvisfalse(o) || tvistrue(o))
#if LJ_64 && !LJ_GC64
#define tvislightud(o)        (((int32_t)itype(o) >> 15) == -2)
#else
#define tvislightud(o)        (itype(o) == LJ_TLIGHTUD)
#endif
#define tvisstr(o)        (itype(o) == LJ_TSTR)
#define tvisfunc(o)        (itype(o) == LJ_TFUNC)
#define tvisthread(o)        (itype(o) == LJ_TTHREAD)
#define tvisproto(o)        (itype(o) == LJ_TPROTO)
#define tviscdata(o)        (itype(o) == LJ_TCDATA)
#define tvistab(o)        (itype(o) == LJ_TTAB)
#define tvisudata(o)        (itype(o) == LJ_TUDATA)
#define tvisnumber(o)        (itype(o) <= LJ_TISNUM)
#define tvisint(o)        (LJ_DUALNUM && itype(o) == LJ_TISNUM)
#define tvisnum(o)        (itype(o) < LJ_TISNUM)

#define tvistruecond(o)        (itype(o) < LJ_TISTRUECOND)
#define tvispri(o)        (itype(o) >= LJ_TISPRI)
#define tvistabud(o)        (itype(o) <= LJ_TISTABUD)  /* && !tvisnum() */
#define tvisgcv(o)        ((itype(o) - LJ_TISGCV) > (LJ_TNUMX - LJ_TISGCV))

/* Special macros to test numbers for NaN, +0, -0, +1 and raw equality. */
#define tvisnan(o)        ((o)->n != (o)->n)
#if LJ_64
#define tviszero(o)        (((o)->u64 << 1) == 0)
#else
#define tviszero(o)        (((o)->u32.lo | ((o)->u32.hi << 1)) == 0)
#endif
#define tvispzero(o)        ((o)->u64 == 0)
#define tvismzero(o)        ((o)->u64 == U64x(80000000,00000000))
#define tvispone(o)        ((o)->u64 == U64x(3ff00000,00000000))
#define rawnumequal(o1, o2)        ((o1)->u64 == (o2)->u64)

/* Macros to convert type ids. */
#if LJ_64 && !LJ_GC64
#define itypemap(o) \
  (tvisnumber(o) ? ~LJ_TNUMX : tvislightud(o) ? ~LJ_TLIGHTUD : ~itype(o))
#else
#define itypemap(o)        (tvisnumber(o) ? ~LJ_TNUMX : ~itype(o))
#endif

/* Macros to get tagged values. */
#if LJ_GC64
#define gcval(o)        ((GCobj *)(gcrefu((o)->gcr) & LJ_GCVMASK))
#else
#define gcval(o)        (gcref((o)->gcr))
#endif
#define boolV(o)        check_exp(tvisbool(o), (LJ_TFALSE - itype(o)))
#if LJ_64
#define lightudV(o) \
  check_exp(tvislightud(o), (void *)((o)->u64 & U64x(00007fff,ffffffff)))
#else
#define lightudV(o)        check_exp(tvislightud(o), gcrefp((o)->gcr, void))
#endif
#define gcV(o)                check_exp(tvisgcv(o), gcval(o))
#define strV(o)                check_exp(tvisstr(o), &gcval(o)->str)
#define funcV(o)        check_exp(tvisfunc(o), &gcval(o)->fn)
#define threadV(o)        check_exp(tvisthread(o), &gcval(o)->th)
#define protoV(o)        check_exp(tvisproto(o), &gcval(o)->pt)
#define cdataV(o)        check_exp(tviscdata(o), &gcval(o)->cd)
#define tabV(o)                check_exp(tvistab(o), &gcval(o)->tab)
#define udataV(o)        check_exp(tvisudata(o), &gcval(o)->ud)
#define numV(o)                check_exp(tvisnum(o), (o)->n)
#define intV(o)                check_exp(tvisint(o), (int32_t)(o)->i)

/* Macros to set tagged values. */
#if LJ_GC64
#define setitype(o, i)                ((o)->it = ((i) << 15))
```

```
#define setnilV(o)               ((o)->it64 = -1)
#define setpriV(o, x)               ((o)->it64 = (int64_t)~((uint64_t)~(x)<<47))
#define setboolV(o, x)               ((o)->it64 = (int64_t)~((uint64_t)((x)+1)<<47))
#else
#define setitype(o, i)               ((o)->it = (i))
#define setnilV(o)               ((o)->it = LJ_TNIL)
#define setboolV(o, x)               ((o)->it = LJ_TFALSE-(uint32_t)(x))
#define setpriV(o, i)               (setitype((o), (i)))
#endif

static LJ_AINLINE void setlightudV(TValue *o, void *p)
{
#if LJ_GC64
  o->u64 = (uint64_t)p | (((uint64_t)LJ_TLIGHTUD) << 47);
#elif LJ_64
  o->u64 = (uint64_t)p | (((uint64_t)0xffff) << 48);
#else
  setgcrefp(o->gcr, p); setitype(o, LJ_TLIGHTUD);
#endif
}

#if LJ_64
#define checklightudptr(L, p) \
  (((uint64_t)(p) >> 47) ? (lj_err_msg(L, LJ_ERR_BADLU), NULL) : (p))
#else
#define checklightudptr(L, p)        (p)
#endif

#if LJ_FR2
#define setcont(o, f)               ((o)->u64 = (uint64_t)(uintptr_t)(void *)(f))
#elif LJ_64
#define setcont(o, f) \
  ((o)->u64 = (uint64_t)(void *)(f) - (uint64_t)lj_vm_asm_begin)
#else
#define setcont(o, f)               setlightudV((o), (void *)(f))
#endif

#define tvchecklive(L, o) \
  UNUSED(L), lua_assert(!tvisgcv(o) || \
  ((~itype(o) == gcval(o)->gch.gct) && !isdead(G(L), gcval(o))))

static LJ_AINLINE void setgcVraw(TValue *o, GCobj *v, uint32_t itype)
{
#if LJ_GC64
  setgcreft(o->gcr, v, itype);
#else
  setgcref(o->gcr, v); setitype(o, itype);
#endif
}

static LJ_AINLINE void setgcV(lua_State *L, TValue *o, GCobj *v, uint32_t it)
{
  setgcVraw(o, v, it); tvchecklive(L, o);
}

#define define_setV(name, type, tag) \
static LJ_AINLINE void name(lua_State *L, TValue *o, type *v) \
{ \
  setgcV(L, o, obj2gco(v), tag); \
}
define_setV(setstrV, GCstr, LJ_TSTR)
define_setV(setthreadV, lua_State, LJ_TTHREAD)
define_setV(setprotoV, GCproto, LJ_TPROTO)
define_setV(setfuncV, GCfunc, LJ_TFUNC)
define_setV(setcdataV, GCcdata, LJ_TCDATA)
define_setV(settabV, GCtab, LJ_TTAB)
define_setV(setudataV, GCudata, LJ_TUDATA)

#define setnumV(o, x)               ((o)->n = (x))
#define setnanV(o)               ((o)->u64 = U64x(fff80000,00000000))
#define setpinfV(o)               ((o)->u64 = U64x(7ff00000,00000000))
#define setminfV(o)               ((o)->u64 = U64x(fff00000,00000000))

static LJ_AINLINE void setintV(TValue *o, int32_t i)
{
#if LJ_DUALNUM
```

```
  o->i = (uint32_t)i; setitype(o, LJ_TISNUM);
#else
  o->n = (lua_Number)i;
#endif
}

static LJ_AINLINE void setint64V(TValue *o, int64_t i)
{
  if (LJ_DUALNUM && LJ_LIKELY(i == (int64_t)(int32_t)i))
    setintV(o, (int32_t)i);
  else
    setnumV(o, (lua_Number)i);
}

#if LJ_64
#define setintptrV(o, i)        setint64V((o), (i))
#else
#define setintptrV(o, i)        setintV((o), (i))
#endif

/* Copy tagged values. */
static LJ_AINLINE void copyTV(lua_State *L, TValue *o1, const TValue *o2)
{
  *o1 = *o2; tvchecklive(L, o1);
}

/* -- Number to integer conversion ---------------------------------------- */

#if LJ_SOFTFP
LJ_ASMF int32_t lj_vm_tobit(double x);
#endif

static LJ_AINLINE int32_t lj_num2bit(lua_Number n)
{
#if LJ_SOFTFP
  return lj_vm_tobit(n);
#else
  TValue o;
  o.n = n + 6755399441055744.0;  /* 2^52 + 2^51 */
  return (int32_t)o.u32.lo;
#endif
}

#define lj_num2int(n)   ((int32_t)(n))

static LJ_AINLINE uint64_t lj_num2u64(lua_Number n)
{
#ifdef _MSC_VER
  if (n >= 9223372036854775808.0)  /* They think it's a feature. */
    return (uint64_t)(int64_t)(n - 18446744073709551616.0);
  else
#endif
    return (uint64_t)n;
}

static LJ_AINLINE int32_t numberVint(cTValue *o)
{
  if (LJ_LIKELY(tvisint(o)))
    return intV(o);
  else
    return lj_num2int(numV(o));
}

static LJ_AINLINE lua_Number numberVnum(cTValue *o)
{
  if (LJ_UNLIKELY(tvisint(o)))
    return (lua_Number)intV(o);
  else
    return numV(o);
}

/* -- Miscellaneous object handling --------------------------------------- */

/* Names and maps for internal and external object tags. */
LJ_DATA const char *const lj_obj_typename[1+LUA_TCDATA+1];
LJ_DATA const char *const lj_obj_itypename[~LJ_TNUMX+1];
```

```
969
970 #define lj_typename(o)        (lj_obj_itypename[itypemap(o)])
971
972 /* Compare two objects without calling metamethods. */
973 LJ_FUNC int LJ_FASTCALL lj_obj_equal(cTValue *o1, cTValue *o2);
974 LJ_FUNC const void * LJ_FASTCALL lj_obj_ptr(cTValue *o);
975
976 #endif
```

One Level Up Top Level

src/lj_tab.c - luajit-2.0-src

Functions defined

- unbound_search

Macros defined

- LUA_CORE
- hashgcref
- hashgcref
- hashlohi
- hashnum
- hashptr
- hashstr
- lj_tab_c

Source code

```
/*
** Table handling.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Major portions taken verbatim or adapted from the Lua interpreter.
** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
*/

#define lj_tab_c
 #define LUA_CORE

 #include "lj_obj.h"
 #include "lj_gc.h"
 #include "lj_err.h"
 #include "lj_tab.h"

 /* -- Object hashing ------------------------------------------------------ */

 /* Hash values are masked with the table hash mask and used as an index. */
 static LJ_AINLINE Node *hashmask(const GCtab *t, uint32_t hash)
 {
   Node *n = noderef(t->node);
   return &n[hash & t->hmask];
 }

 /* String hashes are precomputed when they are interned. */
 #define hashstr(t, s)               hashmask(t, (s)->hash)

 #define hashlohi(t, lo, hi)        hashmask((t), hashrot((lo), (hi)))
 #define hashnum(t, o)               hashlohi((t), (o)->u32.lo, ((o)->u32.hi << 1))
 #define hashptr(t, p)               hashlohi((t), u32ptr(p), u32ptr(p) + HASH_BIAS)
 #if LJ_GC64
 #define hashgcref(t, r) \
   hashlohi((t), (uint32_t)gcrefu(r), (uint32_t)(gcrefu(r) >> 32))
 #else
 #define hashgcref(t, r)               hashlohi((t), gcrefu(r), gcrefu(r) + HASH_BIAS)
 #endif

 /* Hash an arbitrary key and return its anchor position in the hash table. */
 static Node *hashkey(const GCtab *t, cTValue *key)
 {
   lua_assert(!tvisint(key));
   if (tvisstr(key))
     return hashstr(t, strV(key));
   else if (tvisnum(key))
     return hashnum(t, key);
   else if (tvisbool(key))
     return hashmask(t, boolV(key));
```

```
  else
    return hashgcref(t, key->gcr);
  /* Only hash 32 bits of lightuserdata on a 64 bit CPU. Good enough? */
}

/* -- Table creation and destruction ---------------------------------- */

/* Create new hash part for table. */
static LJ_AINLINE void newhpart(lua_State *L, GCtab *t, uint32_t hbits)
{
  uint32_t hsize;
  Node *node;
  lua_assert(hbits != 0);
  if (hbits > LJ_MAX_HBITS)
    lj_err_msg(L, LJ_ERR_TABOV);
  hsize = 1u << hbits;
  node = lj_mem_newvec(L, hsize, Node);
  setmref(t->node, node);
  setfreetop(t, node, &node[hsize]);
  t->hmask = hsize-1;
}

/*
** Q: Why all of these copies of t->hmask, t->node etc. to local variables?
** A: Because alias analysis for C is _really_ tough.
**    Even state-of-the-art C compilers won't produce good code without this.
*/

/* Clear hash part of table. */
static LJ_AINLINE void clearhpart(GCtab *t)
{
  uint32_t i, hmask = t->hmask;
  Node *node = noderef(t->node);
  lua_assert(t->hmask != 0);
  for (i = 0; i <= hmask; i++) {
    Node *n = &node[i];
    setmref(n->next, NULL);
    setnilV(&n->key);
    setnilV(&n->val);
  }
}

/* Clear array part of table. */
static LJ_AINLINE void clearapart(GCtab *t)
{
  uint32_t i, asize = t->asize;
  TValue *array = tvref(t->array);
  for (i = 0; i < asize; i++)
    setnilV(&array[i]);
}

 /* Create a new table. Note: the slots are not initialized (yet). */
 static GCtab *newtab(lua_State *L, uint32_t asize, uint32_t hbits)
 {
   GCtab *t;
   /* First try to colocate the array part. */
   if (LJ_MAX_COLOSIZE != 0 && asize > 0 && asize <= LJ_MAX_COLOSIZE) {
     Node *nilnode;
     lua_assert((sizeof(GCtab) & 7) == 0);
     t = (GCtab *)lj_mem_newgco(L, sizetabcolo(asize));
     t->gct = ~LJ_TTAB;
     t->nomm = (uint8_t)~0;
     t->colo = (int8_t)asize;
     setmref(t->array, (TValue *)((char *)t + sizeof(GCtab)));
     setgcrefnull(t->metatable);
     t->asize = asize;
     t->hmask = 0;
     nilnode = &G(L)->nilnode;
     setmref(t->node, nilnode);
 #if LJ_GC64
     setmref(t->freetop, nilnode);
 #endif
   } else {  /* Otherwise separately allocate the array part. */
     Node *nilnode;
     t = lj_mem_newobj(L, GCtab);
     t->gct = ~LJ_TTAB;
```

```
    t->nomm = (uint8_t)~0;
    t->colo = 0;
    setmref(t->array, NULL);
    setgcrefnull(t->metatable);
    t->asize = 0;  /* In case the array allocation fails. */
    t->hmask = 0;
    nilnode = &G(L)->nilnode;
    setmref(t->node, nilnode);
#if LJ_GC64
    setmref(t->freetop, nilnode);
#endif
    if (asize > 0) {
      if (asize > LJ_MAX_ASIZE)
        lj_err_msg(L, LJ_ERR_TABOV);
      setmref(t->array, lj_mem_newvec(L, asize, TValue));
      t->asize = asize;
    }
  }
  if (hbits)
    newhpart(L, t, hbits);
  return t;
}

/* Create a new table.
**
** IMPORTANT NOTE: The API differs from lua_createtable()!
**
** The array size is non-inclusive. E.g. asize=128 creates array slots
** for 0..127, but not for 128. If you need slots 1..128, pass asize=129
** (slot 0 is wasted in this case).
**
** The hash size is given in hash bits. hbits=0 means no hash part.
** hbits=1 creates 2 hash slots, hbits=2 creates 4 hash slots and so on.
*/
GCtab *lj_tab_new(lua_State *L, uint32_t asize, uint32_t hbits)
{
  GCtab *t = newtab(L, asize, hbits);
  clearapart(t);
  if (t->hmask > 0) clearhpart(t);
  return t;
}

/* The API of this function conforms to lua_createtable(). */
GCtab *lj_tab_new_ah(lua_State *L, int32_t a, int32_t h)
{
  return lj_tab_new(L, (uint32_t)(a > 0 ? a+1 : 0), hsize2hbits(h));
}

#if LJ_HASJIT
GCtab * LJ_FASTCALL lj_tab_new1(lua_State *L, uint32_t ahsize)
{
  GCtab *t = newtab(L, ahsize & 0xffffff, ahsize >> 24);
  clearapart(t);
  if (t->hmask > 0) clearhpart(t);
  return t;
}
#endif

/* Duplicate a table. */
GCtab * LJ_FASTCALL lj_tab_dup(lua_State *L, const GCtab *kt)
{
  GCtab *t;
  uint32_t asize, hmask;
  t = newtab(L, kt->asize, kt->hmask > 0 ? lj_fls(kt->hmask)+1 : 0);
  lua_assert(kt->asize == t->asize && kt->hmask == t->hmask);
  t->nomm = 0;  /* Keys with metamethod names may be present. */
  asize = kt->asize;
  if (asize > 0) {
    TValue *array = tvref(t->array);
    TValue *karray = tvref(kt->array);
    if (asize < 64) {  /* An inlined loop beats memcpy for < 512 bytes. */
      uint32_t i;
      for (i = 0; i < asize; i++)
        copyTV(L, &array[i], &karray[i]);
    } else {
      memcpy(array, karray, asize*sizeof(TValue));
```

```
    }
  }
  hmask = kt->hmask;
  if (hmask > 0) {
    uint32_t i;
    Node *node = noderef(t->node);
    Node *knode = noderef(kt->node);
    ptrdiff_t d = (char *)node - (char *)knode;
    setfreetop(t, node, (Node *)((char *)getfreetop(kt, knode) + d));
    for (i = 0; i <= hmask; i++) {
      Node *kn = &knode[i];
      Node *n = &node[i];
      Node *next = nextnode(kn);
      /* Don't use copyTV here, since it asserts on a copy of a dead key. */
      n->val = kn->val; n->key = kn->key;
      setmref(n->next, next == NULL? next : (Node *)((char *)next + d));
    }
  }
  return t;
}

/* Clear a table. */
void LJ_FASTCALL lj_tab_clear(GCtab *t)
{
  clearapart(t);
  if (t->hmask > 0) {
    Node *node = noderef(t->node);
    setfreetop(t, node, &node[t->hmask+1]);
    clearhpart(t);
  }
}

/* Free a table. */
void LJ_FASTCALL lj_tab_free(global_State *g, GCtab *t)
{
  if (t->hmask > 0)
    lj_mem_freevec(g, noderef(t->node), t->hmask+1, Node);
  if (t->asize > 0 && LJ_MAX_COLOSIZE != 0 && t->colo <= 0)
    lj_mem_freevec(g, tvref(t->array), t->asize, TValue);
  if (LJ_MAX_COLOSIZE != 0 && t->colo)
    lj_mem_free(g, t, sizetabcolo((uint32_t)t->colo & 0x7f));
  else
    lj_mem_freet(g, t);
}

/* -- Table resizing ------------------------------------------------------ */

/* Resize a table to fit the new array/hash part sizes. */
static void resizetab(lua_State *L, GCtab *t, uint32_t asize, uint32_t hbits)
{
  Node *oldnode = noderef(t->node);
  uint32_t oldasize = t->asize;
  uint32_t oldhmask = t->hmask;
  if (asize > oldasize) {  /* Array part grows? */
    TValue *array;
    uint32_t i;
    if (asize > LJ_MAX_ASIZE)
      lj_err_msg(L, LJ_ERR_TABOV);
    if (LJ_MAX_COLOSIZE != 0 && t->colo > 0) {
      /* A colocated array must be separated and copied. */
      TValue *oarray = tvref(t->array);
      array = lj_mem_newvec(L, asize, TValue);
      t->colo = (int8_t)(t->colo | 0x80);  /* Mark as separated (colo < 0). */
      for (i = 0; i < oldasize; i++)
        copyTV(L, &array[i], &oarray[i]);
    } else {
      array = (TValue *)lj_mem_realloc(L, tvref(t->array),
                           oldasize*sizeof(TValue), asize*sizeof(TValue));
    }
    setmref(t->array, array);
    t->asize = asize;
    for (i = oldasize; i < asize; i++)  /* Clear newly allocated slots. */
      setnilV(&array[i]);
  }
  /* Create new (empty) hash part. */
  if (hbits) {
```

```
    newhpart(L, t, hbits);
    clearhpart(t);
  } else {
    global_State *g = G(L);
    setmref(t->node, &g->nilnode);
#if LJ_GC64
    setmref(t->freetop, &g->nilnode);
#endif
    t->hmask = 0;
  }
  if (asize < oldasize) {  /* Array part shrinks? */
    TValue *array = tvref(t->array);
    uint32_t i;
    t->asize = asize;  /* Note: This 'shrinks' even colocated arrays. */
    for (i = asize; i < oldasize; i++)  /* Reinsert old array values. */
      if (!tvisnil(&array[i]))
        copyTV(L, lj_tab_setinth(L, t, (int32_t)i), &array[i]);
    /* Physically shrink only separated arrays. */
    if (LJ_MAX_COLOSIZE != 0 && t->colo <= 0)
      setmref(t->array, lj_mem_realloc(L, array,
              oldasize*sizeof(TValue), asize*sizeof(TValue)));
  }
  if (oldhmask > 0) {  /* Reinsert pairs from old hash part. */
    global_State *g;
    uint32_t i;
    for (i = 0; i <= oldhmask; i++) {
      Node *n = &oldnode[i];
      if (!tvisnil(&n->val))
        copyTV(L, lj_tab_set(L, t, &n->key), &n->val);
    }
    g = G(L);
    lj_mem_freevec(g, oldnode, oldhmask+1, Node);
  }
}

static uint32_t countint(cTValue *key, uint32_t *bins)
{
  lua_assert(!tvisint(key));
  if (tvisnum(key)) {
    lua_Number nk = numV(key);
    int32_t k = lj_num2int(nk);
    if ((uint32_t)k < LJ_MAX_ASIZE && nk == (lua_Number)k) {
      bins[(k > 2 ? lj_fls((uint32_t)(k-1)) : 0)]++;
      return 1;
    }
  }
  return 0;
}

static uint32_t countarray(const GCtab *t, uint32_t *bins)
{
  uint32_t na, b, i;
  if (t->asize == 0) return 0;
  for (na = i = b = 0; b < LJ_MAX_ABITS; b++) {
    uint32_t n, top = 2u << b;
    TValue *array;
    if (top >= t->asize) {
      top = t->asize-1;
      if (i > top)
        break;
    }
    array = tvref(t->array);
    for (n = 0; i <= top; i++)
      if (!tvisnil(&array[i]))
        n++;
    bins[b] += n;
    na += n;
  }
  return na;
}

static uint32_t counthash(const GCtab *t, uint32_t *bins, uint32_t *narray)
{
  uint32_t total, na, i, hmask = t->hmask;
  Node *node = noderef(t->node);
  for (total = na = 0, i = 0; i <= hmask; i++) {
```

```
    Node *n = &node[i];
    if (!tvisnil(&n->val)) {
      na += countint(&n->key, bins);
      total++;
    }
  }
  *narray += na;
  return total;
}

static uint32_t bestasize(uint32_t bins[], uint32_t *narray)
{
  uint32_t b, sum, na = 0, sz = 0, nn = *narray;
  for (b = 0, sum = 0; 2*nn > (1u<<b) && sum != nn; b++)
    if (bins[b] > 0 && 2*(sum += bins[b]) > (1u<<b)) {
      sz = (2u<<b)+1;
      na = sum;
    }
  *narray = sz;
  return na;
}

static void rehashtab(lua_State *L, GCtab *t, cTValue *ek)
{
  uint32_t bins[LJ_MAX_ABITS];
  uint32_t total, asize, na, i;
  for (i = 0; i < LJ_MAX_ABITS; i++) bins[i] = 0;
  asize = countarray(t, bins);
  total = 1 + asize;
  total += counthash(t, bins, &asize);
  asize += countint(ek, bins);
  na = bestasize(bins, &asize);
  total -= na;
  resizetab(L, t, asize, hsize2hbits(total));
}

#if LJ_HASFFI
void lj_tab_rehash(lua_State *L, GCtab *t)
{
  rehashtab(L, t, niltv(L));
}
#endif

void lj_tab_reasize(lua_State *L, GCtab *t, uint32_t nasize)
{
  resizetab(L, t, nasize+1, t->hmask > 0 ? lj_fls(t->hmask)+1 : 0);
}

/* -- Table getters ------------------------------------------------------- */

cTValue * LJ_FASTCALL lj_tab_getinth(GCtab *t, int32_t key)
{
  TValue k;
  Node *n;
  k.n = (lua_Number)key;
  n = hashnum(t, &k);
  do {
    if (tvisnum(&n->key) && n->key.n == k.n)
      return &n->val;
  } while ((n = nextnode(n)));
  return NULL;
}

cTValue *lj_tab_getstr(GCtab *t, GCstr *key)
{
  Node *n = hashstr(t, key);
  do {
    if (tvisstr(&n->key) && strV(&n->key) == key)
      return &n->val;
  } while ((n = nextnode(n)));
  return NULL;
}

cTValue *lj_tab_get(lua_State *L, GCtab *t, cTValue *key)
{
  if (tvisstr(key)) {
```

```
    cTValue *tv = lj_tab_getstr(t, strV(key));
    if (tv)
      return tv;
  } else if (tvisint(key)) {
    cTValue *tv = lj_tab_getint(t, intV(key));
    if (tv)
      return tv;
  } else if (tvisnum(key)) {
    lua_Number nk = numV(key);
    int32_t k = lj_num2int(nk);
    if (nk == (lua_Number)k) {
      cTValue *tv = lj_tab_getint(t, k);
      if (tv)
        return tv;
    } else {
      goto genlookup;  /* Else use the generic lookup. */
    }
  } else if (!tvisnil(key)) {
    Node *n;
  genlookup:
    n = hashkey(t, key);
    do {
      if (lj_obj_equal(&n->key, key))
        return &n->val;
    } while ((n = nextnode(n)));
  }
  return niltv(L);
}

/* -- Table setters ------------------------------------------------------ */

/* Insert new key. Use Brent's variation to optimize the chain length. */
TValue *lj_tab_newkey(lua_State *L, GCtab *t, cTValue *key)
{
  Node *n = hashkey(t, key);
  if (!tvisnil(&n->val) || t->hmask == 0) {
    Node *nodebase = noderef(t->node);
    Node *collide, *freenode = getfreetop(t, nodebase);
    lua_assert(freenode >= nodebase && freenode <= nodebase+t->hmask+1);
    do {
      if (freenode == nodebase) {  /* No free node found? */
        rehashtab(L, t, key);  /* Rehash table. */
        return lj_tab_set(L, t, key);  /* Retry key insertion. */
      }
    } while (!tvisnil(&(--freenode)->key));
    setfreetop(t, nodebase, freenode);
    lua_assert(freenode != &G(L)->nilnode);
    collide = hashkey(t, &n->key);
    if (collide != n) {  /* Colliding node not the main node? */
      while (noderef(collide->next) != n)  /* Find predecessor. */
        collide = nextnode(collide);
      setmref(collide->next, freenode);  /* Relink chain. */
      /* Copy colliding node into free node and free main node. */
      freenode->val = n->val;
      freenode->key = n->key;
      freenode->next = n->next;
      setmref(n->next, NULL);
      setnilV(&n->val);
      /* Rechain pseudo-resurrected string keys with colliding hashes. */
      while (nextnode(freenode)) {
        Node *nn = nextnode(freenode);
        if (tvisstr(&nn->key) && !tvisnil(&nn->val) &&
            hashstr(t, strV(&nn->key)) == n) {
          freenode->next = nn->next;
          nn->next = n->next;
          setmref(n->next, nn);
        } else {
          freenode = nn;
        }
      }
    } else {  /* Otherwise use free node. */
      setmrefr(freenode->next, n->next);  /* Insert into chain. */
      setmref(n->next, freenode);
      n = freenode;
    }
  }
```

```
  n->key.u64 = key->u64;
  if (LJ_UNLIKELY(tvismzero(&n->key)))
    n->key.u64 = 0;
  lj_gc_anybarriert(L, t);
  lua_assert(tvisnil(&n->val));
  return &n->val;
}

TValue *lj_tab_setinth(lua_State *L, GCtab *t, int32_t key)
{
  TValue k;
  Node *n;
  k.n = (lua_Number)key;
  n = hashnum(t, &k);
  do {
    if (tvisnum(&n->key) && n->key.n == k.n)
      return &n->val;
  } while ((n = nextnode(n)));
  return lj_tab_newkey(L, t, &k);
}

TValue *lj_tab_setstr(lua_State *L, GCtab *t, GCstr *key)
{
  TValue k;
  Node *n = hashstr(t, key);
  do {
    if (tvisstr(&n->key) && strV(&n->key) == key)
      return &n->val;
  } while ((n = nextnode(n)));
  setstrV(L, &k, key);
  return lj_tab_newkey(L, t, &k);
}

TValue *lj_tab_set(lua_State *L, GCtab *t, cTValue *key)
{
  Node *n;
  t->nomm = 0;  /* Invalidate negative metamethod cache. */
  if (tvisstr(key)) {
    return lj_tab_setstr(L, t, strV(key));
  } else if (tvisint(key)) {
    return lj_tab_setint(L, t, intV(key));
  } else if (tvisnum(key)) {
    lua_Number nk = numV(key);
    int32_t k = lj_num2int(nk);
    if (nk == (lua_Number)k)
      return lj_tab_setint(L, t, k);
    if (tvisnan(key))
      lj_err_msg(L, LJ_ERR_NANIDX);
    /* Else use the generic lookup. */
  } else if (tvisnil(key)) {
    lj_err_msg(L, LJ_ERR_NILIDX);
  }
  n = hashkey(t, key);
  do {
    if (lj_obj_equal(&n->key, key))
      return &n->val;
  } while ((n = nextnode(n)));
  return lj_tab_newkey(L, t, key);
}

/* -- Table traversal ---------------------------------------------------- */

/* Get the traversal index of a key. */
static uint32_t keyindex(lua_State *L, GCtab *t, cTValue *key)
{
  TValue tmp;
  if (tvisint(key)) {
    int32_t k = intV(key);
    if ((uint32_t)k < t->asize)
      return (uint32_t)k;  /* Array key indexes: [0..t->asize-1] */
    setnumV(&tmp, (lua_Number)k);
    key = &tmp;
  } else if (tvisnum(key)) {
    lua_Number nk = numV(key);
    int32_t k = lj_num2int(nk);
    if ((uint32_t)k < t->asize && nk == (lua_Number)k)
```

```
      return (uint32_t)k;  /* Array key indexes: [0..t->asize-1] */
  }
  if (!tvisnil(key)) {
    Node *n = hashkey(t, key);
    do {
      if (lj_obj_equal(&n->key, key))
        return t->asize + (uint32_t)(n - noderef(t->node));
        /* Hash key indexes: [t->asize..t->asize+t->nmask] */
    } while ((n = nextnode(n)));
    if (key->u32.hi == 0xfffe7fff)  /* ITERN was despecialized while running. */
      return key->u32.lo - 1;
    lj_err_msg(L, LJ_ERR_NEXTIDX);
    return 0;  /* unreachable */
  }
  return ~0u;  /* A nil key starts the traversal. */
}

/* Advance to the next step in a table traversal. */
int lj_tab_next(lua_State *L, GCtab *t, TValue *key)
{
  uint32_t i = keyindex(L, t, key);  /* Find predecessor key index. */
  for (i++; i < t->asize; i++)  /* First traverse the array keys. */
    if (!tvisnil(arrayslot(t, i))) {
      setintV(key, i);
      copyTV(L, key+1, arrayslot(t, i));
      return 1;
    }
  for (i -= t->asize; i <= t->hmask; i++) {  /* Then traverse the hash keys. */
    Node *n = &noderef(t->node)[i];
    if (!tvisnil(&n->val)) {
      copyTV(L, key, &n->key);
      copyTV(L, key+1, &n->val);
      return 1;
    }
  }
  return 0;  /* End of traversal. */
}

/* -- Table length calculation ---------------------------------------- */

static MSize unbound_search(GCtab *t, MSize j)
{
  cTValue *tv;
  MSize i = j;  /* i is zero or a present index */
  j++;
  /* find `i' and `j' such that i is present and j is not */
  while ((tv = lj_tab_getint(t, (int32_t)j)) && !tvisnil(tv)) {
    i = j;
    j *= 2;
    if (j > (MSize)(INT_MAX-2)) {  /* overflow? */
      /* table was built with bad purposes: resort to linear search */
      i = 1;
      while ((tv = lj_tab_getint(t, (int32_t)i)) && !tvisnil(tv)) i++;
      return i - 1;
    }
  }
  /* now do a binary search between them */
  while (j - i > 1) {
    MSize m = (i+j)/2;
    cTValue *tvb = lj_tab_getint(t, (int32_t)m);
    if (tvb && !tvisnil(tvb)) i = m; else j = m;
  }
  return i;
}

/*
** Try to find a boundary in table `t'. A `boundary' is an integer index
** such that t[i] is non-nil and t[i+1] is nil (and 0 if t[1] is nil).
*/
MSize LJ_FASTCALL lj_tab_len(GCtab *t)
{
  MSize j = (MSize)t->asize;
  if (j > 1 && tvisnil(arrayslot(t, j-1))) {
    MSize i = 1;
    while (j - i > 1) {
      MSize m = (i+j)/2;
```

```
657       if (tvisnil(arrayslot(t, m-1))) j = m; else i = m;
658     }
659     return i-1;
660   }
661   if (j) j--;
662   if (t->hmask <= 0)
663     return j;
664   return unbound_search(t, j);
665 }
666
```

One Level Up Top Level

src/lj_tab.h - luajit-2.0-src

Functions defined

- hashrot

Macros defined

- HASH_BIAS
- HASH_ROT1
- HASH_ROT2
- HASH_ROT3
- _LJ_TAB_H
- arrayslot
- hsize2hbits
- inarray
- lj_tab_getint
- lj_tab_setint

Source code

```
1  /*
2  ** Table handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_TAB_H
7  #define _LJ_TAB_H
8
9  #include "lj_obj.h"
10
11  /* Hash constants. Tuned using a brute force search. */
12  #define HASH_BIAS        (-0x04c11db7)
13  #define HASH_ROT1        14
14  #define HASH_ROT2        5
15  #define HASH_ROT3        13
16
17  /* Scramble the bits of numbers and pointers. */
18  static LJ_AINLINE uint32_t hashrot(uint32_t lo, uint32_t hi)
19  {
20  #if LJ_TARGET_X86ORX64
21    /* Prefer variant that compiles well for a 2-operand CPU. */
22    lo ^= hi; hi = lj_rol(hi, HASH_ROT1);
23    lo -= hi; hi = lj_rol(hi, HASH_ROT2);
24    hi ^= lo; hi -= lj_rol(lo, HASH_ROT3);
25  #else
26    lo ^= hi;
27    lo = lo - lj_rol(hi, HASH_ROT1);
28    hi = lo ^ lj_rol(hi, HASH_ROT1 + HASH_ROT2);
29    hi = hi - lj_rol(lo, HASH_ROT3);
30  #endif
31    return hi;
32  }
33
34  #define hsize2hbits(s)        ((s) ? ((s)==1 ? 1 : 1+lj_fls((uint32_t)((s)-1))) : 0)
35
36  LJ_FUNCA GCtab *lj_tab_new(lua_State *L, uint32_t asize, uint32_t hbits);
```

```
LJ_FUNC GCtab *lj_tab_new_ah(lua_State *L, int32_t a, int32_t h);
#if LJ_HASJIT
LJ_FUNC GCtab * LJ_FASTCALL lj_tab_new1(lua_State *L, uint32_t ahsize);
#endif
LJ_FUNCA GCtab * LJ_FASTCALL lj_tab_dup(lua_State *L, const GCtab *kt);
LJ_FUNC void LJ_FASTCALL lj_tab_clear(GCtab *t);
LJ_FUNC void LJ_FASTCALL lj_tab_free(global_State *g, GCtab *t);
#if LJ_HASFFI
LJ_FUNC void lj_tab_rehash(lua_State *L, GCtab *t);
#endif
LJ_FUNCA void lj_tab_reasize(lua_State *L, GCtab *t, uint32_t nasize);

/* Caveat: all getters except lj_tab_get() can return NULL! */

LJ_FUNCA cTValue * LJ_FASTCALL lj_tab_getinth(GCtab *t, int32_t key);
LJ_FUNC cTValue *lj_tab_getstr(GCtab *t, GCstr *key);
LJ_FUNCA cTValue *lj_tab_get(lua_State *L, GCtab *t, cTValue *key);

/* Caveat: all setters require a write barrier for the stored value. */

LJ_FUNCA TValue *lj_tab_newkey(lua_State *L, GCtab *t, cTValue *key);
LJ_FUNCA TValue *lj_tab_setinth(lua_State *L, GCtab *t, int32_t key);
LJ_FUNC TValue *lj_tab_setstr(lua_State *L, GCtab *t, GCstr *key);
LJ_FUNC TValue *lj_tab_set(lua_State *L, GCtab *t, cTValue *key);

#define inarray(t, key)               ((MSize)(key) < (MSize)(t)->asize)
#define arrayslot(t, i)               (&tvref((t)->array)[(i)])
#define lj_tab_getint(t, key) \
  (inarray((t), (key)) ? arrayslot((t), (key)) : lj_tab_getinth((t), (key)))
#define lj_tab_setint(L, t, key) \
  (inarray((t), (key)) ? arrayslot((t), (key)) : lj_tab_setinth(L, (t), (key)))

LJ_FUNCA int lj_tab_next(lua_State *L, GCtab *t, TValue *key);
LJ_FUNCA MSize LJ_FASTCALL lj_tab_len(GCtab *t);

#endif
```

One Level Up Top Level

src/lj_err.c - luajit-2.0-src

Global variables defined

- lj_err_allmsg
- static_uex
- static_uex

Data types defined

- UndocumentedDispatcherContext
- UndocumentedDispatcherContext
- _Unwind_Context
- _Unwind_Exception
- _Unwind_Exception

Functions defined

- RtlUnwindEx_FIXED
- _Unwind_GetGR
- _Unwind_SetGR
- err_argmsg
- err_msgv
- err_raise_ext
- err_raise_ext
- err_unwind
- finderrfunc
- lj_err_arg
- lj_err_argt
- lj_err_argtype
- lj_err_argv
- lj_err_caller
- lj_err_callermsg
- lj_err_callerv
- lj_err_comp
- lj_err_lex
- lj_err_mem

- lj_err_msg
- lj_err_optype
- lj_err_optype_call
- lj_err_run
- lj_err_str
- lj_err_throw
- lj_err_unwind_arm
- lj_err_unwind_dwarf
- luaL_argerror
- luaL_error
- luaL_typerror
- luaL_where
- lua_atpanic
- lua_error
- unwindstack

Macros defined

- ERRDEF
- LJ_EXCODE
- LJ_EXCODE_CHECK
- LJ_EXCODE_ERRCODE
- LJ_EXCODE_MAKE
- LJ_GCC_EXCODE
- LJ_MSVC_EXCODE
- LJ_UEXCLASS
- LJ_UEXCLASS_CHECK
- LJ_UEXCLASS_ERRCODE
- LJ_UEXCLASS_MAKE
- LJ_UNWIND_EXT
- LJ_UNWIND_EXT
- LUA_CORE
- RtlUnwindEx
- WIN32_LEAN_AND_MEAN
- _UA_CLEANUP_PHASE

- _UA_FORCE_UNWIND
- _UA_HANDLER_FRAME
- _UA_SEARCH_PHASE
- _URC_CONTINUE_UNWIND
- _URC_FAILURE
- _URC_FATAL_PHASE1_ERROR
- _URC_HANDLER_FOUND
- _URC_INSTALL_CONTEXT
- _URC_OK
- _US_ACTION_MASK
- _US_FORCE_UNWIND
- _US_UNWIND_FRAME_STARTING
- _US_VIRTUAL_UNWIND_FRAME
- lj_err_c

Source code

```
/*
** Error handling.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_err_c
#define LUA_CORE

#include "lj_obj.h"
 #include "lj_err.h"
 #include "lj_debug.h"
 #include "lj_str.h"
 #include "lj_func.h"
 #include "lj_state.h"
 #include "lj_frame.h"
 #include "lj_ff.h"
 #include "lj_trace.h"
 #include "lj_vm.h"
 #include "lj_strfmt.h"

 /*
 ** LuaJIT can either use internal or external frame unwinding:
 **
 ** - Internal frame unwinding (INT) is free-standing and doesn't require
 **   any OS or library support.
 **
 ** - External frame unwinding (EXT) uses the system-provided unwind handler.
 **
 ** Pros and Cons:
 **
 ** - EXT requires unwind tables for *all* functions on the C stack between
 **   the pcall/catch and the error/throw. This is the default on x64,
 **   but needs to be manually enabled on x86/PPC for non-C++ code.
 **
 ** - INT is faster when actually throwing errors (but this happens rarely).
 **   Setting up error handlers is zero-cost in any case.
 **
 ** - EXT provides full interoperability with C++ exceptions. You can throw
 **   Lua errors or C++ exceptions through a mix of Lua frames and C++ frames.
 **   C++ destructors are called as needed. C++ exceptions caught by pcall
 **   are converted to the string "C++ exception". Lua errors can be caught
```

```
**   with catch (...) in C++.
**
** - INT has only limited support for automatically catching C++ exceptions
**   on POSIX systems using DWARF2 stack unwinding. Other systems may use
**   the wrapper function feature. Lua errors thrown through C++ frames
**   cannot be caught by C++ code and C++ destructors are not run.
**
** EXT is the default on x64 systems, INT is the default on all other systems.
**
** EXT can be manually enabled on POSIX systems using GCC and DWARF2 stack
** unwinding with -DLUAJIT_UNWIND_EXTERNAL. *All* C code must be compiled
** with -funwind-tables (or -fexceptions). This includes LuaJIT itself (set
** TARGET_CFLAGS), all of your C/Lua binding code, all loadable C modules
** and all C libraries that have callbacks which may be used to call back
** into Lua. C++ code must *not* be compiled with -fno-exceptions.
**
** EXT cannot be enabled on WIN32 since system exceptions use code-driven SEH.
** EXT is mandatory on WIN64 since the calling convention has an abundance
** of callee-saved registers (rbx, rbp, rsi, rdi, r12-r15, xmm6-xmm15).
** EXT is mandatory on POSIX/x64 since the interpreter doesn't save r12/r13.
*/

#if defined(__GNUC__) && (LJ_TARGET_X64 || defined(LUAJIT_UNWIND_EXTERNAL))
#define LJ_UNWIND_EXT		1
#elif LJ_TARGET_X64 && LJ_TARGET_WINDOWS
#define LJ_UNWIND_EXT		1
#endif

/* -- Error messages ------------------------------------------------------ */

/* Error message strings. */
LJ_DATADEF const char *lj_err_allmsg =
#define ERRDEF(name, msg)	msg "\0"
#include "lj_errmsg.h"
;

/* -- Internal frame unwinding --------------------------------------------- */

/* Unwind Lua stack and move error message to new top. */
LJ_NOINLINE static void unwindstack(lua_State *L, TValue *top)
{
  lj_func_closeuv(L, top);
  if (top < L->top-1) {
    copyTV(L, top, L->top-1);
    L->top = top+1;
  }
  lj_state_relimitstack(L);
}

/* Unwind until stop frame. Optionally cleanup frames. */
static void *err_unwind(lua_State *L, void *stopcf, int errcode)
{
  TValue *frame = L->base-1;
  void *cf = L->cframe;
  while (cf) {
    int32_t nres = cframe_nres(cframe_raw(cf));
    if (nres < 0) {  /* C frame without Lua frame? */
      TValue *top = restorestack(L, -nres);
      if (frame < top) {  /* Frame reached? */
        if (errcode) {
          L->base = frame+1;
          L->cframe = cframe_prev(cf);
          unwindstack(L, top);
        }
        return cf;
      }
    }
    if (frame <= tvref(L->stack)+LJ_FR2)
      break;
    switch (frame_typep(frame)) {
    case FRAME_LUA:  /* Lua frame. */
    case FRAME_LUAP:
      frame = frame_prevl(frame);
      break;
    case FRAME_C:  /* C frame. */
    unwind_c:
```

```
#if LJ_UNWIND_EXT
      if (errcode) {
        L->base = frame_prevd(frame) + 1;
        L->cframe = cframe_prev(cf);
        unwindstack(L, frame);
      } else if (cf != stopcf) {
        cf = cframe_prev(cf);
        frame = frame_prevd(frame);
        break;
      }
      return NULL;  /* Continue unwinding. */
#else
      UNUSED(stopcf);
      cf = cframe_prev(cf);
      frame = frame_prevd(frame);
      break;
#endif
    case FRAME_CP:  /* Protected C frame. */
      if (cframe_canyield(cf)) {  /* Resume? */
        if (errcode) {
          hook_leave(G(L));  /* Assumes nobody uses coroutines inside hooks. */
          L->cframe = NULL;
          L->status = (uint8_t)errcode;
        }
        return cf;
      }
      if (errcode) {
        L->base = frame_prevd(frame) + 1;
        L->cframe = cframe_prev(cf);
        unwindstack(L, frame);
      }
      return cf;
    case FRAME_CONT:  /* Continuation frame. */
      if (frame_iscont_ficb(frame))
        goto unwind_c;
    case FRAME_VARG:  /* Vararg frame. */
      frame = frame_prevd(frame);
      break;
    case FRAME_PCALL:  /* FF pcall() frame. */
    case FRAME_PCALLH:  /* FF pcall() frame inside hook. */
      if (errcode) {
        if (errcode == LUA_YIELD) {
          frame = frame_prevd(frame);
          break;
        }
        if (frame_typep(frame) == FRAME_PCALL)
          hook_leave(G(L));
        L->base = frame_prevd(frame) + 1;
        L->cframe = cf;
        unwindstack(L, L->base);
      }
      return (void *)((intptr_t)cf | CFRAME_UNWIND_FF);
    }
  }
  /* No C frame. */
  if (errcode) {
    L->base = tvref(L->stack)+1+LJ_FR2;
    L->cframe = NULL;
    unwindstack(L, L->base);
    if (G(L)->panic)
      G(L)->panic(L);
    exit(EXIT_FAILURE);
  }
  return L;  /* Anything non-NULL will do. */
}

/* -- External frame unwinding ------------------------------------------- */

#if defined(__GNUC__) && !LJ_NO_UNWIND && !LJ_TARGET_WINDOWS

/*
** We have to use our own definitions instead of the mandatory (!) unwind.h,
** since various OS, distros and compilers mess up the header installation.
*/

typedef struct _Unwind_Exception
```

```
{
  uint64_t exclass;
  void (*excleanup)(int, struct _Unwind_Exception *);
  uintptr_t p1, p2;
} __attribute__((__aligned__)) _Unwind_Exception;

typedef struct _Unwind_Context _Unwind_Context;

#define _URC_OK                         0
#define _URC_FATAL_PHASE1_ERROR         3
#define _URC_HANDLER_FOUND        6
#define _URC_INSTALL_CONTEXT        7
#define _URC_CONTINUE_UNWIND        8
#define _URC_FAILURE                9

#if !LJ_TARGET_ARM

extern uintptr_t _Unwind_GetCFA(_Unwind_Context *);
extern void _Unwind_SetGR(_Unwind_Context *, int, uintptr_t);
extern void _Unwind_SetIP(_Unwind_Context *, uintptr_t);
extern void _Unwind_DeleteException(_Unwind_Exception *);
extern int _Unwind_RaiseException(_Unwind_Exception *);

#define _UA_SEARCH_PHASE        1
#define _UA_CLEANUP_PHASE        2
#define _UA_HANDLER_FRAME        4
#define _UA_FORCE_UNWIND        8

#define LJ_UEXCLASS                0x4c55414a49543200ULL        /* LUAJIT2\0 */
#define LJ_UEXCLASS_MAKE(c)        (LJ_UEXCLASS | (uint64_t)(c))
#define LJ_UEXCLASS_CHECK(cl)        (((cl) ^ LJ_UEXCLASS) <= 0xff)
#define LJ_UEXCLASS_ERRCODE(cl)        ((int)((cl) & 0xff))

/* DWARF2 personality handler referenced from interpreter .eh_frame. */
LJ_FUNCA int lj_err_unwind_dwarf(int version, int actions,
  uint64_t uexclass, _Unwind_Exception *uex, _Unwind_Context *ctx)
{
  void *cf;
  lua_State *L;
  if (version != 1)
    return _URC_FATAL_PHASE1_ERROR;
  UNUSED(uexclass);
  cf = (void *)_Unwind_GetCFA(ctx);
  L = cframe_L(cf);
  if ((actions & _UA_SEARCH_PHASE)) {
#if LJ_UNWIND_EXT
    if (err_unwind(L, cf, 0) == NULL)
      return _URC_CONTINUE_UNWIND;
#endif
    if (!LJ_UEXCLASS_CHECK(uexclass)) {
      setstrV(L, L->top++, lj_err_str(L, LJ_ERR_ERRCPP));
    }
    return _URC_HANDLER_FOUND;
  }
  if ((actions & _UA_CLEANUP_PHASE)) {
    int errcode;
    if (LJ_UEXCLASS_CHECK(uexclass)) {
      errcode = LJ_UEXCLASS_ERRCODE(uexclass);
    } else {
      if ((actions & _UA_HANDLER_FRAME))
        _Unwind_DeleteException(uex);
      errcode = LUA_ERRRUN;
    }
#if LJ_UNWIND_EXT
    cf = err_unwind(L, cf, errcode);
    if ((actions & _UA_FORCE_UNWIND)) {
      return _URC_CONTINUE_UNWIND;
    } else if (cf) {
      _Unwind_SetGR(ctx, LJ_TARGET_EHRETREG, errcode);
      _Unwind_SetIP(ctx, (uintptr_t)(cframe_unwind_ff(cf) ?
                                     lj_vm_unwind_ff_eh :
                                     lj_vm_unwind_c_eh));
      return _URC_INSTALL_CONTEXT;
    }
#if LJ_TARGET_X86ORX64
    else if ((actions & _UA_HANDLER_FRAME)) {
```

```
      /* Workaround for ancient libgcc bug. Still present in RHEL 5.5. :-/
      ** Real fix: http://gcc.gnu.org/viewcvs/trunk/gcc/unwind-dw2.c?
r1=121165&r2=124837&pathrev=153877&diff_format=h
      */
      _Unwind_SetGR(ctx, LJ_TARGET_EHRETREG, errcode);
      _Unwind_SetIP(ctx, (uintptr_t)lj_vm_unwind_rethrow);
      return _URC_INSTALL_CONTEXT;
    }
#endif
#else
    /* This is not the proper way to escape from the unwinder. We get away with
    ** it on non-x64 because the interpreter restores all callee-saved regs.
    */
    lj_err_throw(L, errcode);
#endif
  }
  return _URC_CONTINUE_UNWIND;
}

#if LJ_UNWIND_EXT
#if LJ_TARGET_OSX || defined(__OpenBSD__)
/* Sorry, no thread safety for OSX. Complain to Apple, not me. */
static _Unwind_Exception static_uex;
#else
static __thread _Unwind_Exception static_uex;
#endif

/* Raise DWARF2 exception. */
static void err_raise_ext(int errcode)
{
  static_uex.exclass = LJ_UEXCLASS_MAKE(errcode);
  static_uex.excleanup = NULL;
  _Unwind_RaiseException(&static_uex);
}
#endif

#else

extern void _Unwind_DeleteException(void *);
extern int __gnu_unwind_frame (void *, _Unwind_Context *);
extern int _Unwind_VRS_Set(_Unwind_Context *, int, uint32_t, int, void *);
extern int _Unwind_VRS_Get(_Unwind_Context *, int, uint32_t, int, void *);

static inline uint32_t _Unwind_GetGR(_Unwind_Context *ctx, int r)
{
  uint32_t v;
  _Unwind_VRS_Get(ctx, 0, r, 0, &v);
  return v;
}

static inline void _Unwind_SetGR(_Unwind_Context *ctx, int r, uint32_t v)
{
  _Unwind_VRS_Set(ctx, 0, r, 0, &v);
}

#define _US_VIRTUAL_UNWIND_FRAME        0
#define _US_UNWIND_FRAME_STARTING        1
#define _US_ACTION_MASK                     3
#define _US_FORCE_UNWIND                8

/* ARM unwinder personality handler referenced from interpreter .ARM.extab. */
LJ_FUNCA int lj_err_unwind_arm(int state, void *ucb, _Unwind_Context *ctx)
{
  void *cf = (void *)_Unwind_GetGR(ctx, 13);
  lua_State *L = cframe_L(cf);
  if ((state & _US_ACTION_MASK) == _US_VIRTUAL_UNWIND_FRAME) {
    setstrV(L, L->top++, lj_err_str(L, LJ_ERR_ERRCPP));
    return _URC_HANDLER_FOUND;
  }
  if ((state&(_US_ACTION_MASK|_US_FORCE_UNWIND)) == _US_UNWIND_FRAME_STARTING) {
    _Unwind_DeleteException(ucb);
    _Unwind_SetGR(ctx, 15, (uint32_t)(void *)lj_err_throw);
    _Unwind_SetGR(ctx, 0, (uint32_t)L);
    _Unwind_SetGR(ctx, 1, (uint32_t)LUA_ERRRUN);
    return _URC_INSTALL_CONTEXT;
  }
```

```
  if (__gnu_unwind_frame(ucb, ctx) != _URC_OK)
    return _URC_FAILURE;
  return _URC_CONTINUE_UNWIND;
}

#endif

#elif LJ_TARGET_X64 && LJ_TARGET_WINDOWS

/*
** Someone in Redmond owes me several days of my life. A lot of this is
** undocumented or just plain wrong on MSDN. Some of it can be gathered
** from 3rd party docs or must be found by trial-and-error. They really
** don't want you to write your own language-specific exception handler
** or to interact gracefully with MSVC. :-(
**
** Apparently MSVC doesn't call C++ destructors for foreign exceptions
** unless you compile your C++ code with /EHa. Unfortunately this means
** catch (...) also catches things like access violations. The use of
** _set_se_translator doesn't really help, because it requires /EHa, too.
*/

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

/* Taken from: http://www.nynaeve.net/?p=99 */
typedef struct UndocumentedDispatcherContext {
  ULONG64 ControlPc;
  ULONG64 ImageBase;
  PRUNTIME_FUNCTION FunctionEntry;
  ULONG64 EstablisherFrame;
  ULONG64 TargetIp;
  PCONTEXT ContextRecord;
  void (*LanguageHandler)(void);
  PVOID HandlerData;
  PUNWIND_HISTORY_TABLE HistoryTable;
  ULONG ScopeIndex;
  ULONG Fill0;
} UndocumentedDispatcherContext;

/* Another wild guess. */
extern void __DestructExceptionObject(EXCEPTION_RECORD *rec, int nothrow);

#ifdef MINGW_SDK_INIT
/* Workaround for broken MinGW64 declaration. */
VOID RtlUnwindEx_FIXED(PVOID,PVOID,PVOID,PVOID,PVOID,PVOID) asm("RtlUnwindEx");
#define RtlUnwindEx RtlUnwindEx_FIXED
#endif

#define LJ_MSVC_EXCODE               ((DWORD)0xe06d7363)
#define LJ_GCC_EXCODE                ((DWORD)0x20474343)

#define LJ_EXCODE                ((DWORD)0xe24c4a00)
#define LJ_EXCODE_MAKE(c)        (LJ_EXCODE | (DWORD)(c))
#define LJ_EXCODE_CHECK(cl)        (((cl) ^ LJ_EXCODE) <= 0xff)
#define LJ_EXCODE_ERRCODE(cl)        ((int)((cl) & 0xff))

/* Win64 exception handler for interpreter frame. */
LJ_FUNCA EXCEPTION_DISPOSITION lj_err_unwind_win64(EXCEPTION_RECORD *rec,
  void *cf, CONTEXT *ctx, UndocumentedDispatcherContext *dispatch)
{
  lua_State *L = cframe_L(cf);
  int errcode = LJ_EXCODE_CHECK(rec->ExceptionCode) ?
                LJ_EXCODE_ERRCODE(rec->ExceptionCode) : LUA_ERRRUN;
  if ((rec->ExceptionFlags & 6)) {  /* EH_UNWINDING|EH_EXIT_UNWIND */
    /* Unwind internal frames. */
    err_unwind(L, cf, errcode);
  } else {
    void *cf2 = err_unwind(L, cf, 0);
    if (cf2) {  /* We catch it, so start unwinding the upper frames. */
      if (rec->ExceptionCode == LJ_MSVC_EXCODE ||
          rec->ExceptionCode == LJ_GCC_EXCODE) {
        __DestructExceptionObject(rec, 1);
        setstrV(L, L->top++, lj_err_str(L, LJ_ERR_ERRCPP));
      } else if (!LJ_EXCODE_CHECK(rec->ExceptionCode)) {
        /* Don't catch access violations etc. */
```

```
421         return ExceptionContinueSearch;
422       }
423       /* Unwind the stack and call all handlers for all lower C frames
424       ** (including ourselves) again with EH_UNWINDING set. Then set
425       ** rsp = cf, rax = errcode and jump to the specified target.
426       */
427       RtlUnwindEx(cf, (void *)((cframe_unwind_ff(cf2) && errcode != LUA_YIELD) ?
428                                lj_vm_unwind_ff_eh :
429                                lj_vm_unwind_c_eh),
430                   rec, (void *)(uintptr_t)errcode, ctx, dispatch->HistoryTable);
431       /* RtlUnwindEx should never return. */
432     }
433   }
434   return ExceptionContinueSearch;
435 }
436
437 /* Raise Windows exception. */
438 static void err_raise_ext(int errcode)
439 {
440   RaiseException(LJ_EXCODE_MAKE(errcode), 1 /* EH_NONCONTINUABLE */, 0, NULL);
441 }
442
443 #endif
444
445 /* -- Error handling ------------------------------------------------------ */
446
447 /* Throw error. Find catch frame, unwind stack and continue. */
448 LJ_NOINLINE void LJ_FASTCALL lj_err_throw(lua_State *L, int errcode)
449 {
450   global_State *g = G(L);
451   lj_trace_abort(g);
452   setmref(g->jit_base, NULL);
453   L->status = 0;
454 #if LJ_UNWIND_EXT
455   err_raise_ext(errcode);
456   /*
457   ** A return from this function signals a corrupt C stack that cannot be
458   ** unwound. We have no choice but to call the panic function and exit.
459   **
460   ** Usually this is caused by a C function without unwind information.
461   ** This should never happen on x64, but may happen if you've manually
462   ** enabled LUAJIT_UNWIND_EXTERNAL and forgot to recompile *every*
463   ** non-C++ file with -funwind-tables.
464   */
465   if (G(L)->panic)
466     G(L)->panic(L);
467 #else
468   {
469     void *cf = err_unwind(L, NULL, errcode);
470     if (cframe_unwind_ff(cf))
471       lj_vm_unwind_ff(cframe_raw(cf));
472     else
473       lj_vm_unwind_c(cframe_raw(cf), errcode);
474   }
475 #endif
476   exit(EXIT_FAILURE);
477 }
478
479 /* Return string object for error message. */
480 LJ_NOINLINE GCstr *lj_err_str(lua_State *L, ErrMsg em)
481 {
482   return lj_str_newz(L, err2msg(em));
483 }
484
485 /* Out-of-memory error. */
486 LJ_NOINLINE void lj_err_mem(lua_State *L)
487 {
488   if (L->status == LUA_ERRERR+1)  /* Don't touch the stack during lua_open. */
489     lj_vm_unwind_c(L->cframe, LUA_ERRMEM);
490   setstrV(L, L->top++, lj_err_str(L, LJ_ERR_ERRMEM));
491   lj_err_throw(L, LUA_ERRMEM);
492 }
493
494 /* Find error function for runtime errors. Requires an extra stack traversal. */
495 static ptrdiff_t finderrfunc(lua_State *L)
496 {
```

```
  cTValue *frame = L->base-1, *bot = tvref(L->stack)+LJ_FR2;
  void *cf = L->cframe;
  while (frame > bot && cf) {
    while (cframe_nres(cframe_raw(cf)) < 0) {  /* cframe without frame? */
      if (frame >= restorestack(L, -cframe_nres(cf)))
        break;
      if (cframe_errfunc(cf) >= 0)  /* Error handler not inherited (-1)? */
        return cframe_errfunc(cf);
      cf = cframe_prev(cf);  /* Else unwind cframe and continue searching. */
      if (cf == NULL)
        return 0;
    }
    switch (frame_typep(frame)) {
    case FRAME_LUA:
    case FRAME_LUAP:
      frame = frame_prevl(frame);
      break;
    case FRAME_C:
      cf = cframe_prev(cf);
      /* fallthrough */
    case FRAME_VARG:
      frame = frame_prevd(frame);
      break;
    case FRAME_CONT:
      if (frame_iscont_fficb(frame))
        cf = cframe_prev(cf);
      frame = frame_prevd(frame);
      break;
    case FRAME_CP:
      if (cframe_canyield(cf)) return 0;
      if (cframe_errfunc(cf) >= 0)
        return cframe_errfunc(cf);
      frame = frame_prevd(frame);
      break;
    case FRAME_PCALL:
    case FRAME_PCALLH:
      if (frame_func(frame_prevd(frame))->c.ffid == FF_xpcall)
        return savestack(L, frame_prevd(frame)+1);  /* xpcall's errorfunc. */
      return 0;
    default:
      lua_assert(0);
      return 0;
    }
  }
  return 0;
}

/* Runtime error. */
LJ_NOINLINE void lj_err_run(lua_State *L)
{
  ptrdiff_t ef = finderrfunc(L);
  if (ef) {
    TValue *errfunc = restorestack(L, ef);
    TValue *top = L->top;
    lj_trace_abort(G(L));
    if (!tvisfunc(errfunc) || L->status == LUA_ERRERR) {
      setstrV(L, top-1, lj_err_str(L, LJ_ERR_ERRERR));
      lj_err_throw(L, LUA_ERRERR);
    }
    L->status = LUA_ERRERR;
    copyTV(L, top+LJ_FR2, top-1);
    copyTV(L, top-1, errfunc);
    if (LJ_FR2) setnilV(top++);
    L->top = top+1;
    lj_vm_call(L, top, 1+1);  /* Stack: |errfunc|msg| -> |msg| */
  }
  lj_err_throw(L, LUA_ERRRUN);
}

/* Formatted runtime error message. */
LJ_NORET LJ_NOINLINE static void err_msgv(lua_State *L, ErrMsg em, ...)
{
  const char *msg;
  va_list argp;
  va_start(argp, em);
  if (curr_funcisL(L)) L->top = curr_topL(L);
```

```
  msg = lj_strfmt_pushvf(L, err2msg(em), argp);
  va_end(argp);
  lj_debug_addloc(L, msg, L->base-1, NULL);
  lj_err_run(L);
}

/* Non-vararg variant for better calling conventions. */
LJ_NOINLINE void lj_err_msg(lua_State *L, ErrMsg em)
{
  err_msgv(L, em);
}

/* Lexer error. */
LJ_NOINLINE void lj_err_lex(lua_State *L, GCstr *src, const char *tok,
                            BCLine line, ErrMsg em, va_list argp)
{
  char buff[LUA_IDSIZE];
  const char *msg;
  lj_debug_shortname(buff, src, line);
  msg = lj_strfmt_pushvf(L, err2msg(em), argp);
  msg = lj_strfmt_pushf(L, "%s:%d: %s", buff, line, msg);
  if (tok)
    lj_strfmt_pushf(L, err2msg(LJ_ERR_XNEAR), msg, tok);
  lj_err_throw(L, LUA_ERRSYNTAX);
}

/* Typecheck error for operands. */
LJ_NOINLINE void lj_err_optype(lua_State *L, cTValue *o, ErrMsg opm)
{
  const char *tname = lj_typename(o);
  const char *opname = err2msg(opm);
  if (curr_funcisL(L)) {
    GCproto *pt = curr_proto(L);
    const BCIns *pc = cframe_Lpc(L) - 1;
    const char *oname = NULL;
    const char *kind = lj_debug_slotname(pt, pc, (BCReg)(o-L->base), &oname);
    if (kind)
      err_msgv(L, LJ_ERR_BADOPRT, opname, kind, oname, tname);
  }
  err_msgv(L, LJ_ERR_BADOPRV, opname, tname);
}

/* Typecheck error for ordered comparisons. */
LJ_NOINLINE void lj_err_comp(lua_State *L, cTValue *o1, cTValue *o2)
{
  const char *t1 = lj_typename(o1);
  const char *t2 = lj_typename(o2);
  err_msgv(L, t1 == t2 ? LJ_ERR_BADCMPV : LJ_ERR_BADCMPT, t1, t2);
  /* This assumes the two "boolean" entries are commoned by the C compiler. */
}

/* Typecheck error for __call. */
LJ_NOINLINE void lj_err_optype_call(lua_State *L, TValue *o)
{
  /* Gross hack if lua_[p]call or pcall/xpcall fail for a non-callable object:
  ** L->base still points to the caller. So add a dummy frame with L instead
  ** of a function. See lua_getstack().
  */
  const BCIns *pc = cframe_Lpc(L);
  if (((ptrdiff_t)pc & FRAME_TYPE) != FRAME_LUA) {
    const char *tname = lj_typename(o);
    if (LJ_FR2) o++;
    setframe_pc(o, pc);
    setframe_gc(o, obj2gco(L), LJ_TTHREAD);
    L->top = L->base = o+1;
    err_msgv(L, LJ_ERR_BADCALL, tname);
  }
  lj_err_optype(L, o, LJ_ERR_OPCALL);
}

/* Error in context of caller. */
LJ_NOINLINE void lj_err_callermsg(lua_State *L, const char *msg)
{
  TValue *frame = L->base-1;
  TValue *pframe = NULL;
  if (frame_islua(frame)) {
```

```
    pframe = frame_prevl(frame);
  } else if (frame_iscont(frame)) {
    if (frame_iscont_fficb(frame)) {
      pframe = frame;
      frame = NULL;
    } else {
      pframe = frame_prevd(frame);
#if LJ_HASFFI
      /* Remove frame for FFI metamethods. */
      if (frame_func(frame)->c.ffid >= FF_ffi_meta___index &&
          frame_func(frame)->c.ffid <= FF_ffi_meta___tostring) {
        L->base = pframe+1;
        L->top = frame;
        setcframe_pc(cframe_raw(L->cframe), frame_contpc(frame));
      }
#endif
    }
  }
  lj_debug_addloc(L, msg, pframe, frame);
  lj_err_run(L);
}

/* Formatted error in context of caller. */
LJ_NOINLINE void lj_err_callerv(lua_State *L, ErrMsg em, ...)
{
  const char *msg;
  va_list argp;
  va_start(argp, em);
  msg = lj_strfmt_pushvf(L, err2msg(em), argp);
  va_end(argp);
  lj_err_callermsg(L, msg);
}

/* Error in context of caller. */
LJ_NOINLINE void lj_err_caller(lua_State *L, ErrMsg em)
{
  lj_err_callermsg(L, err2msg(em));
}

/* Argument error message. */
LJ_NORET LJ_NOINLINE static void err_argmsg(lua_State *L, int narg,
                                            const char *msg)
{
  const char *fname = "?";
  const char *ftype = lj_debug_funcname(L, L->base - 1, &fname);
  if (narg < 0 && narg > LUA_REGISTRYINDEX)
    narg = (int)(L->top - L->base) + narg + 1;
  if (ftype && ftype[3] == 'h' && --narg == 0)  /* Check for "method". */
    msg = lj_strfmt_pushf(L, err2msg(LJ_ERR_BADSELF), fname, msg);
  else
    msg = lj_strfmt_pushf(L, err2msg(LJ_ERR_BADARG), narg, fname, msg);
  lj_err_callermsg(L, msg);
}

/* Formatted argument error. */
LJ_NOINLINE void lj_err_argv(lua_State *L, int narg, ErrMsg em, ...)
{
  const char *msg;
  va_list argp;
  va_start(argp, em);
  msg = lj_strfmt_pushvf(L, err2msg(em), argp);
  va_end(argp);
  err_argmsg(L, narg, msg);
}

/* Argument error. */
LJ_NOINLINE void lj_err_arg(lua_State *L, int narg, ErrMsg em)
{
  err_argmsg(L, narg, err2msg(em));
}

/* Typecheck error for arguments. */
LJ_NOINLINE void lj_err_argtype(lua_State *L, int narg, const char *xname)
{
  const char *tname, *msg;
  if (narg <= LUA_REGISTRYINDEX) {
```

```
    if (narg >= LUA_GLOBALSINDEX) {
      tname = lj_obj_itypename[~LJ_TTAB];
    } else {
      GCfunc *fn = curr_func(L);
      int idx = LUA_GLOBALSINDEX - narg;
      if (idx <= fn->c.nupvalues)
        tname = lj_typename(&fn->c.upvalue[idx-1]);
      else
        tname = lj_obj_typename[0];
    }
  } else {
    TValue *o = narg < 0 ? L->top + narg : L->base + narg-1;
    tname = o < L->top ? lj_typename(o) : lj_obj_typename[0];
  }
  msg = lj_strfmt_pushf(L, err2msg(LJ_ERR_BADTYPE), xname, tname);
  err_argmsg(L, narg, msg);
}

/* Typecheck error for arguments. */
LJ_NOINLINE void lj_err_argt(lua_State *L, int narg, int tt)
{
  lj_err_argtype(L, narg, lj_obj_typename[tt+1]);
}

/* -- Public error handling API ------------------------------------------- */

LUA_API lua_CFunction lua_atpanic(lua_State *L, lua_CFunction panicf)
{
  lua_CFunction old = G(L)->panic;
  G(L)->panic = panicf;
  return old;
}

/* Forwarders for the public API (C calling convention and no LJ_NORET). */
LUA_API int lua_error(lua_State *L)
{
  lj_err_run(L);
  return 0;  /* unreachable */
}

LUALIB_API int luaL_argerror(lua_State *L, int narg, const char *msg)
{
  err_argmsg(L, narg, msg);
  return 0;  /* unreachable */
}

LUALIB_API int luaL_typerror(lua_State *L, int narg, const char *xname)
{
  lj_err_argtype(L, narg, xname);
  return 0;  /* unreachable */
}

LUALIB_API void luaL_where(lua_State *L, int level)
{
  int size;
  cTValue *frame = lj_debug_frame(L, level, &size);
  lj_debug_addloc(L, "", frame, size ? frame+size : NULL);
}

LUALIB_API int luaL_error(lua_State *L, const char *fmt, ...)
{
  const char *msg;
  va_list argp;
  va_start(argp, fmt);
  msg = lj_strfmt_pushvf(L, fmt, argp);
  va_end(argp);
  lj_err_callermsg(L, msg);
  return 0;  /* unreachable */
}

```

One Level Up Top Level

src/lj_func.c - luajit-2.0-src

Functions defined

- func_emptyuv
- func_finduv
- func_newL
- lj_func_closeuv
- lj_func_free
- lj_func_freeproto
- lj_func_freeuv
- lj_func_newC
- lj_func_newL_empty
- lj_func_newL_gc
- unlinkuv

Macros defined

- LUA_CORE
- lj_func_c

Source code

```
1  /*
2  ** Function handling (prototypes, functions and upvalues).
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  **
5  ** Portions taken verbatim or adapted from the Lua interpreter.
6  ** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
7  */
8
9  #define lj_func_c
10  #define LUA_CORE
11
12  #include "lj_obj.h"
13  #include "lj_gc.h"
14  #include "lj_func.h"
15  #include "lj_trace.h"
16  #include "lj_vm.h"
17
18  /* -- Prototypes ---------------------------------------------------------- */
19
20  void LJ_FASTCALL lj_func_freeproto(global_State *g, GCproto *pt)
21  {
22    lj_mem_free(g, pt, pt->sizept);
23  }
24
25  /* -- Upvalues ---------------------------------------------------------- */
26
27  static void unlinkuv(GCupval *uv)
28  {
29    lua_assert(uvprev(uvnext(uv)) == uv && uvnext(uvprev(uv)) == uv);
30    setgcrefr(uvnext(uv)->prev, uv->prev);
31    setgcrefr(uvprev(uv)->next, uv->next);
32  }
```

```
33
34  /* Find existing open upvalue for a stack slot or create a new one. */
35  static GCupval *func_finduv(lua_State *L, TValue *slot)
36  {
37    global_State *g = G(L);
38    GCRef *pp = &L->openupval;
39    GCupval *p;
40    GCupval *uv;
41    /* Search the sorted list of open upvalues. */
42    while (gcref(*pp) != NULL && uvval((p = gco2uv(gcref(*pp)))) >= slot) {
43      lua_assert(!p->closed && uvval(p) != &p->tv);
44      if (uvval(p) == slot) {  /* Found open upvalue pointing to same slot? */
45        if (isdead(g, obj2gco(p)))  /* Resurrect it, if it's dead. */
46          flipwhite(obj2gco(p));
47        return p;
48      }
49      pp = &p->nextgc;
50    }
51    /* No matching upvalue found. Create a new one. */
52    uv = lj_mem_newt(L, sizeof(GCupval), GCupval);
53    newwhite(g, uv);
54    uv->gct = ~LJ_TUPVAL;
55    uv->closed = 0;  /* Still open. */
56    setmref(uv->v, slot);  /* Pointing to the stack slot. */
57    /* NOBARRIER: The GCupval is new (marked white) and open. */
58    setgcrefr(uv->nextgc, *pp);  /* Insert into sorted list of open upvalues. */
59    setgcref(*pp, obj2gco(uv));
60    setgcref(uv->prev, obj2gco(&g->uvhead));  /* Insert into GC list, too. */
61    setgcrefr(uv->next, g->uvhead.next);
62    setgcref(uvnext(uv)->prev, obj2gco(uv));
63    setgcref(g->uvhead.next, obj2gco(uv));
64    lua_assert(uvprev(uvnext(uv)) == uv && uvnext(uvprev(uv)) == uv);
65    return uv;
66  }
67
68  /* Create an empty and closed upvalue. */
69  static GCupval *func_emptyuv(lua_State *L)
70  {
71    GCupval *uv = (GCupval *)lj_mem_newgco(L, sizeof(GCupval));
72    uv->gct = ~LJ_TUPVAL;
73    uv->closed = 1;
74    setnilV(&uv->tv);
75    setmref(uv->v, &uv->tv);
76    return uv;
77  }
78
79  /* Close all open upvalues pointing to some stack level or above. */
80  void LJ_FASTCALL lj_func_closeuv(lua_State *L, TValue *level)
81  {
82    GCupval *uv;
83    global_State *g = G(L);
84    while (gcref(L->openupval) != NULL &&
85           uvval((uv = gco2uv(gcref(L->openupval)))) >= level) {
86      GCobj *o = obj2gco(uv);
87      lua_assert(!isblack(o) && !uv->closed && uvval(uv) != &uv->tv);
88      setgcrefr(L->openupval, uv->nextgc);  /* No longer in open list. */
89      if (isdead(g, o)) {
90        lj_func_freeuv(g, uv);
91      } else {
92        unlinkuv(uv);
93        lj_gc_closeuv(g, uv);
94      }
95    }
96  }
97
98  void LJ_FASTCALL lj_func_freeuv(global_State *g, GCupval *uv)
99  {
100   if (!uv->closed)
101     unlinkuv(uv);
102   lj_mem_freet(g, uv);
103 }
104
105 /* -- Functions (closures) ------------------------------------------------ */
106
107 GCfunc *lj_func_newC(lua_State *L, MSize nelems, GCtab *env)
108 {
```

```c
  GCfunc *fn = (GCfunc *)lj_mem_newgco(L, sizeCfunc(nelems));
  fn->c.gct = ~LJ_TFUNC;
  fn->c.ffid = FF_C;
  fn->c.nupvalues = (uint8_t)nelems;
  /* NOBARRIER: The GCfunc is new (marked white). */
  setmref(fn->c.pc, &G(L)->bc_cfunc_ext);
  setgcref(fn->c.env, obj2gco(env));
  return fn;
}

static GCfunc *func_newL(lua_State *L, GCproto *pt, GCtab *env)
{
  uint32_t count;
  GCfunc *fn = (GCfunc *)lj_mem_newgco(L, sizeLfunc((MSize)pt->sizeuv));
  fn->l.gct = ~LJ_TFUNC;
  fn->l.ffid = FF_LUA;
  fn->l.nupvalues = 0;  /* Set to zero until upvalues are initialized. */
  /* NOBARRIER: Really a setgcref. But the GCfunc is new (marked white). */
  setmref(fn->l.pc, proto_bc(pt));
  setgcref(fn->l.env, obj2gco(env));
  /* Saturating 3 bit counter (0..7) for created closures. */
  count = (uint32_t)pt->flags + PROTO_CLCOUNT;
  pt->flags = (uint8_t)(count - ((count >> PROTO_CLC_BITS) & PROTO_CLCOUNT));
  return fn;
}

/* Create a new Lua function with empty upvalues. */
GCfunc *lj_func_newL_empty(lua_State *L, GCproto *pt, GCtab *env)
{
  GCfunc *fn = func_newL(L, pt, env);
  MSize i, nuv = pt->sizeuv;
  /* NOBARRIER: The GCfunc is new (marked white). */
  for (i = 0; i < nuv; i++) {
    GCupval *uv = func_emptyuv(L);
    uv->dhash = (uint32_t)(uintptr_t)pt ^ ((uint32_t)proto_uv(pt)[i] << 24);
    setgcref(fn->l.uvptr[i], obj2gco(uv));
  }
  fn->l.nupvalues = (uint8_t)nuv;
  return fn;
}

/* Do a GC check and create a new Lua function with inherited upvalues. */
GCfunc *lj_func_newL_gc(lua_State *L, GCproto *pt, GCfuncL *parent)
{
  GCfunc *fn;
  GCRef *puv;
  MSize i, nuv;
  TValue *base;
  lj_gc_check_fixtop(L);
  fn = func_newL(L, pt, tabref(parent->env));
  /* NOBARRIER: The GCfunc is new (marked white). */
  puv = parent->uvptr;
  nuv = pt->sizeuv;
  base = L->base;
  for (i = 0; i < nuv; i++) {
    uint32_t v = proto_uv(pt)[i];
    GCupval *uv;
    if ((v & PROTO_UV_LOCAL)) {
      uv = func_finduv(L, base + (v & 0xff));
      uv->immutable = ((v / PROTO_UV_IMMUTABLE) & 1);
      uv->dhash = (uint32_t)(uintptr_t)mref(parent->pc, char) ^ (v << 24);
    } else {
      uv = &gcref(puv[v])->uv;
    }
    setgcref(fn->l.uvptr[i], obj2gco(uv));
  }
  fn->l.nupvalues = (uint8_t)nuv;
  return fn;
}

void LJ_FASTCALL lj_func_free(global_State *g, GCfunc *fn)
{
  MSize size = isluafunc(fn) ? sizeLfunc((MSize)fn->l.nupvalues) :
                               sizeCfunc((MSize)fn->c.nupvalues);
  lj_mem_free(g, fn, size);
}
```

One Level Up Top Level

src/lj_gc.h - luajit-2.0-src

Functions defined

- lj_gc_barrierback
- lj_mem_free

Macros defined

- LJ_GC_BLACK
- LJ_GC_CDATA_FIN
- LJ_GC_COLORS
- LJ_GC_FINALIZED
- LJ_GC_FIXED
- LJ_GC_SFIXED
- LJ_GC_WEAK
- LJ_GC_WEAKKEY
- LJ_GC_WEAKVAL
- LJ_GC_WHITE0
- LJ_GC_WHITE1
- LJ_GC_WHITES
- _LJ_GC_H
- black2gray
- curwhite
- fixstring
- flipwhite
- isblack
- isdead
- isgray
- iswhite
- lj_gc_anybarriert
- lj_gc_barrier
- lj_gc_barriert
- lj_gc_check
- lj_gc_check_fixtop

- lj_gc_finalize_cdata
- lj_gc_objbarrier
- lj_gc_objbarriert
- lj_mem_freet
- lj_mem_freevec
- lj_mem_growvec
- lj_mem_new
- lj_mem_newobj
- lj_mem_newt
- lj_mem_newvec
- lj_mem_reallocvec
- makewhite
- markfinalized
- newwhite
- otherwhite
- tviswhite

Source code

```
1  /*
2  ** Garbage collector.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_GC_H
7  #define _LJ_GC_H
8
9  #include "lj_obj.h"
10
11  /* Garbage collector states. Order matters. */
12  enum {
13    GCSpause, GCSpropagate, GCSatomic, GCSsweepstring, GCSsweep, GCSfinalize
14  };
15
16  /* Bitmasks for marked field of GCobj. */
17  #define LJ_GC_WHITE0        0x01
18  #define LJ_GC_WHITE1        0x02
19  #define LJ_GC_BLACK         0x04
20  #define LJ_GC_FINALIZED        0x08
21  #define LJ_GC_WEAKKEY        0x08
22  #define LJ_GC_WEAKVAL        0x10
23  #define LJ_GC_CDATA_FIN        0x10
24  #define LJ_GC_FIXED         0x20
25  #define LJ_GC_SFIXED        0x40
26
27  #define LJ_GC_WHITES        (LJ_GC_WHITE0 | LJ_GC_WHITE1)
28  #define LJ_GC_COLORS        (LJ_GC_WHITES | LJ_GC_BLACK)
29  #define LJ_GC_WEAK        (LJ_GC_WEAKKEY | LJ_GC_WEAKVAL)
30
31  /* Macros to test and set GCobj colors. */
32  #define iswhite(x)        ((x)->gch.marked & LJ_GC_WHITES)
33  #define isblack(x)        ((x)->gch.marked & LJ_GC_BLACK)
34  #define isgray(x)        (!((x)->gch.marked & (LJ_GC_BLACK|LJ_GC_WHITES)))
35  #define tviswhite(x)        (tvisgcv(x) && iswhite(gcV(x)))
36  #define otherwhite(g)        (g->gc.currentwhite ^ LJ_GC_WHITES)
```

```
#define isdead(g, v)		((v)->gch.marked & otherwhite(g) & LJ_GC_WHITES)

#define curwhite(g)		((g)->gc.currentwhite & LJ_GC_WHITES)
#define newwhite(g, x)		(obj2gco(x)->gch.marked = (uint8_t)curwhite(g))
#define makewhite(g, x) \
  ((x)->gch.marked = ((x)->gch.marked & (uint8_t)~LJ_GC_COLORS) | curwhite(g))
#define flipwhite(x)		((x)->gch.marked ^= LJ_GC_WHITES)
#define black2gray(x)		((x)->gch.marked &= (uint8_t)~LJ_GC_BLACK)
#define fixstring(s)		((s)->marked |= LJ_GC_FIXED)
#define markfinalized(x)	((x)->gch.marked |= LJ_GC_FINALIZED)

/* Collector. */
LJ_FUNC size_t lj_gc_separateudata(global_State *g, int all);
LJ_FUNC void lj_gc_finalize_udata(lua_State *L);
#if LJ_HASFFI
LJ_FUNC void lj_gc_finalize_cdata(lua_State *L);
#else
#define lj_gc_finalize_cdata(L)		UNUSED(L)
#endif
LJ_FUNC void lj_gc_freeall(global_State *g);
LJ_FUNCA int LJ_FASTCALL lj_gc_step(lua_State *L);
LJ_FUNCA void LJ_FASTCALL lj_gc_step_fixtop(lua_State *L);
#if LJ_HASJIT
LJ_FUNC int LJ_FASTCALL lj_gc_step_jit(global_State *g, MSize steps);
#endif
LJ_FUNC void lj_gc_fullgc(lua_State *L);

/* GC check: drive collector forward if the GC threshold has been reached. */
#define lj_gc_check(L) \
  { if (LJ_UNLIKELY(G(L)->gc.total >= G(L)->gc.threshold)) \
      lj_gc_step(L); }
#define lj_gc_check_fixtop(L) \
  { if (LJ_UNLIKELY(G(L)->gc.total >= G(L)->gc.threshold)) \
      lj_gc_step_fixtop(L); }

/* Write barriers. */
LJ_FUNC void lj_gc_barrierf(global_State *g, GCobj *o, GCobj *v);
LJ_FUNCA void LJ_FASTCALL lj_gc_barrieruv(global_State *g, TValue *tv);
LJ_FUNC void lj_gc_closeuv(global_State *g, GCupval *uv);
#if LJ_HASJIT
LJ_FUNC void lj_gc_barriertrace(global_State *g, uint32_t traceno);
#endif

/* Move the GC propagation frontier back for tables (make it gray again). */
static LJ_AINLINE void lj_gc_barrierback(global_State *g, GCtab *t)
{
  GCobj *o = obj2gco(t);
  lua_assert(isblack(o) && !isdead(g, o));
  lua_assert(g->gc.state != GCSfinalize && g->gc.state != GCSpause);
  black2gray(o);
  setgcrefr(t->gclist, g->gc.grayagain);
  setgcref(g->gc.grayagain, o);
}

/* Barrier for stores to table objects. TValue and GCobj variant. */
#define lj_gc_anybarriert(L, t)  \
  { if (LJ_UNLIKELY(isblack(obj2gco(t)))) lj_gc_barrierback(G(L), (t)); }
#define lj_gc_barriert(L, t, tv) \
  { if (tviswhite(tv) && isblack(obj2gco(t))) \
      lj_gc_barrierback(G(L), (t)); }
#define lj_gc_objbarriert(L, t, o)  \
  { if (iswhite(obj2gco(o)) && isblack(obj2gco(t))) \
      lj_gc_barrierback(G(L), (t)); }

/* Barrier for stores to any other object. TValue and GCobj variant. */
#define lj_gc_barrier(L, p, tv) \
  { if (tviswhite(tv) && isblack(obj2gco(p))) \
      lj_gc_barrierf(G(L), obj2gco(p), gcV(tv)); }
#define lj_gc_objbarrier(L, p, o) \
  { if (iswhite(obj2gco(o)) && isblack(obj2gco(p))) \
      lj_gc_barrierf(G(L), obj2gco(p), obj2gco(o)); }

/* Allocator. */
LJ_FUNC void *lj_mem_realloc(lua_State *L, void *p, GCSize osz, GCSize nsz);
LJ_FUNC void * LJ_FASTCALL lj_mem_newgco(lua_State *L, GCSize size);
LJ_FUNC void *lj_mem_grow(lua_State *L, void *p,
```

```
                        MSize *szp, MSize lim, MSize esz);

#define lj_mem_new(L, s)        lj_mem_realloc(L, NULL, 0, (s))

static LJ_AINLINE void lj_mem_free(global_State *g, void *p, size_t osize)
{
  g->gc.total -= (GCSize)osize;
  g->allocf(g->allocd, p, osize, 0);
}

#define lj_mem_newvec(L, n, t)        ((t *)lj_mem_new(L, (GCSize)((n)*sizeof(t))))
#define lj_mem_reallocvec(L, p, on, n, t) \
  ((p) = (t *)lj_mem_realloc(L, p, (on)*sizeof(t), (GCSize)((n)*sizeof(t))))
#define lj_mem_growvec(L, p, n, m, t) \
  ((p) = (t *)lj_mem_grow(L, (p), &(n), (m), (MSize)sizeof(t)))
#define lj_mem_freevec(g, p, n, t)        lj_mem_free(g, (p), (n)*sizeof(t))

#define lj_mem_newobj(L, t)        ((t *)lj_mem_newgco(L, sizeof(t)))
#define lj_mem_newt(L, s, t)        ((t *)lj_mem_new(L, (s)))
#define lj_mem_freet(g, p)        lj_mem_free(g, (p), sizeof(*(p)))

#endif
```

One Level Up Top Level

src/lj_gc.c - luajit-2.0-src

Global variables defined

- gc_freefunc

Data types defined

- GCFreeFunc

Functions defined

- atomic
- gc_call_finalizer
- gc_clearweak
- gc_finalize
- gc_mark
- gc_mark_gcroot
- gc_mark_mmudata
- gc_mark_start
- gc_mark_uv
- gc_marktrace
- gc_mayclear
- gc_onestep
- gc_propagate_gray
- gc_sweep
- gc_traverse_frames
- gc_traverse_func
- gc_traverse_proto
- gc_traverse_tab
- gc_traverse_thread
- gc_traverse_trace
- lj_gc_barrierf
- lj_gc_barriertrace
- lj_gc_barrieruv
- lj_gc_closeuv
- lj_gc_finalize_cdata

- lj_gc_finalize_udata
- lj_gc_freeall
- lj_gc_fullgc
- lj_gc_separateudata
- lj_gc_step
- lj_gc_step_fixtop
- lj_gc_step_jit
- lj_mem_grow
- lj_mem_newgco
- lj_mem_realloc
- propagatemark

Macros defined

- GCFINALIZECOST
- GCSTEPSIZE
- GCSWEEPCOST
- GCSWEEPMAX
- LUA_CORE
- TV2MARKED
- TV2MARKED
- gc_fullsweep
- gc_mark_str
- gc_markobj
- gc_marktv
- gc_traverse_curtrace
- gc_traverse_curtrace
- gray2black
- isfinalized
- lj_gc_c
- white2gray

Source code

```
/*
** Garbage collector.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Major portions taken verbatim or adapted from the Lua interpreter.
```

```
** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
*/

#define lj_gc_c
#define LUA_CORE

#include "lj_obj.h"
#include "lj_gc.h"
#include "lj_err.h"
#include "lj_buf.h"
#include "lj_str.h"
#include "lj_tab.h"
#include "lj_func.h"
#include "lj_udata.h"
#include "lj_meta.h"
#include "lj_state.h"
#include "lj_frame.h"
#if LJ_HASFFI
#include "lj_ctype.h"
#include "lj_cdata.h"
#endif
#include "lj_trace.h"
#include "lj_vm.h"

#define GCSTEPSIZE	1024u
#define GCSWEEPMAX	40
#define GCSWEEPCOST	10
#define GCFINALIZECOST	100

/* Macros to set GCobj colors and flags. */
#define white2gray(x)		((x)->gch.marked &= (uint8_t)~LJ_GC_WHITES)
#define gray2black(x)		((x)->gch.marked |= LJ_GC_BLACK)
#define isfinalized(u)		((u)->marked & LJ_GC_FINALIZED)

/* -- Mark phase ------------------------------------------------------- */

/* Mark a TValue (if needed). */
#define gc_marktv(g, tv) \
  { lua_assert(!tvisgcv(tv) || (~itype(tv) == gcval(tv)->gch.gct)); \
    if (tviswhite(tv)) gc_mark(g, gcV(tv)); }

/* Mark a GCobj (if needed). */
#define gc_markobj(g, o) \
  { if (iswhite(obj2gco(o))) gc_mark(g, obj2gco(o)); }

/* Mark a string object. */
#define gc_mark_str(s)		((s)->marked &= (uint8_t)~LJ_GC_WHITES)

/* Mark a white GCobj. */
static void gc_mark(global_State *g, GCobj *o)
{
  int gct = o->gch.gct;
  lua_assert(iswhite(o) && !isdead(g, o));
  white2gray(o);
  if (LJ_UNLIKELY(gct == ~LJ_TUDATA)) {
    GCtab *mt = tabref(gco2ud(o)->metatable);
    gray2black(o);  /* Userdata are never gray. */
    if (mt) gc_markobj(g, mt);
    gc_markobj(g, tabref(gco2ud(o)->env));
  } else if (LJ_UNLIKELY(gct == ~LJ_TUPVAL)) {
    GCupval *uv = gco2uv(o);
    gc_marktv(g, uvval(uv));
    if (uv->closed)
      gray2black(o);  /* Closed upvalues are never gray. */
  } else if (gct != ~LJ_TSTR && gct != ~LJ_TCDATA) {
    lua_assert(gct == ~LJ_TFUNC || gct == ~LJ_TTAB ||
               gct == ~LJ_TTHREAD || gct == ~LJ_TPROTO);
    setgcrefr(o->gch.gclist, g->gc.gray);
    setgcref(g->gc.gray, o);
  }
}

/* Mark GC roots. */
static void gc_mark_gcroot(global_State *g)
{
  ptrdiff_t i;
```

```
  for (i = 0; i < GCROOT_MAX; i++)
    if (gcref(g->gcroot[i]) != NULL)
      gc_markobj(g, gcref(g->gcroot[i]));
}

/* Start a GC cycle and mark the root set. */
static void gc_mark_start(global_State *g)
{
  setgcrefnull(g->gc.gray);
  setgcrefnull(g->gc.grayagain);
  setgcrefnull(g->gc.weak);
  gc_markobj(g, mainthread(g));
  gc_markobj(g, tabref(mainthread(g)->env));
  gc_marktv(g, &g->registrytv);
  gc_mark_gcroot(g);
  g->gc.state = GCSpropagate;
}

/* Mark open upvalues. */
static void gc_mark_uv(global_State *g)
{
  GCupval *uv;
  for (uv = uvnext(&g->uvhead); uv != &g->uvhead; uv = uvnext(uv)) {
    lua_assert(uvprev(uvnext(uv)) == uv && uvnext(uvprev(uv)) == uv);
    if (isgray(obj2gco(uv)))
      gc_marktv(g, uvval(uv));
  }
}

/* Mark userdata in mmudata list. */
static void gc_mark_mmudata(global_State *g)
{
  GCobj *root = gcref(g->gc.mmudata);
  GCobj *u = root;
  if (u) {
    do {
      u = gcnext(u);
      makewhite(g, u);  /* Could be from previous GC. */
      gc_mark(g, u);
    } while (u != root);
  }
}

/* Separate userdata objects to be finalized to mmudata list. */
size_t lj_gc_separateudata(global_State *g, int all)
{
  size_t m = 0;
  GCRef *p = &mainthread(g)->nextgc;
  GCobj *o;
  while ((o = gcref(*p)) != NULL) {
    if (!(iswhite(o) || all) || isfinalized(gco2ud(o))) {
      p = &o->gch.nextgc;  /* Nothing to do. */
    } else if (!lj_meta_fastg(g, tabref(gco2ud(o)->metatable), MM_gc)) {
      markfinalized(o);  /* Done, as there's no __gc metamethod. */
      p = &o->gch.nextgc;
    } else {  /* Otherwise move userdata to be finalized to mmudata list. */
      m += sizeudata(gco2ud(o));
      markfinalized(o);
      *p = o->gch.nextgc;
      if (gcref(g->gc.mmudata)) {  /* Link to end of mmudata list. */
        GCobj *root = gcref(g->gc.mmudata);
        setgcrefr(o->gch.nextgc, root->gch.nextgc);
        setgcref(root->gch.nextgc, o);
        setgcref(g->gc.mmudata, o);
      } else {  /* Create circular list. */
        setgcref(o->gch.nextgc, o);
        setgcref(g->gc.mmudata, o);
      }
    }
  }
  return m;
}

/* -- Propagation phase ---------------------------------------------------- */

/* Traverse a table. */
```

```c
static int gc_traverse_tab(global_State *g, GCtab *t)
{
  int weak = 0;
  cTValue *mode;
  GCtab *mt = tabref(t->metatable);
  if (mt)
    gc_markobj(g, mt);
  mode = lj_meta_fastg(g, mt, MM_mode);
  if (mode && tvisstr(mode)) {  /* Valid __mode field? */
    const char *modestr = strVdata(mode);
    int c;
    while ((c = *modestr++)) {
      if (c == 'k') weak |= LJ_GC_WEAKKEY;
      else if (c == 'v') weak |= LJ_GC_WEAKVAL;
      else if (c == 'K') weak = (int)(~0u & ~LJ_GC_WEAKVAL);
    }
    if (weak > 0) {  /* Weak tables are cleared in the atomic phase. */
      t->marked = (uint8_t)((t->marked & ~LJ_GC_WEAK) | weak);
      setgcrefr(t->gclist, g->gc.weak);
      setgcref(g->gc.weak, obj2gco(t));
    }
  }
  if (weak == LJ_GC_WEAK)  /* Nothing to mark if both keys/values are weak. */
    return 1;
  if (!(weak & LJ_GC_WEAKVAL)) {  /* Mark array part. */
    MSize i, asize = t->asize;
    for (i = 0; i < asize; i++)
      gc_marktv(g, arrayslot(t, i));
  }
  if (t->hmask > 0) {  /* Mark hash part. */
    Node *node = noderef(t->node);
    MSize i, hmask = t->hmask;
    for (i = 0; i <= hmask; i++) {
      Node *n = &node[i];
      if (!tvisnil(&n->val)) {  /* Mark non-empty slot. */
        lua_assert(!tvisnil(&n->key));
        if (!(weak & LJ_GC_WEAKKEY)) gc_marktv(g, &n->key);
        if (!(weak & LJ_GC_WEAKVAL)) gc_marktv(g, &n->val);
      }
    }
  }
  return weak;
}

/* Traverse a function. */
static void gc_traverse_func(global_State *g, GCfunc *fn)
{
  gc_markobj(g, tabref(fn->c.env));
  if (isluafunc(fn)) {
    uint32_t i;
    lua_assert(fn->l.nupvalues <= funcproto(fn)->sizeuv);
    gc_markobj(g, funcproto(fn));
    for (i = 0; i < fn->l.nupvalues; i++)  /* Mark Lua function upvalues. */
      gc_markobj(g, &gcref(fn->l.uvptr[i])->uv);
  } else {
    uint32_t i;
    for (i = 0; i < fn->c.nupvalues; i++)  /* Mark C function upvalues. */
      gc_marktv(g, &fn->c.upvalue[i]);
  }
}

#if LJ_HASJIT
/* Mark a trace. */
static void gc_marktrace(global_State *g, TraceNo traceno)
{
  GCobj *o = obj2gco(traceref(G2J(g), traceno));
  lua_assert(traceno != G2J(g)->cur.traceno);
  if (iswhite(o)) {
    white2gray(o);
    setgcrefr(o->gch.gclist, g->gc.gray);
    setgcref(g->gc.gray, o);
  }
}

/* Traverse a trace. */
static void gc_traverse_trace(global_State *g, GCtrace *T)
```

```
{
  IRRef ref;
  if (T->traceno == 0) return;
  for (ref = T->nk; ref < REF_TRUE; ref++) {
    IRIns *ir = &T->ir[ref];
    if (ir->o == IR_KGC)
      gc_markobj(g, ir_kgc(ir));
  }
  if (T->link) gc_marktrace(g, T->link);
  if (T->nextroot) gc_marktrace(g, T->nextroot);
  if (T->nextside) gc_marktrace(g, T->nextside);
  gc_markobj(g, gcref(T->startpt));
}

/* The current trace is a GC root while not anchored in the prototype (yet). */
#define gc_traverse_curtrace(g)        gc_traverse_trace(g, &G2J(g)->cur)
#else
#define gc_traverse_curtrace(g)        UNUSED(g)
#endif

/* Traverse a prototype. */
static void gc_traverse_proto(global_State *g, GCproto *pt)
{
  ptrdiff_t i;
  gc_mark_str(proto_chunkname(pt));
  for (i = -(ptrdiff_t)pt->sizekgc; i < 0; i++)  /* Mark collectable consts. */
    gc_markobj(g, proto_kgc(pt, i));
#if LJ_HASJIT
  if (pt->trace) gc_marktrace(g, pt->trace);
#endif
}

/* Traverse the frame structure of a stack. */
static MSize gc_traverse_frames(global_State *g, lua_State *th)
{
  TValue *frame, *top = th->top-1, *bot = tvref(th->stack);
  /* Note: extra vararg frame not skipped, marks function twice (harmless). */
  for (frame = th->base-1; frame > bot+LJ_FR2; frame = frame_prev(frame)) {
    GCfunc *fn = frame_func(frame);
    TValue *ftop = frame;
    if (isluafunc(fn)) ftop += funcproto(fn)->framesize;
    if (ftop > top) top = ftop;
    if (!LJ_FR2) gc_markobj(g, fn);  /* Need to mark hidden function (or L). */
  }
  top++;  /* Correct bias of -1 (frame == base-1). */
  if (top > tvref(th->maxstack)) top = tvref(th->maxstack);
  return (MSize)(top - bot);  /* Return minimum needed stack size. */
}

/* Traverse a thread object. */
static void gc_traverse_thread(global_State *g, lua_State *th)
{
  TValue *o, *top = th->top;
  for (o = tvref(th->stack)+1+LJ_FR2; o < top; o++)
    gc_marktv(g, o);
  if (g->gc.state == GCSatomic) {
    top = tvref(th->stack) + th->stacksize;
    for (; o < top; o++)  /* Clear unmarked slots. */
      setnilV(o);
  }
  gc_markobj(g, tabref(th->env));
  lj_state_shrinkstack(th, gc_traverse_frames(g, th));
}

/* Propagate one gray object. Traverse it and turn it black. */
static size_t propagatemark(global_State *g)
{
  GCobj *o = gcref(g->gc.gray);
  int gct = o->gch.gct;
  lua_assert(isgray(o));
  gray2black(o);
  setgcrefr(g->gc.gray, o->gch.gclist);  /* Remove from gray list. */
  if (LJ_LIKELY(gct == ~LJ_TTAB)) {
    GCtab *t = gco2tab(o);
    if (gc_traverse_tab(g, t) > 0)
      black2gray(o);  /* Keep weak tables gray. */
```

```
    return sizeof(GCtab) + sizeof(TValue) * t->asize +
                           sizeof(Node) * (t->hmask + 1);
  } else if (LJ_LIKELY(gct == ~LJ_TFUNC)) {
    GCfunc *fn = gco2func(o);
    gc_traverse_func(g, fn);
    return isluafunc(fn) ? sizeLfunc((MSize)fn->l.nupvalues) :
                           sizeCfunc((MSize)fn->c.nupvalues);
  } else if (LJ_LIKELY(gct == ~LJ_TPROTO)) {
    GCproto *pt = gco2pt(o);
    gc_traverse_proto(g, pt);
    return pt->sizept;
  } else if (LJ_LIKELY(gct == ~LJ_TTHREAD)) {
    lua_State *th = gco2th(o);
    setgcrefr(th->gclist, g->gc.grayagain);
    setgcref(g->gc.grayagain, o);
    black2gray(o);  /* Threads are never black. */
    gc_traverse_thread(g, th);
    return sizeof(lua_State) + sizeof(TValue) * th->stacksize;
  } else {
#if LJ_HASJIT
    GCtrace *T = gco2trace(o);
    gc_traverse_trace(g, T);
    return ((sizeof(GCtrace)+7)&~7) + (T->nins-T->nk)*sizeof(IRIns) +
           T->nsnap*sizeof(SnapShot) + T->nsnapmap*sizeof(SnapEntry);
#else
    lua_assert(0);
    return 0;
#endif
  }
}

/* Propagate all gray objects. */
static size_t gc_propagate_gray(global_State *g)
{
  size_t m = 0;
  while (gcref(g->gc.gray) != NULL)
    m += propagatemark(g);
  return m;
}

/* -- Sweep phase ------------------------------------------------------- */

/* Type of GC free functions. */
typedef void (LJ_FASTCALL *GCFreeFunc)(global_State *g, GCobj *o);

/* GC free functions for LJ_TSTR .. LJ_TUDATA. ORDER LJ_T */
static const GCFreeFunc gc_freefunc[] = {
  (GCFreeFunc)lj_str_free,
  (GCFreeFunc)lj_func_freeuv,
  (GCFreeFunc)lj_state_free,
  (GCFreeFunc)lj_func_freeproto,
  (GCFreeFunc)lj_func_free,
#if LJ_HASJIT
  (GCFreeFunc)lj_trace_free,
#else
  (GCFreeFunc)0,
#endif
#if LJ_HASFFI
  (GCFreeFunc)lj_cdata_free,
#else
  (GCFreeFunc)0,
#endif
  (GCFreeFunc)lj_tab_free,
  (GCFreeFunc)lj_udata_free
};

/* Full sweep of a GC list. */
#define gc_fullsweep(g, p)        gc_sweep(g, (p), ~(uint32_t)0)

/* Partial sweep of a GC list. */
static GCRef *gc_sweep(global_State *g, GCRef *p, uint32_t lim)
{
  /* Mask with other white and LJ_GC_FIXED. Or LJ_GC_SFIXED on shutdown. */
  int ow = otherwhite(g);
  GCobj *o;
  while ((o = gcref(*p)) != NULL && lim-- > 0) {
```

```
    if (o->gch.gct == ~LJ_TTHREAD)  /* Need to sweep open upvalues, too. */
      gc_fullsweep(g, &gco2th(o)->openupval);
    if (((o->gch.marked ^ LJ_GC_WHITES) & ow)) {  /* Black or current white? */
      lua_assert(!isdead(g, o) || (o->gch.marked & LJ_GC_FIXED));
      makewhite(g, o);  /* Value is alive, change to the current white. */
      p = &o->gch.nextgc;
    } else {  /* Otherwise value is dead, free it. */
      lua_assert(isdead(g, o) || ow == LJ_GC_SFIXED);
      setgcrefr(*p, o->gch.nextgc);
      if (o == gcref(g->gc.root))
        setgcrefr(g->gc.root, o->gch.nextgc);  /* Adjust list anchor. */
      gc_freefunc[o->gch.gct - ~LJ_TSTR](g, o);
    }
  }
  return p;
}

/* Check whether we can clear a key or a value slot from a table. */
static int gc_mayclear(cTValue *o, int val)
{
  if (tvisgcv(o)) {  /* Only collectable objects can be weak references. */
    if (tvisstr(o)) {  /* But strings cannot be used as weak references. */
      gc_mark_str(strV(o));  /* And need to be marked. */
      return 0;
    }
    if (iswhite(gcV(o)))
      return 1;  /* Object is about to be collected. */
    if (tvisudata(o) && val && isfinalized(udataV(o)))
      return 1;  /* Finalized userdata is dropped only from values. */
  }
  return 0;  /* Cannot clear. */
}

/* Clear collected entries from weak tables. */
static void gc_clearweak(GCobj *o)
{
  while (o) {
    GCtab *t = gco2tab(o);
    lua_assert((t->marked & LJ_GC_WEAK));
    if ((t->marked & LJ_GC_WEAKVAL)) {
      MSize i, asize = t->asize;
      for (i = 0; i < asize; i++) {
        /* Clear array slot when value is about to be collected. */
        TValue *tv = arrayslot(t, i);
        if (gc_mayclear(tv, 1))
          setnilV(tv);
      }
    }
    if (t->hmask > 0) {
      Node *node = noderef(t->node);
      MSize i, hmask = t->hmask;
      for (i = 0; i <= hmask; i++) {
        Node *n = &node[i];
        /* Clear hash slot when key or value is about to be collected. */
        if (!tvisnil(&n->val) && (gc_mayclear(&n->key, 0) ||
                                  gc_mayclear(&n->val, 1)))
          setnilV(&n->val);
      }
    }
    o = gcref(t->gclist);
  }
}

/* Call a userdata or cdata finalizer. */
static void gc_call_finalizer(global_State *g, lua_State *L,
                              cTValue *mo, GCobj *o)
{
  /* Save and restore lots of state around the __gc callback. */
  uint8_t oldh = hook_save(g);
  GCSize oldt = g->gc.threshold;
  int errcode;
  TValue *top;
  lj_trace_abort(g);
  hook_entergc(g);  /* Disable hooks and new traces during __gc. */
  g->gc.threshold = LJ_MAX_MEM;  /* Prevent GC steps. */
  top = L->top;
```

```
  copyTV(L, top++, mo);
  if (LJ_FR2) setnilV(top++);
  setgcV(L, top, o, ~o->gch.gct);
  L->top = top+1;
  errcode = lj_vm_pcall(L, top, 1+0, -1);  /* Stack: |mo|o| -> | */
  hook_restore(g, oldh);
  g->gc.threshold = oldt;  /* Restore GC threshold. */
  if (errcode)
    lj_err_throw(L, errcode);  /* Propagate errors. */
}

/* Finalize one userdata or cdata object from the mmudata list. */
static void gc_finalize(lua_State *L)
{
  global_State *g = G(L);
  GCobj *o = gcnext(gcref(g->gc.mmudata));
  cTValue *mo;
  lua_assert(tvref(g->jit_base) == NULL);  /* Must not be called on trace. */
  /* Unchain from list of userdata to be finalized. */
  if (o == gcref(g->gc.mmudata))
    setgcrefnull(g->gc.mmudata);
  else
    setgcrefr(gcref(g->gc.mmudata)->gch.nextgc, o->gch.nextgc);
#if LJ_HASFFI
  if (o->gch.gct == ~LJ_TCDATA) {
    TValue tmp, *tv;
    /* Add cdata back to the GC list and make it white. */
    setgcrefr(o->gch.nextgc, g->gc.root);
    setgcref(g->gc.root, o);
    makewhite(g, o);
    o->gch.marked &= (uint8_t)~LJ_GC_CDATA_FIN;
    /* Resolve finalizer. */
    setcdataV(L, &tmp, gco2cd(o));
    tv = lj_tab_set(L, ctype_ctsG(g)->finalizer, &tmp);
    if (!tvisnil(tv)) {
      g->gc.nocdatafin = 0;
      copyTV(L, &tmp, tv);
      setnilV(tv);  /* Clear entry in finalizer table. */
      gc_call_finalizer(g, L, &tmp, o);
    }
    return;
  }
#endif
  /* Add userdata back to the main userdata list and make it white. */
  setgcrefr(o->gch.nextgc, mainthread(g)->nextgc);
  setgcref(mainthread(g)->nextgc, o);
  makewhite(g, o);
  /* Resolve the __gc metamethod. */
  mo = lj_meta_fastg(g, tabref(gco2ud(o)->metatable), MM_gc);
  if (mo)
    gc_call_finalizer(g, L, mo, o);
}

/* Finalize all userdata objects from mmudata list. */
void lj_gc_finalize_udata(lua_State *L)
{
  while (gcref(G(L)->gc.mmudata) != NULL)
    gc_finalize(L);
}

#if LJ_HASFFI
/* Finalize all cdata objects from finalizer table. */
void lj_gc_finalize_cdata(lua_State *L)
{
  global_State *g = G(L);
  CTState *cts = ctype_ctsG(g);
  if (cts) {
    GCtab *t = cts->finalizer;
    Node *node = noderef(t->node);
    ptrdiff_t i;
    setgcrefnull(t->metatable);  /* Mark finalizer table as disabled. */
    for (i = (ptrdiff_t)t->hmask; i >= 0; i--)
      if (!tvisnil(&node[i].val) && tviscdata(&node[i].key)) {
        GCobj *o = gcV(&node[i].key);
        TValue tmp;
        makewhite(g, o);
```

```
        o->gch.marked &= (uint8_t)~LJ_GC_CDATA_FIN;
        copyTV(L, &tmp, &node[i].val);
        setnilV(&node[i].val);
        gc_call_finalizer(g, L, &tmp, o);
      }
    }
  }
#endif

/* Free all remaining GC objects. */
void lj_gc_freeall(global_State *g)
{
  MSize i, strmask;
  /* Free everything, except super-fixed objects (the main thread). */
  g->gc.currentwhite = LJ_GC_WHITES | LJ_GC_SFIXED;
  gc_fullsweep(g, &g->gc.root);
  strmask = g->strmask;
  for (i = 0; i <= strmask; i++)  /* Free all string hash chains. */
    gc_fullsweep(g, &g->strhash[i]);
}

/* -- Collector ------------------------------------------------------------ */

/* Atomic part of the GC cycle, transitioning from mark to sweep phase. */
static void atomic(global_State *g, lua_State *L)
{
  size_t udsize;

  gc_mark_uv(g);  /* Need to remark open upvalues (the thread may be dead). */
  gc_propagate_gray(g);  /* Propagate any left-overs. */

  setgcrefr(g->gc.gray, g->gc.weak);  /* Empty the list of weak tables. */
  setgcrefnull(g->gc.weak);
  lua_assert(!iswhite(obj2gco(mainthread(g))));
  gc_markobj(g, L);  /* Mark running thread. */
  gc_traverse_curtrace(g);  /* Traverse current trace. */
  gc_mark_gcroot(g);  /* Mark GC roots (again). */
  gc_propagate_gray(g);  /* Propagate all of the above. */

  setgcrefr(g->gc.gray, g->gc.grayagain);  /* Empty the 2nd chance list. */
  setgcrefnull(g->gc.grayagain);
  gc_propagate_gray(g);  /* Propagate it. */

  udsize = lj_gc_separateudata(g, 0);  /* Separate userdata to be finalized. */
  gc_mark_mmudata(g);  /* Mark them. */
  udsize += gc_propagate_gray(g);  /* And propagate the marks. */

  /* All marking done, clear weak tables. */
  gc_clearweak(gcref(g->gc.weak));

  lj_buf_shrink(L, &g->tmpbuf);  /* Shrink temp buffer. */

  /* Prepare for sweep phase. */
  g->gc.currentwhite = (uint8_t)otherwhite(g);  /* Flip current white. */
  g->strempty.marked = g->gc.currentwhite;
  setmref(g->gc.sweep, &g->gc.root);
  g->gc.estimate = g->gc.total - (GCSize)udsize;  /* Initial estimate. */
}

/* GC state machine. Returns a cost estimate for each step performed. */
static size_t gc_onestep(lua_State *L)
{
  global_State *g = G(L);
  switch (g->gc.state) {
  case GCSpause:
    gc_mark_start(g);  /* Start a new GC cycle by marking all GC roots. */
    return 0;
  case GCSpropagate:
    if (gcref(g->gc.gray) != NULL)
      return propagatemark(g);  /* Propagate one gray object. */
    g->gc.state = GCSatomic;  /* End of mark phase. */
    return 0;
  case GCSatomic:
    if (tvref(g->jit_base))  /* Don't run atomic phase on trace. */
      return LJ_MAX_MEM;
    atomic(g, L);
```

```
    g->gc.state = GCSsweepstring;  /* Start of sweep phase. */
    g->gc.sweepstr = 0;
    return 0;
  case GCSsweepstring: {
    GCSize old = g->gc.total;
    gc_fullsweep(g, &g->strhash[g->gc.sweepstr++]);  /* Sweep one chain. */
    if (g->gc.sweepstr > g->strmask)
      g->gc.state = GCSsweep;  /* All string hash chains sweeped. */
    lua_assert(old >= g->gc.total);
    g->gc.estimate -= old - g->gc.total;
    return GCSWEEPCOST;
    }
  case GCSsweep: {
    GCSize old = g->gc.total;
    setmref(g->gc.sweep, gc_sweep(g, mref(g->gc.sweep, GCRef), GCSWEEPMAX));
    lua_assert(old >= g->gc.total);
    g->gc.estimate -= old - g->gc.total;
    if (gcref(*mref(g->gc.sweep, GCRef)) == NULL) {
      if (g->strnum <= (g->strmask >> 2) && g->strmask > LJ_MIN_STRTAB*2-1)
        lj_str_resize(L, g->strmask >> 1);  /* Shrink string table. */
      if (gcref(g->gc.mmudata)) {  /* Need any finalizations? */
        g->gc.state = GCSfinalize;
#if LJ_HASFFI
        g->gc.nocdatafin = 1;
#endif
      } else {  /* Otherwise skip this phase to help the JIT. */
        g->gc.state = GCSpause;  /* End of GC cycle. */
        g->gc.debt = 0;
      }
    }
    return GCSWEEPMAX*GCSWEEPCOST;
    }
  case GCSfinalize:
    if (gcref(g->gc.mmudata) != NULL) {
      if (tvref(g->jit_base))  /* Don't call finalizers on trace. */
        return LJ_MAX_MEM;
      gc_finalize(L);  /* Finalize one userdata object. */
      if (g->gc.estimate > GCFINALIZECOST)
        g->gc.estimate -= GCFINALIZECOST;
      return GCFINALIZECOST;
    }
#if LJ_HASFFI
    if (!g->gc.nocdatafin) lj_tab_rehash(L, ctype_ctsG(g)->finalizer);
#endif
    g->gc.state = GCSpause;  /* End of GC cycle. */
    g->gc.debt = 0;
    return 0;
  default:
    lua_assert(0);
    return 0;
  }
}

/* Perform a limited amount of incremental GC steps. */
int LJ_FASTCALL lj_gc_step(lua_State *L)
{
  global_State *g = G(L);
  GCSize lim;
  int32_t ostate = g->vmstate;
  setvmstate(g, GC);
  lim = (GCSTEPSIZE/100) * g->gc.stepmul;
  if (lim == 0)
    lim = LJ_MAX_MEM;
  if (g->gc.total > g->gc.threshold)
    g->gc.debt += g->gc.total - g->gc.threshold;
  do {
    lim -= (GCSize)gc_onestep(L);
    if (g->gc.state == GCSpause) {
      g->gc.threshold = (g->gc.estimate/100) * g->gc.pause;
      g->vmstate = ostate;
      return 1;  /* Finished a GC cycle. */
    }
  } while (sizeof(lim) == 8 ? ((int64_t)lim > 0) : ((int32_t)lim > 0));
  if (g->gc.debt < GCSTEPSIZE) {
    g->gc.threshold = g->gc.total + GCSTEPSIZE;
    g->vmstate = ostate;
```

```
    return -1;
  } else {
    g->gc.debt -= GCSTEPSIZE;
    g->gc.threshold = g->gc.total;
    g->vmstate = ostate;
    return 0;
  }
}

/* Ditto, but fix the stack top first. */
void LJ_FASTCALL lj_gc_step_fixtop(lua_State *L)
{
  if (curr_funcisL(L)) L->top = curr_topL(L);
  lj_gc_step(L);
}

#if LJ_HASJIT
/* Perform multiple GC steps. Called from JIT-compiled code. */
int LJ_FASTCALL lj_gc_step_jit(global_State *g, MSize steps)
{
  lua_State *L = gco2th(gcref(g->cur_L));
  L->base = tvref(G(L)->jit_base);
  L->top = curr_topL(L);
  while (steps-- > 0 && lj_gc_step(L) == 0)
    ;
  /* Return 1 to force a trace exit. */
  return (G(L)->gc.state == GCSatomic || G(L)->gc.state == GCSfinalize);
}
#endif

/* Perform a full GC cycle. */
void lj_gc_fullgc(lua_State *L)
{
  global_State *g = G(L);
  int32_t ostate = g->vmstate;
  setvmstate(g, GC);
  if (g->gc.state <= GCSatomic) {  /* Caught somewhere in the middle. */
    setmref(g->gc.sweep, &g->gc.root);  /* Sweep everything (preserving it). */
    setgcrefnull(g->gc.gray);  /* Reset lists from partial propagation. */
    setgcrefnull(g->gc.grayagain);
    setgcrefnull(g->gc.weak);
    g->gc.state = GCSsweepstring;  /* Fast forward to the sweep phase. */
    g->gc.sweepstr = 0;
  }
  while (g->gc.state == GCSsweepstring || g->gc.state == GCSsweep)
    gc_onestep(L);  /* Finish sweep. */
  lua_assert(g->gc.state == GCSfinalize || g->gc.state == GCSpause);
  /* Now perform a full GC. */
  g->gc.state = GCSpause;
  do { gc_onestep(L); } while (g->gc.state != GCSpause);
  g->gc.threshold = (g->gc.estimate/100) * g->gc.pause;
  g->vmstate = ostate;
}

/* -- Write barriers ------------------------------------------------------ */

/* Move the GC propagation frontier forward. */
void lj_gc_barrierf(global_State *g, GCobj *o, GCobj *v)
{
  lua_assert(isblack(o) && iswhite(v) && !isdead(g, v) && !isdead(g, o));
  lua_assert(g->gc.state != GCSfinalize && g->gc.state != GCSpause);
  lua_assert(o->gch.gct != ~LJ_TTAB);
  /* Preserve invariant during propagation. Otherwise it doesn't matter. */
  if (g->gc.state == GCSpropagate || g->gc.state == GCSatomic)
    gc_mark(g, v);  /* Move frontier forward. */
  else
    makewhite(g, o);  /* Make it white to avoid the following barrier. */
}

/* Specialized barrier for closed upvalue. Pass &uv->tv. */
void LJ_FASTCALL lj_gc_barrieruv(global_State *g, TValue *tv)
{
#define TV2MARKED(x) \
  (*((uint8_t *)(x) - offsetof(GCupval, tv) + offsetof(GCupval, marked)))
  if (g->gc.state == GCSpropagate || g->gc.state == GCSatomic)
    gc_mark(g, gcV(tv));
```

```
  else
    TV2MARKED(tv) = (TV2MARKED(tv) & (uint8_t)~LJ_GC_COLORS) | curwhite(g);
#undef TV2MARKED
}

/* Close upvalue. Also needs a write barrier. */
void lj_gc_closeuv(global_State *g, GCupval *uv)
{
  GCobj *o = obj2gco(uv);
  /* Copy stack slot to upvalue itself and point to the copy. */
  copyTV(mainthread(g), &uv->tv, uvval(uv));
  setmref(uv->v, &uv->tv);
  uv->closed = 1;
  setgcrefr(o->gch.nextgc, g->gc.root);
  setgcref(g->gc.root, o);
  if (isgray(o)) {  /* A closed upvalue is never gray, so fix this. */
    if (g->gc.state == GCSpropagate || g->gc.state == GCSatomic) {
      gray2black(o);  /* Make it black and preserve invariant. */
      if (tviswhite(&uv->tv))
        lj_gc_barrierf(g, o, gcV(&uv->tv));
    } else {
      makewhite(g, o);  /* Make it white, i.e. sweep the upvalue. */
      lua_assert(g->gc.state != GCSfinalize && g->gc.state != GCSpause);
    }
  }
}

#if LJ_HASJIT
/* Mark a trace if it's saved during the propagation phase. */
void lj_gc_barriertrace(global_State *g, uint32_t traceno)
{
  if (g->gc.state == GCSpropagate || g->gc.state == GCSatomic)
    gc_marktrace(g, traceno);
}
#endif

/* -- Allocator ------------------------------------------------------------ */

/* Call pluggable memory allocator to allocate or resize a fragment. */
void *lj_mem_realloc(lua_State *L, void *p, GCSize osz, GCSize nsz)
{
  global_State *g = G(L);
  lua_assert((osz == 0) == (p == NULL));
  p = g->allocf(g->allocd, p, osz, nsz);
  if (p == NULL && nsz > 0)
    lj_err_mem(L);
  lua_assert((nsz == 0) == (p == NULL));
  lua_assert(checkptrGC(p));
  g->gc.total = (g->gc.total - osz) + nsz;
  return p;
}

/* Allocate new GC object and link it to the root set. */
void * LJ_FASTCALL lj_mem_newgco(lua_State *L, GCSize size)
{
  global_State *g = G(L);
  GCobj *o = (GCobj *)g->allocf(g->allocd, NULL, 0, size);
  if (o == NULL)
    lj_err_mem(L);
  lua_assert(checkptrGC(o));
  g->gc.total += size;
  setgcrefr(o->gch.nextgc, g->gc.root);
  setgcref(g->gc.root, o);
  newwhite(g, o);
  return o;
}

/* Resize growable vector. */
void *lj_mem_grow(lua_State *L, void *p, MSize *szp, MSize lim, MSize esz)
{
  MSize sz = (*szp) << 1;
  if (sz < LJ_MIN_VECSZ)
    sz = LJ_MIN_VECSZ;
  if (sz > lim)
    sz = lim;
  p = lj_mem_realloc(L, p, (*szp)*esz, sz*esz);
```

```
842   *szp = sz;
843   return p;
844 }
845
```

One Level Up Top Level

src/lj_meta.h - luajit-2.0-src

Macros defined

- _LJ_META_H
- lj_meta_fast
- lj_meta_fastg

Source code

```
1  /*
2  ** Metamethod handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_META_H
7  #define _LJ_META_H
8
9  #include "lj_obj.h"
10
11 /* Metamethod handling */
12 LJ_FUNC void lj_meta_init(lua_State *L);
13 LJ_FUNC cTValue *lj_meta_cache(GCtab *mt, MMS mm, GCstr *name);
14 LJ_FUNC cTValue *lj_meta_lookup(lua_State *L, cTValue *o, MMS mm);
15 #if LJ_HASFFI
16 LJ_FUNC int lj_meta_tailcall(lua_State *L, cTValue *tv);
17 #endif
18
19 #define lj_meta_fastg(g, mt, mm) \
20   ((mt) == NULL ? NULL : ((mt)->nomm & (1u<<(mm))) ? NULL : \
21    lj_meta_cache(mt, mm, mmname_str(g, mm)))
22 #define lj_meta_fast(L, mt, mm)        lj_meta_fastg(G(L), mt, mm)
23
24 /* C helpers for some instructions, called from assembler VM. */
25 LJ_FUNCA cTValue *lj_meta_tget(lua_State *L, cTValue *o, cTValue *k);
26 LJ_FUNCA TValue *lj_meta_tset(lua_State *L, cTValue *o, cTValue *k);
27 LJ_FUNCA TValue *lj_meta_arith(lua_State *L, TValue *ra, cTValue *rb,
28                                cTValue *rc, BCReg op);
29 LJ_FUNCA TValue *lj_meta_cat(lua_State *L, TValue *top, int left);
30 LJ_FUNCA TValue * LJ_FASTCALL lj_meta_len(lua_State *L, cTValue *o);
31 LJ_FUNCA TValue *lj_meta_equal(lua_State *L, GCobj *o1, GCobj *o2, int ne);
32 LJ_FUNCA TValue * LJ_FASTCALL lj_meta_equal_cd(lua_State *L, BCIns ins);
33 LJ_FUNCA TValue *lj_meta_comp(lua_State *L, cTValue *o1, cTValue *o2, int op);
34 LJ_FUNCA void lj_meta_istype(lua_State *L, BCReg ra, BCReg tp);
35 LJ_FUNCA void lj_meta_call(lua_State *L, TValue *func, TValue *top);
36 LJ_FUNCA void LJ_FASTCALL lj_meta_for(lua_State *L, TValue *o);
37
38 #endif
```

src/lj_meta.c - luajit-2.0-src

Functions defined

Macros defined

Source code

```
1  /*
2  ** Metamethod handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  **
5  ** Portions taken verbatim or adapted from the Lua interpreter.
6  ** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
7  */
8
9  #define lj_meta_c
10  #define LUA_CORE
11
12  #include "lj_obj.h"
13  #include "lj_gc.h"
14  #include "lj_err.h"
```

```c
#include "lj_buf.h"
#include "lj_str.h"
#include "lj_tab.h"
#include "lj_meta.h"
#include "lj_frame.h"
#include "lj_bc.h"
#include "lj_vm.h"
#include "lj_strscan.h"
#include "lj_strfmt.h"
#include "lj_lib.h"

/* -- Metamethod handling ------------------------------------------------ */

/* String interning of metamethod names for fast indexing. */
void lj_meta_init(lua_State *L)
{
#define MMNAME(name)        "__" #name
  const char *metanames = MMDEF(MMNAME);
#undef MMNAME
  global_State *g = G(L);
  const char *p, *q;
  uint32_t mm;
  for (mm = 0, p = metanames; *p; mm++, p = q) {
    GCstr *s;
    for (q = p+2; *q && *q != '_'; q++) ;
    s = lj_str_new(L, p, (size_t)(q-p));
    /* NOBARRIER: g->gcroot[] is a GC root. */
    setgcref(g->gcroot[GCROOT_MMNAME+mm], obj2gco(s));
  }
}

/* Negative caching of a few fast metamethods. See the lj_meta_fast() macro. */
cTValue *lj_meta_cache(GCtab *mt, MMS mm, GCstr *name)
{
  cTValue *mo = lj_tab_getstr(mt, name);
  lua_assert(mm <= MM_FAST);
  if (!mo || tvisnil(mo)) {  /* No metamethod? */
    mt->nomm |= (uint8_t)(1u<<mm);  /* Set negative cache flag. */
    return NULL;
  }
  return mo;
}

/* Lookup metamethod for object. */
cTValue *lj_meta_lookup(lua_State *L, cTValue *o, MMS mm)
{
  GCtab *mt;
  if (tvistab(o))
    mt = tabref(tabV(o)->metatable);
  else if (tvisudata(o))
    mt = tabref(udataV(o)->metatable);
  else
    mt = tabref(basemt_obj(G(L), o));
  if (mt) {
    cTValue *mo = lj_tab_getstr(mt, mmname_str(G(L), mm));
    if (mo)
      return mo;
  }
  return niltv(L);
}

#if LJ_HASFFI
/* Tailcall from C function. */
int lj_meta_tailcall(lua_State *L, cTValue *tv)
{
  TValue *base = L->base;
  TValue *top = L->top;
  const BCIns *pc = frame_pc(base-1);  /* Preserve old PC from frame. */
  copyTV(L, base-1-LJ_FR2, tv);  /* Replace frame with new object. */
  if (LJ_FR2)
    (top++)->u64 = LJ_CONT_TAILCALL;
  else
    top->u32.lo = LJ_CONT_TAILCALL;
  setframe_pc(top++, pc);
  if (LJ_FR2) top++;
  setframe_gc(top, obj2gco(L), LJ_TTHREAD);  /* Dummy frame object. */
```

```
  setframe_ftsz(top, ((char *)(top+1) - (char *)base) + FRAME_CONT);
  L->base = L->top = top+1;
  /*
  ** before:   [old_mo|PC]    [... ...]
  **                         ^base     ^top
  ** after:    [new_mo|itype] [... ...] [NULL|PC] [dummy|delta]
  **                                                    ^base/top
  ** tailcall: [new_mo|PC]    [... ...]
  **                         ^base     ^top
   */
  return 0;
}
#endif

/* Setup call to metamethod to be run by Assembler VM. */
static TValue *mmcall(lua_State *L, ASMFunction cont, cTValue *mo,
                      cTValue *a, cTValue *b)
{
  /*
  **           |-- framesize -> top       top+1       top+2 top+3
  ** before:   [func slots ...]
  ** mm setup: [func slots ...] [cont|?]  [mo|tmtype] [a]   [b]
  ** in asm:   [func slots ...] [cont|PC] [mo|delta]  [a]   [b]
  **           ^-- func base                         ^-- mm base
  ** after mm: [func slots ...]           [result]
  **                ^-- copy to base[PC_RA] --/     for lj_cont_ra
  **                           istruecond + branch   for lj_cont_cond*
  **                                     ignore      for lj_cont_nop
  ** next PC:  [func slots ...]
  */
  TValue *top = L->top;
  if (curr_funcisL(L)) top = curr_topL(L);
  setcont(top++, cont);  /* Assembler VM stores PC in upper word or FR2. */
  if (LJ_FR2) setnilV(top++);
  copyTV(L, top++, mo);  /* Store metamethod and two arguments. */
  if (LJ_FR2) setnilV(top++);
  copyTV(L, top, a);
  copyTV(L, top+1, b);
  return top;  /* Return new base. */
}

/* -- C helpers for some instructions, called from assembler VM ----------- */

/* Helper for TGET*. __index chain and metamethod. */
cTValue *lj_meta_tget(lua_State *L, cTValue *o, cTValue *k)
{
  int loop;
  for (loop = 0; loop < LJ_MAX_IDXCHAIN; loop++) {
    cTValue *mo;
    if (LJ_LIKELY(tvistab(o))) {
      GCtab *t = tabV(o);
      cTValue *tv = lj_tab_get(L, t, k);
      if (!tvisnil(tv) ||
          !(mo = lj_meta_fast(L, tabref(t->metatable), MM_index)))
        return tv;
    } else if (tvisnil(mo = lj_meta_lookup(L, o, MM_index))) {
      lj_err_optype(L, o, LJ_ERR_OPINDEX);
      return NULL;  /* unreachable */
    }
    if (tvisfunc(mo)) {
      L->top = mmcall(L, lj_cont_ra, mo, o, k);
      return NULL;  /* Trigger metamethod call. */
    }
    o = mo;
  }
  lj_err_msg(L, LJ_ERR_GETLOOP);
  return NULL;  /* unreachable */
}

/* Helper for TSET*. __newindex chain and metamethod. */
TValue *lj_meta_tset(lua_State *L, cTValue *o, cTValue *k)
{
  TValue tmp;
  int loop;
  for (loop = 0; loop < LJ_MAX_IDXCHAIN; loop++) {
    cTValue *mo;
```

```
    if (LJ_LIKELY(tvistab(o))) {
      GCtab *t = tabV(o);
      cTValue *tv = lj_tab_get(L, t, k);
      if (LJ_LIKELY(!tvisnil(tv))) {
        t->nomm = 0;  /* Invalidate negative metamethod cache. */
        lj_gc_anybarriert(L, t);
        return (TValue *)tv;
      } else if (!(mo = lj_meta_fast(L, tabref(t->metatable), MM_newindex))) {
        t->nomm = 0;  /* Invalidate negative metamethod cache. */
        lj_gc_anybarriert(L, t);
        if (tv != niltv(L))
          return (TValue *)tv;
        if (tvisnil(k)) lj_err_msg(L, LJ_ERR_NILIDX);
        else if (tvisint(k)) { setnumV(&tmp, (lua_Number)intV(k)); k = &tmp; }
        else if (tvisnum(k) && tvisnan(k)) lj_err_msg(L, LJ_ERR_NANIDX);
        return lj_tab_newkey(L, t, k);
      }
    } else if (tvisnil(mo = lj_meta_lookup(L, o, MM_newindex))) {
      lj_err_optype(L, o, LJ_ERR_OPINDEX);
      return NULL;  /* unreachable */
    }
    if (tvisfunc(mo)) {
      L->top = mmcall(L, lj_cont_nop, mo, o, k);
      /* L->top+2 = v filled in by caller. */
      return NULL;  /* Trigger metamethod call. */
    }
    copyTV(L, &tmp, mo);
    o = &tmp;
  }
  lj_err_msg(L, LJ_ERR_SETLOOP);
  return NULL;  /* unreachable */
}

static cTValue *str2num(cTValue *o, TValue *n)
{
  if (tvisnum(o))
    return o;
  else if (tvisint(o))
    return (setnumV(n, (lua_Number)intV(o)), n);
  else if (tvisstr(o) && lj_strscan_num(strV(o), n))
    return n;
  else
    return NULL;
}

/* Helper for arithmetic instructions. Coercion, metamethod. */
TValue *lj_meta_arith(lua_State *L, TValue *ra, cTValue *rb, cTValue *rc,
                      BCReg op)
{
  MMS mm = bcmode_mm(op);
  TValue tempb, tempc;
  cTValue *b, *c;
  if ((b = str2num(rb, &tempb)) != NULL &&
      (c = str2num(rc, &tempc)) != NULL) {  /* Try coercion first. */
    setnumV(ra, lj_vm_foldarith(numV(b), numV(c), (int)mm-MM_add));
    return NULL;
  } else {
    cTValue *mo = lj_meta_lookup(L, rb, mm);
    if (tvisnil(mo)) {
      mo = lj_meta_lookup(L, rc, mm);
      if (tvisnil(mo)) {
        if (str2num(rb, &tempb) == NULL) rc = rb;
        lj_err_optype(L, rc, LJ_ERR_OPARITH);
        return NULL;  /* unreachable */
      }
    }
    return mmcall(L, lj_cont_ra, mo, rb, rc);
  }
}

/* Helper for CAT. Coercion, iterative concat, __concat metamethod. */
TValue *lj_meta_cat(lua_State *L, TValue *top, int left)
{
  int fromc = 0;
  if (left < 0) { left = -left; fromc = 1; }
  do {
```

```
    if (!(tvisstr(top) || tvisnumber(top)) ||
        !(tvisstr(top-1) || tvisnumber(top-1))) {
      cTValue *mo = lj_meta_lookup(L, top-1, MM_concat);
      if (tvisnil(mo)) {
        mo = lj_meta_lookup(L, top, MM_concat);
        if (tvisnil(mo)) {
          if (tvisstr(top-1) || tvisnumber(top-1)) top++;
          lj_err_optype(L, top-1, LJ_ERR_OPCAT);
          return NULL;  /* unreachable */
        }
      }
      /* One of the top two elements is not a string, call __cat metamethod:
      **
      ** before:    [...][CAT stack .......................]
      **                                  top-1     top         top+1 top+2
      ** pick two:  [...][CAT stack ...] [o1]      [o2]
      ** setup mm:  [...][CAT stack ...] [cont|?]  [mo|tmtype] [o1]  [o2]
      ** in asm:    [...][CAT stack ...] [cont|PC] [mo|delta]  [o1]  [o2]
      **            ^-- func base                              ^-- mm base
      ** after mm:  [...][CAT stack ...] <--push-- [result]
      ** next step: [...][CAT stack .............]
      */
      copyTV(L, top+2*LJ_FR2+2, top);  /* Carefully ordered stack copies! */
      copyTV(L, top+2*LJ_FR2+1, top-1);
      copyTV(L, top+LJ_FR2, mo);
      setcont(top-1, lj_cont_cat);
      if (LJ_FR2) { setnilV(top); setnilV(top+2); top += 2; }
      return top+1;  /* Trigger metamethod call. */
    } else {
      /* Pick as many strings as possible from the top and concatenate them:
      **
      ** before:    [...][CAT stack ........................]
      ** pick str:  [...][CAT stack ...] [..... strings .....]
      ** concat:    [...][CAT stack ...] [result]
      ** next step: [...][CAT stack ............]
      */
      TValue *e, *o = top;
      uint64_t tlen = tvisstr(o) ? strV(o)->len : STRFMT_MAXBUF_NUM;
      char *p, *buf;
      do {
        o--; tlen += tvisstr(o) ? strV(o)->len : STRFMT_MAXBUF_NUM;
      } while (--left > 0 && (tvisstr(o-1) || tvisnumber(o-1)));
      if (tlen >= LJ_MAX_STR) lj_err_msg(L, LJ_ERR_STROV);
      p = buf = lj_buf_tmp(L, (MSize)tlen);
      for (e = top, top = o; o <= e; o++) {
        if (tvisstr(o)) {
          GCstr *s = strV(o);
          MSize len = s->len;
          p = lj_buf_wmem(p, strdata(s), len);
        } else if (tvisint(o)) {
          p = lj_strfmt_wint(p, intV(o));
        } else {
          lua_assert(tvisnum(o));
          p = lj_strfmt_wnum(p, o);
        }
      }
      setstrV(L, top, lj_str_new(L, buf, (size_t)(p-buf)));
    }
  } while (left >= 1);
  if (LJ_UNLIKELY(G(L)->gc.total >= G(L)->gc.threshold)) {
    if (!fromc) L->top = curr_topL(L);
    lj_gc_step(L);
  }
  return NULL;
}

/* Helper for LEN. __len metamethod. */
TValue * LJ_FASTCALL lj_meta_len(lua_State *L, cTValue *o)
{
  cTValue *mo = lj_meta_lookup(L, o, MM_len);
  if (tvisnil(mo)) {
    if (LJ_52 && tvistab(o))
      tabref(tabV(o)->metatable)->nomm |= (uint8_t)(1u<<MM_len);
    else
      lj_err_optype(L, o, LJ_ERR_OPLEN);
    return NULL;
```

```
  }
  return mmcall(L, lj_cont_ra, mo, o, LJ_52 ? o : niltv(L));
}

/* Helper for equality comparisons. __eq metamethod. */
TValue *lj_meta_equal(lua_State *L, GCobj *o1, GCobj *o2, int ne)
{
  /* Field metatable must be at same offset for GCtab and GCudata! */
  cTValue *mo = lj_meta_fast(L, tabref(o1->gch.metatable), MM_eq);
  if (mo) {
    TValue *top;
    uint32_t it;
    if (tabref(o1->gch.metatable) != tabref(o2->gch.metatable)) {
      cTValue *mo2 = lj_meta_fast(L, tabref(o2->gch.metatable), MM_eq);
      if (mo2 == NULL || !lj_obj_equal(mo, mo2))
        return (TValue *)(intptr_t)ne;
    }
    top = curr_top(L);
    setcont(top++, ne ? lj_cont_condf : lj_cont_condt);
    if (LJ_FR2) setnilV(top++);
    copyTV(L, top++, mo);
    if (LJ_FR2) setnilV(top++);
    it = ~(uint32_t)o1->gch.gct;
    setgcV(L, top, o1, it);
    setgcV(L, top+1, o2, it);
    return top;  /* Trigger metamethod call. */
  }
  return (TValue *)(intptr_t)ne;
}

#if LJ_HASFFI
TValue * LJ_FASTCALL lj_meta_equal_cd(lua_State *L, BCIns ins)
{
  ASMFunction cont = (bc_op(ins) & 1) ? lj_cont_condf : lj_cont_condt;
  int op = (int)bc_op(ins) & ~1;
  TValue tv;
  cTValue *mo, *o2, *o1 = &L->base[bc_a(ins)];
  cTValue *o1mm = o1;
  if (op == BC_ISEQV) {
    o2 = &L->base[bc_d(ins)];
    if (!tviscdata(o1mm)) o1mm = o2;
  } else if (op == BC_ISEQS) {
    setstrV(L, &tv, gco2str(proto_kgc(curr_proto(L), ~(ptrdiff_t)bc_d(ins))));
    o2 = &tv;
  } else if (op == BC_ISEQN) {
    o2 = &mref(curr_proto(L)->k, cTValue)[bc_d(ins)];
  } else {
    lua_assert(op == BC_ISEQP);
    setpriV(&tv, ~bc_d(ins));
    o2 = &tv;
  }
  mo = lj_meta_lookup(L, o1mm, MM_eq);
  if (LJ_LIKELY(!tvisnil(mo)))
    return mmcall(L, cont, mo, o1, o2);
  else
    return (TValue *)(intptr_t)(bc_op(ins) & 1);
}
#endif

/* Helper for ordered comparisons. String compare, __lt/__le metamethods. */
TValue *lj_meta_comp(lua_State *L, cTValue *o1, cTValue *o2, int op)
{
  if (LJ_HASFFI && (tviscdata(o1) || tviscdata(o2))) {
    ASMFunction cont = (op & 1) ? lj_cont_condf : lj_cont_condt;
    MMS mm = (op & 2) ? MM_le : MM_lt;
    cTValue *mo = lj_meta_lookup(L, tviscdata(o1) ? o1 : o2, mm);
    if (LJ_UNLIKELY(tvisnil(mo))) goto err;
    return mmcall(L, cont, mo, o1, o2);
  } else if (LJ_52 || itype(o1) == itype(o2)) {
    /* Never called with two numbers. */
    if (tvisstr(o1) && tvisstr(o2)) {
      int32_t res = lj_str_cmp(strV(o1), strV(o2));
      return (TValue *)(intptr_t)(((op&2) ? res <= 0 : res < 0) ^ (op&1));
    } else {
    trymt:
      while (1) {
```

```
        ASMFunction cont = (op & 1) ? lj_cont_condf : lj_cont_condt;
        MMS mm = (op & 2) ? MM_le : MM_lt;
        cTValue *mo = lj_meta_lookup(L, o1, mm);
#if LJ_52
        if (tvisnil(mo) && tvisnil((mo = lj_meta_lookup(L, o2, mm))))
#else
        cTValue *mo2 = lj_meta_lookup(L, o2, mm);
        if (tvisnil(mo) || !lj_obj_equal(mo, mo2))
#endif
        {
          if (op & 2) {  /* MM_le not found: retry with MM_lt. */
            cTValue *ot = o1; o1 = o2; o2 = ot;  /* Swap operands. */
            op ^= 3;  /* Use LT and flip condition. */
            continue;
          }
          goto err;
        }
        return mmcall(L, cont, mo, o1, o2);
      }
    }
  } else if (tvisbool(o1) && tvisbool(o2)) {
    goto trymt;
  } else {
  err:
    lj_err_comp(L, o1, o2);
    return NULL;
  }
}

/* Helper for ISTYPE and ISNUM. Implicit coercion or error. */
void lj_meta_istype(lua_State *L, BCReg ra, BCReg tp)
{
  L->top = curr_topL(L);
  ra++; tp--;
  lua_assert(LJ_DUALNUM || tp != ~LJ_TNUMX);  /* ISTYPE -> ISNUM broken. */
  if (LJ_DUALNUM && tp == ~LJ_TNUMX) lj_lib_checkint(L, ra);
  else if (tp == ~LJ_TNUMX+1) lj_lib_checknum(L, ra);
  else if (tp == ~LJ_TSTR) lj_lib_checkstr(L, ra);
  else lj_err_argtype(L, ra, lj_obj_itypename[tp]);
}

/* Helper for calls. __call metamethod. */
void lj_meta_call(lua_State *L, TValue *func, TValue *top)
{
  cTValue *mo = lj_meta_lookup(L, func, MM_call);
  TValue *p;
  if (!tvisfunc(mo))
    lj_err_optype_call(L, func);
  for (p = top; p > func+2*LJ_FR2; p--) copyTV(L, p, p-1);
  if (LJ_FR2) copyTV(L, func+2, func);
  copyTV(L, func, mo);
}

/* Helper for FORI. Coercion. */
void LJ_FASTCALL lj_meta_for(lua_State *L, TValue *o)
{
  if (!lj_strscan_numberobj(o)) lj_err_msg(L, LJ_ERR_FORINIT);
  if (!lj_strscan_numberobj(o+1)) lj_err_msg(L, LJ_ERR_FORLIM);
  if (!lj_strscan_numberobj(o+2)) lj_err_msg(L, LJ_ERR_FORSTEP);
  if (LJ_DUALNUM) {
    /* Ensure all slots are integers or all slots are numbers. */
    int32_t k[3];
    int nint = 0;
    ptrdiff_t i;
    for (i = 0; i <= 2; i++) {
      if (tvisint(o+i)) {
        k[i] = intV(o+i); nint++;
      } else {
        k[i] = lj_num2int(numV(o+i)); nint += ((lua_Number)k[i] == numV(o+i));
      }
    }
    if (nint == 3) {  /* Narrow to integers. */
      setintV(o, k[0]);
      setintV(o+1, k[1]);
      setintV(o+2, k[2]);
    } else if (nint != 0) {  /* Widen to numbers. */
```

```
471       if (tvisint(o)) setnumV(o, (lua_Number)intV(o));
472       if (tvisint(o+1)) setnumV(o+1, (lua_Number)intV(o+1));
473       if (tvisint(o+2)) setnumV(o+2, (lua_Number)intV(o+2));
474     }
475   }
476 }
477
```

One Level Up Top Level

src/lj_str.c - luajit-2.0-src

Functions defined

Macros defined

Source code

```
1  /*
2  ** String handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lj_str_c
7  #define LUA_CORE
8
9  #include "lj_obj.h"
10 #include "lj_gc.h"
11 #include "lj_err.h"
12 #include "lj_str.h"
13 #include "lj_char.h"
14
15 /* -- String helpers ------------------------------------------------------ */
16
17 /* Ordered compare of strings. Assumes string data is 4-byte aligned. */
18 int32_t LJ_FASTCALL lj_str_cmp(GCstr *a, GCstr *b)
19 {
20   MSize i, n = a->len > b->len ? b->len : a->len;
21   for (i = 0; i < n; i += 4) {
22     /* Note: innocuous access up to end of string + 3. */
23     uint32_t va = *(const uint32_t *)(strdata(a)+i);
24     uint32_t vb = *(const uint32_t *)(strdata(b)+i);
25     if (va != vb) {
26 #if LJ_LE
27       va = lj_bswap(va); vb = lj_bswap(vb);
28 #endif
29       i -= n;
30       if ((int32_t)i >= -3) {
31         va >>= 32+(i<<3); vb >>= 32+(i<<3);
32         if (va == vb) break;
33       }
34       return va < vb ? -1 : 1;
35     }
36   }
37   return (int32_t)(a->len - b->len);
38 }
39
40 /* Fast string data comparison. Caveat: unaligned access to 1st string! */
41 static LJ_AINLINE int str_fastcmp(const char *a, const char *b, MSize len)
```

```c
{
  MSize i = 0;
  lua_assert(len > 0);
  lua_assert((((uintptr_t)a+len-1) & (LJ_PAGESIZE-1)) <= LJ_PAGESIZE-4);
  do {  /* Note: innocuous access up to end of string + 3. */
    uint32_t v = lj_getu32(a+i) ^ *(const uint32_t *)(b+i);
    if (v) {
      i -= len;
#if LJ_LE
      return (int32_t)i >= -3 ? (v << (32+(i<<3))) : 1;
#else
      return (int32_t)i >= -3 ? (v >> (32+(i<<3))) : 1;
#endif
    }
    i += 4;
  } while (i < len);
  return 0;
}

/* Find fixed string p inside string s. */
const char *lj_str_find(const char *s, const char *p, MSize slen, MSize plen)
{
  if (plen <= slen) {
    if (plen == 0) {
      return s;
    } else {
      int c = *(const uint8_t *)p++;
      plen--; slen -= plen;
      while (slen) {
        const char *q = (const char *)memchr(s, c, slen);
        if (!q) break;
        if (memcmp(q+1, p, plen) == 0) return q;
        q++; slen -= (MSize)(q-s); s = q;
      }
    }
  }
  return NULL;
}

/* Check whether a string has a pattern matching character. */
int lj_str_haspattern(GCstr *s)
{
  const char *p = strdata(s), *q = p + s->len;
  while (p < q) {
    int c = *(const uint8_t *)p++;
    if (lj_char_ispunct(c) && strchr("^$*+?.([%-", c))
      return 1;  /* Found a pattern matching char. */
  }
  return 0;  /* No pattern matching chars found. */
}

/* -- String interning ----------------------------------------------- */

/* Resize the string hash table (grow and shrink). */
void lj_str_resize(lua_State *L, MSize newmask)
{
  global_State *g = G(L);
  GCRef *newhash;
   MSize i;
   if (g->gc.state == GCSsweepstring || newmask >= LJ_MAX_STRTAB-1)
     return;  /* No resizing during GC traversal or if already too big. */
   newhash = lj_mem_newvec(L, newmask+1, GCRef);
   memset(newhash, 0, (newmask+1)*sizeof(GCRef));
   for (i = g->strmask; i != ~(MSize)0; i--) {  /* Rehash old table. */
     GCobj *p = gcref(g->strhash[i]);
     while (p) {  /* Follow each hash chain and reinsert all strings. */
       MSize h = gco2str(p)->hash & newmask;
       GCobj *next = gcnext(p);
       /* NOBARRIER: The string table is a GC root. */
       setgcrefr(p->gch.nextgc, newhash[h]);
       setgcref(newhash[h], p);
       p = next;
     }
   }
   lj_mem_freevec(g, g->strhash, g->strmask+1, GCRef);
   g->strmask = newmask;
```

```
  g->strhash = newhash;
}

/* Intern a string and return string object. */
GCstr *lj_str_new(lua_State *L, const char *str, size_t lenx)
{
  global_State *g;
  GCstr *s;
  GCobj *o;
  MSize len = (MSize)lenx;
  MSize a, b, h = len;
  if (lenx >= LJ_MAX_STR)
    lj_err_msg(L, LJ_ERR_STROV);
  g = G(L);
  /* Compute string hash. Constants taken from lookup3 hash by Bob Jenkins. */
  if (len >= 4) {  /* Caveat: unaligned access! */
    a = lj_getu32(str);
    h ^= lj_getu32(str+len-4);
    b = lj_getu32(str+(len>>1)-2);
    h ^= b; h -= lj_rol(b, 14);
    b += lj_getu32(str+(len>>2)-1);
  } else if (len > 0) {
    a = *(const uint8_t *)str;
    h ^= *(const uint8_t *)(str+len-1);
    b = *(const uint8_t *)(str+(len>>1));
    h ^= b; h -= lj_rol(b, 14);
  } else {
    return &g->strempty;
  }
  a ^= h; a -= lj_rol(h, 11);
  b ^= a; b -= lj_rol(a, 25);
  h ^= b; h -= lj_rol(b, 16);
  /* Check if the string has already been interned. */
  o = gcref(g->strhash[h & g->strmask]);
  if (LJ_LIKELY((((uintptr_t)str+len-1) & (LJ_PAGESIZE-1)) <= LJ_PAGESIZE-4)) {
    while (o != NULL) {
      GCstr *sx = gco2str(o);
      if (sx->len == len && str_fastcmp(str, strdata(sx), len) == 0) {
        /* Resurrect if dead. Can only happen with fixstring() (keywords). */
        if (isdead(g, o)) flipwhite(o);
        return sx;  /* Return existing string. */
      }
      o = gcnext(o);
    }
  } else {  /* Slow path: end of string is too close to a page boundary. */
    while (o != NULL) {
      GCstr *sx = gco2str(o);
      if (sx->len == len && memcmp(str, strdata(sx), len) == 0) {
        /* Resurrect if dead. Can only happen with fixstring() (keywords). */
        if (isdead(g, o)) flipwhite(o);
        return sx;  /* Return existing string. */
      }
      o = gcnext(o);
    }
  }
  /* Nope, create a new string. */
  s = lj_mem_newt(L, sizeof(GCstr)+len+1, GCstr);
  newwhite(g, s);
  s->gct = ~LJ_TSTR;
  s->len = len;
  s->hash = h;
  s->reserved = 0;
  memcpy(strdatawr(s), str, len);
  strdatawr(s)[len] = '\0';  /* Zero-terminate string. */
  /* Add it to string hash table. */
  h &= g->strmask;
  s->nextgc = g->strhash[h];
  /* NOBARRIER: The string table is a GC root. */
  setgcref(g->strhash[h], obj2gco(s));
  if (g->strnum++ > g->strmask)  /* Allow a 100% load factor. */
    lj_str_resize(L, (g->strmask<<1)+1);  /* Grow string table. */
  return s;  /* Return newly interned string. */
}

void LJ_FASTCALL lj_str_free(global_State *g, GCstr *s)
{
```

```
194   g->strnum--;
195   lj_mem_free(g, s, sizestring(s));
196 }
197
```

One Level Up Top Level

src/lj_char.h - luajit-2.0-src

Global variables defined

- lj_char_bits

Macros defined

- LJ_CHAR_ALNUM
- LJ_CHAR_ALPHA
- LJ_CHAR_CNTRL
- LJ_CHAR_DIGIT
- LJ_CHAR_GRAPH
- LJ_CHAR_IDENT
- LJ_CHAR_LOWER
- LJ_CHAR_PUNCT
- LJ_CHAR_SPACE
- LJ_CHAR_UPPER
- LJ_CHAR_XDIGIT
- _LJ_CHAR_H
- lj_char_isa
- lj_char_isalnum
- lj_char_isalpha
- lj_char_iscntrl
- lj_char_isdigit
- lj_char_isgraph
- lj_char_isident
- lj_char_islower
- lj_char_ispunct
- lj_char_isspace
- lj_char_isupper
- lj_char_isxdigit
- lj_char_tolower
- lj_char_toupper

Source code

```c
/*
** Character types.
** Donated to the public domain.
*/

#ifndef _LJ_CHAR_H
#define _LJ_CHAR_H

#include "lj_def.h"

#define LJ_CHAR_CNTRL        0x01
#define LJ_CHAR_SPACE        0x02
#define LJ_CHAR_PUNCT        0x04
#define LJ_CHAR_DIGIT        0x08
#define LJ_CHAR_XDIGIT        0x10
#define LJ_CHAR_UPPER        0x20
#define LJ_CHAR_LOWER        0x40
#define LJ_CHAR_IDENT        0x80
#define LJ_CHAR_ALPHA        (LJ_CHAR_LOWER|LJ_CHAR_UPPER)
#define LJ_CHAR_ALNUM        (LJ_CHAR_ALPHA|LJ_CHAR_DIGIT)
#define LJ_CHAR_GRAPH        (LJ_CHAR_ALNUM|LJ_CHAR_PUNCT)

/* Only pass -1 or 0..255 to these macros. Never pass a signed char! */
#define lj_char_isa(c, t)        ((lj_char_bits+1)[(c)] & t)
#define lj_char_iscntrl(c)        lj_char_isa((c), LJ_CHAR_CNTRL)
#define lj_char_isspace(c)        lj_char_isa((c), LJ_CHAR_SPACE)
#define lj_char_ispunct(c)        lj_char_isa((c), LJ_CHAR_PUNCT)
#define lj_char_isdigit(c)        lj_char_isa((c), LJ_CHAR_DIGIT)
#define lj_char_isxdigit(c)        lj_char_isa((c), LJ_CHAR_XDIGIT)
#define lj_char_isupper(c)        lj_char_isa((c), LJ_CHAR_UPPER)
#define lj_char_islower(c)        lj_char_isa((c), LJ_CHAR_LOWER)
#define lj_char_isident(c)        lj_char_isa((c), LJ_CHAR_IDENT)
#define lj_char_isalpha(c)        lj_char_isa((c), LJ_CHAR_ALPHA)
#define lj_char_isalnum(c)        lj_char_isa((c), LJ_CHAR_ALNUM)
#define lj_char_isgraph(c)        lj_char_isa((c), LJ_CHAR_GRAPH)

#define lj_char_toupper(c)        ((c) - (lj_char_islower(c) >> 1))
#define lj_char_tolower(c)        ((c) + lj_char_isupper(c))

LJ_DATA const uint8_t lj_char_bits[257];

#endif
```

One Level Up Top Level

src/lj_frame.h - luajit-2.0-src

Macros defined

- CFRAME_OFS_ERRF
- CFRAME_OFS_ERRF
- CFRAME_OFS_ERRF
- CFRAME_OFS_ERRF
- CFRAME_OFS_ERRF
- CFRAME_OFS_ERRF
- CFRAME_OFS_ERRF
- CFRAME_OFS_ERRF
- CFRAME_OFS_ERRF
- CFRAME_OFS_L
- CFRAME_OFS_L
- CFRAME_OFS_L
- CFRAME_OFS_L
- CFRAME_OFS_L
- CFRAME_OFS_L
- CFRAME_OFS_L
- CFRAME_OFS_L
- CFRAME_OFS_L
- CFRAME_OFS_MULTRES
- CFRAME_OFS_MULTRES
- CFRAME_OFS_MULTRES
- CFRAME_OFS_MULTRES
- CFRAME_OFS_MULTRES
- CFRAME_OFS_MULTRES
- CFRAME_OFS_MULTRES
- CFRAME_OFS_MULTRES
- CFRAME_OFS_MULTRES
- CFRAME_OFS_NRES
- CFRAME_OFS_NRES
- CFRAME_OFS_NRES

- CFRAME_OFS_NRES
- CFRAME_OFS_NRES
- CFRAME_OFS_NRES
- CFRAME_OFS_NRES
- CFRAME_OFS_NRES
- CFRAME_OFS_NRES
- CFRAME_OFS_PC
- CFRAME_OFS_PC
- CFRAME_OFS_PC
- CFRAME_OFS_PC
- CFRAME_OFS_PC
- CFRAME_OFS_PC
- CFRAME_OFS_PC
- CFRAME_OFS_PC
- CFRAME_OFS_PC
- CFRAME_OFS_PREV
- CFRAME_OFS_PREV
- CFRAME_OFS_PREV
- CFRAME_OFS_PREV
- CFRAME_OFS_PREV
- CFRAME_OFS_PREV
- CFRAME_OFS_PREV
- CFRAME_OFS_PREV
- CFRAME_OFS_PREV
- CFRAME_RAWMASK
- CFRAME_RESUME
- CFRAME_SHIFT_MULTRES
- CFRAME_SHIFT_MULTRES
- CFRAME_SHIFT_MULTRES
- CFRAME_SHIFT_MULTRES
- CFRAME_SHIFT_MULTRES
- CFRAME_SHIFT_MULTRES
- CFRAME_SHIFT_MULTRES

- CFRAME_SHIFT_MULTRES
- CFRAME_SHIFT_MULTRES
- CFRAME_SIZE
- CFRAME_SIZE
- CFRAME_SIZE
- CFRAME_SIZE
- CFRAME_SIZE
- CFRAME_SIZE
- CFRAME_SIZE
- CFRAME_SIZE
- CFRAME_SIZE
- CFRAME_SIZE
- CFRAME_SIZE_JIT
- CFRAME_SIZE_JIT
- CFRAME_SIZE_JIT
- CFRAME_UNWIND_FF
- FRAME_P
- FRAME_TYPE
- FRAME_TYPEP
- _LJ_FRAME_H
- cframe_L
- cframe_Lpc
- cframe_canyield
- cframe_errfunc
- cframe_multres
- cframe_multres_n
- cframe_nres
- cframe_pc
- cframe_prev
- cframe_raw
- cframe_unwind_ff
- frame_contf
- frame_contf

- frame_contf
- frame_contpc
- frame_contpc
- frame_contv
- frame_contv
- frame_delta
- frame_ftsz
- frame_ftsz
- frame_func
- frame_gc
- frame_gc
- frame_isc
- frame_iscont
- frame_iscont_fficb
- frame_islua
- frame_ispcall
- frame_isvarg
- frame_pc
- frame_pc
- frame_prev
- frame_prevd
- frame_prevl
- frame_sized
- frame_type
- frame_typep
- setcframe_L
- setcframe_pc
- setframe_ftsz
- setframe_ftsz
- setframe_gc
- setframe_gc
- setframe_pc
- setframe_pc

Source code

```
/*
** Stack frames.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_FRAME_H
#define _LJ_FRAME_H

#include "lj_obj.h"
#include "lj_bc.h"

/* -- Lua stack frame ----------------------------------------------------- */

/* Frame type markers in LSB of PC (4-byte aligned) or delta (8-byte aligned:
**
**    PC  00  Lua frame
** delta 001  C frame
** delta 010  Continuation frame
** delta 011  Lua vararg frame
** delta 101  cpcall() frame
** delta 110  ff pcall() frame
** delta 111  ff pcall() frame with active hook
*/
enum {
  FRAME_LUA, FRAME_C, FRAME_CONT, FRAME_VARG,
  FRAME_LUAP, FRAME_CP, FRAME_PCALL, FRAME_PCALLH
};
#define FRAME_TYPE               3
#define FRAME_P                     4
#define FRAME_TYPEP               (FRAME_TYPE|FRAME_P)

/* Macros to access and modify Lua frames. */
#if LJ_FR2
/* Two-slot frame info, required for 64 bit PC/GCRef:
**
**                  base-2  base-1      |  base  base+1 ...
**                 [func   PC/delta/ft] | [slots ...]
**                 ^-- frame            | ^-- base   ^-- top
**
** Continuation frames:
**
**   base-4  base-3  base-2  base-1      |  base  base+1 ...
**  [cont      PC ] [func   PC/delta/ft] | [slots ...]
**                  ^-- frame           | ^-- base   ^-- top
*/
#define frame_gc(f)               (gcval((f)-1))
#define frame_ftsz(f)               ((ptrdiff_t)(f)->ftsz)
#define frame_pc(f)               ((const BCIns *)frame_ftsz(f))
#define setframe_gc(f, p, tp)       (setgcVraw((f)-1, (p), (tp)))
#define setframe_ftsz(f, sz)       ((f)->ftsz = (sz))
#define setframe_pc(f, pc)       ((f)->ftsz = (int64_t)(intptr_t)(pc))
#else
/* One-slot frame info, sufficient for 32 bit PC/GCRef:
**
**             base-1              |  base  base+1 ...
**             lo     hi           |
**            [func | PC/delta/ft] | [slots ...]
**            ^-- frame            | ^-- base   ^-- top
**
** Continuation frames:
**
**  base-2      base-1              |  base  base+1 ...
**  lo     hi   lo     hi           |
** [cont | PC] [func | PC/delta/ft] | [slots ...]
**             ^-- frame            | ^-- base   ^-- top
*/
#define frame_gc(f)               (gcref((f)->fr.func))
#define frame_ftsz(f)               ((ptrdiff_t)(f)->fr.tp.ftsz)
#define frame_pc(f)               (mref((f)->fr.tp.pcr, const BCIns))
#define setframe_gc(f, p, tp)       (setgcref((f)->fr.func, (p)), UNUSED(tp))
#define setframe_ftsz(f, sz)       ((f)->fr.tp.ftsz = (int32_t)(sz))
#define setframe_pc(f, pc)       (setmref((f)->fr.tp.pcr, (pc)))
#endif
```

```
74
75  #define frame_type(f)                (frame_ftsz(f) & FRAME_TYPE)
76  #define frame_typep(f)                (frame_ftsz(f) & FRAME_TYPEP)
77  #define frame_islua(f)                (frame_type(f) == FRAME_LUA)
78  #define frame_isc(f)                (frame_type(f) == FRAME_C)
79  #define frame_iscont(f)                (frame_typep(f) == FRAME_CONT)
80  #define frame_isvarg(f)                (frame_typep(f) == FRAME_VARG)
81  #define frame_ispcall(f)        ((frame_ftsz(f) & 6) == FRAME_PCALL)
82
83  #define frame_func(f)                (&frame_gc(f)->fn)
84  #define frame_delta(f)                (frame_ftsz(f) >> 3)
85  #define frame_sized(f)                (frame_ftsz(f) & ~FRAME_TYPEP)
86
87  enum { LJ_CONT_TAILCALL, LJ_CONT_FFI_CALLBACK };  /* Special continuations. */
88
89  #if LJ_FR2
90  #define frame_contpc(f)                (frame_pc((f)-2))
91  #define frame_contv(f)                (((f)-3)->u64)
92  #else
93  #define frame_contpc(f)                (frame_pc((f)-1))
94  #define frame_contv(f)                (((f)-1)->u32.lo)
95  #endif
96  #if LJ_FR2
97  #define frame_contf(f)                ((ASMFunction)(uintptr_t)((f)-3)->u64)
98  #elif LJ_64
99  #define frame_contf(f) \
100   ((ASMFunction)(void *)((intptr_t)lj_vm_asm_begin + \
101                          (intptr_t)(int32_t)((f)-1)->u32.lo))
102 #else
103 #define frame_contf(f)                ((ASMFunction)gcrefp(((f)-1)->gcr, void))
104 #endif
105 #define frame_iscont_ficb(f) \
106   (LJ_HASFFI && frame_contv(f) == LJ_CONT_FFI_CALLBACK)
107
108 #define frame_prevl(f)                ((f) - (1+LJ_FR2+bc_a(frame_pc(f)[-1])))
109 #define frame_prevd(f)                ((TValue *)((char *)(f) - frame_sized(f)))
110 #define frame_prev(f)                (frame_islua(f)?frame_prevl(f):frame_prevd(f))
111 /* Note: this macro does not skip over FRAME_VARG. */
112
113 /* -- C stack frame ------------------------------------------------------- */
114
115 /* Macros to access and modify the C stack frame chain. */
116
117 /* These definitions must match with the arch-specific *.dasc files. */
118 #if LJ_TARGET_X86
119 #define CFRAME_OFS_ERRF                (15*4)
120 #define CFRAME_OFS_NRES                (14*4)
121 #define CFRAME_OFS_PREV                (13*4)
122 #define CFRAME_OFS_L                (12*4)
123 #define CFRAME_OFS_PC                (6*4)
124 #define CFRAME_OFS_MULTRES        (5*4)
125 #define CFRAME_SIZE                (12*4)
126 #define CFRAME_SHIFT_MULTRES        0
127 #elif LJ_TARGET_X64
128 #if LJ_ABI_WIN
129 #define CFRAME_OFS_PREV                (13*8)
130 #define CFRAME_OFS_PC                (25*4)
131 #define CFRAME_OFS_L                (24*4)
132 #define CFRAME_OFS_ERRF                (23*4)
133 #define CFRAME_OFS_NRES                (22*4)
134 #define CFRAME_OFS_MULTRES        (21*4)
135 #define CFRAME_SIZE                (10*8)
136 #define CFRAME_SIZE_JIT                (CFRAME_SIZE + 9*16 + 4*8)
137 #define CFRAME_SHIFT_MULTRES        0
138 #else
139 #define CFRAME_OFS_PREV                (4*8)
140 #define CFRAME_OFS_PC                (7*4)
141 #define CFRAME_OFS_L                (6*4)
142 #define CFRAME_OFS_ERRF                (5*4)
143 #define CFRAME_OFS_NRES                (4*4)
144 #define CFRAME_OFS_MULTRES        (1*4)
145 #define CFRAME_SIZE                (10*8)
146 #define CFRAME_SIZE_JIT                (CFRAME_SIZE + 16)
147 #define CFRAME_SHIFT_MULTRES        0
148 #endif
149 #elif LJ_TARGET_ARM
```

```
#define CFRAME_OFS_ERRF               24
#define CFRAME_OFS_NRES               20
#define CFRAME_OFS_PREV               16
#define CFRAME_OFS_L                  12
#define CFRAME_OFS_PC                  8
#define CFRAME_OFS_MULTRES             4
#if LJ_ARCH_HASFPU
#define CFRAME_SIZE                  128
#else
#define CFRAME_SIZE                   64
#endif
#define CFRAME_SHIFT_MULTRES           3
#elif LJ_TARGET_ARM64
#define CFRAME_OFS_ERRF              196
#define CFRAME_OFS_NRES              200
#define CFRAME_OFS_PREV              160
#define CFRAME_OFS_L                 176
#define CFRAME_OFS_PC                168
#define CFRAME_OFS_MULTRES           192
#define CFRAME_SIZE                  208
#define CFRAME_SHIFT_MULTRES           3
#elif LJ_TARGET_PPC
#if LJ_TARGET_XBOX360
#define CFRAME_OFS_ERRF              424
#define CFRAME_OFS_NRES              420
#define CFRAME_OFS_PREV              400
#define CFRAME_OFS_L                 416
#define CFRAME_OFS_PC                412
#define CFRAME_OFS_MULTRES           408
#define CFRAME_SIZE                  384
#define CFRAME_SHIFT_MULTRES           3
#elif LJ_ARCH_PPC32ON64
#define CFRAME_OFS_ERRF              472
#define CFRAME_OFS_NRES              468
#define CFRAME_OFS_PREV              448
#define CFRAME_OFS_L                 464
#define CFRAME_OFS_PC                460
#define CFRAME_OFS_MULTRES           456
#define CFRAME_SIZE                  400
#define CFRAME_SHIFT_MULTRES           3
#else
#define CFRAME_OFS_ERRF               48
#define CFRAME_OFS_NRES               44
#define CFRAME_OFS_PREV               40
#define CFRAME_OFS_L                  36
#define CFRAME_OFS_PC                 32
#define CFRAME_OFS_MULTRES            28
#define CFRAME_SIZE                  272
#define CFRAME_SHIFT_MULTRES           3
#endif
#elif LJ_TARGET_MIPS
#define CFRAME_OFS_ERRF              124
#define CFRAME_OFS_NRES              120
#define CFRAME_OFS_PREV              116
#define CFRAME_OFS_L                 112
#define CFRAME_OFS_PC                 20
#define CFRAME_OFS_MULTRES            16
#define CFRAME_SIZE                  112
#define CFRAME_SHIFT_MULTRES           3
#else
#error "Missing CFRAME_* definitions for this architecture"
#endif

#ifndef CFRAME_SIZE_JIT
#define CFRAME_SIZE_JIT              CFRAME_SIZE
#endif

#define CFRAME_RESUME                  1
#define CFRAME_UNWIND_FF             2  /* Only used in unwinder. */
#define CFRAME_RAWMASK                 (~(intptr_t)(CFRAME_RESUME|CFRAME_UNWIND_FF))

#define cframe_errfunc(cf)           (*(int32_t *)(((char *)(cf))+CFRAME_OFS_ERRF))
#define cframe_nres(cf)                (*(int32_t *)(((char *)(cf))+CFRAME_OFS_NRES))
#define cframe_prev(cf)                (*(void **)(((char *)(cf))+CFRAME_OFS_PREV))
#define cframe_multres(cf)  (*(uint32_t *)(((char *)(cf))+CFRAME_OFS_MULTRES))
#define cframe_multres_n(cf)         (cframe_multres((cf)) >> CFRAME_SHIFT_MULTRES)
```

```
226 #define cframe_L(cf) \
227   (&gcref(*(GCRef *)(((char *)(cf))+CFRAME_OFS_L))->th)
228 #define cframe_pc(cf) \
229   (mref(*(MRef *)(((char *)(cf))+CFRAME_OFS_PC), const BCIns))
230 #define setcframe_L(cf, L) \
231   (setmref(*(MRef *)(((char *)(cf))+CFRAME_OFS_L), (L)))
232 #define setcframe_pc(cf, pc) \
233   (setmref(*(MRef *)(((char *)(cf))+CFRAME_OFS_PC), (pc)))
234 #define cframe_canyield(cf)        ((intptr_t)(cf) & CFRAME_RESUME)
235 #define cframe_unwind_ff(cf)        ((intptr_t)(cf) & CFRAME_UNWIND_FF)
236 #define cframe_raw(cf)               ((void *)((intptr_t)(cf) & CFRAME_RAWMASK))
237 #define cframe_Lpc(L)               cframe_pc(cframe_raw(L->cframe))
238
239 #endif
```

One Level Up Top Level

src/lj_api.c - luajit-2.0-src

Functions defined

- lua_gettop
- lua_getupvalue
- lua_insert
- lua_iscfunction
- lua_isnumber
- lua_isstring
- lua_isuserdata
- lua_lessthan
- lua_newthread
- lua_newuserdata
- lua_next
- lua_objlen
- lua_pcall
- lua_pushboolean
- lua_pushcclosure
- lua_pushfstring
- lua_pushinteger
- lua_pushlightuserdata
- lua_pushlstring
- lua_pushnil
- lua_pushnumber
- lua_pushstring
- lua_pushthread
- lua_pushvalue
- lua_pushvfstring
- lua_rawequal
- lua_rawget
- lua_rawgeti
- lua_rawset
- lua_rawseti
- lua_remove
- lua_replace
- lua_resume

- lua_setallocf
- lua_setfenv
- lua_setfield
- lua_setmetatable
- lua_settable
- lua_settop
- lua_setupvalue
- lua_status
- lua_toboolean
- lua_tocfunction
- lua_tointeger
- lua_tolstring
- lua_tonumber
- lua_topointer
- lua_tothread
- lua_touserdata
- lua_type
- lua_typename
- lua_upvalueid
- lua_upvaluejoin
- lua_xmove
- lua_yield
- stkindex2adr

Macros defined

- LUA_CORE
- api_call_base
- api_checknelems
- api_checkvalidindex
- lj_api_c

Source code

```
/*
** Public Lua/C API.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Major portions taken verbatim or adapted from the Lua interpreter.
```

```
** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
*/

#define lj_api_c
#define LUA_CORE

#include "lj_obj.h"
#include "lj_gc.h"
#include "lj_err.h"
#include "lj_debug.h"
#include "lj_str.h"
#include "lj_tab.h"
#include "lj_func.h"
#include "lj_udata.h"
#include "lj_meta.h"
#include "lj_state.h"
#include "lj_bc.h"
#include "lj_frame.h"
#include "lj_trace.h"
#include "lj_vm.h"
#include "lj_strscan.h"
#include "lj_strfmt.h"

/* -- Common helper functions ---------------------------------------------- */

#define api_checknelems(L, n)               api_check(L, (n) <= (L->top - L->base))
#define api_checkvalidindex(L, i)        api_check(L, (i) != niltv(L))

static TValue *index2adr(lua_State *L, int idx)
{
  if (idx > 0) {
    TValue *o = L->base + (idx - 1);
    return o < L->top ? o : niltv(L);
  } else if (idx > LUA_REGISTRYINDEX) {
    api_check(L, idx != 0 && -idx <= L->top - L->base);
    return L->top + idx;
  } else if (idx == LUA_GLOBALSINDEX) {
    TValue *o = &G(L)->tmptv;
    settabV(L, o, tabref(L->env));
    return o;
  } else if (idx == LUA_REGISTRYINDEX) {
    return registry(L);
  } else {
    GCfunc *fn = curr_func(L);
    api_check(L, fn->c.gct == ~LJ_TFUNC && !isluafunc(fn));
    if (idx == LUA_ENVIRONINDEX) {
      TValue *o = &G(L)->tmptv;
      settabV(L, o, tabref(fn->c.env));
      return o;
    } else {
      idx = LUA_GLOBALSINDEX - idx;
      return idx <= fn->c.nupvalues ? &fn->c.upvalue[idx-1] : niltv(L);
    }
  }
}

static TValue *stkindex2adr(lua_State *L, int idx)
{
  if (idx > 0) {
    TValue *o = L->base + (idx - 1);
    return o < L->top ? o : niltv(L);
  } else {
    api_check(L, idx != 0 && -idx <= L->top - L->base);
    return L->top + idx;
  }
}

static GCtab *getcurrenv(lua_State *L)
{
  GCfunc *fn = curr_func(L);
  return fn->c.gct == ~LJ_TFUNC ? tabref(fn->c.env) : tabref(L->env);
}

/* -- Miscellaneous API functions ------------------------------------------ */

LUA_API int lua_status(lua_State *L)
```

```
{
  return L->status;
}

LUA_API int lua_checkstack(lua_State *L, int size)
{
  if (size > LUAI_MAXCSTACK || (L->top - L->base + size) > LUAI_MAXCSTACK) {
    return 0;  /* Stack overflow. */
  } else if (size > 0) {
    lj_state_checkstack(L, (MSize)size);
  }
  return 1;
}

LUALIB_API void luaL_checkstack(lua_State *L, int size, const char *msg)
{
  if (!lua_checkstack(L, size))
    lj_err_callerv(L, LJ_ERR_STKOVM, msg);
}

LUA_API void lua_xmove(lua_State *from, lua_State *to, int n)
{
  TValue *f, *t;
  if (from == to) return;
  api_checknelems(from, n);
  api_check(from, G(from) == G(to));
  lj_state_checkstack(to, (MSize)n);
  f = from->top;
  t = to->top = to->top + n;
  while (--n >= 0) copyTV(to, --t, --f);
  from->top = f;
}

/* -- Stack manipulation ------------------------------------------------- */

LUA_API int lua_gettop(lua_State *L)
{
  return (int)(L->top - L->base);
}

LUA_API void lua_settop(lua_State *L, int idx)
{
  if (idx >= 0) {
    api_check(L, idx <= tvref(L->maxstack) - L->base);
    if (L->base + idx > L->top) {
      if (L->base + idx >= tvref(L->maxstack))
        lj_state_growstack(L, (MSize)idx - (MSize)(L->top - L->base));
      do { setnilV(L->top++); } while (L->top < L->base + idx);
    } else {
      L->top = L->base + idx;
    }
  } else {
    api_check(L, -(idx+1) <= (L->top - L->base));
    L->top += idx+1;  /* Shrinks top (idx < 0). */
  }
}

LUA_API void lua_remove(lua_State *L, int idx)
{
  TValue *p = stkindex2adr(L, idx);
  api_checkvalidindex(L, p);
  while (++p < L->top) copyTV(L, p-1, p);
  L->top--;
}

LUA_API void lua_insert(lua_State *L, int idx)
{
  TValue *q, *p = stkindex2adr(L, idx);
  api_checkvalidindex(L, p);
  for (q = L->top; q > p; q--) copyTV(L, q, q-1);
  copyTV(L, p, L->top);
}

LUA_API void lua_replace(lua_State *L, int idx)
{
  api_checknelems(L, 1);
```

```
  if (idx == LUA_GLOBALSINDEX) {
    api_check(L, tvistab(L->top-1));
    /* NOBARRIER: A thread (i.e. L) is never black. */
    setgcref(L->env, obj2gco(tabV(L->top-1)));
  } else if (idx == LUA_ENVIRONINDEX) {
    GCfunc *fn = curr_func(L);
    if (fn->c.gct != ~LJ_TFUNC)
      lj_err_msg(L, LJ_ERR_NOENV);
    api_check(L, tvistab(L->top-1));
    setgcref(fn->c.env, obj2gco(tabV(L->top-1)));
    lj_gc_barrier(L, fn, L->top-1);
  } else {
    TValue *o = index2adr(L, idx);
    api_checkvalidindex(L, o);
    copyTV(L, o, L->top-1);
    if (idx < LUA_GLOBALSINDEX)  /* Need a barrier for upvalues. */
      lj_gc_barrier(L, curr_func(L), L->top-1);
  }
  L->top--;
}

LUA_API void lua_pushvalue(lua_State *L, int idx)
{
  copyTV(L, L->top, index2adr(L, idx));
  incr_top(L);
}

/* -- Stack getters ------------------------------------------------------- */

LUA_API int lua_type(lua_State *L, int idx)
{
  cTValue *o = index2adr(L, idx);
  if (tvisnumber(o)) {
    return LUA_TNUMBER;
#if LJ_64 && !LJ_GC64
  } else if (tvislightud(o)) {
    return LUA_TLIGHTUSERDATA;
#endif
  } else if (o == niltv(L)) {
    return LUA_TNONE;
  } else {  /* Magic internal/external tag conversion. ORDER LJ_T */
    uint32_t t = ~itype(o);
#if LJ_64
    int tt = (int)((U64x(75a06,98042110) >> 4*t) & 15u);
#else
    int tt = (int)(((t < 8 ? 0x98042110u : 0x75a06u) >> 4*(t&7)) & 15u);
#endif
    lua_assert(tt != LUA_TNIL || tvisnil(o));
    return tt;
  }
}

LUALIB_API void luaL_checktype(lua_State *L, int idx, int tt)
{
  if (lua_type(L, idx) != tt)
    lj_err_argt(L, idx, tt);
}

LUALIB_API void luaL_checkany(lua_State *L, int idx)
{
  if (index2adr(L, idx) == niltv(L))
    lj_err_arg(L, idx, LJ_ERR_NOVAL);
}

LUA_API const char *lua_typename(lua_State *L, int t)
{
  UNUSED(L);
  return lj_obj_typename[t+1];
}

LUA_API int lua_iscfunction(lua_State *L, int idx)
{
  cTValue *o = index2adr(L, idx);
  return tvisfunc(o) && !isluafunc(funcV(o));
}
```

```
LUA_API int lua_isnumber(lua_State *L, int idx)
{
  cTValue *o = index2adr(L, idx);
  TValue tmp;
  return (tvisnumber(o) || (tvisstr(o) && lj_strscan_number(strV(o), &tmp)));
}

LUA_API int lua_isstring(lua_State *L, int idx)
{
  cTValue *o = index2adr(L, idx);
  return (tvisstr(o) || tvisnumber(o));
}

LUA_API int lua_isuserdata(lua_State *L, int idx)
{
  cTValue *o = index2adr(L, idx);
  return (tvisudata(o) || tvislightud(o));
}

LUA_API int lua_rawequal(lua_State *L, int idx1, int idx2)
{
  cTValue *o1 = index2adr(L, idx1);
  cTValue *o2 = index2adr(L, idx2);
  return (o1 == niltv(L) || o2 == niltv(L)) ? 0 : lj_obj_equal(o1, o2);
}

LUA_API int lua_equal(lua_State *L, int idx1, int idx2)
{
  cTValue *o1 = index2adr(L, idx1);
  cTValue *o2 = index2adr(L, idx2);
  if (tvisint(o1) && tvisint(o2)) {
    return intV(o1) == intV(o2);
  } else if (tvisnumber(o1) && tvisnumber(o2)) {
    return numberVnum(o1) == numberVnum(o2);
  } else if (itype(o1) != itype(o2)) {
    return 0;
  } else if (tvispri(o1)) {
    return o1 != niltv(L) && o2 != niltv(L);
#if LJ_64 && !LJ_GC64
  } else if (tvislightud(o1)) {
    return o1->u64 == o2->u64;
#endif
  } else if (gcrefeq(o1->gcr, o2->gcr)) {
    return 1;
  } else if (!tvistabud(o1)) {
    return 0;
  } else {
    TValue *base = lj_meta_equal(L, gcV(o1), gcV(o2), 0);
    if ((uintptr_t)base <= 1) {
      return (int)(uintptr_t)base;
    } else {
      L->top = base+2;
      lj_vm_call(L, base, 1+1);
      L->top -= 2+LJ_FR2;
      return tvistruecond(L->top+1+LJ_FR2);
    }
  }
}

LUA_API int lua_lessthan(lua_State *L, int idx1, int idx2)
{
  cTValue *o1 = index2adr(L, idx1);
  cTValue *o2 = index2adr(L, idx2);
  if (o1 == niltv(L) || o2 == niltv(L)) {
    return 0;
  } else if (tvisint(o1) && tvisint(o2)) {
    return intV(o1) < intV(o2);
  } else if (tvisnumber(o1) && tvisnumber(o2)) {
    return numberVnum(o1) < numberVnum(o2);
  } else {
    TValue *base = lj_meta_comp(L, o1, o2, 0);
    if ((uintptr_t)base <= 1) {
      return (int)(uintptr_t)base;
    } else {
      L->top = base+2;
      lj_vm_call(L, base, 1+1);
```

```
      L->top -= 2+LJ_FR2;
      return tvistruecond(L->top+1+LJ_FR2);
    }
  }
}

LUA_API lua_Number lua_tonumber(lua_State *L, int idx)
{
  cTValue *o = index2adr(L, idx);
  TValue tmp;
  if (LJ_LIKELY(tvisnumber(o)))
    return numberVnum(o);
  else if (tvisstr(o) && lj_strscan_num(strV(o), &tmp))
    return numV(&tmp);
  else
    return 0;
}

LUALIB_API lua_Number luaL_checknumber(lua_State *L, int idx)
{
  cTValue *o = index2adr(L, idx);
  TValue tmp;
  if (LJ_LIKELY(tvisnumber(o)))
    return numberVnum(o);
  else if (!(tvisstr(o) && lj_strscan_num(strV(o), &tmp)))
    lj_err_argt(L, idx, LUA_TNUMBER);
  return numV(&tmp);
}

LUALIB_API lua_Number luaL_optnumber(lua_State *L, int idx, lua_Number def)
{
  cTValue *o = index2adr(L, idx);
  TValue tmp;
  if (LJ_LIKELY(tvisnumber(o)))
    return numberVnum(o);
  else if (tvisnil(o))
    return def;
  else if (!(tvisstr(o) && lj_strscan_num(strV(o), &tmp)))
    lj_err_argt(L, idx, LUA_TNUMBER);
  return numV(&tmp);
}

LUA_API lua_Integer lua_tointeger(lua_State *L, int idx)
{
  cTValue *o = index2adr(L, idx);
  TValue tmp;
  lua_Number n;
  if (LJ_LIKELY(tvisint(o))) {
    return intV(o);
  } else if (LJ_LIKELY(tvisnum(o))) {
    n = numV(o);
  } else {
    if (!(tvisstr(o) && lj_strscan_number(strV(o), &tmp)))
      return 0;
    if (tvisint(&tmp))
      return (lua_Integer)intV(&tmp);
    n = numV(&tmp);
  }
#if LJ_64
  return (lua_Integer)n;
#else
  return lj_num2int(n);
#endif
}

LUALIB_API lua_Integer luaL_checkinteger(lua_State *L, int idx)
{
  cTValue *o = index2adr(L, idx);
  TValue tmp;
  lua_Number n;
  if (LJ_LIKELY(tvisint(o))) {
    return intV(o);
  } else if (LJ_LIKELY(tvisnum(o))) {
    n = numV(o);
  } else {
    if (!(tvisstr(o) && lj_strscan_number(strV(o), &tmp)))
```

```
      lj_err_argt(L, idx, LUA_TNUMBER);
    if (tvisint(&tmp))
      return (lua_Integer)intV(&tmp);
    n = numV(&tmp);
  }
#if LJ_64
  return (lua_Integer)n;
#else
  return lj_num2int(n);
#endif
}

LUALIB_API lua_Integer luaL_optinteger(lua_State *L, int idx, lua_Integer def)
{
  cTValue *o = index2adr(L, idx);
  TValue tmp;
  lua_Number n;
  if (LJ_LIKELY(tvisint(o))) {
    return intV(o);
  } else if (LJ_LIKELY(tvisnum(o))) {
    n = numV(o);
  } else if (tvisnil(o)) {
    return def;
  } else {
    if (!(tvisstr(o) && lj_strscan_number(strV(o), &tmp)))
      lj_err_argt(L, idx, LUA_TNUMBER);
    if (tvisint(&tmp))
      return (lua_Integer)intV(&tmp);
    n = numV(&tmp);
  }
#if LJ_64
  return (lua_Integer)n;
#else
  return lj_num2int(n);
#endif
}

LUA_API int lua_toboolean(lua_State *L, int idx)
{
  cTValue *o = index2adr(L, idx);
  return tvistruecond(o);
}

LUA_API const char *lua_tolstring(lua_State *L, int idx, size_t *len)
{
  TValue *o = index2adr(L, idx);
  GCstr *s;
  if (LJ_LIKELY(tvisstr(o))) {
    s = strV(o);
  } else if (tvisnumber(o)) {
    lj_gc_check(L);
    o = index2adr(L, idx);  /* GC may move the stack. */
    s = lj_strfmt_number(L, o);
    setstrV(L, o, s);
  } else {
    if (len != NULL) *len = 0;
    return NULL;
  }
  if (len != NULL) *len = s->len;
  return strdata(s);
}

LUALIB_API const char *luaL_checklstring(lua_State *L, int idx, size_t *len)
{
  TValue *o = index2adr(L, idx);
  GCstr *s;
  if (LJ_LIKELY(tvisstr(o))) {
    s = strV(o);
  } else if (tvisnumber(o)) {
    lj_gc_check(L);
    o = index2adr(L, idx);  /* GC may move the stack. */
    s = lj_strfmt_number(L, o);
    setstrV(L, o, s);
  } else {
    lj_err_argt(L, idx, LUA_TSTRING);
  }
```

```
462   if (len != NULL) *len = s->len;
463   return strdata(s);
464 }
465 
466 LUALIB_API const char *luaL_optlstring(lua_State *L, int idx,
467                                        const char *def, size_t *len)
468 {
469   TValue *o = index2adr(L, idx);
470   GCstr *s;
471   if (LJ_LIKELY(tvisstr(o))) {
472     s = strV(o);
473   } else if (tvisnil(o)) {
474     if (len != NULL) *len = def ? strlen(def) : 0;
475     return def;
476   } else if (tvisnumber(o)) {
477     lj_gc_check(L);
478     o = index2adr(L, idx);  /* GC may move the stack. */
479     s = lj_strfmt_number(L, o);
480     setstrV(L, o, s);
481   } else {
482     lj_err_argt(L, idx, LUA_TSTRING);
483   }
484   if (len != NULL) *len = s->len;
485   return strdata(s);
486 }
487 
488 LUALIB_API int luaL_checkoption(lua_State *L, int idx, const char *def,
489                                 const char *const lst[])
490 {
491   ptrdiff_t i;
492   const char *s = lua_tolstring(L, idx, NULL);
493   if (s == NULL && (s = def) == NULL)
494     lj_err_argt(L, idx, LUA_TSTRING);
495   for (i = 0; lst[i]; i++)
496     if (strcmp(lst[i], s) == 0)
497       return (int)i;
498   lj_err_argv(L, idx, LJ_ERR_INVOPTM, s);
499 }
500 
501 LUA_API size_t lua_objlen(lua_State *L, int idx)
502 {
503   TValue *o = index2adr(L, idx);
504   if (tvisstr(o)) {
505     return strV(o)->len;
506   } else if (tvistab(o)) {
507     return (size_t)lj_tab_len(tabV(o));
508   } else if (tvisudata(o)) {
509     return udataV(o)->len;
510   } else if (tvisnumber(o)) {
511     GCstr *s = lj_strfmt_number(L, o);
512     setstrV(L, o, s);
513     return s->len;
514   } else {
515     return 0;
516   }
517 }
518 
519 LUA_API lua_CFunction lua_tocfunction(lua_State *L, int idx)
520 {
521   cTValue *o = index2adr(L, idx);
522   if (tvisfunc(o)) {
523     BCOp op = bc_op(*mref(funcV(o)->c.pc, BCIns));
524     if (op == BC_FUNCC || op == BC_FUNCCW)
525       return funcV(o)->c.f;
526   }
527   return NULL;
528 }
529 
530 LUA_API void *lua_touserdata(lua_State *L, int idx)
531 {
532   cTValue *o = index2adr(L, idx);
533   if (tvisudata(o))
534     return uddata(udataV(o));
535   else if (tvislightud(o))
536     return lightudV(o);
537   else
```

```
538     return NULL;
539 }
540 
541 LUA_API lua_State *lua_tothread(lua_State *L, int idx)
542 {
543   cTValue *o = index2adr(L, idx);
544   return (!tvisthread(o)) ? NULL : threadV(o);
545 }
546 
547 LUA_API const void *lua_topointer(lua_State *L, int idx)
548 {
549   return lj_obj_ptr(index2adr(L, idx));
550 }
551 
552 /* -- Stack setters (object creation) ------------------------------------ */
553 
554 LUA_API void lua_pushnil(lua_State *L)
555 {
556   setnilV(L->top);
557   incr_top(L);
558 }
559 
560 LUA_API void lua_pushnumber(lua_State *L, lua_Number n)
561 {
562   setnumV(L->top, n);
563   if (LJ_UNLIKELY(tvisnan(L->top)))
564     setnanV(L->top);  /* Canonicalize injected NaNs. */
565   incr_top(L);
566 }
567 
568 LUA_API void lua_pushinteger(lua_State *L, lua_Integer n)
569 {
570   setintptrV(L->top, n);
571   incr_top(L);
572 }
573 
574 LUA_API void lua_pushlstring(lua_State *L, const char *str, size_t len)
575 {
576   GCstr *s;
577   lj_gc_check(L);
578   s = lj_str_new(L, str, len);
579   setstrV(L, L->top, s);
580   incr_top(L);
581 }
582 
583 LUA_API void lua_pushstring(lua_State *L, const char *str)
584 {
585   if (str == NULL) {
586     setnilV(L->top);
587   } else {
588     GCstr *s;
589     lj_gc_check(L);
590     s = lj_str_newz(L, str);
591     setstrV(L, L->top, s);
592   }
593   incr_top(L);
594 }
595 
596 LUA_API const char *lua_pushvfstring(lua_State *L, const char *fmt,
597                                      va_list argp)
598 {
599   lj_gc_check(L);
600   return lj_strfmt_pushvf(L, fmt, argp);
601 }
602 
603 LUA_API const char *lua_pushfstring(lua_State *L, const char *fmt, ...)
604 {
605   const char *ret;
606   va_list argp;
607   lj_gc_check(L);
608   va_start(argp, fmt);
609   ret = lj_strfmt_pushvf(L, fmt, argp);
610   va_end(argp);
611   return ret;
612 }
613 
```

```
LUA_API void lua_pushcclosure(lua_State *L, lua_CFunction f, int n)
{
  GCfunc *fn;
  lj_gc_check(L);
  api_checknelems(L, n);
  fn = lj_func_newC(L, (MSize)n, getcurrenv(L));
  fn->c.f = f;
  L->top -= n;
  while (n--)
    copyTV(L, &fn->c.upvalue[n], L->top+n);
  setfuncV(L, L->top, fn);
  lua_assert(iswhite(obj2gco(fn)));
  incr_top(L);
}

LUA_API void lua_pushboolean(lua_State *L, int b)
{
  setboolV(L->top, (b != 0));
  incr_top(L);
}

LUA_API void lua_pushlightuserdata(lua_State *L, void *p)
{
  setlightudV(L->top, checklightudptr(L, p));
  incr_top(L);
}

LUA_API void lua_createtable(lua_State *L, int narray, int nrec)
{
  lj_gc_check(L);
  settabV(L, L->top, lj_tab_new_ah(L, narray, nrec));
  incr_top(L);
}

LUALIB_API int luaL_newmetatable(lua_State *L, const char *tname)
{
  GCtab *regt = tabV(registry(L));
  TValue *tv = lj_tab_setstr(L, regt, lj_str_newz(L, tname));
  if (tvisnil(tv)) {
    GCtab *mt = lj_tab_new(L, 0, 1);
    settabV(L, tv, mt);
    settabV(L, L->top++, mt);
    lj_gc_anybarriert(L, regt);
    return 1;
  } else {
    copyTV(L, L->top++, tv);
    return 0;
  }
}

LUA_API int lua_pushthread(lua_State *L)
{
  setthreadV(L, L->top, L);
  incr_top(L);
  return (mainthread(G(L)) == L);
}

LUA_API lua_State *lua_newthread(lua_State *L)
{
  lua_State *L1;
  lj_gc_check(L);
  L1 = lj_state_new(L);
  setthreadV(L, L->top, L1);
  incr_top(L);
  return L1;
}

LUA_API void *lua_newuserdata(lua_State *L, size_t size)
{
  GCudata *ud;
  lj_gc_check(L);
  if (size > LJ_MAX_UDATA)
    lj_err_msg(L, LJ_ERR_UDATAOV);
  ud = lj_udata_new(L, (MSize)size, getcurrenv(L));
  setudataV(L, L->top, ud);
  incr_top(L);
```

```
  return uddata(ud);
}

LUA_API void lua_concat(lua_State *L, int n)
{
  api_checknelems(L, n);
  if (n >= 2) {
    n--;
    do {
      TValue *top = lj_meta_cat(L, L->top-1, -n);
      if (top == NULL) {
        L->top -= n;
        break;
      }
      n -= (int)(L->top - top);
      L->top = top+2;
      lj_vm_call(L, top, 1+1);
      L->top -= 1+LJ_FR2;
      copyTV(L, L->top-1, L->top+LJ_FR2);
    } while (--n > 0);
  } else if (n == 0) {  /* Push empty string. */
    setstrV(L, L->top, &G(L)->strempty);
    incr_top(L);
  }
  /* else n == 1: nothing to do. */
}

/* -- Object getters ------------------------------------------------------ */

LUA_API void lua_gettable(lua_State *L, int idx)
{
  cTValue *v, *t = index2adr(L, idx);
  api_checkvalidindex(L, t);
  v = lj_meta_tget(L, t, L->top-1);
  if (v == NULL) {
    L->top += 2;
    lj_vm_call(L, L->top-2, 1+1);
    L->top -= 2+LJ_FR2;
    v = L->top+1+LJ_FR2;
  }
  copyTV(L, L->top-1, v);
}

LUA_API void lua_getfield(lua_State *L, int idx, const char *k)
{
  cTValue *v, *t = index2adr(L, idx);
  TValue key;
  api_checkvalidindex(L, t);
  setstrV(L, &key, lj_str_newz(L, k));
  v = lj_meta_tget(L, t, &key);
  if (v == NULL) {
    L->top += 2;
    lj_vm_call(L, L->top-2, 1+1);
    L->top -= 2+LJ_FR2;
    v = L->top+1+LJ_FR2;
  }
  copyTV(L, L->top, v);
  incr_top(L);
}

LUA_API void lua_rawget(lua_State *L, int idx)
{
  cTValue *t = index2adr(L, idx);
  api_check(L, tvistab(t));
  copyTV(L, L->top-1, lj_tab_get(L, tabV(t), L->top-1));
}

LUA_API void lua_rawgeti(lua_State *L, int idx, int n)
{
  cTValue *v, *t = index2adr(L, idx);
  api_check(L, tvistab(t));
  v = lj_tab_getint(tabV(t), n);
  if (v) {
    copyTV(L, L->top, v);
  } else {
    setnilV(L->top);
```

```
  }
  incr_top(L);
}

LUA_API int lua_getmetatable(lua_State *L, int idx)
{
  cTValue *o = index2adr(L, idx);
  GCtab *mt = NULL;
  if (tvistab(o))
    mt = tabref(tabV(o)->metatable);
  else if (tvisudata(o))
    mt = tabref(udataV(o)->metatable);
  else
    mt = tabref(basemt_obj(G(L), o));
  if (mt == NULL)
    return 0;
  settabV(L, L->top, mt);
  incr_top(L);
  return 1;
}

LUALIB_API int luaL_getmetafield(lua_State *L, int idx, const char *field)
{
  if (lua_getmetatable(L, idx)) {
    cTValue *tv = lj_tab_getstr(tabV(L->top-1), lj_str_newz(L, field));
    if (tv && !tvisnil(tv)) {
      copyTV(L, L->top-1, tv);
      return 1;
    }
    L->top--;
  }
  return 0;
}

LUA_API void lua_getfenv(lua_State *L, int idx)
{
  cTValue *o = index2adr(L, idx);
  api_checkvalidindex(L, o);
  if (tvisfunc(o)) {
    settabV(L, L->top, tabref(funcV(o)->c.env));
  } else if (tvisudata(o)) {
    settabV(L, L->top, tabref(udataV(o)->env));
  } else if (tvisthread(o)) {
    settabV(L, L->top, tabref(threadV(o)->env));
  } else {
    setnilV(L->top);
  }
  incr_top(L);
}

LUA_API int lua_next(lua_State *L, int idx)
{
  cTValue *t = index2adr(L, idx);
  int more;
  api_check(L, tvistab(t));
  more = lj_tab_next(L, tabV(t), L->top-1);
  if (more) {
    incr_top(L);  /* Return new key and value slot. */
  } else {  /* End of traversal. */
    L->top--;  /* Remove key slot. */
  }
  return more;
}

LUA_API const char *lua_getupvalue(lua_State *L, int idx, int n)
{
  TValue *val;
  const char *name = lj_debug_uvnamev(index2adr(L, idx), (uint32_t)(n-1), &val);
  if (name) {
    copyTV(L, L->top, val);
    incr_top(L);
  }
  return name;
}

LUA_API void *lua_upvalueid(lua_State *L, int idx, int n)
```

```
{
  GCfunc *fn = funcV(index2adr(L, idx));
  n--;
  api_check(L, (uint32_t)n < fn->l.nupvalues);
  return isluafunc(fn) ? (void *)gcref(fn->l.uvptr[n]) :
                         (void *)&fn->c.upvalue[n];
}

LUA_API void lua_upvaluejoin(lua_State *L, int idx1, int n1, int idx2, int n2)
{
  GCfunc *fn1 = funcV(index2adr(L, idx1));
  GCfunc *fn2 = funcV(index2adr(L, idx2));
  n1--; n2--;
  api_check(L, isluafunc(fn1) && (uint32_t)n1 < fn1->l.nupvalues);
  api_check(L, isluafunc(fn2) && (uint32_t)n2 < fn2->l.nupvalues);
  setgcrefr(fn1->l.uvptr[n1], fn2->l.uvptr[n2]);
  lj_gc_objbarrier(L, fn1, gcref(fn1->l.uvptr[n1]));
}

LUALIB_API void *luaL_checkudata(lua_State *L, int idx, const char *tname)
{
  cTValue *o = index2adr(L, idx);
  if (tvisudata(o)) {
    GCudata *ud = udataV(o);
    cTValue *tv = lj_tab_getstr(tabV(registry(L)), lj_str_newz(L, tname));
    if (tv && tvistab(tv) && tabV(tv) == tabref(ud->metatable))
      return uddata(ud);
  }
  lj_err_argtype(L, idx, tname);
  return NULL;  /* unreachable */
}

/* -- Object setters ------------------------------------------------------ */

LUA_API void lua_settable(lua_State *L, int idx)
{
  TValue *o;
  cTValue *t = index2adr(L, idx);
  api_checknelems(L, 2);
  api_checkvalidindex(L, t);
  o = lj_meta_tset(L, t, L->top-2);
  if (o) {
    /* NOBARRIER: lj_meta_tset ensures the table is not black. */
    L->top -= 2;
    copyTV(L, o, L->top+1);
  } else {
    TValue *base = L->top;
    copyTV(L, base+2, base-3-2*LJ_FR2);
    L->top = base+3;
    lj_vm_call(L, base, 0+1);
    L->top -= 3+LJ_FR2;
  }
}

LUA_API void lua_setfield(lua_State *L, int idx, const char *k)
{
  TValue *o;
  TValue key;
  cTValue *t = index2adr(L, idx);
  api_checknelems(L, 1);
  api_checkvalidindex(L, t);
  setstrV(L, &key, lj_str_newz(L, k));
  o = lj_meta_tset(L, t, &key);
  if (o) {
    /* NOBARRIER: lj_meta_tset ensures the table is not black. */
    copyTV(L, o, --L->top);
  } else {
    TValue *base = L->top;
    copyTV(L, base+2, base-3-2*LJ_FR2);
    L->top = base+3;
    lj_vm_call(L, base, 0+1);
    L->top -= 2+LJ_FR2;
  }
}

LUA_API void lua_rawset(lua_State *L, int idx)
```

```
{
  GCtab *t = tabV(index2adr(L, idx));
  TValue *dst, *key;
  api_checknelems(L, 2);
  key = L->top-2;
  dst = lj_tab_set(L, t, key);
  copyTV(L, dst, key+1);
  lj_gc_anybarriert(L, t);
  L->top = key;
}

LUA_API void lua_rawseti(lua_State *L, int idx, int n)
{
  GCtab *t = tabV(index2adr(L, idx));
  TValue *dst, *src;
  api_checknelems(L, 1);
  dst = lj_tab_setint(L, t, n);
  src = L->top-1;
  copyTV(L, dst, src);
  lj_gc_barriert(L, t, dst);
  L->top = src;
}

LUA_API int lua_setmetatable(lua_State *L, int idx)
{
  global_State *g;
  GCtab *mt;
  cTValue *o = index2adr(L, idx);
  api_checknelems(L, 1);
  api_checkvalidindex(L, o);
  if (tvisnil(L->top-1)) {
    mt = NULL;
  } else {
    api_check(L, tvistab(L->top-1));
    mt = tabV(L->top-1);
  }
  g = G(L);
  if (tvistab(o)) {
    setgcref(tabV(o)->metatable, obj2gco(mt));
    if (mt)
      lj_gc_objbarriert(L, tabV(o), mt);
  } else if (tvisudata(o)) {
    setgcref(udataV(o)->metatable, obj2gco(mt));
    if (mt)
      lj_gc_objbarrier(L, udataV(o), mt);
  } else {
    /* Flush cache, since traces specialize to basemt. But not during __gc. */
    if (lj_trace_flushall(L))
      lj_err_caller(L, LJ_ERR_NOGCMM);
    if (tvisbool(o)) {
      /* NOBARRIER: basemt is a GC root. */
      setgcref(basemt_it(g, LJ_TTRUE), obj2gco(mt));
      setgcref(basemt_it(g, LJ_TFALSE), obj2gco(mt));
    } else {
      /* NOBARRIER: basemt is a GC root. */
      setgcref(basemt_obj(g, o), obj2gco(mt));
    }
  }
  L->top--;
  return 1;
}

LUA_API int lua_setfenv(lua_State *L, int idx)
{
  cTValue *o = index2adr(L, idx);
  GCtab *t;
  api_checknelems(L, 1);
  api_checkvalidindex(L, o);
  api_check(L, tvistab(L->top-1));
  t = tabV(L->top-1);
  if (tvisfunc(o)) {
    setgcref(funcV(o)->c.env, obj2gco(t));
  } else if (tvisudata(o)) {
    setgcref(udataV(o)->env, obj2gco(t));
  } else if (tvisthread(o)) {
    setgcref(threadV(o)->env, obj2gco(t));
```

```
 } else {
   L->top--;
   return 0;
 }
 lj_gc_objbarrier(L, gcV(o), t);
 L->top--;
  return 1;
}

LUA_API const char *lua_setupvalue(lua_State *L, int idx, int n)
{
  cTValue *f = index2adr(L, idx);
  TValue *val;
  const char *name;
  api_checknelems(L, 1);
  name = lj_debug_uvnamev(f, (uint32_t)(n-1), &val);
  if (name) {
    L->top--;
    copyTV(L, val, L->top);
    lj_gc_barrier(L, funcV(f), L->top);
  }
  return name;
}

/* -- Calls ---------------------------------------------------------- */

#if LJ_FR2
static TValue *api_call_base(lua_State *L, int nargs)
{
  TValue *o = L->top, *base = o - nargs;
  L->top = o+1;
  for (; o > base; o--) copyTV(L, o, o-1);
  setnilV(o);
  return o+1;
}
#else
#define api_call_base(L, nargs)        (L->top - (nargs))
#endif

LUA_API void lua_call(lua_State *L, int nargs, int nresults)
{
  api_check(L, L->status == 0 || L->status == LUA_ERRERR);
  api_checknelems(L, nargs+1);
  lj_vm_call(L, api_call_base(L, nargs), nresults+1);
}

LUA_API int lua_pcall(lua_State *L, int nargs, int nresults, int errfunc)
{
  global_State *g = G(L);
  uint8_t oldh = hook_save(g);
  ptrdiff_t ef;
  int status;
  api_check(L, L->status == 0 || L->status == LUA_ERRERR);
  api_checknelems(L, nargs+1);
  if (errfunc == 0) {
    ef = 0;
  } else {
    cTValue *o = stkindex2adr(L, errfunc);
    api_checkvalidindex(L, o);
    ef = savestack(L, o);
  }
  status = lj_vm_pcall(L, api_call_base(L, nargs), nresults+1, ef);
  if (status) hook_restore(g, oldh);
  return status;
}

static TValue *cpcall(lua_State *L, lua_CFunction func, void *ud)
{
  GCfunc *fn = lj_func_newC(L, 0, getcurrenv(L));
  TValue *top = L->top;
  fn->c.f = func;
  setfuncV(L, top++, fn);
  if (LJ_FR2) setnilV(top++);
  setlightudV(top++, checklightudptr(L, ud));
  cframe_nres(L->cframe) = 1+0;  /* Zero results. */
  L->top = top;
```

```
  return top-1;  /* Now call the newly allocated C function. */
}

LUA_API int lua_cpcall(lua_State *L, lua_CFunction func, void *ud)
{
  global_State *g = G(L);
  uint8_t oldh = hook_save(g);
  int status;
  api_check(L, L->status == 0 || L->status == LUA_ERRERR);
  status = lj_vm_cpcall(L, func, ud, cpcall);
  if (status) hook_restore(g, oldh);
  return status;
}

LUALIB_API int luaL_callmeta(lua_State *L, int idx, const char *field)
{
  if (luaL_getmetafield(L, idx, field)) {
    TValue *top = L->top--;
    if (LJ_FR2) setnilV(top++);
    copyTV(L, top++, index2adr(L, idx));
    L->top = top;
    lj_vm_call(L, top-1, 1+1);
    return 1;
  }
  return 0;
}

/* -- Coroutine yield and resume ----------------------------------------- */

LUA_API int lua_yield(lua_State *L, int nresults)
{
  void *cf = L->cframe;
  global_State *g = G(L);
  if (cframe_canyield(cf)) {
    cf = cframe_raw(cf);
    if (!hook_active(g)) {  /* Regular yield: move results down if needed. */
      cTValue *f = L->top - nresults;
      if (f > L->base) {
        TValue *t = L->base;
        while (--nresults >= 0) copyTV(L, t++, f++);
        L->top = t;
      }
      L->cframe = NULL;
      L->status = LUA_YIELD;
      return -1;
    } else {  /* Yield from hook: add a pseudo-frame. */
      TValue *top = L->top;
      hook_leave(g);
      (top++)->u64 = cframe_multres(cf);
      setcont(top, lj_cont_hook);
      if (LJ_FR2) top++;
      setframe_pc(top, cframe_pc(cf)-1);
      if (LJ_FR2) top++;
      setframe_gc(top, obj2gco(L), LJ_TTHREAD);
      setframe_ftsz(top, ((char *)(top+1)-(char *)L->base)+FRAME_CONT);
      L->top = L->base = top+1;
#if LJ_TARGET_X64
      lj_err_throw(L, LUA_YIELD);
#else
      L->cframe = NULL;
      L->status = LUA_YIELD;
      lj_vm_unwind_c(cf, LUA_YIELD);
#endif
    }
  }
  lj_err_msg(L, LJ_ERR_CYIELD);
  return 0;  /* unreachable */
}

LUA_API int lua_resume(lua_State *L, int nargs)
{
  if (L->cframe == NULL && L->status <= LUA_YIELD)
    return lj_vm_resume(L,
      L->status == 0 ? api_call_base(L, nargs) : L->top - nargs,
      0, 0);
  L->top = L->base;
```

```
1146   setstrV(L, L->top, lj_err_str(L, LJ_ERR_COSUSP));
1147   incr_top(L);
1148   return LUA_ERRRUN;
1149 }
1150
1151 /* -- GC and memory management ------------------------------------------ */
1152
1153 LUA_API int lua_gc(lua_State *L, int what, int data)
1154 {
1155   global_State *g = G(L);
1156   int res = 0;
1157   switch (what) {
1158   case LUA_GCSTOP:
1159     g->gc.threshold = LJ_MAX_MEM;
1160     break;
1161   case LUA_GCRESTART:
1162     g->gc.threshold = data == -1 ? (g->gc.total/100)*g->gc.pause : g->gc.total;
1163     break;
1164   case LUA_GCCOLLECT:
1165     lj_gc_fullgc(L);
1166     break;
1167   case LUA_GCCOUNT:
1168     res = (int)(g->gc.total >> 10);
1169     break;
1170   case LUA_GCCOUNTB:
1171     res = (int)(g->gc.total & 0x3ff);
1172     break;
1173   case LUA_GCSTEP: {
1174     GCSize a = (GCSize)data << 10;
1175     g->gc.threshold = (a <= g->gc.total) ? (g->gc.total - a) : 0;
1176     while (g->gc.total >= g->gc.threshold)
1177       if (lj_gc_step(L) > 0) {
1178         res = 1;
1179         break;
1180       }
1181     break;
1182   }
1183   case LUA_GCSETPAUSE:
1184     res = (int)(g->gc.pause);
1185     g->gc.pause = (MSize)data;
1186     break;
1187   case LUA_GCSETSTEPMUL:
1188     res = (int)(g->gc.stepmul);
1189     g->gc.stepmul = (MSize)data;
1190     break;
1191   default:
1192     res = -1;  /* Invalid option. */
1193   }
1194   return res;
1195 }
1196
1197 LUA_API lua_Alloc lua_getallocf(lua_State *L, void **ud)
1198 {
1199   global_State *g = G(L);
1200   if (ud) *ud = g->allocd;
1201   return g->allocf;
1202 }
1203
1204 LUA_API void lua_setallocf(lua_State *L, lua_Alloc f, void *ud)
1205 {
1206   global_State *g = G(L);
1207   g->allocd = ud;
1208   g->allocf = f;
1209 }
1210
```

One Level Up Top Level

src/lua.h - luajit-2.0-src

Data types defined

- lua_Alloc
- lua_CFunction
- lua_Debug
- lua_Debug
- lua_Hook
- lua_Integer
- lua_Number
- lua_Reader
- lua_State
- lua_Writer

Macros defined

- LUA_AUTHORS
- LUA_COPYRIGHT
- LUA_ENVIRONINDEX
- LUA_ERRERR
- LUA_ERRMEM
- LUA_ERRRUN
- LUA_ERRSYNTAX
- LUA_GCCOLLECT
- LUA_GCCOUNT
- LUA_GCCOUNTB
- LUA_GCRESTART
- LUA_GCSETPAUSE
- LUA_GCSETSTEPMUL
- LUA_GCSTEP
- LUA_GCSTOP
- LUA_GLOBALSINDEX
- LUA_HOOKCALL
- LUA_HOOKCOUNT

- LUA_HOOKLINE
- LUA_HOOKRET
- LUA_HOOKTAILRET
- LUA_MASKCALL
- LUA_MASKCOUNT
- LUA_MASKLINE
- LUA_MASKRET
- LUA_MINSTACK
- LUA_MULTRET
- LUA_REGISTRYINDEX
- LUA_RELEASE
- LUA_SIGNATURE
- LUA_TBOOLEAN
- LUA_TFUNCTION
- LUA_TLIGHTUSERDATA
- LUA_TNIL
- LUA_TNONE
- LUA_TNUMBER
- LUA_TSTRING
- LUA_TTABLE
- LUA_TTHREAD
- LUA_TUSERDATA
- LUA_VERSION
- LUA_VERSION_NUM
- LUA_YIELD
- lua_Chunkreader
- lua_Chunkwriter
- lua_getgccount
- lua_getglobal
- lua_getregistry
- lua_h
- lua_isboolean
- lua_isfunction

- lua_islightuserdata
- lua_isnil
- lua_isnone
- lua_isnoneornil
- lua_istable
- lua_isthread
- lua_newtable
- lua_open
- lua_pop
- lua_pushcfunction
- lua_pushliteral
- lua_register
- lua_setglobal
- lua_strlen
- lua_tostring
- lua_upvalueindex

Source code

```
/*
** $Id: lua.h,v 1.218.1.5 2008/08/06 13:30:12 roberto Exp $
** Lua - An Extensible Extension Language
** Lua.org, PUC-Rio, Brazil (http://www.lua.org)
** See Copyright Notice at the end of this file
*/


#ifndef lua_h
 #define lua_h

 #include <stdarg.h>
 #include <stddef.h>


 #include "luaconf.h"


 #define LUA_VERSION        "Lua 5.1"
 #define LUA_RELEASE        "Lua 5.1.4"
 #define LUA_VERSION_NUM        501
 #define LUA_COPYRIGHT        "Copyright (C) 1994-2008 Lua.org, PUC-Rio"
 #define LUA_AUTHORS        "R. Ierusalimschy, L. H. de Figueiredo & W. Celes"


 /* mark for precompiled code (`<esc>Lua') */
 #define        LUA_SIGNATURE        "\033Lua"

 /* option for multiple returns in `lua_pcall' and `lua_call' */
 #define LUA_MULTRET        (-1)


 /*
 ** pseudo-indices
 */
 #define LUA_REGISTRYINDEX        (-10000)
```

```c
#define LUA_ENVIRONINDEX        (-10001)
#define LUA_GLOBALSINDEX        (-10002)
#define lua_upvalueindex(i)        (LUA_GLOBALSINDEX-(i))


/* thread status; 0 is OK */
#define LUA_YIELD        1
#define LUA_ERRRUN        2
#define LUA_ERRSYNTAX        3
#define LUA_ERRMEM        4
#define LUA_ERRERR        5


typedef struct lua_State lua_State;

typedef int (*lua_CFunction) (lua_State *L);


/*
** functions that read/write blocks when loading/dumping Lua chunks
*/
typedef const char * (*lua_Reader) (lua_State *L, void *ud, size_t *sz);

typedef int (*lua_Writer) (lua_State *L, const void* p, size_t sz, void* ud);


/*
** prototype for memory-allocation functions
*/
typedef void * (*lua_Alloc) (void *ud, void *ptr, size_t osize, size_t nsize);


/*
** basic types
*/
#define LUA_TNONE                (-1)

#define LUA_TNIL                0
#define LUA_TBOOLEAN                1
#define LUA_TLIGHTUSERDATA        2
#define LUA_TNUMBER                3
#define LUA_TSTRING                4
#define LUA_TTABLE                5
#define LUA_TFUNCTION                6
#define LUA_TUSERDATA                7
#define LUA_TTHREAD                8



/* minimum Lua stack available to a C function */
#define LUA_MINSTACK        20


/*
** generic extra include file
*/
#if defined(LUA_USER_H)
#include LUA_USER_H
#endif


/* type of numbers in Lua */
typedef LUA_NUMBER lua_Number;


 /* type for integer functions */
 typedef LUA_INTEGER lua_Integer;



 /*
 ** state manipulation
 */
 LUA_API lua_State *(lua_newstate) (lua_Alloc f, void *ud);
 LUA_API void       (lua_close) (lua_State *L);
 LUA_API lua_State *(lua_newthread) (lua_State *L);
```

```
113
114 LUA_API lua_CFunction (lua_atpanic) (lua_State *L, lua_CFunction panicf);
115
116
117 /*
118 ** basic stack manipulation
119 */
120 LUA_API int   (lua_gettop) (lua_State *L);
121 LUA_API void  (lua_settop) (lua_State *L, int idx);
122 LUA_API void  (lua_pushvalue) (lua_State *L, int idx);
123 LUA_API void  (lua_remove) (lua_State *L, int idx);
124 LUA_API void  (lua_insert) (lua_State *L, int idx);
125 LUA_API void  (lua_replace) (lua_State *L, int idx);
126 LUA_API int   (lua_checkstack) (lua_State *L, int sz);
127
128 LUA_API void  (lua_xmove) (lua_State *from, lua_State *to, int n);
129
130
131 /*
132 ** access functions (stack -> C)
133 */
134
135 LUA_API int             (lua_isnumber) (lua_State *L, int idx);
136 LUA_API int             (lua_isstring) (lua_State *L, int idx);
137 LUA_API int             (lua_iscfunction) (lua_State *L, int idx);
138 LUA_API int             (lua_isuserdata) (lua_State *L, int idx);
139 LUA_API int             (lua_type) (lua_State *L, int idx);
140 LUA_API const char     *(lua_typename) (lua_State *L, int tp);
141
142 LUA_API int            (lua_equal) (lua_State *L, int idx1, int idx2);
143 LUA_API int            (lua_rawequal) (lua_State *L, int idx1, int idx2);
144 LUA_API int            (lua_lessthan) (lua_State *L, int idx1, int idx2);
145
146 LUA_API lua_Number      (lua_tonumber) (lua_State *L, int idx);
147 LUA_API lua_Integer     (lua_tointeger) (lua_State *L, int idx);
148 LUA_API int             (lua_toboolean) (lua_State *L, int idx);
149 LUA_API const char     *(lua_tolstring) (lua_State *L, int idx, size_t *len);
150 LUA_API size_t          (lua_objlen) (lua_State *L, int idx);
151 LUA_API lua_CFunction   (lua_tocfunction) (lua_State *L, int idx);
152 LUA_API void               *(lua_touserdata) (lua_State *L, int idx);
153 LUA_API lua_State      *(lua_tothread) (lua_State *L, int idx);
154 LUA_API const void     *(lua_topointer) (lua_State *L, int idx);
155
156
157 /*
158 ** push functions (C -> stack)
159 */
160 LUA_API void  (lua_pushnil) (lua_State *L);
161 LUA_API void  (lua_pushnumber) (lua_State *L, lua_Number n);
162 LUA_API void  (lua_pushinteger) (lua_State *L, lua_Integer n);
163 LUA_API void  (lua_pushlstring) (lua_State *L, const char *s, size_t l);
164 LUA_API void  (lua_pushstring) (lua_State *L, const char *s);
165 LUA_API const char *(lua_pushvfstring) (lua_State *L, const char *fmt,
166                                                       va_list argp);
167 LUA_API const char *(lua_pushfstring) (lua_State *L, const char *fmt, ...);
168 LUA_API void  (lua_pushcclosure) (lua_State *L, lua_CFunction fn, int n);
169 LUA_API void  (lua_pushboolean) (lua_State *L, int b);
170 LUA_API void  (lua_pushlightuserdata) (lua_State *L, void *p);
171 LUA_API int   (lua_pushthread) (lua_State *L);
172
173
174 /*
175 ** get functions (Lua -> stack)
176 */
177 LUA_API void  (lua_gettable) (lua_State *L, int idx);
178 LUA_API void  (lua_getfield) (lua_State *L, int idx, const char *k);
179 LUA_API void  (lua_rawget) (lua_State *L, int idx);
180 LUA_API void  (lua_rawgeti) (lua_State *L, int idx, int n);
181 LUA_API void  (lua_createtable) (lua_State *L, int narr, int nrec);
182 LUA_API void *(lua_newuserdata) (lua_State *L, size_t sz);
183 LUA_API int   (lua_getmetatable) (lua_State *L, int objindex);
184 LUA_API void  (lua_getfenv) (lua_State *L, int idx);
185
186
187 /*
188 ** set functions (stack -> Lua)
```

```
*/
LUA_API void  (lua_settable) (lua_State *L, int idx);
LUA_API void  (lua_setfield) (lua_State *L, int idx, const char *k);
LUA_API void  (lua_rawset) (lua_State *L, int idx);
LUA_API void  (lua_rawseti) (lua_State *L, int idx, int n);
LUA_API int   (lua_setmetatable) (lua_State *L, int objindex);
LUA_API int   (lua_setfenv) (lua_State *L, int idx);


/*
** `load' and `call' functions (load and run Lua code)
*/
LUA_API void  (lua_call) (lua_State *L, int nargs, int nresults);
LUA_API int   (lua_pcall) (lua_State *L, int nargs, int nresults, int errfunc);
LUA_API int   (lua_cpcall) (lua_State *L, lua_CFunction func, void *ud);
LUA_API int   (lua_load) (lua_State *L, lua_Reader reader, void *dt,
                                        const char *chunkname);

LUA_API int (lua_dump) (lua_State *L, lua_Writer writer, void *data);


/*
** coroutine functions
*/
LUA_API int  (lua_yield) (lua_State *L, int nresults);
LUA_API int  (lua_resume) (lua_State *L, int narg);
LUA_API int  (lua_status) (lua_State *L);

/*
** garbage-collection function and options
*/

#define LUA_GCSTOP                0
#define LUA_GCRESTART               1
#define LUA_GCCOLLECT               2
#define LUA_GCCOUNT                3
#define LUA_GCCOUNTB                4
#define LUA_GCSTEP                5
#define LUA_GCSETPAUSE                6
#define LUA_GCSETSTEPMUL        7

LUA_API int (lua_gc) (lua_State *L, int what, int data);


/*
** miscellaneous functions
*/

LUA_API int   (lua_error) (lua_State *L);

LUA_API int   (lua_next) (lua_State *L, int idx);

LUA_API void  (lua_concat) (lua_State *L, int n);

LUA_API lua_Alloc (lua_getallocf) (lua_State *L, void **ud);
LUA_API void lua_setallocf (lua_State *L, lua_Alloc f, void *ud);



/*
** ===============================================================
** some useful macros
** ===============================================================
*/

#define lua_pop(L,n)                lua_settop(L, -(n)-1)

#define lua_newtable(L)                lua_createtable(L, 0, 0)

#define lua_register(L,n,f) (lua_pushcfunction(L, (f)), lua_setglobal(L, (n)))

#define lua_pushcfunction(L,f)        lua_pushcclosure(L, (f), 0)

#define lua_strlen(L,i)                lua_objlen(L, (i))

#define lua_isfunction(L,n)        (lua_type(L, (n)) == LUA_TFUNCTION)
```

```
#define lua_istable(L,n)        (lua_type(L, (n)) == LUA_TTABLE)
#define lua_islightuserdata(L,n)        (lua_type(L, (n)) == LUA_TLIGHTUSERDATA)
#define lua_isnil(L,n)                (lua_type(L, (n)) == LUA_TNIL)
#define lua_isboolean(L,n)        (lua_type(L, (n)) == LUA_TBOOLEAN)
#define lua_isthread(L,n)        (lua_type(L, (n)) == LUA_TTHREAD)
#define lua_isnone(L,n)                (lua_type(L, (n)) == LUA_TNONE)
#define lua_isnoneornil(L, n)        (lua_type(L, (n)) <= 0)

#define lua_pushliteral(L, s)        \
        lua_pushlstring(L, "" s, (sizeof(s)/sizeof(char))-1)

#define lua_setglobal(L,s)        lua_setfield(L, LUA_GLOBALSINDEX, (s))
#define lua_getglobal(L,s)        lua_getfield(L, LUA_GLOBALSINDEX, (s))

#define lua_tostring(L,i)        lua_tolstring(L, (i), NULL)



/*
** compatibility macros and functions
*/

#define lua_open()        luaL_newstate()

#define lua_getregistry(L)        lua_pushvalue(L, LUA_REGISTRYINDEX)

#define lua_getgccount(L)        lua_gc(L, LUA_GCCOUNT, 0)

#define lua_Chunkreader                lua_Reader
#define lua_Chunkwriter                lua_Writer


/* hack */
LUA_API void lua_setlevel        (lua_State *from, lua_State *to);


/*
** {======================================================================
** Debug API
** =======================================================================
*/


/*
** Event codes
*/
#define LUA_HOOKCALL        0
#define LUA_HOOKRET        1
#define LUA_HOOKLINE        2
#define LUA_HOOKCOUNT        3
#define LUA_HOOKTAILRET 4


/*
** Event masks
*/
#define LUA_MASKCALL        (1 << LUA_HOOKCALL)
#define LUA_MASKRET        (1 << LUA_HOOKRET)
#define LUA_MASKLINE        (1 << LUA_HOOKLINE)
#define LUA_MASKCOUNT        (1 << LUA_HOOKCOUNT)

typedef struct lua_Debug lua_Debug;  /* activation record */


/* Functions to be called by the debuger in specific events */
typedef void (*lua_Hook) (lua_State *L, lua_Debug *ar);


LUA_API int lua_getstack (lua_State *L, int level, lua_Debug *ar);
LUA_API int lua_getinfo (lua_State *L, const char *what, lua_Debug *ar);
LUA_API const char *lua_getlocal (lua_State *L, const lua_Debug *ar, int n);
LUA_API const char *lua_setlocal (lua_State *L, const lua_Debug *ar, int n);
LUA_API const char *lua_getupvalue (lua_State *L, int funcindex, int n);
LUA_API const char *lua_setupvalue (lua_State *L, int funcindex, int n);
LUA_API int lua_sethook (lua_State *L, lua_Hook func, int mask, int count);
LUA_API lua_Hook lua_gethook (lua_State *L);
```

```
LUA_API int lua_gethookmask (lua_State *L);
LUA_API int lua_gethookcount (lua_State *L);

/* From Lua 5.2. */
LUA_API void *lua_upvalueid (lua_State *L, int idx, int n);
LUA_API void lua_upvaluejoin (lua_State *L, int idx1, int n1, int idx2, int n2);
LUA_API int lua_loadx (lua_State *L, lua_Reader reader, void *dt,
                       const char *chunkname, const char *mode);


struct lua_Debug {
  int event;
  const char *name;        /* (n) */
  const char *namewhat;        /* (n) `global', `local', `field', `method' */
  const char *what;        /* (S) `Lua', `C', `main', `tail' */
  const char *source;        /* (S) */
  int currentline;        /* (l) */
  int nups;                /* (u) number of upvalues */
  int linedefined;        /* (S) */
  int lastlinedefined;        /* (S) */
  char short_src[LUA_IDSIZE]; /* (S) */
  /* private part */
  int i_ci;  /* active function */
};

/* }====================================================================== */


/******************************************************************************
* Copyright (C) 1994-2008 Lua.org, PUC-Rio.  All rights reserved.
*
* Permission is hereby granted, free of charge, to any person obtaining
* a copy of this software and associated documentation files (the
* "Software"), to deal in the Software without restriction, including
* without limitation the rights to use, copy, modify, merge, publish,
* distribute, sublicense, and/or sell copies of the Software, and to
* permit persons to whom the Software is furnished to do so, subject to
* the following conditions:
*
* The above copyright notice and this permission notice shall be
* included in all copies or substantial portions of the Software.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
* EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
* MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
* IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
* CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
* TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
* SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
******************************************************************************/


#endif
```

One Level Up Top Level

src/luaconf.h - luajit-2.0-src

Macros defined

- LUAI_GCMUL
- LUAI_GCPAUSE
- LUAI_MAXCSTACK
- LUAI_MAXNUMBER2STR
- LUAI_MAXSTACK
- LUAI_UACNUMBER
- LUALIB_API
- LUAL_BUFFERSIZE
- LUA_API
- LUA_API
- LUA_API
- LUA_CDIR
- LUA_COMPAT_GFIND
- LUA_COMPAT_MOD
- LUA_CPATH
- LUA_CPATH_DEFAULT
- LUA_CPATH_DEFAULT
- LUA_DIRSEP
- LUA_DIRSEP
- LUA_EXECDIR
- LUA_IDSIZE
- LUA_IGMARK
- LUA_INIT
- LUA_INTEGER
- LUA_INTFRMLEN
- LUA_INTFRM_T
- LUA_JPATH
- LUA_JROOT
- LUA_JROOT
- LUA_LCDIR

- LUA_LCPATH1
- LUA_LCPATH2
- LUA_LDIR
- LUA_LJDIR
- LUA_LLDIR
- LUA_LLPATH
- LUA_LMULTILIB
- LUA_LROOT
- LUA_LUADIR
- LUA_MAXCAPTURES
- LUA_MAXINPUT
- LUA_MULTILIB
- LUA_NUMBER
- LUA_NUMBER_DOUBLE
- LUA_NUMBER_FMT
- LUA_NUMBER_SCAN
- LUA_PATH
- LUA_PATHSEP
- LUA_PATH_CONFIG
- LUA_PATH_DEFAULT
- LUA_PATH_DEFAULT
- LUA_PATH_MARK
- LUA_PROGNAME
- LUA_PROMPT
- LUA_PROMPT2
- LUA_QL
- LUA_QS
- LUA_RCDIR
- LUA_RCPATH
- LUA_RCPATH
- LUA_RLDIR
- LUA_RLPATH
- LUA_RLPATH

- WINVER
- lua_assert
- lua_number2str
- luaconf_h
- luai_apicheck
- luai_apicheck

Source code

```
/*
** Configuration header.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef luaconf_h
#define luaconf_h

#ifndef WINVER
 #define WINVER 0x0501
 #endif
 #include <limits.h>
 #include <stddef.h>

 /* Default path for loading Lua and C modules with require(). */
 #if defined(_WIN32)
 /*
 ** In Windows, any exclamation mark ('!') in the path is replaced by the
 ** path of the directory of the executable file of the current process.
 */
 #define LUA_LDIR        "!\\lua\\"
 #define LUA_CDIR        "!\\"
 #define LUA_PATH_DEFAULT \
   ".\\?.lua;" LUA_LDIR"?.lua;" LUA_LDIR"?\\init.lua;"
 #define LUA_CPATH_DEFAULT \
   ".\\?.dll;" LUA_CDIR"?.dll;" LUA_CDIR"loadall.dll"
 #else
 /*
 ** Note to distribution maintainers: do NOT patch the following lines!
 ** Please read ../doc/install.html#distro and pass PREFIX=/usr instead.
 */
 #ifndef LUA_MULTILIB
 #define LUA_MULTILIB        "lib"
 #endif
 #ifndef LUA_LMULTILIB
 #define LUA_LMULTILIB        "lib"
 #endif
 #define LUA_LROOT        "/usr/local"
 #define LUA_LUADIR        "/lua/5.1/"
 #define LUA_LJDIR        "/luajit-2.1.0-alpha/"

 #ifdef LUA_ROOT
 #define LUA_JROOT        LUA_ROOT
 #define LUA_RLDIR        LUA_ROOT "/share" LUA_LUADIR
 #define LUA_RCDIR        LUA_ROOT "/" LUA_MULTILIB LUA_LUADIR
 #define LUA_RLPATH        ";" LUA_RLDIR "?.lua;" LUA_RLDIR "?/init.lua"
 #define LUA_RCPATH        ";" LUA_RCDIR "?.so"
 #else
 #define LUA_JROOT        LUA_LROOT
 #define LUA_RLPATH
 #define LUA_RCPATH
 #endif

 #define LUA_JPATH        ";" LUA_JROOT "/share" LUA_LJDIR "?.lua"
 #define LUA_LLDIR        LUA_LROOT "/share" LUA_LUADIR
 #define LUA_LCDIR        LUA_LROOT "/" LUA_LMULTILIB LUA_LUADIR
 #define LUA_LLPATH        ";" LUA_LLDIR "?.lua;" LUA_LLDIR "?/init.lua"
 #define LUA_LCPATH1        ";" LUA_LCDIR "?.so"
 #define LUA_LCPATH2        ";" LUA_LCDIR "loadall.so"
```

```
60
61 #define LUA_PATH_DEFAULT        "./?.lua" LUA_JPATH LUA_LLPATH LUA_RLPATH
62 #define LUA_CPATH_DEFAULT        "./?.so" LUA_LCPATH1 LUA_RCPATH LUA_LCPATH2
63 #endif
64
65 /* Environment variable names for path overrides and initialization code. */
66 #define LUA_PATH        "LUA_PATH"
67 #define LUA_CPATH        "LUA_CPATH"
68 #define LUA_INIT        "LUA_INIT"
69
70 /* Special file system characters. */
71 #if defined(_WIN32)
72 #define LUA_DIRSEP        "\\"
73 #else
74 #define LUA_DIRSEP        "/"
75 #endif
76 #define LUA_PATHSEP        ";"
77 #define LUA_PATH_MARK        "?"
78 #define LUA_EXECDIR        "!"
79 #define LUA_IGMARK        "-"
80 #define LUA_PATH_CONFIG \
81   LUA_DIRSEP "\n" LUA_PATHSEP "\n" LUA_PATH_MARK "\n" \
82   LUA_EXECDIR "\n" LUA_IGMARK
83
84 /* Quoting in error messages. */
85 #define LUA_QL(x)        "'" x "'"
86 #define LUA_QS                LUA_QL("%s")
87
88 /* Various tunables. */
89 #define LUAI_MAXSTACK        65500        /* Max. # of stack slots for a thread (<64K). */
90 #define LUAI_MAXCSTACK        8000        /* Max. # of stack slots for a C func (<10K). */
91 #define LUAI_GCPAUSE        200        /* Pause GC until memory is at 200%. */
92 #define LUAI_GCMUL        200        /* Run GC at 200% of allocation speed. */
93 #define LUA_MAXCAPTURES        32        /* Max. pattern captures. */
94
95 /* Compatibility with older library function names. */
96 #define LUA_COMPAT_MOD                /* OLD: math.mod, NEW: math.fmod */
97 #define LUA_COMPAT_GFIND        /* OLD: string.gfind, NEW: string.gmatch */
98
99 /* Configuration for the frontend (the luajit executable). */
100 #if defined(luajit_c)
101 #define LUA_PROGNAME        "luajit"  /* Fallback frontend name. */
102 #define LUA_PROMPT        "> "        /* Interactive prompt. */
103 #define LUA_PROMPT2        ">> "        /* Continuation prompt. */
104 #define LUA_MAXINPUT        512        /* Max. input line length. */
105 #endif
106
107 /* Note: changing the following defines breaks the Lua 5.1 ABI. */
108 #define LUA_INTEGER        ptrdiff_t
109 #define LUA_IDSIZE        60        /* Size of lua_Debug.short_src. */
110 /*
111 ** Size of lauxlib and io.* on-stack buffers. Weird workaround to avoid using
112 ** unreasonable amounts of stack space, but still retain ABI compatibility.
113 ** Blame Lua for depending on BUFSIZ in the ABI, blame **** for wrecking it.
114 */
115 #define LUAL_BUFFERSIZE        (BUFSIZ > 16384 ? 8192 : BUFSIZ)
116
117 /* The following defines are here only for compatibility with luaconf.h
118 ** from the standard Lua distribution. They must not be changed for LuaJIT.
119 */
120 #define LUA_NUMBER_DOUBLE
121 #define LUA_NUMBER                double
122 #define LUAI_UACNUMBER                double
123 #define LUA_NUMBER_SCAN                "%lf"
124 #define LUA_NUMBER_FMT                "%.14g"
125 #define lua_number2str(s, n)        sprintf((s), LUA_NUMBER_FMT, (n))
126 #define LUAI_MAXNUMBER2STR        32
127 #define LUA_INTFRMLEN                "l"
128 #define LUA_INTFRM_T                long
129
130 /* Linkage of public API functions. */
131 #if defined(LUA_BUILD_AS_DLL)
132 #if defined(LUA_CORE) || defined(LUA_LIB)
133 #define LUA_API                __declspec(dllexport)
134 #else
135 #define LUA_API                __declspec(dllimport)
```

```
#endif
#else
#define LUA_API                extern
#endif

#define LUALIB_API        LUA_API

/* Support for internal assertions. */
#if defined(LUA_USE_ASSERT) || defined(LUA_USE_APICHECK)
#include <assert.h>
#endif
#ifdef LUA_USE_ASSERT
#define lua_assert(x)                assert(x)
#endif
#ifdef LUA_USE_APICHECK
#define luai_apicheck(L, o)        { (void)L; assert(o); }
#else
#define luai_apicheck(L, o)        { (void)L; }
#endif

#endif
```

One Level Up Top Level

src/luajit.c - luajit-2.0-src

Global variables defined

- globalL
- progname
- smain

Data types defined

- Smain

Functions defined

- collectargs
- dobytecode
- docall
- dofile
- dojitcmd
- dojitopt
- dolibrary
- dostring
- dotty
- getargs
- handle_luainit
- handle_script
- incomplete
- l_message
- laction
- loadjitmodule
- loadline
- lstop
- main
- pmain
- print_jit_status
- print_usage
- print_version

- pushline
- report
- runargs
- runcmdopt
- traceback
- write_prompt

Macros defined

- FLAGS_EXEC
- FLAGS_INTERACTIVE
- FLAGS_NOENV
- FLAGS_OPTION
- FLAGS_VERSION
- lua_stdin_is_tty
- lua_stdin_is_tty
- lua_stdin_is_tty
- lua_stdin_is_tty
- luajit_c
- notail

Source code

```
/*
** LuaJIT frontend. Runs commands, scripts, read-eval-print (REPL) etc.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Major portions taken verbatim or adapted from the Lua interpreter.
** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
*/

#include <stdio.h>
 #include <stdlib.h>
 #include <string.h>

 #define luajit_c

 #include "lua.h"
 #include "lauxlib.h"
 #include "lualib.h"
 #include "luajit.h"

 #include "lj_arch.h"

 #if LJ_TARGET_POSIX
 #include <unistd.h>
 #define lua_stdin_is_tty()        isatty(0)
 #elif LJ_TARGET_WINDOWS
 #include <io.h>
 #ifdef __BORLANDC__
 #define lua_stdin_is_tty()        isatty(_fileno(stdin))
 #else
 #define lua_stdin_is_tty()        _isatty(_fileno(stdin))
```

```
#endif
#else
#define lua_stdin_is_tty()        1
#endif

#if !LJ_TARGET_CONSOLE
#include <signal.h>
#endif

static lua_State *globalL = NULL;
static const char *progname = LUA_PROGNAME;

#if !LJ_TARGET_CONSOLE
static void lstop(lua_State *L, lua_Debug *ar)
{
  (void)ar;  /* unused arg. */
  lua_sethook(L, NULL, 0, 0);
  /* Avoid luaL_error -- a C hook doesn't add an extra frame. */
  luaL_where(L, 0);
  lua_pushfstring(L, "%sinterrupted!", lua_tostring(L, -1));
  lua_error(L);
}

static void laction(int i)
{
  signal(i, SIG_DFL); /* if another SIGINT happens before lstop,
                         terminate process (default action) */
  lua_sethook(globalL, lstop, LUA_MASKCALL | LUA_MASKRET | LUA_MASKCOUNT, 1);
}
#endif

static void print_usage(void)
{
  fputs("usage: ", stderr);
  fputs(progname, stderr);
  fputs(" [options]... [script [args]...].\n"
  "Available options are:\n"
  "  -e chunk  Execute string " LUA_QL("chunk") ".\n"
  "  -l name   Require library " LUA_QL("name") ".\n"
  "  -b ...    Save or list bytecode.\n"
  "  -j cmd    Perform LuaJIT control command.\n"
  "  -O[opt]   Control LuaJIT optimizations.\n"
  "  -i        Enter interactive mode after executing " LUA_QL("script") ".\n"
  "  -v        Show version information.\n"
  "  -E        Ignore environment variables.\n"
  "  --        Stop handling options.\n"
  "  -         Execute stdin and stop handling options.\n", stderr);
  fflush(stderr);
}

static void l_message(const char *pname, const char *msg)
{
  if (pname) { fputs(pname, stderr); fputc(':', stderr); fputc(' ', stderr); }
  fputs(msg, stderr); fputc('\n', stderr);
  fflush(stderr);
}

static int report(lua_State *L, int status)
{
  if (status && !lua_isnil(L, -1)) {
    const char *msg = lua_tostring(L, -1);
    if (msg == NULL) msg = "(error object is not a string)";
    l_message(progname, msg);
    lua_pop(L, 1);
  }
  return status;
}

static int traceback(lua_State *L)
 {
   if (!lua_isstring(L, 1)) { /* Non-string error object? Try metamethod. */
     if (lua_isnoneornil(L, 1) ||
         !luaL_callmeta(L, 1, "__tostring") ||
         !lua_isstring(L, -1))
       return 1;  /* Return non-string error object. */
     lua_remove(L, 1);  /* Replace object by result of __tostring metamethod. */
```

```
  }
  luaL_traceback(L, L, lua_tostring(L, 1), 1);
  return 1;
}

static int docall(lua_State *L, int narg, int clear)
{
  int status;
  int base = lua_gettop(L) - narg;  /* function index */
  lua_pushcfunction(L, traceback);  /* push traceback function */
  lua_insert(L, base);  /* put it under chunk and args */
#if !LJ_TARGET_CONSOLE
  signal(SIGINT, laction);
#endif
  status = lua_pcall(L, narg, (clear ? 0 : LUA_MULTRET), base);
#if !LJ_TARGET_CONSOLE
  signal(SIGINT, SIG_DFL);
#endif
  lua_remove(L, base);  /* remove traceback function */
  /* force a complete garbage collection in case of errors */
  if (status != 0) lua_gc(L, LUA_GCCOLLECT, 0);
  return status;
}

static void print_version(void)
{
  fputs(LUAJIT_VERSION " -- " LUAJIT_COPYRIGHT ". " LUAJIT_URL "\n", stdout);
}

static void print_jit_status(lua_State *L)
{
  int n;
  const char *s;
  lua_getfield(L, LUA_REGISTRYINDEX, "_LOADED");
  lua_getfield(L, -1, "jit");  /* Get jit.* module table. */
  lua_remove(L, -2);
  lua_getfield(L, -1, "status");
  lua_remove(L, -2);
  n = lua_gettop(L);
  lua_call(L, 0, LUA_MULTRET);
  fputs(lua_toboolean(L, n) ? "JIT: ON" : "JIT: OFF", stdout);
  for (n++; (s = lua_tostring(L, n)); n++) {
    putc(' ', stdout);
    fputs(s, stdout);
  }
  putc('\n', stdout);
}

static int getargs(lua_State *L, char **argv, int n)
{
  int narg;
  int i;
  int argc = 0;
  while (argv[argc]) argc++;  /* count total number of arguments */
  narg = argc - (n + 1);  /* number of arguments to the script */
  luaL_checkstack(L, narg + 3, "too many arguments to script");
  for (i = n+1; i < argc; i++)
    lua_pushstring(L, argv[i]);
  lua_createtable(L, narg, n + 1);
  for (i = 0; i < argc; i++) {
    lua_pushstring(L, argv[i]);
    lua_rawseti(L, -2, i - n);
  }
  return narg;
}

static int dofile(lua_State *L, const char *name)
{
  int status = luaL_loadfile(L, name) || docall(L, 0, 1);
  return report(L, status);
}

static int dostring(lua_State *L, const char *s, const char *name)
{
  int status = luaL_loadbuffer(L, s, strlen(s), name) || docall(L, 0, 1);
  return report(L, status);
```

```
}

static int dolibrary(lua_State *L, const char *name)
{
  lua_getglobal(L, "require");
  lua_pushstring(L, name);
  return report(L, docall(L, 1, 1));
}

static void write_prompt(lua_State *L, int firstline)
{
  const char *p;
  lua_getfield(L, LUA_GLOBALSINDEX, firstline ? "_PROMPT" : "_PROMPT2");
  p = lua_tostring(L, -1);
  if (p == NULL) p = firstline ? LUA_PROMPT : LUA_PROMPT2;
  fputs(p, stdout);
  fflush(stdout);
  lua_pop(L, 1);  /* remove global */
}

static int incomplete(lua_State *L, int status)
{
  if (status == LUA_ERRSYNTAX) {
    size_t lmsg;
    const char *msg = lua_tolstring(L, -1, &lmsg);
    const char *tp = msg + lmsg - (sizeof(LUA_QL("<eof>")) - 1);
    if (strstr(msg, LUA_QL("<eof>")) == tp) {
      lua_pop(L, 1);
      return 1;
    }
  }
  return 0;  /* else... */
}

static int pushline(lua_State *L, int firstline)
{
  char buf[LUA_MAXINPUT];
  write_prompt(L, firstline);
  if (fgets(buf, LUA_MAXINPUT, stdin)) {
    size_t len = strlen(buf);
    if (len > 0 && buf[len-1] == '\n')
      buf[len-1] = '\0';
    if (firstline && buf[0] == '=')
      lua_pushfstring(L, "return %s", buf+1);
    else
      lua_pushstring(L, buf);
    return 1;
  }
  return 0;
}

static int loadline(lua_State *L)
{
  int status;
  lua_settop(L, 0);
  if (!pushline(L, 1))
    return -1;  /* no input */
  for (;;) {  /* repeat until gets a complete line */
    status = luaL_loadbuffer(L, lua_tostring(L, 1), lua_strlen(L, 1), "=stdin");
    if (!incomplete(L, status)) break;  /* cannot try to add lines? */
    if (!pushline(L, 0))  /* no more input? */
      return -1;
    lua_pushliteral(L, "\n");  /* add a new line... */
    lua_insert(L, -2);  /* ...between the two lines */
    lua_concat(L, 3);  /* join them */
  }
  lua_remove(L, 1);  /* remove line */
  return status;
}

static void dotty(lua_State *L)
{
  int status;
  const char *oldprogname = progname;
  progname = NULL;
  while ((status = loadline(L)) != -1) {
```

```
    if (status == 0) status = docall(L, 0, 0);
    report(L, status);
    if (status == 0 && lua_gettop(L) > 0) {  /* any result to print? */
      lua_getglobal(L, "print");
      lua_insert(L, 1);
      if (lua_pcall(L, lua_gettop(L)-1, 0, 0) != 0)
        l_message(progname,
          lua_pushfstring(L, "error calling " LUA_QL("print") " (%s)",
                             lua_tostring(L, -1)));
    }
  }
  lua_settop(L, 0);  /* clear stack */
  fputs("\n", stdout);
  fflush(stdout);
  progname = oldprogname;
}

static int handle_script(lua_State *L, char **argv, int n)
{
  int status;
  const char *fname;
  int narg = getargs(L, argv, n);  /* collect arguments */
  lua_setglobal(L, "arg");
  fname = argv[n];
  if (strcmp(fname, "-") == 0 && strcmp(argv[n-1], "--") != 0)
    fname = NULL;  /* stdin */
  status = luaL_loadfile(L, fname);
  lua_insert(L, -(narg+1));
  if (status == 0)
    status = docall(L, narg, 0);
  else
    lua_pop(L, narg);
  return report(L, status);
}

/* Load add-on module. */
static int loadjitmodule(lua_State *L)
{
  lua_getglobal(L, "require");
  lua_pushliteral(L, "jit.");
  lua_pushvalue(L, -3);
  lua_concat(L, 2);
  if (lua_pcall(L, 1, 1, 0)) {
    const char *msg = lua_tostring(L, -1);
    if (msg && !strncmp(msg, "module ", 7))
      goto nomodule;
    return report(L, 1);
  }
  lua_getfield(L, -1, "start");
  if (lua_isnil(L, -1)) {
  nomodule:
    l_message(progname,
              "unknown luaJIT command or jit.* modules not installed");
    return 1;
  }
  lua_remove(L, -2);  /* Drop module table. */
  return 0;
}

/* Run command with options. */
static int runcmdopt(lua_State *L, const char *opt)
{
  int narg = 0;
  if (opt && *opt) {
    for (;;) {  /* Split arguments. */
      const char *p = strchr(opt, ',');
      narg++;
      if (!p) break;
      if (p == opt)
        lua_pushnil(L);
      else
        lua_pushlstring(L, opt, (size_t)(p - opt));
      opt = p + 1;
    }
    if (*opt)
      lua_pushstring(L, opt);
```

```
    else
      lua_pushnil(L);
  }
  return report(L, lua_pcall(L, narg, 0, 0));
}

/* JIT engine control command: try jit library first or load add-on module. */
static int dojitcmd(lua_State *L, const char *cmd)
{
  const char *opt = strchr(cmd, '=');
  lua_pushlstring(L, cmd, opt ? (size_t)(opt - cmd) : strlen(cmd));
  lua_getfield(L, LUA_REGISTRYINDEX, "_LOADED");
  lua_getfield(L, -1, "jit");  /* Get jit.* module table. */
  lua_remove(L, -2);
  lua_pushvalue(L, -2);
  lua_gettable(L, -2);  /* Lookup library function. */
  if (!lua_isfunction(L, -1)) {
    lua_pop(L, 2);  /* Drop non-function and jit.* table, keep module name. */
    if (loadjitmodule(L))
      return 1;
  } else {
    lua_remove(L, -2);  /* Drop jit.* table. */
  }
  lua_remove(L, -2);  /* Drop module name. */
  return runcmdopt(L, opt ? opt+1 : opt);
}

/* Optimization flags. */
static int dojitopt(lua_State *L, const char *opt)
{
  lua_getfield(L, LUA_REGISTRYINDEX, "_LOADED");
  lua_getfield(L, -1, "jit.opt");  /* Get jit.opt.* module table. */
  lua_remove(L, -2);
  lua_getfield(L, -1, "start");
  lua_remove(L, -2);
  return runcmdopt(L, opt);
}

/* Save or list bytecode. */
static int dobytecode(lua_State *L, char **argv)
{
  int narg = 0;
  lua_pushliteral(L, "bcsave");
  if (loadjitmodule(L))
    return 1;
  if (argv[0][2]) {
    narg++;
    argv[0][1] = '-';
    lua_pushstring(L, argv[0]+1);
  }
  for (argv++; *argv != NULL; narg++, argv++)
    lua_pushstring(L, *argv);
  return report(L, lua_pcall(L, narg, 0, 0));
}

/* check that argument has no extra characters at the end */
#define notail(x)        {if ((x)[2] != '\0') return -1;}

#define FLAGS_INTERACTIVE        1
#define FLAGS_VERSION            2
#define FLAGS_EXEC               4
#define FLAGS_OPTION             8
#define FLAGS_NOENV              16

static int collectargs(char **argv, int *flags)
{
  int i;
  for (i = 1; argv[i] != NULL; i++) {
    if (argv[i][0] != '-')  /* Not an option? */
      return i;
    switch (argv[i][1]) {  /* Check option. */
    case '-':
      notail(argv[i]);
      return (argv[i+1] != NULL ? i+1 : 0);
    case '\0':
      return i;
```

```
    case 'i':
      notail(argv[i]);
      *flags |= FLAGS_INTERACTIVE;
      /* fallthrough */
    case 'v':
      notail(argv[i]);
      *flags |= FLAGS_VERSION;
      break;
    case 'e':
      *flags |= FLAGS_EXEC;
    case 'j':  /* LuaJIT extension */
    case 'l':
      *flags |= FLAGS_OPTION;
      if (argv[i][2] == '\0') {
        i++;
        if (argv[i] == NULL) return -1;
      }
      break;
    case 'O': break;  /* LuaJIT extension */
    case 'b':  /* LuaJIT extension */
      if (*flags) return -1;
      *flags |= FLAGS_EXEC;
      return 0;
    case 'E':
      *flags |= FLAGS_NOENV;
      break;
    default: return -1;  /* invalid option */
    }
  }
  return 0;
}

static int runargs(lua_State *L, char **argv, int n)
{
  int i;
  for (i = 1; i < n; i++) {
    if (argv[i] == NULL) continue;
    lua_assert(argv[i][0] == '-');
    switch (argv[i][1]) {  /* option */
    case 'e': {
      const char *chunk = argv[i] + 2;
      if (*chunk == '\0') chunk = argv[++i];
      lua_assert(chunk != NULL);
      if (dostring(L, chunk, "=(command line)") != 0)
        return 1;
      break;
      }
    case 'l': {
      const char *filename = argv[i] + 2;
      if (*filename == '\0') filename = argv[++i];
      lua_assert(filename != NULL);
      if (dolibrary(L, filename))
        return 1;  /* stop if file fails */
      break;
      }
    case 'j': {  /* LuaJIT extension */
      const char *cmd = argv[i] + 2;
      if (*cmd == '\0') cmd = argv[++i];
      lua_assert(cmd != NULL);
      if (dojitcmd(L, cmd))
        return 1;
      break;
      }
    case 'O':  /* LuaJIT extension */
      if (dojitopt(L, argv[i] + 2))
        return 1;
      break;
    case 'b':  /* LuaJIT extension */
      return dobytecode(L, argv+i);
    default: break;
    }
  }
  return 0;
}

static int handle_luainit(lua_State *L)
```

```
{
#if LJ_TARGET_CONSOLE
  const char *init = NULL;
#else
  const char *init = getenv(LUA_INIT);
#endif
  if (init == NULL)
    return 0;  /* status OK */
  else if (init[0] == '@')
    return dofile(L, init+1);
  else
    return dostring(L, init, "=" LUA_INIT);
}

static struct Smain {
  char **argv;
  int argc;
  int status;
} smain;

static int pmain(lua_State *L)
{
  struct Smain *s = &smain;
  char **argv = s->argv;
  int script;
  int flags = 0;
  globalL = L;
  if (argv[0] && argv[0][0]) progname = argv[0];
  LUAJIT_VERSION_SYM();  /* linker-enforced version check */
  script = collectargs(argv, &flags);
  if (script < 0) {  /* invalid args? */
    print_usage();
    s->status = 1;
    return 0;
  }
  if ((flags & FLAGS_NOENV)) {
    lua_pushboolean(L, 1);
    lua_setfield(L, LUA_REGISTRYINDEX, "LUA_NOENV");
  }
  lua_gc(L, LUA_GCSTOP, 0);  /* stop collector during initialization */
  luaL_openlibs(L);  /* open libraries */
  lua_gc(L, LUA_GCRESTART, -1);
  if (!(flags & FLAGS_NOENV)) {
    s->status = handle_luainit(L);
    if (s->status != 0) return 0;
  }
  if ((flags & FLAGS_VERSION)) print_version();
  s->status = runargs(L, argv, (script > 0) ? script : s->argc);
  if (s->status != 0) return 0;
  if (script) {
    s->status = handle_script(L, argv, script);
    if (s->status != 0) return 0;
  }
  if ((flags & FLAGS_INTERACTIVE)) {
    print_jit_status(L);
    dotty(L);
  } else if (script == 0 && !(flags & (FLAGS_EXEC|FLAGS_VERSION))) {
    if (lua_stdin_is_tty()) {
      print_version();
      print_jit_status(L);
      dotty(L);
    } else {
      dofile(L, NULL);  /* executes stdin as a file */
    }
  }
  return 0;
}

int main(int argc, char **argv)
{
  int status;
  lua_State *L = lua_open();  /* create state */
  if (L == NULL) {
    l_message(argv[0], "cannot create state: not enough memory");
    return EXIT_FAILURE;
  }
```

```
563   smain.argc = argc;
564   smain.argv = argv;
565   status = lua_cpcall(L, pmain, NULL);
566   report(L, status);
567   lua_close(L);
568   return (status || smain.status) ? EXIT_FAILURE : EXIT_SUCCESS;
569 }
570
```

One Level Up Top Level

src/lj_dispatch.c - luajit-2.0-src

Global variables defined

- dispatch_got

Functions defined

- LUA_API void LUAJIT_VERSION_SYM(void)
- call_init
- callhook
- cur_topslot
- lj_dispatch_call
- lj_dispatch_init
- lj_dispatch_init_hotcount
- lj_dispatch_ins
- lj_dispatch_profile
- lj_dispatch_stitch
- lj_dispatch_update
- luaJIT_setmode
- lua_gethook
- lua_gethookcount
- lua_gethookmask
- lua_sethook
- setptmode
- setptmode_all

Macros defined

- DISPMODE_CALL
- DISPMODE_INS
- DISPMODE_JIT
- DISPMODE_PROF
- DISPMODE_REC
- DISPMODE_RET
- GOTFUNC
- GOTFUNC

- LUA_CORE
- lj_dispatch_c
- lj_dispatch_profile
- lj_dispatch_stitch

Source code

```
/*
** Instruction dispatch handling.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_dispatch_c
#define LUA_CORE

#include "lj_obj.h"
 #include "lj_err.h"
 #include "lj_buf.h"
 #include "lj_func.h"
 #include "lj_str.h"
 #include "lj_tab.h"
 #include "lj_meta.h"
 #include "lj_debug.h"
 #include "lj_state.h"
 #include "lj_frame.h"
 #include "lj_bc.h"
 #include "lj_ff.h"
 #include "lj_strfmt.h"
 #if LJ_HASJIT
 #include "lj_jit.h"
 #endif
 #if LJ_HASFFI
 #include "lj_ccallback.h"
 #endif
 #include "lj_trace.h"
 #include "lj_dispatch.h"
 #if LJ_HASPROFILE
 #include "lj_profile.h"
 #endif
 #include "lj_vm.h"
 #include "luajit.h"

 /* Bump GG_NUM_ASMFF in lj_dispatch.h as needed. Ugly. */
 LJ_STATIC_ASSERT(GG_NUM_ASMFF == FF_NUM_ASMFUNC);

 /* -- Dispatch table management ------------------------------------------ */

 #if LJ_TARGET_MIPS
 #include <math.h>
 LJ_FUNCA_NORET void LJ_FASTCALL lj_ffh_coroutine_wrap_err(lua_State *L,
                                                           lua_State *co);
 #if !LJ_HASJIT
 #define lj_dispatch_stitch        lj_dispatch_ins
 #endif
 #if !LJ_HASPROFILE
 #define lj_dispatch_profile        lj_dispatch_ins
 #endif

 #define GOTFUNC(name)        (ASMFunction)name,
 static const ASMFunction dispatch_got[] = {
   GOTDEF(GOTFUNC)
 };
 #undef GOTFUNC
 #endif

 /* Initialize instruction dispatch table and hot counters. */
 void lj_dispatch_init(GG_State *GG)
 {
   uint32_t i;
   ASMFunction *disp = GG->dispatch;
```

```
  for (i = 0; i < GG_LEN_SDISP; i++)
    disp[GG_LEN_DDISP+i] = disp[i] = makeasmfunc(lj_bc_ofs[i]);
  for (i = GG_LEN_SDISP; i < GG_LEN_DDISP; i++)
    disp[i] = makeasmfunc(lj_bc_ofs[i]);
  /* The JIT engine is off by default. luaopen_jit() turns it on. */
  disp[BC_FORL] = disp[BC_IFORL];
  disp[BC_ITERL] = disp[BC_IITERL];
  disp[BC_LOOP] = disp[BC_ILOOP];
  disp[BC_FUNCF] = disp[BC_IFUNCF];
  disp[BC_FUNCV] = disp[BC_IFUNCV];
  GG->g.bc_cfunc_ext = GG->g.bc_cfunc_int = BCINS_AD(BC_FUNCC, LUA_MINSTACK, 0);
  for (i = 0; i < GG_NUM_ASMFF; i++)
    GG->bcff[i] = BCINS_AD(BC__MAX+i, 0, 0);
#if LJ_TARGET_MIPS
  memcpy(GG->got, dispatch_got, LJ_GOT__MAX*4);
#endif
}

#if LJ_HASJIT
/* Initialize hotcount table. */
void lj_dispatch_init_hotcount(global_State *g)
{
  int32_t hotloop = G2J(g)->param[JIT_P_hotloop];
  HotCount start = (HotCount)(hotloop*HOTCOUNT_LOOP - 1);
  HotCount *hotcount = G2GG(g)->hotcount;
  uint32_t i;
  for (i = 0; i < HOTCOUNT_SIZE; i++)
    hotcount[i] = start;
}
#endif

/* Internal dispatch mode bits. */
#define DISPMODE_CALL        0x01        /* Override call dispatch. */
#define DISPMODE_RET         0x02        /* Override return dispatch. */
#define DISPMODE_INS         0x04        /* Override instruction dispatch. */
#define DISPMODE_JIT         0x10        /* JIT compiler on. */
#define DISPMODE_REC         0x20        /* Recording active. */
#define DISPMODE_PROF        0x40        /* Profiling active. */

/* Update dispatch table depending on various flags. */
void lj_dispatch_update(global_State *g)
{
  uint8_t oldmode = g->dispatchmode;
  uint8_t mode = 0;
#if LJ_HASJIT
  mode |= (G2J(g)->flags & JIT_F_ON) ? DISPMODE_JIT : 0;
  mode |= G2J(g)->state != LJ_TRACE_IDLE ?
            (DISPMODE_REC|DISPMODE_INS|DISPMODE_CALL) : 0;
#endif
#if LJ_HASPROFILE
  mode |= (g->hookmask & HOOK_PROFILE) ? (DISPMODE_PROF|DISPMODE_INS) : 0;
#endif
  mode |= (g->hookmask & (LUA_MASKLINE|LUA_MASKCOUNT)) ? DISPMODE_INS : 0;
  mode |= (g->hookmask & LUA_MASKCALL) ? DISPMODE_CALL : 0;
  mode |= (g->hookmask & LUA_MASKRET) ? DISPMODE_RET : 0;
  if (oldmode != mode) {  /* Mode changed? */
    ASMFunction *disp = G2GG(g)->dispatch;
    ASMFunction f_forl, f_iterl, f_loop, f_funcf, f_funcv;
    g->dispatchmode = mode;

    /* Hotcount if JIT is on, but not while recording. */
    if ((mode & (DISPMODE_JIT|DISPMODE_REC)) == DISPMODE_JIT) {
      f_forl = makeasmfunc(lj_bc_ofs[BC_FORL]);
      f_iterl = makeasmfunc(lj_bc_ofs[BC_ITERL]);
      f_loop = makeasmfunc(lj_bc_ofs[BC_LOOP]);
      f_funcf = makeasmfunc(lj_bc_ofs[BC_FUNCF]);
      f_funcv = makeasmfunc(lj_bc_ofs[BC_FUNCV]);
    } else {  /* Otherwise use the non-hotcounting instructions. */
      f_forl = disp[GG_LEN_DDISP+BC_IFORL];
      f_iterl = disp[GG_LEN_DDISP+BC_IITERL];
      f_loop = disp[GG_LEN_DDISP+BC_ILOOP];
      f_funcf = makeasmfunc(lj_bc_ofs[BC_IFUNCF]);
      f_funcv = makeasmfunc(lj_bc_ofs[BC_IFUNCV]);
    }
    /* Init static counting instruction dispatch first (may be copied below). */
    disp[GG_LEN_DDISP+BC_FORL] = f_forl;
```

```
    disp[GG_LEN_DDISP+BC_ITERL] = f_iterl;
    disp[GG_LEN_DDISP+BC_LOOP] = f_loop;

    /* Set dynamic instruction dispatch. */
    if ((oldmode ^ mode) & (DISPMODE_PROF|DISPMODE_REC|DISPMODE_INS)) {
      /* Need to update the whole table. */
      if (!(mode & DISPMODE_INS)) {  /* No ins dispatch? */
        /* Copy static dispatch table to dynamic dispatch table. */
        memcpy(&disp[0], &disp[GG_LEN_DDISP], GG_LEN_SDISP*sizeof(ASMFunction));
        /* Overwrite with dynamic return dispatch. */
        if ((mode & DISPMODE_RET)) {
          disp[BC_RETM] = lj_vm_rethook;
          disp[BC_RET] = lj_vm_rethook;
          disp[BC_RET0] = lj_vm_rethook;
          disp[BC_RET1] = lj_vm_rethook;
        }
      } else {
        /* The recording dispatch also checks for hooks. */
        ASMFunction f = (mode & DISPMODE_PROF) ? lj_vm_profhook :
                        (mode & DISPMODE_REC) ? lj_vm_record : lj_vm_inshook;
        uint32_t i;
        for (i = 0; i < GG_LEN_SDISP; i++)
          disp[i] = f;
      }
    } else if (!(mode & DISPMODE_INS)) {
      /* Otherwise set dynamic counting ins. */
      disp[BC_FORL] = f_forl;
      disp[BC_ITERL] = f_iterl;
      disp[BC_LOOP] = f_loop;
      /* Set dynamic return dispatch. */
      if ((mode & DISPMODE_RET)) {
        disp[BC_RETM] = lj_vm_rethook;
        disp[BC_RET] = lj_vm_rethook;
        disp[BC_RET0] = lj_vm_rethook;
        disp[BC_RET1] = lj_vm_rethook;
      } else {
        disp[BC_RETM] = disp[GG_LEN_DDISP+BC_RETM];
        disp[BC_RET] = disp[GG_LEN_DDISP+BC_RET];
        disp[BC_RET0] = disp[GG_LEN_DDISP+BC_RET0];
        disp[BC_RET1] = disp[GG_LEN_DDISP+BC_RET1];
      }
    }

    /* Set dynamic call dispatch. */
    if ((oldmode ^ mode) & DISPMODE_CALL) {  /* Update the whole table? */
      uint32_t i;
      if ((mode & DISPMODE_CALL) == 0) {  /* No call hooks? */
        for (i = GG_LEN_SDISP; i < GG_LEN_DDISP; i++)
          disp[i] = makeasmfunc(lj_bc_ofs[i]);
      } else {
        for (i = GG_LEN_SDISP; i < GG_LEN_DDISP; i++)
          disp[i] = lj_vm_callhook;
      }
    }
    if (!(mode & DISPMODE_CALL)) {  /* Overwrite dynamic counting ins. */
      disp[BC_FUNCF] = f_funcf;
      disp[BC_FUNCV] = f_funcv;
    }

#if LJ_HASJIT
    /* Reset hotcounts for JIT off to on transition. */
    if ((mode & DISPMODE_JIT) && !(oldmode & DISPMODE_JIT))
      lj_dispatch_init_hotcount(g);
#endif
  }
}

/* -- JIT mode setting ---------------------------------------------------- */

#if LJ_HASJIT
/* Set JIT mode for a single prototype. */
static void setptmode(global_State *g, GCproto *pt, int mode)
{
  if ((mode & LUAJIT_MODE_ON)) {  /* (Re-)enable JIT compilation. */
    pt->flags &= ~PROTO_NOJIT;
    lj_trace_reenableproto(pt);  /* Unpatch all ILOOP etc. bytecodes. */
```

```
  } else {  /* Flush and/or disable JIT compilation. */
    if (!(mode & LUAJIT_MODE_FLUSH))
      pt->flags |= PROTO_NOJIT;
    lj_trace_flushproto(g, pt);  /* Flush all traces of prototype. */
  }
}

/* Recursively set the JIT mode for all children of a prototype. */
static void setptmode_all(global_State *g, GCproto *pt, int mode)
{
  ptrdiff_t i;
  if (!(pt->flags & PROTO_CHILD)) return;
  for (i = -(ptrdiff_t)pt->sizekgc; i < 0; i++) {
    GCobj *o = proto_kgc(pt, i);
    if (o->gch.gct == ~LJ_TPROTO) {
      setptmode(g, gco2pt(o), mode);
      setptmode_all(g, gco2pt(o), mode);
    }
  }
}
#endif

/* Public API function: control the JIT engine. */
int luaJIT_setmode(lua_State *L, int idx, int mode)
{
  global_State *g = G(L);
  int mm = mode & LUAJIT_MODE_MASK;
  lj_trace_abort(g);  /* Abort recording on any state change. */
  /* Avoid pulling the rug from under our own feet. */
  if ((g->hookmask & HOOK_GC))
    lj_err_caller(L, LJ_ERR_NOGCMM);
  switch (mm) {
#if LJ_HASJIT
  case LUAJIT_MODE_ENGINE:
    if ((mode & LUAJIT_MODE_FLUSH)) {
      lj_trace_flushall(L);
    } else {
      if (!(mode & LUAJIT_MODE_ON))
        G2J(g)->flags &= ~(uint32_t)JIT_F_ON;
#if LJ_TARGET_X86ORX64
      else if ((G2J(g)->flags & JIT_F_SSE2))
        G2J(g)->flags |= (uint32_t)JIT_F_ON;
      else
        return 0;  /* Don't turn on JIT compiler without SSE2 support. */
#else
      else
        G2J(g)->flags |= (uint32_t)JIT_F_ON;
#endif
      lj_dispatch_update(g);
    }
    break;
  case LUAJIT_MODE_FUNC:
  case LUAJIT_MODE_ALLFUNC:
  case LUAJIT_MODE_ALLSUBFUNC: {
    cTValue *tv = idx == 0 ? frame_prev(L->base-1) :
                  idx > 0 ? L->base + (idx-1) : L->top + idx;
    GCproto *pt;
    if ((idx == 0 || tvisfunc(tv)) && isluafunc(&gcval(tv)->fn))
      pt = funcproto(&gcval(tv)->fn);  /* Cannot use funcV() for frame slot. */
    else if (tvisproto(tv))
      pt = protoV(tv);
    else
      return 0;  /* Failed. */
    if (mm != LUAJIT_MODE_ALLSUBFUNC)
      setptmode(g, pt, mode);
    if (mm != LUAJIT_MODE_FUNC)
      setptmode_all(g, pt, mode);
    break;
    }
  case LUAJIT_MODE_TRACE:
    if (!(mode & LUAJIT_MODE_FLUSH))
      return 0;  /* Failed. */
    lj_trace_flush(G2J(g), idx);
    break;
#else
  case LUAJIT_MODE_ENGINE:
```

```
  case LUAJIT_MODE_FUNC:
  case LUAJIT_MODE_ALLFUNC:
  case LUAJIT_MODE_ALLSUBFUNC:
    UNUSED(idx);
    if ((mode & LUAJIT_MODE_ON))
      return 0;  /* Failed. */
    break;
#endif
  case LUAJIT_MODE_WRAPCFUNC:
    if ((mode & LUAJIT_MODE_ON)) {
      if (idx != 0) {
        cTValue *tv = idx > 0 ? L->base + (idx-1) : L->top + idx;
        if (tvislightud(tv))
          g->wrapf = (lua_CFunction)lightudV(tv);
        else
          return 0;  /* Failed. */
      } else {
        return 0;  /* Failed. */
      }
      g->bc_cfunc_ext = BCINS_AD(BC_FUNCCW, 0, 0);
    } else {
      g->bc_cfunc_ext = BCINS_AD(BC_FUNCC, 0, 0);
    }
    break;
  default:
    return 0;  /* Failed. */
  }
  return 1;  /* OK. */
}

/* Enforce (dynamic) linker error for version mismatches. See luajit.c. */
LUA_API void LUAJIT_VERSION_SYM(void)
{
}

/* -- Hooks ---------------------------------------------------------------- */

/* This function can be called asynchronously (e.g. during a signal). */
LUA_API int lua_sethook(lua_State *L, lua_Hook func, int mask, int count)
{
  global_State *g = G(L);
  mask &= HOOK_EVENTMASK;
  if (func == NULL || mask == 0) { mask = 0; func = NULL; }  /* Consistency. */
  g->hookf = func;
  g->hookcount = g->hookcstart = (int32_t)count;
  g->hookmask = (uint8_t)((g->hookmask & ~HOOK_EVENTMASK) | mask);
  lj_trace_abort(g);  /* Abort recording on any hook change. */
  lj_dispatch_update(g);
  return 1;
}

LUA_API lua_Hook lua_gethook(lua_State *L)
{
  return G(L)->hookf;
}

LUA_API int lua_gethookmask(lua_State *L)
{
  return G(L)->hookmask & HOOK_EVENTMASK;
}

LUA_API int lua_gethookcount(lua_State *L)
{
  return (int)G(L)->hookcstart;
}

/* Call a hook. */
static void callhook(lua_State *L, int event, BCLine line)
{
  global_State *g = G(L);
  lua_Hook hookf = g->hookf;
  if (hookf && !hook_active(g)) {
    lua_Debug ar;
    lj_trace_abort(g);  /* Abort recording on any hook call. */
    ar.event = event;
    ar.currentline = line;
```

```
    /* Top frame, nextframe = NULL. */
    ar.i_ci = (int)((L->base-1) - tvref(L->stack));
    lj_state_checkstack(L, 1+LUA_MINSTACK);
#if LJ_HASPROFILE && !LJ_PROFILE_SIGPROF
    lj_profile_hook_enter(g);
#else
    hook_enter(g);
#endif
    hookf(L, &ar);
    lua_assert(hook_active(g));
    setgcref(g->cur_L, obj2gco(L));
#if LJ_HASPROFILE && !LJ_PROFILE_SIGPROF
    lj_profile_hook_leave(g);
#else
    hook_leave(g);
#endif
  }
}

/* -- Dispatch callbacks ---------------------------------------------- */

/* Calculate number of used stack slots in the current frame. */
static BCReg cur_topslot(GCproto *pt, const BCIns *pc, uint32_t nres)
{
  BCIns ins = pc[-1];
  if (bc_op(ins) == BC_UCLO)
    ins = pc[bc_j(ins)];
  switch (bc_op(ins)) {
  case BC_CALLM: case BC_CALLMT: return bc_a(ins) + bc_c(ins) + nres-1+1+LJ_FR2;
  case BC_RETM: return bc_a(ins) + bc_d(ins) + nres-1;
  case BC_TSETM: return bc_a(ins) + nres-1;
  default: return pt->framesize;
  }
}

/* Instruction dispatch. Used by instr/line/return hooks or when recording. */
void LJ_FASTCALL lj_dispatch_ins(lua_State *L, const BCIns *pc)
{
  ERRNO_SAVE
  GCfunc *fn = curr_func(L);
  GCproto *pt = funcproto(fn);
  void *cf = cframe_raw(L->cframe);
  const BCIns *oldpc = cframe_pc(cf);
  global_State *g = G(L);
  BCReg slots;
  setcframe_pc(cf, pc);
  slots = cur_topslot(pt, pc, cframe_multres_n(cf));
  L->top = L->base + slots;  /* Fix top. */
#if LJ_HASJIT
  {
    jit_State *J = G2J(g);
    if (J->state != LJ_TRACE_IDLE) {
#ifdef LUA_USE_ASSERT
      ptrdiff_t delta = L->top - L->base;
#endif
      J->L = L;
      lj_trace_ins(J, pc-1);  /* The interpreter bytecode PC is offset by 1. */
      lua_assert(L->top - L->base == delta);
    }
  }
#endif
  if ((g->hookmask & LUA_MASKCOUNT) && g->hookcount == 0) {
    g->hookcount = g->hookcstart;
    callhook(L, LUA_HOOKCOUNT, -1);
    L->top = L->base + slots;  /* Fix top again. */
  }
  if ((g->hookmask & LUA_MASKLINE)) {
    BCPos npc = proto_bcpos(pt, pc) - 1;
    BCPos opc = proto_bcpos(pt, oldpc) - 1;
    BCLine line = lj_debug_line(pt, npc);
    if (pc <= oldpc || opc >= pt->sizebc || line != lj_debug_line(pt, opc)) {
      callhook(L, LUA_HOOKLINE, line);
      L->top = L->base + slots;  /* Fix top again. */
    }
  }
  if ((g->hookmask & LUA_MASKRET) && bc_isret(bc_op(pc[-1])))
```

```
444     callhook(L, LUA_HOOKRET, -1);
445   ERRNO_RESTORE
446 }
447
448 /* Initialize call. Ensure stack space and return # of missing parameters. */
449 static int call_init(lua_State *L, GCfunc *fn)
450 {
451   if (isluafunc(fn)) {
452     GCproto *pt = funcproto(fn);
453     int numparams = pt->numparams;
454     int gotparams = (int)(L->top - L->base);
455     int need = pt->framesize;
456     if ((pt->flags & PROTO_VARARG)) need += 1+gotparams;
457     lj_state_checkstack(L, (MSize)need);
458     numparams -= gotparams;
459     return numparams >= 0 ? numparams : 0;
460   } else {
461     lj_state_checkstack(L, LUA_MINSTACK);
462     return 0;
463   }
464 }
465
466 /* Call dispatch. Used by call hooks, hot calls or when recording. */
467 ASMFunction LJ_FASTCALL lj_dispatch_call(lua_State *L, const BCIns *pc)
468 {
469   ERRNO_SAVE
470   GCfunc *fn = curr_func(L);
471   BCOp op;
472   global_State *g = G(L);
473 #if LJ_HASJIT
474   jit_State *J = G2J(g);
475 #endif
476   int missing = call_init(L, fn);
477 #if LJ_HASJIT
478   J->L = L;
479   if ((uintptr_t)pc & 1) {  /* Marker for hot call. */
480 #ifdef LUA_USE_ASSERT
481     ptrdiff_t delta = L->top - L->base;
482 #endif
483     pc = (const BCIns *)((uintptr_t)pc & ~(uintptr_t)1);
484     lj_trace_hot(J, pc);
485     lua_assert(L->top - L->base == delta);
486     goto out;
487   } else if (J->state != LJ_TRACE_IDLE &&
488              !(g->hookmask & (HOOK_GC|HOOK_VMEVENT))) {
489 #ifdef LUA_USE_ASSERT
490     ptrdiff_t delta = L->top - L->base;
491 #endif
492     /* Record the FUNC* bytecodes, too. */
493     lj_trace_ins(J, pc-1);  /* The interpreter bytecode PC is offset by 1. */
494     lua_assert(L->top - L->base == delta);
495   }
496 #endif
497   if ((g->hookmask & LUA_MASKCALL)) {
498     int i;
499     for (i = 0; i < missing; i++)  /* Add missing parameters. */
500       setnilV(L->top++);
501     callhook(L, LUA_HOOKCALL, -1);
502     /* Preserve modifications of missing parameters by lua_setlocal(). */
503     while (missing-- > 0 && tvisnil(L->top - 1))
504       L->top--;
505   }
506 #if LJ_HASJIT
507 out:
508 #endif
509   op = bc_op(pc[-1]);  /* Get FUNC* op. */
510 #if LJ_HASJIT
511   /* Use the non-hotcounting variants if JIT is off or while recording. */
512   if ((!(J->flags & JIT_F_ON) || J->state != LJ_TRACE_IDLE) &&
513       (op == BC_FUNCF || op == BC_FUNCV))
514     op = (BCOp)((int)op+(int)BC_IFUNCF-(int)BC_FUNCF);
515 #endif
516   ERRNO_RESTORE
517   return makeasmfunc(lj_bc_ofs[op]);  /* Return static dispatch target. */
518 }
519
```

```
520 #if LJ_HASJIT
521 /* Stitch a new trace. */
522 void LJ_FASTCALL lj_dispatch_stitch(jit_State *J, const BCIns *pc)
523 {
524   ERRNO_SAVE
525   lua_State *L = J->L;
526   void *cf = cframe_raw(L->cframe);
527   const BCIns *oldpc = cframe_pc(cf);
528   setcframe_pc(cf, pc);
529   /* Before dispatch, have to bias PC by 1. */
530   L->top = L->base + cur_topslot(curr_proto(L), pc+1, cframe_multres_n(cf));
531   lj_trace_stitch(J, pc-1);  /* Point to the CALL instruction. */
532   setcframe_pc(cf, oldpc);
533   ERRNO_RESTORE
534 }
535 #endif
536
537 #if LJ_HASPROFILE
538 /* Profile dispatch. */
539 void LJ_FASTCALL lj_dispatch_profile(lua_State *L, const BCIns *pc)
540 {
541   ERRNO_SAVE
542   GCfunc *fn = curr_func(L);
543   GCproto *pt = funcproto(fn);
544   void *cf = cframe_raw(L->cframe);
545   const BCIns *oldpc = cframe_pc(cf);
546   global_State *g;
547   setcframe_pc(cf, pc);
548   L->top = L->base + cur_topslot(pt, pc, cframe_multres_n(cf));
549   lj_profile_interpreter(L);
550   setcframe_pc(cf, oldpc);
551   g = G(L);
552   setgcref(g->cur_L, obj2gco(L));
553   setvmstate(g, INTERP);
554   ERRNO_RESTORE
555 }
556 #endif
557
```

One Level Up Top Level

src/lj_dispatch.h - luajit-2.0-src

Data types defined

- GG_State
- GG_State
- HotCount

Macros defined

- ERRNO_RESTORE
- ERRNO_RESTORE
- ERRNO_RESTORE
- ERRNO_SAVE
- ERRNO_SAVE
- ERRNO_SAVE
- FFIGOTDEF
- FFIGOTDEF
- G2GG
- G2J
- GG_DISP2G
- GG_DISP2HOT
- GG_DISP2J
- GG_DISP2STATIC
- GG_G2DISP
- GG_LEN_DDISP
- GG_LEN_DISP
- GG_LEN_SDISP
- GG_NUM_ASMFF
- GG_OFS
- GOTDEF
- GOTENUM
- GOTENUM
- HOTCOUNT_CALL
- HOTCOUNT_LOOP

- HOTCOUNT_PCMASK
- HOTCOUNT_SIZE
- J2G
- J2GG
- JITGOTDEF
- JITGOTDEF
- L2GG
- L2J
- WIN32_LEAN_AND_MEAN
- _LJ_DISPATCH_H
- hotcount_get
- hotcount_set

Source code

```
1  /*
2  ** Instruction dispatch handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_DISPATCH_H
7  #define _LJ_DISPATCH_H
8
9  #include "lj_obj.h"
10  #include "lj_bc.h"
11  #if LJ_HASJIT
12  #include "lj_jit.h"
13  #endif
14
15  #if LJ_TARGET_MIPS
16  /* Need our own global offset table for the dreaded MIPS calling conventions. */
17  #if LJ_HASJIT
18  #define JITGOTDEF(_)        _(lj_trace_exit) _(lj_trace_hot)
19  #else
20  #define JITGOTDEF(_)
21  #endif
22  #if LJ_HASFFI
23  #define FFIGOTDEF(_) \
24    _(lj_meta_equal_cd) _(lj_ccallback_enter) _(lj_ccallback_leave)
25  #else
26  #define FFIGOTDEF(_)
27  #endif
28  #define GOTDEF(_) \
29    _(floor) _(ceil) _(trunc) _(log) _(log10) _(exp) _(sin) _(cos) _(tan) \
30    _(asin) _(acos) _(atan) _(sinh) _(cosh) _(tanh) _(frexp) _(modf) _(atan2) \
31    _(pow) _(fmod) _(ldexp) \
32    _(lj_dispatch_call) _(lj_dispatch_ins) _(lj_dispatch_stitch) \
33    _(lj_dispatch_profile) _(lj_err_throw) \
34    _(lj_ffh_coroutine_wrap_err) _(lj_func_closeuv) _(lj_func_newL_gc) \
35    _(lj_gc_barrieruv) _(lj_gc_step) _(lj_gc_step_fixtop) _(lj_meta_arith) \
36    _(lj_meta_call) _(lj_meta_cat) _(lj_meta_comp) _(lj_meta_equal) \
37    _(lj_meta_for) _(lj_meta_istype) _(lj_meta_len) _(lj_meta_tget) \
38    _(lj_meta_tset) _(lj_state_growstack) _(lj_strfmt_num) \
39    _(lj_str_new) _(lj_tab_dup) _(lj_tab_get) _(lj_tab_getinth) _(lj_tab_len) \
40    _(lj_tab_new) _(lj_tab_newkey) _(lj_tab_next) _(lj_tab_reasize) \
41    _(lj_tab_setinth) _(lj_buf_putstr_reverse) _(lj_buf_putstr_lower) \
42    _(lj_buf_putstr_upper) _(lj_buf_tostr) JITGOTDEF(_) FFIGOTDEF(_)
43
44  enum {
45  #define GOTENUM(name) LJ_GOT_##name,
```

```
GOTDEF(GOTENUM)
#undef GOTENUM
  LJ_GOT__MAX
};
#endif

/* Type of hot counter. Must match the code in the assembler VM. */
/* 16 bits are sufficient. Only 0.0015% overhead with maximum slot penalty. */
typedef uint16_t HotCount;

/* Number of hot counter hash table entries (must be a power of two). */
#define HOTCOUNT_SIZE               64
#define HOTCOUNT_PCMASK               ((HOTCOUNT_SIZE-1)*sizeof(HotCount))

/* Hotcount decrements. */
#define HOTCOUNT_LOOP               2
#define HOTCOUNT_CALL               1

/* This solves a circular dependency problem -- bump as needed. Sigh. */
#define GG_NUM_ASMFF        57

#define GG_LEN_DDISP        (BC__MAX + GG_NUM_ASMFF)
#define GG_LEN_SDISP        BC_FUNCF
#define GG_LEN_DISP        (GG_LEN_DDISP + GG_LEN_SDISP)

/* Global state, main thread and extra fields are allocated together. */
typedef struct GG_State {
  lua_State L;                                /* Main thread. */
  global_State g;                          /* Global state. */
#if LJ_TARGET_MIPS
  ASMFunction got[LJ_GOT__MAX];                /* Global offset table. */
#endif
#if LJ_HASJIT
  jit_State J;                                /* JIT state. */
  HotCount hotcount[HOTCOUNT_SIZE];        /* Hot counters. */
#endif
  ASMFunction dispatch[GG_LEN_DISP];        /* Instruction dispatch tables. */
  BCIns bcff[GG_NUM_ASMFF];                /* Bytecode for ASM fast functions. */
} GG_State;

#define GG_OFS(field)        ((int)offsetof(GG_State, field))
#define G2GG(gl)        ((GG_State *)((char *)(gl) - GG_OFS(g)))
#define J2GG(j)               ((GG_State *)((char *)(j) - GG_OFS(J)))
#define L2GG(L)               (G2GG(G(L)))
#define J2G(J)               (&J2GG(J)->g)
#define G2J(gl)               (&G2GG(gl)->J)
#define L2J(L)               (&L2GG(L)->J)
#define GG_G2DISP        (GG_OFS(dispatch) - GG_OFS(g))
#define GG_DISP2G        (GG_OFS(g) - GG_OFS(dispatch))
#define GG_DISP2J        (GG_OFS(J) - GG_OFS(dispatch))
#define GG_DISP2HOT        (GG_OFS(hotcount) - GG_OFS(dispatch))
#define GG_DISP2STATIC        (GG_LEN_DDISP*(int)sizeof(ASMFunction))

#define hotcount_get(gg, pc) \
   (gg)->hotcount[(u32ptr(pc)>>2) & (HOTCOUNT_SIZE-1)]
 #define hotcount_set(gg, pc, val) \
   (hotcount_get((gg), (pc)) = (HotCount)(val))

 /* Dispatch table management. */
 LJ_FUNC void lj_dispatch_init(GG_State *GG);
 #if LJ_HASJIT
 LJ_FUNC void lj_dispatch_init_hotcount(global_State *g);
 #endif
 LJ_FUNC void lj_dispatch_update(global_State *g);

 /* Instruction dispatch callback for hooks or when recording. */
 LJ_FUNCA void LJ_FASTCALL lj_dispatch_ins(lua_State *L, const BCIns *pc);
 LJ_FUNCA ASMFunction LJ_FASTCALL lj_dispatch_call(lua_State *L, const BCIns*pc);
 #if LJ_HASJIT
 LJ_FUNCA void LJ_FASTCALL lj_dispatch_stitch(jit_State *J, const BCIns *pc);
 #endif
 #if LJ_HASPROFILE
 LJ_FUNCA void LJ_FASTCALL lj_dispatch_profile(lua_State *L, const BCIns *pc);
 #endif

 #if LJ_HASFFI && !defined(_BUILDVM_H)
```

```
122 /* Save/restore errno and GetLastError() around hooks, exits and recording. */
123 #include <errno.h>
124 #if LJ_TARGET_WINDOWS
125 #define WIN32_LEAN_AND_MEAN
126 #include <windows.h>
127 #define ERRNO_SAVE        int olderr = errno; DWORD oldwerr = GetLastError();
128 #define ERRNO_RESTORE        errno = olderr; SetLastError(oldwerr);
129 #else
130 #define ERRNO_SAVE        int olderr = errno;
131 #define ERRNO_RESTORE        errno = olderr;
132 #endif
133 #else
134 #define ERRNO_SAVE
135 #define ERRNO_RESTORE
136 #endif
137
138 #endif
```

One Level Up Top Level

src/lj_trace.c - luajit-2.0-src

Data types defined

- ExitDataCP
- ExitDataCP

Functions defined

- blacklist_pc
- lj_trace_err
- lj_trace_err_info
- lj_trace_exit
- lj_trace_flush
- lj_trace_flushall
- lj_trace_flushproto
- lj_trace_free
- lj_trace_freestate
- lj_trace_hot
- lj_trace_initstate
- lj_trace_ins
- lj_trace_reenableproto
- lj_trace_stitch
- penalty_pc
- perftools_addtrace
- trace_abort
- trace_downrec
- trace_exit_cp
- trace_exit_find
- trace_exit_regs
- trace_findfree
- trace_flushroot
- trace_hotside
- trace_pendpatch
- trace_save

- trace_start
- trace_state
- trace_stop
- trace_unpatch

Macros defined

- LUA_CORE
- TRACE_APPENDVEC
- lj_trace_c

Source code

```
1  /*
2  ** Trace management.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lj_trace_c
7  #define LUA_CORE
8
9  #include "lj_obj.h"
10
11 #if LJ_HASJIT
12
13 #include "lj_gc.h"
14 #include "lj_err.h"
15 #include "lj_debug.h"
16 #include "lj_str.h"
17 #include "lj_frame.h"
18 #include "lj_state.h"
19 #include "lj_bc.h"
20 #include "lj_ir.h"
21 #include "lj_jit.h"
22 #include "lj_iropt.h"
23 #include "lj_mcode.h"
24 #include "lj_trace.h"
25 #include "lj_snap.h"
26 #include "lj_gdbjit.h"
27 #include "lj_record.h"
28 #include "lj_asm.h"
29 #include "lj_dispatch.h"
30 #include "lj_vm.h"
31 #include "lj_vmevent.h"
32 #include "lj_target.h"
33
34 /* -- Error handling ------------------------------------------------------ */
35
36 /* Synchronous abort with error message. */
37 void lj_trace_err(jit_State *J, TraceError e)
38 {
39   setnilV(&J->errinfo);  /* No error info. */
40   setintV(J->L->top++, (int32_t)e);
41   lj_err_throw(J->L, LUA_ERRRUN);
42 }
43
44 /* Synchronous abort with error message and error info. */
45 void lj_trace_err_info(jit_State *J, TraceError e)
46 {
47   setintV(J->L->top++, (int32_t)e);
48   lj_err_throw(J->L, LUA_ERRRUN);
49 }
50
51 /* -- Trace management ---------------------------------------------------- */
52
53 /* The current trace is first assembled in J->cur. The variable length
```

```
** arrays point to shared, growable buffers (J->irbuf etc.). When trace
** recording ends successfully, the current trace and its data structures
** are copied to a new (compact) GCtrace object.
*/

/* Find a free trace number. */
static TraceNo trace_findfree(jit_State *J)
{
  MSize osz, lim;
  if (J->freetrace == 0)
    J->freetrace = 1;
  for (; J->freetrace < J->sizetrace; J->freetrace++)
    if (traceref(J, J->freetrace) == NULL)
      return J->freetrace++;
  /* Need to grow trace array. */
  lim = (MSize)J->param[JIT_P_maxtrace] + 1;
  if (lim < 2) lim = 2; else if (lim > 65535) lim = 65535;
  osz = J->sizetrace;
  if (osz >= lim)
    return 0;  /* Too many traces. */
  lj_mem_growvec(J->L, J->trace, J->sizetrace, lim, GCRef);
  for (; osz < J->sizetrace; osz++)
    setgcrefnull(J->trace[osz]);
  return J->freetrace;
}

#define TRACE_APPENDVEC(field, szfield, tp) \
  T->field = (tp *)p; \
  memcpy(p, J->cur.field, J->cur.szfield*sizeof(tp)); \
  p += J->cur.szfield*sizeof(tp);

#ifdef LUAJIT_USE_PERFTOOLS
/*
** Create symbol table of JIT-compiled code. For use with Linux perf tools.
** Example usage:
**   perf record -f -e cycles luajit test.lua
**   perf report -s symbol
**   rm perf.data /tmp/perf-*.map
*/
#include <stdio.h>
#include <unistd.h>

static void perftools_addtrace(GCtrace *T)
{
  static FILE *fp;
  GCproto *pt = &gcref(T->startpt)->pt;
  const BCIns *startpc = mref(T->startpc, const BCIns);
  const char *name = proto_chunknamestr(pt);
  BCLine lineno;
  if (name[0] == '@' || name[0] == '=')
    name++;
  else
    name = "(string)";
  lua_assert(startpc >= proto_bc(pt) && startpc < proto_bc(pt) + pt->sizebc);
  lineno = lj_debug_line(pt, proto_bcpos(pt, startpc));
  if (!fp) {
    char fname[40];
    sprintf(fname, "/tmp/perf-%d.map", getpid());
    if (!(fp = fopen(fname, "w"))) return;
    setlinebuf(fp);
  }
  fprintf(fp, "%lx %x TRACE_%d::%s:%u\n",
          (long)T->mcode, T->szmcode, T->traceno, name, lineno);
}
#endif

/* Save current trace by copying and compacting it. */
static void trace_save(jit_State *J)
{
  size_t sztr = ((sizeof(GCtrace)+7)&~7);
  size_t szins = (J->cur.nins-J->cur.nk)*sizeof(IRIns);
  size_t sz = sztr + szins +
              J->cur.nsnap*sizeof(SnapShot) +
              J->cur.nsnapmap*sizeof(SnapEntry);
  GCtrace *T = lj_mem_newt(J->L, (MSize)sz, GCtrace);
  char *p = (char *)T + sztr;
```

```
  memcpy(T, &J->cur, sizeof(GCtrace));
  setgcrefr(T->nextgc, J2G(J)->gc.root);
  setgcrefp(J2G(J)->gc.root, T);
  newwhite(J2G(J), T);
  T->gct = ~LJ_TTRACE;
  T->ir = (IRIns *)p - J->cur.nk;
  memcpy(p, J->cur.ir+J->cur.nk, szins);
  p += szins;
  TRACE_APPENDVEC(snap, nsnap, SnapShot)
  TRACE_APPENDVEC(snapmap, nsnapmap, SnapEntry)
  J->cur.traceno = 0;
  setgcrefp(J->trace[T->traceno], T);
  lj_gc_barriertrace(J2G(J), T->traceno);
  lj_gdbjit_addtrace(J, T);
#ifdef LUAJIT_USE_PERFTOOLS
  perftools_addtrace(T);
#endif
}

void LJ_FASTCALL lj_trace_free(global_State *g, GCtrace *T)
{
  jit_State *J = G2J(g);
  if (T->traceno) {
    lj_gdbjit_deltrace(J, T);
    if (T->traceno < J->freetrace)
      J->freetrace = T->traceno;
    setgcrefnull(J->trace[T->traceno]);
  }
  lj_mem_free(g, T,
    ((sizeof(GCtrace)+7)&~7) + (T->nins-T->nk)*sizeof(IRIns) +
    T->nsnap*sizeof(SnapShot) + T->nsnapmap*sizeof(SnapEntry));
}

/* Re-enable compiling a prototype by unpatching any modified bytecode. */
void lj_trace_reenableproto(GCproto *pt)
{
  if ((pt->flags & PROTO_ILOOP)) {
    BCIns *bc = proto_bc(pt);
    BCPos i, sizebc = pt->sizebc;;
    pt->flags &= ~PROTO_ILOOP;
    if (bc_op(bc[0]) == BC_IFUNCF)
      setbc_op(&bc[0], BC_FUNCF);
    for (i = 1; i < sizebc; i++) {
      BCOp op = bc_op(bc[i]);
      if (op == BC_IFORL || op == BC_IITERL || op == BC_ILOOP)
        setbc_op(&bc[i], (int)op+(int)BC_LOOP-(int)BC_ILOOP);
    }
  }
}

/* Unpatch the bytecode modified by a root trace. */
static void trace_unpatch(jit_State *J, GCtrace *T)
{
  BCOp op = bc_op(T->startins);
  BCIns *pc = mref(T->startpc, BCIns);
  UNUSED(J);
  if (op == BC_JMP)
    return;  /* No need to unpatch branches in parent traces (yet). */
  switch (bc_op(*pc)) {
  case BC_JFORL:
    lua_assert(traceref(J, bc_d(*pc)) == T);
    *pc = T->startins;
    pc += bc_j(T->startins);
    lua_assert(bc_op(*pc) == BC_JFORI);
    setbc_op(pc, BC_FORI);
    break;
  case BC_JITERL:
  case BC_JLOOP:
    lua_assert(op == BC_ITERL || op == BC_LOOP || bc_isret(op));
    *pc = T->startins;
    break;
  case BC_JMP:
    lua_assert(op == BC_ITERL);
    pc += bc_j(*pc)+2;
    if (bc_op(*pc) == BC_JITERL) {
      lua_assert(traceref(J, bc_d(*pc)) == T);
```

```
      *pc = T->startins;
    }
    break;
  case BC_JFUNCF:
    lua_assert(op == BC_FUNCF);
    *pc = T->startins;
    break;
  default:  /* Already unpatched. */
    break;
  }
}

/* Flush a root trace. */
static void trace_flushroot(jit_State *J, GCtrace *T)
{
  GCproto *pt = &gcref(T->startpt)->pt;
  lua_assert(T->root == 0 && pt != NULL);
  /* First unpatch any modified bytecode. */
  trace_unpatch(J, T);
  /* Unlink root trace from chain anchored in prototype. */
  if (pt->trace == T->traceno) {  /* Trace is first in chain. Easy. */
    pt->trace = T->nextroot;
  } else if (pt->trace) {  /* Otherwise search in chain of root traces. */
    GCtrace *T2 = traceref(J, pt->trace);
    if (T2) {
      for (; T2->nextroot; T2 = traceref(J, T2->nextroot))
        if (T2->nextroot == T->traceno) {
          T2->nextroot = T->nextroot;  /* Unlink from chain. */
          break;
        }
    }
  }
}

/* Flush a trace. Only root traces are considered. */
void lj_trace_flush(jit_State *J, TraceNo traceno)
{
  if (traceno > 0 && traceno < J->sizetrace) {
    GCtrace *T = traceref(J, traceno);
    if (T && T->root == 0)
      trace_flushroot(J, T);
  }
}

/* Flush all traces associated with a prototype. */
void lj_trace_flushproto(global_State *g, GCproto *pt)
{
  while (pt->trace != 0)
    trace_flushroot(G2J(g), traceref(G2J(g), pt->trace));
}

/* Flush all traces. */
int lj_trace_flushall(lua_State *L)
{
  jit_State *J = L2J(L);
  ptrdiff_t i;
  if ((J2G(J)->hookmask & HOOK_GC))
    return 1;
  for (i = (ptrdiff_t)J->sizetrace-1; i > 0; i--) {
    GCtrace *T = traceref(J, i);
    if (T) {
      if (T->root == 0)
        trace_flushroot(J, T);
      lj_gdbjit_deltrace(J, T);
      T->traceno = 0;
      setgcrefnull(J->trace[i]);
    }
  }
  J->cur.traceno = 0;
  J->freetrace = 0;
  /* Clear penalty cache. */
  memset(J->penalty, 0, sizeof(J->penalty));
  /* Free the whole machine code and invalidate all exit stub groups. */
  lj_mcode_free(J);
  memset(J->exitstubgroup, 0, sizeof(J->exitstubgroup));
  lj_vmevent_send(L, TRACE,
```

```
    setstrV(L, L->top++, lj_str_newlit(L, "flush"));
  );
  return 0;
}

/* Initialize JIT compiler state. */
void lj_trace_initstate(global_State *g)
{
  jit_State *J = G2J(g);
  TValue *tv;
  /* Initialize SIMD constants. */
  tv = LJ_KSIMD(J, LJ_KSIMD_ABS);
  tv[0].u64 = U64x(7fffffff,ffffffff);
  tv[1].u64 = U64x(7fffffff,ffffffff);
  tv = LJ_KSIMD(J, LJ_KSIMD_NEG);
  tv[0].u64 = U64x(80000000,00000000);
  tv[1].u64 = U64x(80000000,00000000);
}

/* Free everything associated with the JIT compiler state. */
void lj_trace_freestate(global_State *g)
{
  jit_State *J = G2J(g);
#ifdef LUA_USE_ASSERT
  {  /* This assumes all traces have already been freed. */
    ptrdiff_t i;
    for (i = 1; i < (ptrdiff_t)J->sizetrace; i++)
      lua_assert(i == (ptrdiff_t)J->cur.traceno || traceref(J, i) == NULL);
  }
#endif
  lj_mcode_free(J);
  lj_ir_k64_freeall(J);
  lj_mem_freevec(g, J->snapmapbuf, J->sizesnapmap, SnapEntry);
  lj_mem_freevec(g, J->snapbuf, J->sizesnap, SnapShot);
  lj_mem_freevec(g, J->irbuf + J->irbotlim, J->irtoplim - J->irbotlim, IRIns);
  lj_mem_freevec(g, J->trace, J->sizetrace, GCRef);
}

/* -- Penalties and blacklisting ------------------------------------------ */

/* Blacklist a bytecode instruction. */
static void blacklist_pc(GCproto *pt, BCIns *pc)
{
  setbc_op(pc, (int)bc_op(*pc)+(int)BC_ILOOP-(int)BC_LOOP);
  pt->flags |= PROTO_ILOOP;
}

/* Penalize a bytecode instruction. */
static void penalty_pc(jit_State *J, GCproto *pt, BCIns *pc, TraceError e)
{
  uint32_t i, val = PENALTY_MIN;
  for (i = 0; i < PENALTY_SLOTS; i++)
    if (mref(J->penalty[i].pc, const BCIns) == pc) {  /* Cache slot found? */
      /* First try to bump its hotcount several times. */
      val = ((uint32_t)J->penalty[i].val << 1) +
            LJ_PRNG_BITS(J, PENALTY_RNDBITS);
      if (val > PENALTY_MAX) {
        blacklist_pc(pt, pc);  /* Blacklist it, if that didn't help. */
        return;
      }
      goto setpenalty;
    }
  /* Assign a new penalty cache slot. */
  i = J->penaltyslot;
  J->penaltyslot = (J->penaltyslot + 1) & (PENALTY_SLOTS-1);
  setmref(J->penalty[i].pc, pc);
setpenalty:
  J->penalty[i].val = (uint16_t)val;
  J->penalty[i].reason = e;
  hotcount_set(J2GG(J), pc+1, val);
}

/* -- Trace compiler state machine ---------------------------------------- */

/* Start tracing. */
static void trace_start(jit_State *J)
```

```
{
  lua_State *L;
  TraceNo traceno;

  if ((J->pt->flags & PROTO_NOJIT)) {  /* JIT disabled for this proto? */
    if (J->parent == 0 && J->exitno == 0) {
      /* Lazy bytecode patching to disable hotcount events. */
      lua_assert(bc_op(*J->pc) == BC_FORL || bc_op(*J->pc) == BC_ITERL ||
                 bc_op(*J->pc) == BC_LOOP || bc_op(*J->pc) == BC_FUNCF);
      setbc_op(J->pc, (int)bc_op(*J->pc)+(int)BC_ILOOP-(int)BC_LOOP);
      J->pt->flags |= PROTO_ILOOP;
    }
    J->state = LJ_TRACE_IDLE;  /* Silently ignored. */
    return;
  }

  /* Get a new trace number. */
  traceno = trace_findfree(J);
  if (LJ_UNLIKELY(traceno == 0)) {  /* No free trace? */
    lua_assert((J2G(J)->hookmask & HOOK_GC) == 0);
    lj_trace_flushall(J->L);
    J->state = LJ_TRACE_IDLE;  /* Silently ignored. */
    return;
  }
  setgcrefp(J->trace[traceno], &J->cur);

  /* Setup enough of the current trace to be able to send the vmevent. */
  memset(&J->cur, 0, sizeof(GCtrace));
  J->cur.traceno = traceno;
  J->cur.nins = J->cur.nk = REF_BASE;
  J->cur.ir = J->irbuf;
  J->cur.snap = J->snapbuf;
  J->cur.snapmap = J->snapmapbuf;
  J->mergesnap = 0;
  J->needsnap = 0;
  J->bcskip = 0;
  J->guardemit.irt = 0;
  J->postproc = LJ_POST_NONE;
  lj_resetsplit(J);
  setgcref(J->cur.startpt, obj2gco(J->pt));

  L = J->L;
  lj_vmevent_send(L, TRACE,
    setstrV(L, L->top++, lj_str_newlit(L, "start"));
    setintV(L->top++, traceno);
    setfuncV(L, L->top++, J->fn);
    setintV(L->top++, proto_bcpos(J->pt, J->pc));
    if (J->parent) {
      setintV(L->top++, J->parent);
      setintV(L->top++, J->exitno);
    }
  );
  lj_record_setup(J);
}

/* Stop tracing. */
static void trace_stop(jit_State *J)
{
  BCIns *pc = mref(J->cur.startpc, BCIns);
  BCOp op = bc_op(J->cur.startins);
  GCproto *pt = &gcref(J->cur.startpt)->pt;
  TraceNo traceno = J->cur.traceno;
  lua_State *L;

  switch (op) {
  case BC_FORL:
    setbc_op(pc+bc_j(J->cur.startins), BC_JFORI);  /* Patch FORI, too. */
    /* fallthrough */
  case BC_LOOP:
  case BC_ITERL:
  case BC_FUNCF:
    /* Patch bytecode of starting instruction in root trace. */
    setbc_op(pc, (int)op+(int)BC_JLOOP-(int)BC_LOOP);
    setbc_d(pc, traceno);
  addroot:
    /* Add to root trace chain in prototype. */
```

```
    J->cur.nextroot = pt->trace;
    pt->trace = (TraceNo1)traceno;
    break;
  case BC_RET:
  case BC_RET0:
  case BC_RET1:
    *pc = BCINS_AD(BC_JLOOP, J->cur.snap[0].nslots, traceno);
    goto addroot;
  case BC_JMP:
    /* Patch exit branch in parent to side trace entry. */
    lua_assert(J->parent != 0 && J->cur.root != 0);
    lj_asm_patchexit(J, traceref(J, J->parent), J->exitno, J->cur.mcode);
    /* Avoid compiling a side trace twice (stack resizing uses parent exit). */
    traceref(J, J->parent)->snap[J->exitno].count = SNAPCOUNT_DONE;
    /* Add to side trace chain in root trace. */
    {
      GCtrace *root = traceref(J, J->cur.root);
      root->nchild++;
      J->cur.nextside = root->nextside;
      root->nextside = (TraceNo1)traceno;
    }
    break;
  case BC_CALLM:
  case BC_CALL:
  case BC_ITERC:
    /* Trace stitching: patch link of previous trace. */
    traceref(J, J->exitno)->link = traceno;
    break;
  default:
    lua_assert(0);
    break;
  }

  /* Commit new mcode only after all patching is done. */
  lj_mcode_commit(J, J->cur.mcode);
  J->postproc = LJ_POST_NONE;
  trace_save(J);

  L = J->L;
  lj_vmevent_send(L, TRACE,
    setstrV(L, L->top++, lj_str_newlit(L, "stop"));
    setintV(L->top++, traceno);
    setfuncV(L, L->top++, J->fn);
  );
}

/* Start a new root trace for down-recursion. */
static int trace_downrec(jit_State *J)
{
  /* Restart recording at the return instruction. */
  lua_assert(J->pt != NULL);
  lua_assert(bc_isret(bc_op(*J->pc)));
  if (bc_op(*J->pc) == BC_RETM)
    return 0;  /* NYI: down-recursion with RETM. */
  J->parent = 0;
  J->exitno = 0;
  J->state = LJ_TRACE_RECORD;
  trace_start(J);
  return 1;
}

/* Abort tracing. */
static int trace_abort(jit_State *J)
{
  lua_State *L = J->L;
  TraceError e = LJ_TRERR_RECERR;
  TraceNo traceno;

  J->postproc = LJ_POST_NONE;
  lj_mcode_abort(J);
  if (tvisnumber(L->top-1))
    e = (TraceError)numberVint(L->top-1);
  if (e == LJ_TRERR_MCODELM) {
    L->top--;  /* Remove error object */
    J->state = LJ_TRACE_ASM;
    return 1;  /* Retry ASM with new MCode area. */
```

```
  }
  /* Penalize or blacklist starting bytecode instruction. */
  if (J->parent == 0 && !bc_isret(bc_op(J->cur.startins))) {
    if (J->exitno == 0)
      penalty_pc(J, &gcref(J->cur.startpt)->pt, mref(J->cur.startpc, BCIns), e);
    else
      traceref(J, J->exitno)->link = J->exitno;  /* Self-link is blacklisted. */
  }

  /* Is there anything to abort? */
  traceno = J->cur.traceno;
  if (traceno) {
    ptrdiff_t errobj = savestack(L, L->top-1);  /* Stack may be resized. */
    J->cur.link = 0;
    J->cur.linktype = LJ_TRLINK_NONE;
    lj_vmevent_send(L, TRACE,
      TValue *frame;
      const BCIns *pc;
      GCfunc *fn;
      setstrV(L, L->top++, lj_str_newlit(L, "abort"));
      setintV(L->top++, traceno);
      /* Find original Lua function call to generate a better error message. */
      frame = J->L->base-1;
      pc = J->pc;
      while (!isluafunc(frame_func(frame))) {
        pc = (frame_iscont(frame) ? frame_contpc(frame) : frame_pc(frame)) - 1;
        frame = frame_prev(frame);
      }
      fn = frame_func(frame);
      setfuncV(L, L->top++, fn);
      setintV(L->top++, proto_bcpos(funcproto(fn), pc));
      copyTV(L, L->top++, restorestack(L, errobj));
      copyTV(L, L->top++, &J->errinfo);
    );
    /* Drop aborted trace after the vmevent (which may still access it). */
    setgcrefnull(J->trace[traceno]);
    if (traceno < J->freetrace)
      J->freetrace = traceno;
    J->cur.traceno = 0;
  }
  L->top--;  /* Remove error object */
  if (e == LJ_TRERR_DOWNREC)
    return trace_downrec(J);
  else if (e == LJ_TRERR_MCODEAL)
    lj_trace_flushall(L);
  return 0;
}

/* Perform pending re-patch of a bytecode instruction. */
static LJ_AINLINE void trace_pendpatch(jit_State *J, int force)
{
  if (LJ_UNLIKELY(J->patchpc)) {
    if (force || J->bcskip == 0) {
      *J->patchpc = J->patchins;
      J->patchpc = NULL;
    } else {
      J->bcskip = 0;
    }
  }
}

/* State machine for the trace compiler. Protected callback. */
static TValue *trace_state(lua_State *L, lua_CFunction dummy, void *ud)
{
  jit_State *J = (jit_State *)ud;
  UNUSED(dummy);
  do {
  retry:
    switch (J->state) {
    case LJ_TRACE_START:
      J->state = LJ_TRACE_RECORD;  /* trace_start() may change state. */
      trace_start(J);
      lj_dispatch_update(J2G(J));
      break;

    case LJ_TRACE_RECORD:
```

```
      trace_pendpatch(J, 0);
      setvmstate(J2G(J), RECORD);
      lj_vmevent_send(L, RECORD,
        /* Save/restore tmptv state for trace recorder. */
        TValue savetv = J2G(J)->tmptv;
        TValue savetv2 = J2G(J)->tmptv2;
        setintV(L->top++, J->cur.traceno);
        setfuncV(L, L->top++, J->fn);
        setintV(L->top++, J->pt ? (int32_t)proto_bcpos(J->pt, J->pc) : -1);
        setintV(L->top++, J->framedepth);
      ,
        J2G(J)->tmptv = savetv;
        J2G(J)->tmptv2 = savetv2;
      );
      lj_record_ins(J);
      break;

    case LJ_TRACE_END:
      trace_pendpatch(J, 1);
      J->loopref = 0;
      if ((J->flags & JIT_F_OPT_LOOP) &&
          J->cur.link == J->cur.traceno && J->framedepth + J->retdepth == 0) {
        setvmstate(J2G(J), OPT);
        lj_opt_dce(J);
        if (lj_opt_loop(J)) {  /* Loop optimization failed? */
          J->cur.link = 0;
          J->cur.linktype = LJ_TRLINK_NONE;
          J->loopref = J->cur.nins;
          J->state = LJ_TRACE_RECORD;  /* Try to continue recording. */
          break;
        }
        J->loopref = J->chain[IR_LOOP];  /* Needed by assembler. */
      }
      lj_opt_split(J);
      lj_opt_sink(J);
      if (!J->loopref) J->cur.snap[J->cur.nsnap-1].count = SNAPCOUNT_DONE;
      J->state = LJ_TRACE_ASM;
      break;

    case LJ_TRACE_ASM:
      setvmstate(J2G(J), ASM);
      lj_asm_trace(J, &J->cur);
      trace_stop(J);
      setvmstate(J2G(J), INTERP);
      J->state = LJ_TRACE_IDLE;
      lj_dispatch_update(J2G(J));
      return NULL;

    default:  /* Trace aborted asynchronously. */
      setintV(L->top++, (int32_t)LJ_TRERR_RECERR);
      /* fallthrough */
    case LJ_TRACE_ERR:
      trace_pendpatch(J, 1);
      if (trace_abort(J))
        goto retry;
      setvmstate(J2G(J), INTERP);
      J->state = LJ_TRACE_IDLE;
      lj_dispatch_update(J2G(J));
      return NULL;
    }
  } while (J->state > LJ_TRACE_RECORD);
  return NULL;
}

/* -- Event handling ------------------------------------------------------ */

/* A bytecode instruction is about to be executed. Record it. */
void lj_trace_ins(jit_State *J, const BCIns *pc)
{
  /* Note: J->L must already be set. pc is the true bytecode PC here. */
  J->pc = pc;
  J->fn = curr_func(J->L);
  J->pt = isluafunc(J->fn) ? funcproto(J->fn) : NULL;
  while (lj_vm_cpcall(J->L, NULL, (void *)J, trace_state) != 0)
    J->state = LJ_TRACE_ERR;
}
```

```
662
663 /* A hotcount triggered. Start recording a root trace. */
664 void LJ_FASTCALL lj_trace_hot(jit_State *J, const BCIns *pc)
665 {
666   /* Note: pc is the interpreter bytecode PC here. It's offset by 1. */
667   ERRNO_SAVE
668   /* Reset hotcount. */
669   hotcount_set(J2GG(J), pc, J->param[JIT_P_hotloop]*HOTCOUNT_LOOP);
670   /* Only start a new trace if not recording or inside __gc call or vmevent. */
671   if (J->state == LJ_TRACE_IDLE &&
672       !(J2G(J)->hookmask & (HOOK_GC|HOOK_VMEVENT))) {
673     J->parent = 0;  /* Root trace. */
674     J->exitno = 0;
675     J->state = LJ_TRACE_START;
676     lj_trace_ins(J, pc-1);
677   }
678   ERRNO_RESTORE
679 }
680
681 /* Check for a hot side exit. If yes, start recording a side trace. */
682 static void trace_hotside(jit_State *J, const BCIns *pc)
683 {
684   SnapShot *snap = &traceref(J, J->parent)->snap[J->exitno];
685   if (!(J2G(J)->hookmask & (HOOK_GC|HOOK_VMEVENT)) &&
686       isluafunc(curr_func(J->L)) &&
687       snap->count != SNAPCOUNT_DONE &&
688       ++snap->count >= J->param[JIT_P_hotexit]) {
689     lua_assert(J->state == LJ_TRACE_IDLE);
690     /* J->parent is non-zero for a side trace. */
691     J->state = LJ_TRACE_START;
692     lj_trace_ins(J, pc);
693   }
694 }
695
696 /* Stitch a new trace to the previous trace. */
697 void LJ_FASTCALL lj_trace_stitch(jit_State *J, const BCIns *pc)
698 {
699   /* Only start a new trace if not recording or inside __gc call or vmevent. */
700   if (J->state == LJ_TRACE_IDLE &&
701       !(J2G(J)->hookmask & (HOOK_GC|HOOK_VMEVENT))) {
702     J->parent = 0;  /* Have to treat it like a root trace. */
703     /* J->exitno is set to the invoking trace. */
704     J->state = LJ_TRACE_START;
705     lj_trace_ins(J, pc);
706   }
707 }
708
709
710 /* Tiny struct to pass data to protected call. */
711 typedef struct ExitDataCP {
712   jit_State *J;
713   void *exptr;                /* Pointer to exit state. */
714   const BCIns *pc;        /* Restart interpreter at this PC. */
715 } ExitDataCP;
716
717 /* Need to protect lj_snap_restore because it may throw. */
718 static TValue *trace_exit_cp(lua_State *L, lua_CFunction dummy, void *ud)
719 {
720   ExitDataCP *exd = (ExitDataCP *)ud;
721   cframe_errfunc(L->cframe) = -1;  /* Inherit error function. */
722   exd->pc = lj_snap_restore(exd->J, exd->exptr);
723   UNUSED(dummy);
724   return NULL;
725 }
726
727 #ifndef LUAJIT_DISABLE_VMEVENT
728 /* Push all registers from exit state. */
729 static void trace_exit_regs(lua_State *L, ExitState *ex)
730 {
731   int32_t i;
732   setintV(L->top++, RID_NUM_GPR);
733   setintV(L->top++, RID_NUM_FPR);
734   for (i = 0; i < RID_NUM_GPR; i++) {
735     if (sizeof(ex->gpr[i]) == sizeof(int32_t))
736       setintV(L->top++, (int32_t)ex->gpr[i]);
737     else
```

```
      setnumV(L->top++, (lua_Number)ex->gpr[i]);
  }
#if !LJ_SOFTFP
  for (i = 0; i < RID_NUM_FPR; i++) {
    setnumV(L->top, ex->fpr[i]);
    if (LJ_UNLIKELY(tvisnan(L->top)))
      setnanV(L->top);
    L->top++;
  }
#endif
}
#endif

#ifdef EXITSTATE_PCREG
/* Determine trace number from pc of exit instruction. */
static TraceNo trace_exit_find(jit_State *J, MCode *pc)
{
  TraceNo traceno;
  for (traceno = 1; traceno < J->sizetrace; traceno++) {
    GCtrace *T = traceref(J, traceno);
    if (T && pc >= T->mcode && pc < (MCode *)((char *)T->mcode + T->szmcode))
      return traceno;
  }
  lua_assert(0);
  return 0;
}
#endif

/* A trace exited. Restore interpreter state. */
int LJ_FASTCALL lj_trace_exit(jit_State *J, void *exptr)
{
  ERRNO_SAVE
  lua_State *L = J->L;
  ExitState *ex = (ExitState *)exptr;
  ExitDataCP exd;
  int errcode;
  const BCIns *pc;
  void *cf;
  GCtrace *T;
#ifdef EXITSTATE_PCREG
  J->parent = trace_exit_find(J, (MCode *)(intptr_t)ex->gpr[EXITSTATE_PCREG]);
#endif
  T = traceref(J, J->parent); UNUSED(T);
#ifdef EXITSTATE_CHECKEXIT
  if (J->exitno == T->nsnap) {  /* Treat stack check like a parent exit. */
    lua_assert(T->root != 0);
    J->exitno = T->ir[REF_BASE].op2;
    J->parent = T->ir[REF_BASE].op1;
    T = traceref(J, J->parent);
  }
#endif
  lua_assert(T != NULL && J->exitno < T->nsnap);
  exd.J = J;
  exd.exptr = exptr;
  errcode = lj_vm_cpcall(L, NULL, &exd, trace_exit_cp);
  if (errcode)
    return -errcode;  /* Return negated error code. */

  if (!(LJ_HASPROFILE && (G(L)->hookmask & HOOK_PROFILE)))
    lj_vmevent_send(L, TEXIT,
      lj_state_checkstack(L, 4+RID_NUM_GPR+RID_NUM_FPR+LUA_MINSTACK);
      setintV(L->top++, J->parent);
      setintV(L->top++, J->exitno);
      trace_exit_regs(L, ex);
    );

  pc = exd.pc;
  cf = cframe_raw(L->cframe);
  setcframe_pc(cf, pc);
  if (LJ_HASPROFILE && (G(L)->hookmask & HOOK_PROFILE)) {
    /* Just exit to interpreter. */
  } else if (G(L)->gc.state == GCSatomic || G(L)->gc.state == GCSfinalize) {
    if (!(G(L)->hookmask & HOOK_GC))
      lj_gc_step(L);  /* Exited because of GC: drive GC forward. */
  } else {
    trace_hotside(J, pc);
```

```
  }
  if (bc_op(*pc) == BC_JLOOP) {
    BCIns *retpc = &traceref(J, bc_d(*pc))->startins;
    if (bc_isret(bc_op(*retpc))) {
      if (J->state == LJ_TRACE_RECORD) {
        J->patchins = *pc;
        J->patchpc = (BCIns *)pc;
        *J->patchpc = *retpc;
        J->bcskip = 1;
      } else {
        pc = retpc;
        setcframe_pc(cf, pc);
      }
    }
  }
  /* Return MULTRES or 0. */
  ERRNO_RESTORE
  switch (bc_op(*pc)) {
  case BC_CALLM: case BC_CALLMT:
    return (int)((BCReg)(L->top - L->base) - bc_a(*pc) - bc_c(*pc) + LJ_FR2);
  case BC_RETM:
    return (int)((BCReg)(L->top - L->base) + 1 - bc_a(*pc) - bc_d(*pc));
  case BC_TSETM:
    return (int)((BCReg)(L->top - L->base) + 1 - bc_a(*pc));
  default:
    if (bc_op(*pc) >= BC_FUNCF)
      return (int)((BCReg)(L->top - L->base) + 1);
    return 0;
  }
}

#endif
```

One Level Up Top Level

src/lj_jit.h - luajit-2.0-src

Global variables defined

- offsetof

Data types defined

- BPropEntry
- BPropEntry
- ExitNo
- FoldState
- FoldState
- GCtrace
- GCtrace
- HotPenalty
- HotPenalty
- MCode
- MCode
- PostProc
- ScEvEntry
- ScEvEntry
- SnapEntry
- SnapNo
- SnapShot
- SnapShot
- TraceLink
- TraceNo
- TraceNo1
- TraceState
- jit_State

Functions defined

- static LJ_AINLINE uint32_t LJ_PRNG_BITS(jit_State *J, int bits)
- snap_nextofs

Macros defined

- BPROP_SLOTS
- JIT_F_ARMV6
- JIT_F_ARMV6T2
- JIT_F_ARMV6T2_
- JIT_F_ARMV6_
- JIT_F_ARMV7
- JIT_F_CPUSTRING
- JIT_F_CPUSTRING
- JIT_F_CPUSTRING
- JIT_F_CPUSTRING
- JIT_F_CPUSTRING
- JIT_F_CPU_FIRST
- JIT_F_CPU_FIRST
- JIT_F_CPU_FIRST
- JIT_F_CPU_FIRST
- JIT_F_CPU_FIRST
- JIT_F_LEA_AGU
- JIT_F_MIPS32R2
- JIT_F_ON
- JIT_F_OPTSTRING
- JIT_F_OPT_0
- JIT_F_OPT_1
- JIT_F_OPT_2
- JIT_F_OPT_3
- JIT_F_OPT_ABC
- JIT_F_OPT_CSE
- JIT_F_OPT_DCE
- JIT_F_OPT_DEFAULT
- JIT_F_OPT_DSE
- JIT_F_OPT_FIRST
- JIT_F_OPT_FOLD
- JIT_F_OPT_FUSE
- JIT_F_OPT_FWD

- JIT_F_OPT_LOOP
- JIT_F_OPT_MASK
- JIT_F_OPT_NARROW
- JIT_F_OPT_SINK
- JIT_F_PREFER_IMUL
- JIT_F_ROUND
- JIT_F_SQRT
- JIT_F_SSE2
- JIT_F_SSE3
- JIT_F_SSE4_1
- JIT_F_VFP
- JIT_F_VFPV2
- JIT_F_VFPV3
- JIT_PARAMDEF
- JIT_PARAMENUM
- JIT_PARAMENUM
- JIT_PARAMSTR
- JIT_P_STRING
- JIT_P_sizemcode_DEFAULT
- JIT_P_sizemcode_DEFAULT
- LJ_KSIMD
- PENALTY_MAX
- PENALTY_MIN
- PENALTY_RNDBITS
- PENALTY_SLOTS
- SNAP
- SNAPCOUNT_DONE
- SNAP_CONT
- SNAP_FRAME
- SNAP_MKFTSZ
- SNAP_MKPC
- SNAP_NORESTORE
- SNAP_SOFTFPNUM

- SNAP_TR
- _LJ_JIT_H
- gco2trace
- lj_needsplit
- lj_needsplit
- lj_resetsplit
- lj_resetsplit
- snap_isframe
- snap_pc
- snap_ref
- snap_setref
- snap_slot
- traceref

Source code

```
1  /*
2  ** Common definitions for the JIT compiler.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_JIT_H
7  #define _LJ_JIT_H
8
9  #include "lj_obj.h"
10 #include "lj_ir.h"
11
12 /* JIT engine flags. */
13 #define JIT_F_ON                0x00000001
14
15 /* CPU-specific JIT engine flags. */
16 #if LJ_TARGET_X86ORX64
17 #define JIT_F_SSE2                0x00000010
18 #define JIT_F_SSE3                0x00000020
19 #define JIT_F_SSE4_1                0x00000040
20 #define JIT_F_PREFER_IMUL        0x00000080
21 #define JIT_F_LEA_AGU                0x00000100
22
23 /* Names for the CPU-specific flags. Must match the order above. */
24 #define JIT_F_CPU_FIRST                JIT_F_SSE2
25 #define JIT_F_CPUSTRING                "\4SSE2\4SSE3\6SSE4.1\3AMD\4ATOM"
26 #elif LJ_TARGET_ARM
27 #define JIT_F_ARMV6_                0x00000010
28 #define JIT_F_ARMV6T2_                0x00000020
29 #define JIT_F_ARMV7                0x00000040
30 #define JIT_F_VFPV2                0x00000080
31 #define JIT_F_VFPV3                0x00000100
32
33 #define JIT_F_ARMV6                (JIT_F_ARMV6_|JIT_F_ARMV6T2_|JIT_F_ARMV7)
34 #define JIT_F_ARMV6T2                (JIT_F_ARMV6T2_|JIT_F_ARMV7)
35 #define JIT_F_VFP                (JIT_F_VFPV2|JIT_F_VFPV3)
36
37 /* Names for the CPU-specific flags. Must match the order above. */
38 #define JIT_F_CPU_FIRST                JIT_F_ARMV6_
39 #define JIT_F_CPUSTRING                "\5ARMv6\7ARMv6T2\5ARMv7\5VFPv2\5VFPv3"
40 #elif LJ_TARGET_PPC
41 #define JIT_F_SQRT                0x00000010
42 #define JIT_F_ROUND                0x00000020
43
```

```
/* Names for the CPU-specific flags. Must match the order above. */
#define JIT_F_CPU_FIRST               JIT_F_SQRT
#define JIT_F_CPUSTRING               "\4SQRT\5ROUND"
#elif LJ_TARGET_MIPS
#define JIT_F_MIPS32R2                0x00000010

/* Names for the CPU-specific flags. Must match the order above. */
#define JIT_F_CPU_FIRST               JIT_F_MIPS32R2
#define JIT_F_CPUSTRING               "\010MIPS32R2"
#else
#define JIT_F_CPU_FIRST               0
#define JIT_F_CPUSTRING               ""
#endif

/* Optimization flags. */
#define JIT_F_OPT_MASK                0x0fff0000

#define JIT_F_OPT_FOLD                0x00010000
#define JIT_F_OPT_CSE                 0x00020000
#define JIT_F_OPT_DCE                 0x00040000
#define JIT_F_OPT_FWD                 0x00080000
#define JIT_F_OPT_DSE                 0x00100000
#define JIT_F_OPT_NARROW        0x00200000
#define JIT_F_OPT_LOOP                0x00400000
#define JIT_F_OPT_ABC                 0x00800000
#define JIT_F_OPT_SINK                0x01000000
#define JIT_F_OPT_FUSE                0x02000000

/* Optimizations names for -O. Must match the order above. */
#define JIT_F_OPT_FIRST               JIT_F_OPT_FOLD
#define JIT_F_OPTSTRING       \
  "\4fold\3cse\3dce\3fwd\3dse\6narrow\4loop\3abc\4sink\4fuse"

/* Optimization levels set a fixed combination of flags. */
#define JIT_F_OPT_0       0
#define JIT_F_OPT_1       (JIT_F_OPT_FOLD|JIT_F_OPT_CSE|JIT_F_OPT_DCE)
#define JIT_F_OPT_2       (JIT_F_OPT_1|JIT_F_OPT_NARROW|JIT_F_OPT_LOOP)
#define JIT_F_OPT_3       (JIT_F_OPT_2|\
  JIT_F_OPT_FWD|JIT_F_OPT_DSE|JIT_F_OPT_ABC|JIT_F_OPT_SINK|JIT_F_OPT_FUSE)
#define JIT_F_OPT_DEFAULT       JIT_F_OPT_3

#if LJ_TARGET_WINDOWS || LJ_64
/* See: http://blogs.msdn.com/oldnewthing/archive/2003/10/08/55239.aspx */
#define JIT_P_sizemcode_DEFAULT               64
#else
/* Could go as low as 4K, but the mmap() overhead would be rather high. */
#define JIT_P_sizemcode_DEFAULT               32
#endif

/* Optimization parameters and their defaults. Length is a char in octal! */
#define JIT_PARAMDEF(_) \
  _(\010, maxtrace,       1000)       /* Max. # of traces in cache. */ \
  _(\011, maxrecord,       4000)       /* Max. # of recorded IR instructions. */ \
  _(\012, maxirconst,       500)       /* Max. # of IR constants of a trace. */ \
  _(\007, maxside,        100)       /* Max. # of side traces of a root trace. */ \
  _(\007, maxsnap,        500)       /* Max. # of snapshots for a trace. */ \
  _(\011, minstitch,        0)       /* Min. # of IR ins for a stitched trace. */ \
  \
  _(\007, hotloop,        56)       /* # of iter. to detect a hot loop/call. */ \
  _(\007, hotexit,        10)       /* # of taken exits to start a side trace. */ \
  _(\007, tryside,        4)       /* # of attempts to compile a side trace. */ \
  \
  _(\012, instunroll,        4)       /* Max. unroll for instable loops. */ \
  _(\012, loopunroll,        15)       /* Max. unroll for loop ops in side traces. */ \
  _(\012, callunroll,        3)       /* Max. unroll for recursive calls. */ \
  _(\011, recunroll,        2)       /* Min. unroll for true recursion. */ \
  \
  /* Size of each machine code area (in KBytes). */ \
  _(\011, sizemcode,       JIT_P_sizemcode_DEFAULT) \
  /* Max. total size of all machine code areas (in KBytes). */ \
  _(\010, maxmcode,       512) \
  /* End of list. */

enum {
#define JIT_PARAMENUM(len, name, value)       JIT_P_##name,
JIT_PARAMDEF(JIT_PARAMENUM)
```

```
#undef JIT_PARAMENUM
  JIT_P__MAX
};

#define JIT_PARAMSTR(len, name, value)        #len #name
#define JIT_P_STRING        JIT_PARAMDEF(JIT_PARAMSTR)

/* Trace compiler state. */
typedef enum {
  LJ_TRACE_IDLE,        /* Trace compiler idle. */
  LJ_TRACE_ACTIVE = 0x10,
  LJ_TRACE_RECORD,        /* Bytecode recording active. */
  LJ_TRACE_START,        /* New trace started. */
  LJ_TRACE_END,                /* End of trace. */
  LJ_TRACE_ASM,                /* Assemble trace. */
  LJ_TRACE_ERR                /* Trace aborted with error. */
} TraceState;

/* Post-processing action. */
typedef enum {
  LJ_POST_NONE,                /* No action. */
  LJ_POST_FIXCOMP,        /* Fixup comparison and emit pending guard. */
  LJ_POST_FIXGUARD,        /* Fixup and emit pending guard. */
  LJ_POST_FIXGUARDSNAP,        /* Fixup and emit pending guard and snapshot. */
  LJ_POST_FIXBOOL,        /* Fixup boolean result. */
  LJ_POST_FIXCONST,        /* Fixup constant results. */
  LJ_POST_FFRETRY        /* Suppress recording of retried fast functions. */
} PostProc;

/* Machine code type. */
#if LJ_TARGET_X86ORX64
typedef uint8_t MCode;
#else
typedef uint32_t MCode;
#endif

/* Stack snapshot header. */
typedef struct SnapShot {
  uint16_t mapofs;        /* Offset into snapshot map. */
  IRRef1 ref;                /* First IR ref for this snapshot. */
  uint8_t nslots;        /* Number of valid slots. */
  uint8_t topslot;        /* Maximum frame extent. */
  uint8_t nent;                /* Number of compressed entries. */
  uint8_t count;        /* Count of taken exits for this snapshot. */
} SnapShot;

#define SNAPCOUNT_DONE        255        /* Already compiled and linked a side trace. */

/* Compressed snapshot entry. */
typedef uint32_t SnapEntry;

#define SNAP_FRAME                0x010000        /* Frame slot. */
#define SNAP_CONT                0x020000        /* Continuation slot. */
#define SNAP_NORESTORE                0x040000        /* No need to restore slot. */
#define SNAP_SOFTFPNUM                0x080000        /* Soft-float number. */
LJ_STATIC_ASSERT(SNAP_FRAME == TREF_FRAME);
LJ_STATIC_ASSERT(SNAP_CONT == TREF_CONT);

#define SNAP(slot, flags, ref)        (((SnapEntry)(slot) << 24) + (flags) + (ref))
#define SNAP_TR(slot, tr) \
  (((SnapEntry)(slot) << 24) + ((tr) & (TREF_CONT|TREF_FRAME|TREF_REFMASK)))
#define SNAP_MKPC(pc)                ((SnapEntry)u32ptr(pc))
#define SNAP_MKFTSZ(ftsz)        ((SnapEntry)(ftsz))
#define snap_ref(sn)                ((sn) & 0xffff)
#define snap_slot(sn)                ((BCReg)((sn) >> 24))
#define snap_isframe(sn)        ((sn) & SNAP_FRAME)
#define snap_pc(sn)                ((const BCIns *)(uintptr_t)(sn))
#define snap_setref(sn, ref)        (((sn) & (0xffff0000&~SNAP_NORESTORE)) | (ref))

/* Snapshot and exit numbers. */
typedef uint32_t SnapNo;
typedef uint32_t ExitNo;

/* Trace number. */
typedef uint32_t TraceNo;        /* Used to pass around trace numbers. */
typedef uint16_t TraceNo1;        /* Stored trace number. */
```

```
196
197 /* Type of link. ORDER LJ_TRLINK */
198 typedef enum {
199   LJ_TRLINK_NONE,               /* Incomplete trace. No link, yet. */
200   LJ_TRLINK_ROOT,               /* Link to other root trace. */
201   LJ_TRLINK_LOOP,               /* Loop to same trace. */
202   LJ_TRLINK_TAILREC,               /* Tail-recursion. */
203   LJ_TRLINK_UPREC,              /* Up-recursion. */
204   LJ_TRLINK_DOWNREC,               /* Down-recursion. */
205   LJ_TRLINK_INTERP,             /* Fallback to interpreter. */
206   LJ_TRLINK_RETURN,             /* Return to interpreter. */
207   LJ_TRLINK_STITCH              /* Trace stitching. */
208 } TraceLink;
209
210 /* Trace object. */
211 typedef struct GCtrace {
212   GCHeader;
213   uint8_t topslot;        /* Top stack slot already checked to be allocated. */
214   uint8_t linktype;       /* Type of link. */
215   IRRef nins;               /* Next IR instruction. Biased with REF_BIAS. */
216 #if LJ_GC64
217   uint32_t unused_gc64;
218 #endif
219   GCRef gclist;
220   IRIns *ir;                /* IR instructions/constants. Biased with REF_BIAS. */
221   IRRef nk;               /* Lowest IR constant. Biased with REF_BIAS. */
222   uint16_t nsnap;        /* Number of snapshots. */
223   uint16_t nsnapmap;        /* Number of snapshot map elements. */
224   SnapShot *snap;        /* Snapshot array. */
225   SnapEntry *snapmap;        /* Snapshot map. */
226   GCRef startpt;        /* Starting prototype. */
227   MRef startpc;                /* Bytecode PC of starting instruction. */
228   BCIns startins;        /* Original bytecode of starting instruction. */
229   MSize szmcode;        /* Size of machine code. */
230   MCode *mcode;                /* Start of machine code. */
231   MSize mcloop;                /* Offset of loop start in machine code. */
232   uint16_t nchild;        /* Number of child traces (root trace only). */
233   uint16_t spadjust;        /* Stack pointer adjustment (offset in bytes). */
234   TraceNo1 traceno;        /* Trace number. */
235   TraceNo1 link;        /* Linked trace (or self for loops). */
236   TraceNo1 root;        /* Root trace of side trace (or 0 for root traces). */
237   TraceNo1 nextroot;        /* Next root trace for same prototype. */
238   TraceNo1 nextside;        /* Next side trace of same root trace. */
239   uint8_t sinktags;        /* Trace has SINK tags. */
240   uint8_t unused1;
241 #ifdef LUAJIT_USE_GDBJIT
242   void *gdbjit_entry;        /* GDB JIT entry. */
243 #endif
244 } GCtrace;
245
246 #define gco2trace(o)        check_exp((o)->gch.gct == ~LJ_TTRACE, (GCtrace *)(o))
247 #define traceref(J, n) \
248   check_exp((n)>0 && (MSize)(n)<J->sizetrace, (GCtrace *)gcref(J->trace[(n)]))
249
250 LJ_STATIC_ASSERT(offsetof(GChead, gclist) == offsetof(GCtrace, gclist));
251
252 static LJ_AINLINE MSize snap_nextofs(GCtrace *T, SnapShot *snap)
253 {
254   if (snap+1 == &T->snap[T->nsnap])
255     return T->nsnapmap;
256   else
257     return (snap+1)->mapofs;
258 }
259
260 /* Round-robin penalty cache for bytecodes leading to aborted traces. */
261 typedef struct HotPenalty {
262   MRef pc;                /* Starting bytecode PC. */
263   uint16_t val;                /* Penalty value, i.e. hotcount start. */
264   uint16_t reason;        /* Abort reason (really TraceErr). */
265 } HotPenalty;
266
267 #define PENALTY_SLOTS        64        /* Penalty cache slot. Must be a power of 2. */
268 #define PENALTY_MIN        (36*2)        /* Minimum penalty value. */
269 #define PENALTY_MAX        60000        /* Maximum penalty value. */
270 #define PENALTY_RNDBITS        4        /* # of random bits to add to penalty value. */
271
```

```
/* Round-robin backpropagation cache for narrowing conversions. */
typedef struct BPropEntry {
  IRRef1 key;		/* Key: original reference. */
  IRRef1 val;		/* Value: reference after conversion. */
  IRRef mode;		/* Mode for this entry (currently IRCONV_*). */
} BPropEntry;

/* Number of slots for the backpropagation cache. Must be a power of 2. */
#define BPROP_SLOTS	16

/* Scalar evolution analysis cache. */
typedef struct ScEvEntry {
  MRef pc;		/* Bytecode PC of FORI. */
  IRRef1 idx;		/* Index reference. */
  IRRef1 start;		/* Constant start reference. */
  IRRef1 stop;		/* Constant stop reference. */
  IRRef1 step;		/* Constant step reference. */
  IRType1 t;		/* Scalar type. */
  uint8_t dir;		/* Direction. 1: +, 0: -. */
} ScEvEntry;

/* 128 bit SIMD constants. */
enum {
  LJ_KSIMD_ABS,
  LJ_KSIMD_NEG,
  LJ_KSIMD__MAX
};

/* Get 16 byte aligned pointer to SIMD constant. */
#define LJ_KSIMD(J, n) \
  ((TValue *)(((intptr_t)&J->ksimd[2*(n)] + 15) & ~(intptr_t)15))

/* Set/reset flag to activate the SPLIT pass for the current trace. */
#if LJ_SOFTFP || (LJ_32 && LJ_HASFFI)
#define lj_needsplit(J)		(J->needsplit = 1)
#define lj_resetsplit(J)	(J->needsplit = 0)
#else
#define lj_needsplit(J)		UNUSED(J)
#define lj_resetsplit(J)	UNUSED(J)
#endif

/* Fold state is used to fold instructions on-the-fly. */
typedef struct FoldState {
  IRIns ins;		/* Currently emitted instruction. */
  IRIns left;		/* Instruction referenced by left operand. */
  IRIns right;		/* Instruction referenced by right operand. */
} FoldState;

/* JIT compiler state. */
typedef struct jit_State {
  GCtrace cur;		/* Current trace. */

  lua_State *L;		/* Current Lua state. */
  const BCIns *pc;	/* Current PC. */
  GCfunc *fn;		/* Current function. */
  GCproto *pt;		/* Current prototype. */
  TRef *base;		/* Current frame base, points into J->slots. */

  uint32_t flags;	/* JIT engine flags. */
  BCReg maxslot;	/* Relative to baseslot. */
  BCReg baseslot;	/* Current frame base, offset into J->slots. */

  uint8_t mergesnap;	/* Allowed to merge with next snapshot. */
  uint8_t needsnap;	/* Need snapshot before recording next bytecode. */
  IRType1 guardemit;	/* Accumulated IRT_GUARD for emitted instructions. */
  uint8_t bcskip;	/* Number of bytecode instructions to skip. */

  FoldState fold;	/* Fold state. */

  const BCIns *bc_min;	/* Start of allowed bytecode range for root trace. */
  MSize bc_extent;	/* Extent of the range. */

  TraceState state;	/* Trace compiler state. */

  int32_t instunroll;	/* Unroll counter for instable loops. */
  int32_t loopunroll;	/* Unroll counter for loop ops in side traces. */
```

```
  int32_t tailcalled;        /* Number of successive tailcalls. */
  int32_t framedepth;        /* Current frame depth. */
  int32_t retdepth;        /* Return frame depth (count of RETF). */

  MRef k64;                /* Pointer to chained array of 64 bit constants. */
  TValue ksimd[LJ_KSIMD__MAX*2+1];  /* 16 byte aligned SIMD constants. */

  IRIns *irbuf;                /* Temp. IR instruction buffer. Biased with REF_BIAS. */
  IRRef irtoplim;        /* Upper limit of instuction buffer (biased). */
  IRRef irbotlim;        /* Lower limit of instuction buffer (biased). */
  IRRef loopref;        /* Last loop reference or ref of final LOOP (or 0). */

  MSize sizesnap;        /* Size of temp. snapshot buffer. */
  SnapShot *snapbuf;        /* Temp. snapshot buffer. */
  SnapEntry *snapmapbuf;  /* Temp. snapshot map buffer. */
  MSize sizesnapmap;        /* Size of temp. snapshot map buffer. */

  PostProc postproc;        /* Required post-processing after execution. */
#if LJ_SOFTFP || (LJ_32 && LJ_HASFFI)
  int needsplit;        /* Need SPLIT pass. */
#endif

  GCRef *trace;                /* Array of traces. */
  TraceNo freetrace;        /* Start of scan for next free trace. */
  MSize sizetrace;        /* Size of trace array. */

  IRRef1 chain[IR__MAX];  /* IR instruction skip-list chain anchors. */
  TRef slot[LJ_MAX_JSLOTS+LJ_STACK_EXTRA];  /* Stack slot map. */

  int32_t param[JIT_P__MAX];  /* JIT engine parameters. */

  MCode *exitstubgroup[LJ_MAX_EXITSTUBGR];  /* Exit stub group addresses. */

  HotPenalty penalty[PENALTY_SLOTS];  /* Penalty slots. */
  uint32_t penaltyslot;        /* Round-robin index into penalty slots. */
  uint32_t prngstate;        /* PRNG state. */

  BPropEntry bpropcache[BPROP_SLOTS];  /* Backpropagation cache slots. */
  uint32_t bpropslot;        /* Round-robin index into bpropcache slots. */

  ScEvEntry scev;        /* Scalar evolution analysis cache slots. */

  const BCIns *startpc;        /* Bytecode PC of starting instruction. */
  TraceNo parent;        /* Parent of current side trace (0 for root traces). */
  ExitNo exitno;        /* Exit number in parent of current side trace. */

  BCIns *patchpc;        /* PC for pending re-patch. */
  BCIns patchins;        /* Instruction for pending re-patch. */

  int mcprot;                /* Protection of current mcode area. */
  MCode *mcarea;        /* Base of current mcode area. */
  MCode *mctop;                /* Top of current mcode area. */
  MCode *mcbot;                /* Bottom of current mcode area. */
  size_t szmcarea;        /* Size of current mcode area. */
  size_t szallmcarea;        /* Total size of all allocated mcode areas. */

  TValue errinfo;        /* Additional info element for trace errors. */

#if LJ_HASPROFILE
  GCproto *prev_pt;        /* Previous prototype. */
  BCLine prev_line;        /* Previous line. */
  int prof_mode;        /* Profiling mode: 0, 'f', 'l'. */
#endif
}
#if LJ_TARGET_ARM
LJ_ALIGN(16)                /* For DISPATCH-relative addresses in assembler part. */
#endif
jit_State;

/* Trivial PRNG e.g. used for penalty randomization. */
static LJ_AINLINE uint32_t LJ_PRNG_BITS(jit_State *J, int bits)
{
  /* Yes, this LCG is very weak, but that doesn't matter for our use case. */
  J->prngstate = J->prngstate * 1103515245 + 12345;
  return J->prngstate >> (32-bits);
}
```

```
424
425 #endif
```

One Level Up Top Level

src/lib_string.c - luajit-2.0-src

Global variables defined

- match_class_map

Data types defined

- MatchState
- MatchState

Functions defined

- LJLIB_ASM(string_char) LJLIB_REC(.)
- LJLIB_ASM(string_sub) LJLIB_REC(string_range 1)
- LJLIB_ASM(string_reverse) LJLIB_REC(string_op IRCALL_lj_buf_putstr_reverse)
- LJLIB_CF(string_rep) LJLIB_REC(.)
- LJLIB_CF(string_dump)
- LJLIB_CF(string_find) LJLIB_REC(.)
- LJLIB_CF(string_match)
- LJLIB_NOREG LJLIB_CF(string_gmatch_aux)
- LJLIB_CF(string_gmatch)
- LJLIB_CF(string_gsub)
- LJLIB_CF(string_format) LJLIB_REC(.)
- LJLIB_LUA(string_len) /*
- add_s
- add_value
- capture_to_close
- check_capture
- classend
- end_capture
- luaopen_string
- match
- match_capture
- match_class
- matchbalance
- matchbracketclass

- max_expand
- min_expand
- push_captures
- push_onecapture
- singlematch
- start_capture
- str_find_aux
- string_fmt_tostring
- writer_buf

Macros defined

- CAP_POSITION
- CAP_UNFINISHED
- LJLIB_MODULE_string
- LUA_LIB
- L_ESC
- lib_string_c
- uchar

Source code

```
/*
** String library.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Major portions taken verbatim or adapted from the Lua interpreter.
** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
*/

#define lib_string_c
 #define LUA_LIB

 #include "lua.h"
 #include "lauxlib.h"
 #include "lualib.h"

 #include "lj_obj.h"
 #include "lj_gc.h"
 #include "lj_err.h"
 #include "lj_buf.h"
 #include "lj_str.h"
 #include "lj_tab.h"
 #include "lj_meta.h"
 #include "lj_state.h"
 #include "lj_ff.h"
 #include "lj_bcdump.h"
 #include "lj_char.h"
 #include "lj_strfmt.h"
 #include "lj_lib.h"

 /* ------------------------------------------------------------------------ */

 #define LJLIB_MODULE_string
```

```
33
34 LJLIB_LUA(string_len) /*
35   function(s)
36     CHECK_str(s)
37     return #s
38   end
39 */
40
41 LJLIB_ASM(string_byte)                LJLIB_REC(string_range 0)
42 {
43   GCstr *s = lj_lib_checkstr(L, 1);
44   int32_t len = (int32_t)s->len;
45   int32_t start = lj_lib_optint(L, 2, 1);
46   int32_t stop = lj_lib_optint(L, 3, start);
47   int32_t n, i;
48   const unsigned char *p;
49   if (stop < 0) stop += len+1;
50   if (start < 0) start += len+1;
51   if (start <= 0) start = 1;
52   if (stop > len) stop = len;
53   if (start > stop) return FFH_RES(0);  /* Empty interval: return no results. */
54   start--;
55   n = stop - start;
56   if ((uint32_t)n > LUAI_MAXCSTACK)
57     lj_err_caller(L, LJ_ERR_STRSLC);
58   lj_state_checkstack(L, (MSize)n);
59   p = (const unsigned char *)strdata(s) + start;
60   for (i = 0; i < n; i++)
61     setintV(L->base + i-1-LJ_FR2, p[i]);
62   return FFH_RES(n);
63 }
64
65 LJLIB_ASM(string_char)                LJLIB_REC(.)
66 {
67   int i, nargs = (int)(L->top - L->base);
68   char *buf = lj_buf_tmp(L, (MSize)nargs);
69   for (i = 1; i <= nargs; i++) {
70     int32_t k = lj_lib_checkint(L, i);
71     if (!checku8(k))
72       lj_err_arg(L, i, LJ_ERR_BADVAL);
73     buf[i-1] = (char)k;
74   }
75   setstrV(L, L->base-1-LJ_FR2, lj_str_new(L, buf, (size_t)nargs));
76   return FFH_RES(1);
77 }
78
79 LJLIB_ASM(string_sub)                LJLIB_REC(string_range 1)
80 {
81   lj_lib_checkstr(L, 1);
82   lj_lib_checkint(L, 2);
83   setintV(L->base+2, lj_lib_optint(L, 3, -1));
84   return FFH_RETRY;
85 }
86
87 LJLIB_CF(string_rep)                LJLIB_REC(.)
88 {
89   GCstr *s = lj_lib_checkstr(L, 1);
90   int32_t rep = lj_lib_checkint(L, 2);
91   GCstr *sep = lj_lib_optstr(L, 3);
92   SBuf *sb = lj_buf_tmp_(L);
93   if (sep && rep > 1) {
94     GCstr *s2 = lj_buf_cat2str(L, sep, s);
95     lj_buf_reset(sb);
96     lj_buf_putstr(sb, s);
97     s = s2;
98     rep--;
99   }
100   sb = lj_buf_putstr_rep(sb, s, rep);
101   setstrV(L, L->top-1, lj_buf_str(L, sb));
102   lj_gc_check(L);
103   return 1;
104 }
105
106 LJLIB_ASM(string_reverse)  LJLIB_REC(string_op IRCALL_lj_buf_putstr_reverse)
107 {
108   lj_lib_checkstr(L, 1);
```

```
  return FFH_RETRY;
}
LJLIB_ASM_(string_lower)  LJLIB_REC(string_op IRCALL_lj_buf_putstr_lower)
LJLIB_ASM_(string_upper)  LJLIB_REC(string_op IRCALL_lj_buf_putstr_upper)

/* ------------------------------------------------------------------------ */

static int writer_buf(lua_State *L, const void *p, size_t size, void *sb)
{
  lj_buf_putmem((SBuf *)sb, p, (MSize)size);
  UNUSED(L);
  return 0;
}

LJLIB_CF(string_dump)
{
  GCfunc *fn = lj_lib_checkfunc(L, 1);
  int strip = L->base+1 < L->top && tvistruecond(L->base+1);
  SBuf *sb = lj_buf_tmp_(L);  /* Assumes lj_bcwrite() doesn't use tmpbuf. */
  L->top = L->base+1;
  if (!isluafunc(fn) || lj_bcwrite(L, funcproto(fn), writer_buf, sb, strip))
    lj_err_caller(L, LJ_ERR_STRDUMP);
  setstrV(L, L->top-1, lj_buf_str(L, sb));
  lj_gc_check(L);
  return 1;
}

/* ------------------------------------------------------------------------ */

/* macro to `unsign' a character */
#define uchar(c)        ((unsigned char)(c))

#define CAP_UNFINISHED        (-1)
#define CAP_POSITION        (-2)

typedef struct MatchState {
  const char *src_init;  /* init of source string */
  const char *src_end;  /* end (`\0') of source string */
  lua_State *L;
  int level;  /* total number of captures (finished or unfinished) */
  int depth;
  struct {
    const char *init;
    ptrdiff_t len;
  } capture[LUA_MAXCAPTURES];
} MatchState;

#define L_ESC                '%'

static int check_capture(MatchState *ms, int l)
{
  l -= '1';
  if (l < 0 || l >= ms->level || ms->capture[l].len == CAP_UNFINISHED)
    lj_err_caller(ms->L, LJ_ERR_STRCAPI);
  return l;
}

static int capture_to_close(MatchState *ms)
{
  int level = ms->level;
  for (level--; level>=0; level--)
    if (ms->capture[level].len == CAP_UNFINISHED) return level;
  lj_err_caller(ms->L, LJ_ERR_STRPATC);
  return 0;  /* unreachable */
}

static const char *classend(MatchState *ms, const char *p)
{
  switch (*p++) {
  case L_ESC:
    if (*p == '\0')
      lj_err_caller(ms->L, LJ_ERR_STRPATE);
    return p+1;
  case '[':
    if (*p == '^') p++;
    do {  /* look for a `]' */
```

```
      if (*p == '\0')
        lj_err_caller(ms->L, LJ_ERR_STRPATM);
      if (*(p++) == L_ESC && *p != '\0')
        p++;  /* skip escapes (e.g. `%]') */
    } while (*p != ']');
    return p+1;
  default:
    return p;
  }
}

static const unsigned char match_class_map[32] = {
  0,LJ_CHAR_ALPHA,0,LJ_CHAR_CNTRL,LJ_CHAR_DIGIT,0,0,LJ_CHAR_GRAPH,0,0,0,0,
  LJ_CHAR_LOWER,0,0,0,LJ_CHAR_PUNCT,0,0,LJ_CHAR_SPACE,0,
  LJ_CHAR_UPPER,0,LJ_CHAR_ALNUM,LJ_CHAR_XDIGIT,0,0,0,0,0,0,0,0
};

static int match_class(int c, int cl)
{
  if ((cl & 0xc0) == 0x40) {
    int t = match_class_map[(cl&0x1f)];
    if (t) {
      t = lj_char_isa(c, t);
      return (cl & 0x20) ? t : !t;
    }
    if (cl == 'z') return c == 0;
    if (cl == 'Z') return c != 0;
  }
  return (cl == c);
}

static int matchbracketclass(int c, const char *p, const char *ec)
{
  int sig = 1;
  if (*(p+1) == '^') {
    sig = 0;
    p++;  /* skip the `^' */
  }
  while (++p < ec) {
    if (*p == L_ESC) {
      p++;
      if (match_class(c, uchar(*p)))
        return sig;
    }
    else if ((*(p+1) == '-') && (p+2 < ec)) {
      p+=2;
      if (uchar(*(p-2)) <= c && c <= uchar(*p))
        return sig;
    }
    else if (uchar(*p) == c) return sig;
  }
  return !sig;
}

static int singlematch(int c, const char *p, const char *ep)
{
  switch (*p) {
  case '.': return 1;  /* matches any char */
  case L_ESC: return match_class(c, uchar(*(p+1)));
  case '[': return matchbracketclass(c, p, ep-1);
  default:  return (uchar(*p) == c);
  }
}

static const char *match(MatchState *ms, const char *s, const char *p);

static const char *matchbalance(MatchState *ms, const char *s, const char *p)
{
  if (*p == 0 || *(p+1) == 0)
    lj_err_caller(ms->L, LJ_ERR_STRPATU);
  if (*s != *p) {
    return NULL;
  } else {
    int b = *p;
    int e = *(p+1);
    int cont = 1;
```

```
    while (++s < ms->src_end) {
      if (*s == e) {
        if (--cont == 0) return s+1;
      } else if (*s == b) {
        cont++;
      }
    }
  }
  return NULL;  /* string ends out of balance */
}

static const char *max_expand(MatchState *ms, const char *s,
                              const char *p, const char *ep)
{
  ptrdiff_t i = 0;  /* counts maximum expand for item */
  while ((s+i)<ms->src_end && singlematch(uchar(*(s+i)), p, ep))
    i++;
  /* keeps trying to match with the maximum repetitions */
  while (i>=0) {
    const char *res = match(ms, (s+i), ep+1);
    if (res) return res;
    i--;  /* else didn't match; reduce 1 repetition to try again */
  }
  return NULL;
}

static const char *min_expand(MatchState *ms, const char *s,
                              const char *p, const char *ep)
{
  for (;;) {
    const char *res = match(ms, s, ep+1);
    if (res != NULL)
      return res;
    else if (s<ms->src_end && singlematch(uchar(*s), p, ep))
      s++;  /* try with one more repetition */
    else
      return NULL;
  }
}

static const char *start_capture(MatchState *ms, const char *s,
                                 const char *p, int what)
{
  const char *res;
  int level = ms->level;
  if (level >= LUA_MAXCAPTURES) lj_err_caller(ms->L, LJ_ERR_STRCAPN);
  ms->capture[level].init = s;
  ms->capture[level].len = what;
  ms->level = level+1;
  if ((res=match(ms, s, p)) == NULL)  /* match failed? */
    ms->level--;  /* undo capture */
  return res;
}

static const char *end_capture(MatchState *ms, const char *s,
                               const char *p)
{
  int l = capture_to_close(ms);
  const char *res;
  ms->capture[l].len = s - ms->capture[l].init;  /* close capture */
  if ((res = match(ms, s, p)) == NULL)  /* match failed? */
    ms->capture[l].len = CAP_UNFINISHED;  /* undo capture */
  return res;
}

static const char *match_capture(MatchState *ms, const char *s, int l)
{
  size_t len;
  l = check_capture(ms, l);
  len = (size_t)ms->capture[l].len;
  if ((size_t)(ms->src_end-s) >= len &&
      memcmp(ms->capture[l].init, s, len) == 0)
    return s+len;
  else
    return NULL;
}
```

```
337
338 static const char *match(MatchState *ms, const char *s, const char *p)
339 {
340   if (++ms->depth > LJ_MAX_XLEVEL)
341     lj_err_caller(ms->L, LJ_ERR_STRPATX);
342   init: /* using goto's to optimize tail recursion */
343   switch (*p) {
344   case '(':  /* start capture */
345     if (*(p+1) == ')')  /* position capture? */
346       s = start_capture(ms, s, p+2, CAP_POSITION);
347     else
348       s = start_capture(ms, s, p+1, CAP_UNFINISHED);
349     break;
350   case ')':  /* end capture */
351     s = end_capture(ms, s, p+1);
352     break;
353   case L_ESC:
354     switch (*(p+1)) {
355     case 'b':  /* balanced string? */
356       s = matchbalance(ms, s, p+2);
357       if (s == NULL) break;
358       p+=4;
359       goto init;  /* else s = match(ms, s, p+4); */
360     case 'f': {  /* frontier? */
361       const char *ep; char previous;
362       p += 2;
363       if (*p != '[')
364         lj_err_caller(ms->L, LJ_ERR_STRPATB);
365       ep = classend(ms, p);  /* points to what is next */
366       previous = (s == ms->src_init) ? '\0' : *(s-1);
367       if (matchbracketclass(uchar(previous), p, ep-1) ||
368          !matchbracketclass(uchar(*s), p, ep-1)) { s = NULL; break; }
369       p=ep;
370       goto init;  /* else s = match(ms, s, ep); */
371       }
372     default:
373       if (lj_char_isdigit(uchar(*(p+1)))) {  /* capture results (%0-%9)? */
374         s = match_capture(ms, s, uchar(*(p+1)));
375         if (s == NULL) break;
376         p+=2;
377         goto init;  /* else s = match(ms, s, p+2) */
378       }
379       goto dflt;  /* case default */
380     }
381     break;
382   case '\0':  /* end of pattern */
383     break;  /* match succeeded */
384   case '$':
385     /* is the `$' the last char in pattern? */
386     if (*(p+1) != '\0') goto dflt;
387     if (s != ms->src_end) s = NULL;  /* check end of string */
388     break;
389   default: dflt: {  /* it is a pattern item */
390     const char *ep = classend(ms, p);  /* points to what is next */
391     int m = s<ms->src_end && singlematch(uchar(*s), p, ep);
392     switch (*ep) {
393     case '?': {  /* optional */
394       const char *res;
395       if (m && ((res=match(ms, s+1, ep+1)) != NULL)) {
396         s = res;
397         break;
398       }
399       p=ep+1;
400       goto init;  /* else s = match(ms, s, ep+1); */
401       }
402     case '*':  /* 0 or more repetitions */
403       s = max_expand(ms, s, p, ep);
404       break;
405     case '+':  /* 1 or more repetitions */
406       s = (m ? max_expand(ms, s+1, p, ep) : NULL);
407       break;
408     case '-':  /* 0 or more repetitions (minimum) */
409       s = min_expand(ms, s, p, ep);
410       break;
411     default:
412       if (m) { s++; p=ep; goto init; }  /* else s = match(ms, s+1, ep); */
```

```
      s = NULL;
      break;
    }
    break;
    }
  }
  ms->depth--;
  return s;
}

static void push_onecapture(MatchState *ms, int i, const char *s, const char *e)
{
  if (i >= ms->level) {
    if (i == 0)  /* ms->level == 0, too */
      lua_pushlstring(ms->L, s, (size_t)(e - s));  /* add whole match */
    else
      lj_err_caller(ms->L, LJ_ERR_STRCAPI);
  } else {
    ptrdiff_t l = ms->capture[i].len;
    if (l == CAP_UNFINISHED) lj_err_caller(ms->L, LJ_ERR_STRCAPU);
    if (l == CAP_POSITION)
      lua_pushinteger(ms->L, ms->capture[i].init - ms->src_init + 1);
    else
      lua_pushlstring(ms->L, ms->capture[i].init, (size_t)l);
  }
}

static int push_captures(MatchState *ms, const char *s, const char *e)
{
  int i;
  int nlevels = (ms->level == 0 && s) ? 1 : ms->level;
  luaL_checkstack(ms->L, nlevels, "too many captures");
  for (i = 0; i < nlevels; i++)
    push_onecapture(ms, i, s, e);
  return nlevels;  /* number of strings pushed */
}

static int str_find_aux(lua_State *L, int find)
{
  GCstr *s = lj_lib_checkstr(L, 1);
  GCstr *p = lj_lib_checkstr(L, 2);
  int32_t start = lj_lib_optint(L, 3, 1);
  MSize st;
  if (start < 0) start += (int32_t)s->len; else start--;
  if (start < 0) start = 0;
  st = (MSize)start;
  if (st > s->len) {
#if LJ_52
    setnilV(L->top-1);
    return 1;
#else
    st = s->len;
#endif
  }
  if (find && ((L->base+3 < L->top && tvistruecond(L->base+3)) ||
               !lj_str_haspattern(p))) {  /* Search for fixed string. */
    const char *q = lj_str_find(strdata(s)+st, strdata(p), s->len-st, p->len);
    if (q) {
      setintV(L->top-2, (int32_t)(q-strdata(s)) + 1);
      setintV(L->top-1, (int32_t)(q-strdata(s)) + (int32_t)p->len);
      return 2;
    }
  } else {  /* Search for pattern. */
    MatchState ms;
    const char *pstr = strdata(p);
    const char *sstr = strdata(s) + st;
    int anchor = 0;
    if (*pstr == '^') { pstr++; anchor = 1; }
    ms.L = L;
    ms.src_init = strdata(s);
    ms.src_end = strdata(s) + s->len;
    do {  /* Loop through string and try to match the pattern. */
      const char *q;
      ms.level = ms.depth = 0;
      q = match(&ms, sstr, pstr);
      if (q) {
```

```
        if (find) {
          setintV(L->top++, (int32_t)(sstr-(strdata(s)-1)));
          setintV(L->top++, (int32_t)(q-strdata(s)));
          return push_captures(&ms, NULL, NULL) + 2;
        } else {
          return push_captures(&ms, sstr, q);
        }
      }
    } while (sstr++ < ms.src_end && !anchor);
  }
  setnilV(L->top-1);  /* Not found. */
  return 1;
}

LJLIB_CF(string_find)               LJLIB_REC(.)
{
  return str_find_aux(L, 1);
}

LJLIB_CF(string_match)
{
  return str_find_aux(L, 0);
}

LJLIB_NOREG LJLIB_CF(string_gmatch_aux)
{
  const char *p = strVdata(lj_lib_upvalue(L, 2));
  GCstr *str = strV(lj_lib_upvalue(L, 1));
  const char *s = strdata(str);
  TValue *tvpos = lj_lib_upvalue(L, 3);
  const char *src = s + tvpos->u32.lo;
  MatchState ms;
  ms.L = L;
  ms.src_init = s;
  ms.src_end = s + str->len;
  for (; src <= ms.src_end; src++) {
    const char *e;
    ms.level = ms.depth = 0;
    if ((e = match(&ms, src, p)) != NULL) {
      int32_t pos = (int32_t)(e - s);
      if (e == src) pos++;  /* Ensure progress for empty match. */
      tvpos->u32.lo = (uint32_t)pos;
      return push_captures(&ms, src, e);
    }
  }
  return 0;  /* not found */
}

LJLIB_CF(string_gmatch)
{
  lj_lib_checkstr(L, 1);
  lj_lib_checkstr(L, 2);
  L->top = L->base+3;
  (L->top-1)->u64 = 0;
  lj_lib_pushcc(L, lj_cf_string_gmatch_aux, FF_string_gmatch_aux, 3);
  return 1;
}

static void add_s(MatchState *ms, luaL_Buffer *b, const char *s, const char *e)
{
  size_t l, i;
  const char *news = lua_tolstring(ms->L, 3, &l);
  for (i = 0; i < l; i++) {
    if (news[i] != L_ESC) {
      luaL_addchar(b, news[i]);
    } else {
      i++;  /* skip ESC */
      if (!lj_char_isdigit(uchar(news[i]))) {
        luaL_addchar(b, news[i]);
      } else if (news[i] == '0') {
        luaL_addlstring(b, s, (size_t)(e - s));
      } else {
        push_onecapture(ms, news[i] - '1', s, e);
        luaL_addvalue(b);  /* add capture to accumulated result */
      }
    }
```

```
  }
}

static void add_value(MatchState *ms, luaL_Buffer *b,
                      const char *s, const char *e)
{
  lua_State *L = ms->L;
  switch (lua_type(L, 3)) {
    case LUA_TNUMBER:
    case LUA_TSTRING: {
      add_s(ms, b, s, e);
      return;
    }
    case LUA_TFUNCTION: {
      int n;
      lua_pushvalue(L, 3);
      n = push_captures(ms, s, e);
      lua_call(L, n, 1);
      break;
    }
    case LUA_TTABLE: {
      push_onecapture(ms, 0, s, e);
      lua_gettable(L, 3);
      break;
    }
  }
  if (!lua_toboolean(L, -1)) {  /* nil or false? */
    lua_pop(L, 1);
    lua_pushlstring(L, s, (size_t)(e - s));  /* keep original text */
  } else if (!lua_isstring(L, -1)) {
    lj_err_callerv(L, LJ_ERR_STRGSRV, luaL_typename(L, -1));
  }
  luaL_addvalue(b);  /* add result to accumulator */
}

LJLIB_CF(string_gsub)
{
  size_t srcl;
  const char *src = luaL_checklstring(L, 1, &srcl);
  const char *p = luaL_checkstring(L, 2);
  int  tr = lua_type(L, 3);
  int max_s = luaL_optint(L, 4, (int)(srcl+1));
  int anchor = (*p == '^') ? (p++, 1) : 0;
  int n = 0;
  MatchState ms;
  luaL_Buffer b;
  if (!(tr == LUA_TNUMBER || tr == LUA_TSTRING ||
        tr == LUA_TFUNCTION || tr == LUA_TTABLE))
    lj_err_arg(L, 3, LJ_ERR_NOSFT);
  luaL_buffinit(L, &b);
  ms.L = L;
  ms.src_init = src;
  ms.src_end = src+srcl;
  while (n < max_s) {
    const char *e;
    ms.level = ms.depth = 0;
    e = match(&ms, src, p);
    if (e) {
      n++;
      add_value(&ms, &b, src, e);
    }
    if (e && e>src) /* non empty match? */
      src = e;  /* skip it */
    else if (src < ms.src_end)
      luaL_addchar(&b, *src++);
    else
      break;
    if (anchor)
      break;
  }
  luaL_addlstring(&b, src, (size_t)(ms.src_end-src));
  luaL_pushresult(&b);
  lua_pushinteger(L, n);  /* number of substitutions */
  return 2;
}
```

```c
/* ------------------------------------------------------------------------ */

/* Emulate tostring() inline. */
static GCstr *string_fmt_tostring(lua_State *L, int arg, int retry)
{
  TValue *o = L->base+arg-1;
  cTValue *mo;
  lua_assert(o < L->top);  /* Caller already checks for existence. */
  if (LJ_LIKELY(tvisstr(o)))
    return strV(o);
  if (retry != 2 && !tvisnil(mo = lj_meta_lookup(L, o, MM_tostring))) {
    copyTV(L, L->top++, mo);
    copyTV(L, L->top++, o);
    lua_call(L, 1, 1);
    copyTV(L, L->base+arg-1, --L->top);
    return NULL;  /* Buffer may be overwritten, retry. */
  }
  return lj_strfmt_obj(L, o);
}

LJLIB_CF(string_format)               LJLIB_REC(.)
{
  int arg, top = (int)(L->top - L->base);
  GCstr *fmt;
  SBuf *sb;
  FormatState fs;
  SFormat sf;
  int retry = 0;
again:
  arg = 1;
  sb = lj_buf_tmp_(L);
  fmt = lj_lib_checkstr(L, arg);
  lj_strfmt_init(&fs, strdata(fmt), fmt->len);
  while ((sf = lj_strfmt_parse(&fs)) != STRFMT_EOF) {
    if (sf == STRFMT_LIT) {
      lj_buf_putmem(sb, fs.str, fs.len);
    } else if (sf == STRFMT_ERR) {
      lj_err_callerv(L, LJ_ERR_STRFMT, strdata(lj_str_new(L, fs.str, fs.len)));
    } else {
      if (++arg > top)
        luaL_argerror(L, arg, lj_obj_typename[0]);
      switch (STRFMT_TYPE(sf)) {
      case STRFMT_INT:
        if (tvisint(L->base+arg-1)) {
          int32_t k = intV(L->base+arg-1);
          if (sf == STRFMT_INT)
            lj_strfmt_putint(sb, k);  /* Shortcut for plain %d. */
          else
            lj_strfmt_putfxint(sb, sf, k);
        } else {
          lj_strfmt_putfnum_int(sb, sf, lj_lib_checknum(L, arg));
        }
        break;
      case STRFMT_UINT:
        if (tvisint(L->base+arg-1))
          lj_strfmt_putfxint(sb, sf, intV(L->base+arg-1));
        else
          lj_strfmt_putfnum_uint(sb, sf, lj_lib_checknum(L, arg));
        break;
      case STRFMT_NUM:
        lj_strfmt_putfnum(sb, sf, lj_lib_checknum(L, arg));
        break;
      case STRFMT_STR: {
        GCstr *str = string_fmt_tostring(L, arg, retry);
        if (str == NULL)
          retry = 1;
        else if ((sf & STRFMT_T_QUOTED))
          lj_strfmt_putquoted(sb, str);  /* No formatting. */
        else
          lj_strfmt_putfstr(sb, sf, str);
        break;
        }
      case STRFMT_CHAR:
        lj_strfmt_putfchar(sb, sf, lj_lib_checkint(L, arg));
        break;
      case STRFMT_PTR:  /* No formatting. */
```

```
717         lj_strfmt_putptr(sb, lj_obj_ptr(L->base+arg-1));
718         break;
719       default:
720         lua_assert(0);
721         break;
722       }
723     }
724   }
725   if (retry++ == 1) goto again;
726   setstrV(L, L->top-1, lj_buf_str(L, sb));
727   lj_gc_check(L);
728   return 1;
729 }
730
731 /* ------------------------------------------------------------------------ */
732
733 #include "lj_libdef.h"
734
735 LUALIB_API int luaopen_string(lua_State *L)
736 {
737   GCtab *mt;
738   global_State *g;
739   LJ_LIB_REG(L, LUA_STRLIBNAME, string);
740 #if defined(LUA_COMPAT_GFIND) && !LJ_52
741   lua_getfield(L, -1, "gmatch");
742   lua_setfield(L, -2, "gfind");
743 #endif
744   mt = lj_tab_new(L, 0, 1);
745   /* NOBARRIER: basemt is a GC root. */
746   g = G(L);
747   setgcref(basemt_it(g, LJ_TSTR), obj2gco(mt));
748   settabV(L, lj_tab_setstr(L, mt, mmname_str(g, MM_index)), tabV(L->top-1));
749   mt->nomm = (uint8_t)(~(1u<<MM_index));
750   return 1;
751 }
752
```

One Level Up Top Level

src/lib_base.c - luajit-2.0-src

Functions defined

- LJLIB_ASM(assert) LJLIB_REC(.)
- LJLIB_NOREGUV LJLIB_ASM(ipairs_aux) LJLIB_REC(.)
- LJLIB_ASM(rawget) LJLIB_REC(.)
- LJLIB_ASM(tonumber) LJLIB_REC(.)
- LJLIB_ASM(tostring) LJLIB_REC(.)
- LJLIB_ASM(pcall) LJLIB_REC(.)
- LJLIB_ASM(coroutine_yield)
- LJLIB_ASM(coroutine_resume)
- LJLIB_NOREG LJLIB_ASM(coroutine_wrap_aux)
- LJLIB_ASM_(getmetatable) LJLIB_REC(.)
- LJLIB_CF(getfenv) LJLIB_REC(.)
- LJLIB_CF(setfenv)
- LJLIB_CF(rawset) LJLIB_REC(.)
- LJLIB_CF(rawequal) LJLIB_REC(.)
- LJLIB_CF(rawlen) LJLIB_REC(.)
- LJLIB_CF(unpack)
- LJLIB_CF(select) LJLIB_REC(.)
- LJLIB_CF(error)
- LJLIB_CF(loadfile)
- LJLIB_CF(load)
- LJLIB_CF(loadstring)
- LJLIB_CF(dofile)
- LJLIB_CF(gcinfo)
- LJLIB_CF(collectgarbage)
- LJLIB_CF(newproxy)
- LJLIB_CF(print)
- LJLIB_CF(coroutine_running)
- LJLIB_CF(coroutine_create)
- LJLIB_CF(coroutine_wrap)
- LJLIB_PUSH(lastcl)

- LJLIB_PUSH(lastcl)
- LJLIB_SET(_VERSION)
- ffh_pairs
- ffh_resume
- lj_ffh_coroutine_wrap_err
- load_aux
- luaopen_base
- newproxy_weaktable
- reader_func
- setpc_wrap_aux

Macros defined

- LJLIB_MODULE_base
- LJLIB_MODULE_coroutine
- LUA_LIB
- ffh_pairs
- lib_base_c

Source code

```
/*
** Base and coroutine library.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Major portions taken verbatim or adapted from the Lua interpreter.
** Copyright (C) 1994-2011 Lua.org, PUC-Rio. See Copyright Notice in lua.h
*/

#include <stdio.h>

 #define lib_base_c
 #define LUA_LIB

 #include "lua.h"
 #include "lauxlib.h"
 #include "lualib.h"

 #include "lj_obj.h"
 #include "lj_gc.h"
 #include "lj_err.h"
 #include "lj_debug.h"
 #include "lj_str.h"
 #include "lj_tab.h"
 #include "lj_meta.h"
 #include "lj_state.h"
 #if LJ_HASFFI
 #include "lj_ctype.h"
 #include "lj_cconv.h"
 #endif
 #include "lj_bc.h"
 #include "lj_ff.h"
 #include "lj_dispatch.h"
 #include "lj_char.h"
 #include "lj_strscan.h"
 #include "lj_strfmt.h"
```

```c
#include "lj_lib.h"

/* -- Base library: checks ---------------------------------------------- */

#define LJLIB_MODULE_base

LJLIB_ASM(assert)               LJLIB_REC(.)
{
  GCstr *s;
  lj_lib_checkany(L, 1);
  s = lj_lib_optstr(L, 2);
  if (s)
    lj_err_callermsg(L, strdata(s));
  else
    lj_err_caller(L, LJ_ERR_ASSERT);
  return FFH_UNREACHABLE;
}

/* ORDER LJ_T */
LJLIB_PUSH("nil")
LJLIB_PUSH("boolean")
LJLIB_PUSH(top-1)  /* boolean */
LJLIB_PUSH("userdata")
LJLIB_PUSH("string")
LJLIB_PUSH("upval")
LJLIB_PUSH("thread")
LJLIB_PUSH("proto")
LJLIB_PUSH("function")
LJLIB_PUSH("trace")
LJLIB_PUSH("cdata")
LJLIB_PUSH("table")
LJLIB_PUSH(top-9)  /* userdata */
LJLIB_PUSH("number")
LJLIB_ASM_(type)                LJLIB_REC(.)
/* Recycle the lj_lib_checkany(L, 1) from assert. */

/* -- Base library: iterators ------------------------------------------- */

/* This solves a circular dependency problem -- change FF_next_N as needed. */
LJ_STATIC_ASSERT((int)FF_next == FF_next_N);

LJLIB_ASM(next)
{
  lj_lib_checktab(L, 1);
  return FFH_UNREACHABLE;
}

#if LJ_52 || LJ_HASFFI
static int ffh_pairs(lua_State *L, MMS mm)
{
  TValue *o = lj_lib_checkany(L, 1);
  cTValue *mo = lj_meta_lookup(L, o, mm);
  if ((LJ_52 || tviscdata(o)) && !tvisnil(mo)) {
    L->top = o+1;  /* Only keep one argument. */
    copyTV(L, L->base-1-LJ_FR2, mo);  /* Replace callable. */
    return FFH_TAILCALL;
  } else {
    if (!tvistab(o)) lj_err_argt(L, 1, LUA_TTABLE);
    if (LJ_FR2) { copyTV(L, o-1, o); o--; }
    setfuncV(L, o-1, funcV(lj_lib_upvalue(L, 1)));
    if (mm == MM_pairs) setnilV(o+1); else setintV(o+1, 0);
    return FFH_RES(3);
  }
}
#else
#define ffh_pairs(L, mm)        (lj_lib_checktab(L, 1), FFH_UNREACHABLE)
#endif

LJLIB_PUSH(lastcl)
LJLIB_ASM(pairs)                LJLIB_REC(xpairs 0)
{
  return ffh_pairs(L, MM_pairs);
}

LJLIB_NOREGUV LJLIB_ASM(ipairs_aux)        LJLIB_REC(.)
{
```

```
  lj_lib_checktab(L, 1);
  lj_lib_checkint(L, 2);
  return FFH_UNREACHABLE;
}

LJLIB_PUSH(lastcl)
LJLIB_ASM(ipairs)               LJLIB_REC(xpairs 1)
{
  return ffh_pairs(L, MM_ipairs);
}

/* -- Base library: getters and setters ---------------------------------- */

LJLIB_ASM_(getmetatable)        LJLIB_REC(.)
/* Recycle the lj_lib_checkany(L, 1) from assert. */

LJLIB_ASM(setmetatable)               LJLIB_REC(.)
{
  GCtab *t = lj_lib_checktab(L, 1);
  GCtab *mt = lj_lib_checktabornil(L, 2);
  if (!tvisnil(lj_meta_lookup(L, L->base, MM_metatable)))
    lj_err_caller(L, LJ_ERR_PROTMT);
  setgcref(t->metatable, obj2gco(mt));
  if (mt) { lj_gc_objbarriert(L, t, mt); }
  settabV(L, L->base-1-LJ_FR2, t);
  return FFH_RES(1);
}

LJLIB_CF(getfenv)               LJLIB_REC(.)
{
  GCfunc *fn;
  cTValue *o = L->base;
  if (!(o < L->top && tvisfunc(o))) {
    int level = lj_lib_optint(L, 1, 1);
    o = lj_debug_frame(L, level, &level);
    if (o == NULL)
      lj_err_arg(L, 1, LJ_ERR_INVLVL);
    if (LJ_FR2) o--;
  }
  fn = &gcval(o)->fn;
  settabV(L, L->top++, isluafunc(fn) ? tabref(fn->l.env) : tabref(L->env));
  return 1;
}

LJLIB_CF(setfenv)
{
  GCfunc *fn;
  GCtab *t = lj_lib_checktab(L, 2);
  cTValue *o = L->base;
  if (!(o < L->top && tvisfunc(o))) {
    int level = lj_lib_checkint(L, 1);
    if (level == 0) {
      /* NOBARRIER: A thread (i.e. L) is never black. */
      setgcref(L->env, obj2gco(t));
      return 0;
    }
    o = lj_debug_frame(L, level, &level);
    if (o == NULL)
      lj_err_arg(L, 1, LJ_ERR_INVLVL);
    if (LJ_FR2) o--;
  }
  fn = &gcval(o)->fn;
  if (!isluafunc(fn))
    lj_err_caller(L, LJ_ERR_SETFENV);
  setgcref(fn->l.env, obj2gco(t));
  lj_gc_objbarrier(L, obj2gco(fn), t);
  setfuncV(L, L->top++, fn);
  return 1;
}

LJLIB_ASM(rawget)               LJLIB_REC(.)
{
  lj_lib_checktab(L, 1);
  lj_lib_checkany(L, 2);
  return FFH_UNREACHABLE;
}
```

```
188
189 LJLIB_CF(rawset)                LJLIB_REC(.)
190 {
191   lj_lib_checktab(L, 1);
192   lj_lib_checkany(L, 2);
193   L->top = 1+lj_lib_checkany(L, 3);
194   lua_rawset(L, 1);
195   return 1;
196 }
197
198 LJLIB_CF(rawequal)                LJLIB_REC(.)
199 {
200   cTValue *o1 = lj_lib_checkany(L, 1);
201   cTValue *o2 = lj_lib_checkany(L, 2);
202   setboolV(L->top-1, lj_obj_equal(o1, o2));
203   return 1;
204 }
205
206 #if LJ_52
207 LJLIB_CF(rawlen)                LJLIB_REC(.)
208 {
209   cTValue *o = L->base;
210   int32_t len;
211   if (L->top > o && tvisstr(o))
212     len = (int32_t)strV(o)->len;
213   else
214     len = (int32_t)lj_tab_len(lj_lib_checktab(L, 1));
215   setintV(L->top-1, len);
216   return 1;
217 }
218 #endif
219
220 LJLIB_CF(unpack)
221 {
222   GCtab *t = lj_lib_checktab(L, 1);
223   int32_t n, i = lj_lib_optint(L, 2, 1);
224   int32_t e = (L->base+3-1 < L->top && !tvisnil(L->base+3-1)) ?
225               lj_lib_checkint(L, 3) : (int32_t)lj_tab_len(t);
226   if (i > e) return 0;
227   n = e - i + 1;
228   if (n <= 0 || !lua_checkstack(L, n))
229     lj_err_caller(L, LJ_ERR_UNPACK);
230   do {
231     cTValue *tv = lj_tab_getint(t, i);
232     if (tv) {
233       copyTV(L, L->top++, tv);
234     } else {
235       setnilV(L->top++);
236     }
237   } while (i++ < e);
238   return n;
239 }
240
241 LJLIB_CF(select)                LJLIB_REC(.)
242 {
243   int32_t n = (int32_t)(L->top - L->base);
244   if (n >= 1 && tvisstr(L->base) && *strVdata(L->base) == '#') {
245     setintV(L->top-1, n-1);
246     return 1;
247   } else {
248     int32_t i = lj_lib_checkint(L, 1);
249     if (i < 0) i = n + i; else if (i > n) i = n;
250     if (i < 1)
251       lj_err_arg(L, 1, LJ_ERR_IDXRNG);
252     return n - i;
253   }
254 }
255
256 /* -- Base library: conversions ------------------------------------------ */
257
258 LJLIB_ASM(tonumber)                LJLIB_REC(.)
259 {
260   int32_t base = lj_lib_optint(L, 2, 10);
261   if (base == 10) {
262     TValue *o = lj_lib_checkany(L, 1);
263     if (lj_strscan_numberobj(o)) {
```

```
      copyTV(L, L->base-1-LJ_FR2, o);
      return FFH_RES(1);
    }
#if LJ_HASFFI
    if (tviscdata(o)) {
      CTState *cts = ctype_cts(L);
      CType *ct = lj_ctype_rawref(cts, cdataV(o)->ctypeid);
      if (ctype_isenum(ct->info)) ct = ctype_child(cts, ct);
      if (ctype_isnum(ct->info) || ctype_iscomplex(ct->info)) {
        if (LJ_DUALNUM && ctype_isinteger_or_bool(ct->info) &&
            ct->size <= 4 && !(ct->size == 4 && (ct->info & CTF_UNSIGNED))) {
          int32_t i;
          lj_cconv_ct_tv(cts, ctype_get(cts, CTID_INT32), (uint8_t *)&i, o, 0);
          setintV(L->base-1-LJ_FR2, i);
          return FFH_RES(1);
        }
        lj_cconv_ct_tv(cts, ctype_get(cts, CTID_DOUBLE),
                       (uint8_t *)&(L->base-1-LJ_FR2)->n, o, 0);
        return FFH_RES(1);
      }
    }
#endif
  } else {
    const char *p = strdata(lj_lib_checkstr(L, 1));
    char *ep;
    unsigned long ul;
    if (base < 2 || base > 36)
      lj_err_arg(L, 2, LJ_ERR_BASERNG);
    ul = strtoul(p, &ep, base);
    if (p != ep) {
      while (lj_char_isspace((unsigned char)(*ep))) ep++;
      if (*ep == '\0') {
        if (LJ_DUALNUM && LJ_LIKELY(ul < 0x80000000u))
          setintV(L->base-1-LJ_FR2, (int32_t)ul);
        else
          setnumV(L->base-1-LJ_FR2, (lua_Number)ul);
        return FFH_RES(1);
      }
    }
  }
  setnilV(L->base-1-LJ_FR2);
  return FFH_RES(1);
}

LJLIB_ASM(tostring)               LJLIB_REC(.)
{
  TValue *o = lj_lib_checkany(L, 1);
  cTValue *mo;
  L->top = o+1;  /* Only keep one argument. */
  if (!tvisnil(mo = lj_meta_lookup(L, o, MM_tostring))) {
    copyTV(L, L->base-1-LJ_FR2, mo);  /* Replace callable. */
    return FFH_TAILCALL;
  }
  lj_gc_check(L);
  setstrV(L, L->base-1-LJ_FR2, lj_strfmt_obj(L, L->base));
  return FFH_RES(1);
}

/* -- Base library: throw and catch errors ---------------------------------- */

LJLIB_CF(error)
{
  int32_t level = lj_lib_optint(L, 2, 1);
  lua_settop(L, 1);
  if (lua_isstring(L, 1) && level > 0) {
    luaL_where(L, level);
    lua_pushvalue(L, 1);
    lua_concat(L, 2);
  }
  return lua_error(L);
}

LJLIB_ASM(pcall)                LJLIB_REC(.)
{
  lj_lib_checkany(L, 1);
  lj_lib_checkfunc(L, 2);  /* For xpcall only. */
```

```
  return FFH_UNREACHABLE;
}
LJLIB_ASM_(xpcall)               LJLIB_REC(.)

/* -- Base library: load Lua code ----------------------------------------- */

static int load_aux(lua_State *L, int status, int envarg)
{
  if (status == 0) {
    if (tvistab(L->base+envarg-1)) {
      GCfunc *fn = funcV(L->top-1);
      GCtab *t = tabV(L->base+envarg-1);
      setgcref(fn->c.env, obj2gco(t));
      lj_gc_objbarrier(L, fn, t);
    }
    return 1;
  } else {
    setnilV(L->top-2);
    return 2;
  }
}

LJLIB_CF(loadfile)
{
  GCstr *fname = lj_lib_optstr(L, 1);
  GCstr *mode = lj_lib_optstr(L, 2);
  int status;
  lua_settop(L, 3);  /* Ensure env arg exists. */
  status = luaL_loadfilex(L, fname ? strdata(fname) : NULL,
                          mode ? strdata(mode) : NULL);
  return load_aux(L, status, 3);
}

static const char *reader_func(lua_State *L, void *ud, size_t *size)
{
  UNUSED(ud);
  luaL_checkstack(L, 2, "too many nested functions");
  copyTV(L, L->top++, L->base);
  lua_call(L, 0, 1);  /* Call user-supplied function. */
  L->top--;
  if (tvisnil(L->top)) {
    *size = 0;
    return NULL;
  } else if (tvisstr(L->top) || tvisnumber(L->top)) {
    copyTV(L, L->base+4, L->top);  /* Anchor string in reserved stack slot. */
    return lua_tolstring(L, 5, size);
  } else {
    lj_err_caller(L, LJ_ERR_RDRSTR);
    return NULL;
  }
}

LJLIB_CF(load)
{
  GCstr *name = lj_lib_optstr(L, 2);
  GCstr *mode = lj_lib_optstr(L, 3);
  int status;
  if (L->base < L->top && (tvisstr(L->base) || tvisnumber(L->base))) {
    GCstr *s = lj_lib_checkstr(L, 1);
    lua_settop(L, 4);  /* Ensure env arg exists. */
    status = luaL_loadbufferx(L, strdata(s), s->len, strdata(name ? name : s),
                              mode ? strdata(mode) : NULL);
  } else {
    lj_lib_checkfunc(L, 1);
    lua_settop(L, 5);  /* Reserve a slot for the string from the reader. */
    status = lua_loadx(L, reader_func, NULL, name ? strdata(name) : "=(load)",
                       mode ? strdata(mode) : NULL);
  }
  return load_aux(L, status, 4);
}

LJLIB_CF(loadstring)
{
  return lj_cf_load(L);
}
```

```
LJLIB_CF(dofile)
{
  GCstr *fname = lj_lib_optstr(L, 1);
  setnilV(L->top);
  L->top = L->base+1;
  if (luaL_loadfile(L, fname ? strdata(fname) : NULL) != 0)
    lua_error(L);
  lua_call(L, 0, LUA_MULTRET);
  return (int)(L->top - L->base) - 1;
}

/* -- Base library: GC control ------------------------------------------- */

LJLIB_CF(gcinfo)
{
  setintV(L->top++, (G(L)->gc.total >> 10));
  return 1;
}

LJLIB_CF(collectgarbage)
{
  int opt = lj_lib_checkopt(L, 1, LUA_GCCOLLECT,  /* ORDER LUA_GC* */
    "\4stop\7restart\7collect\5count\1\377\4step\10setpause\12setstepmul");
  int32_t data = lj_lib_optint(L, 2, 0);
  if (opt == LUA_GCCOUNT) {
    setnumV(L->top, (lua_Number)G(L)->gc.total/1024.0);
  } else {
    int res = lua_gc(L, opt, data);
    if (opt == LUA_GCSTEP)
      setboolV(L->top, res);
    else
      setintV(L->top, res);
  }
  L->top++;
  return 1;
}

/* -- Base library: miscellaneous functions ------------------------------ */

LJLIB_PUSH(top-2)  /* Upvalue holds weak table. */
LJLIB_CF(newproxy)
{
  lua_settop(L, 1);
  lua_newuserdata(L, 0);
  if (lua_toboolean(L, 1) == 0) {  /* newproxy(): without metatable. */
    return 1;
  } else if (lua_isboolean(L, 1)) {  /* newproxy(true): with metatable. */
    lua_newtable(L);
    lua_pushvalue(L, -1);
    lua_pushboolean(L, 1);
    lua_rawset(L, lua_upvalueindex(1));  /* Remember mt in weak table. */
  } else {  /* newproxy(proxy): inherit metatable. */
    int validproxy = 0;
    if (lua_getmetatable(L, 1)) {
      lua_rawget(L, lua_upvalueindex(1));
      validproxy = lua_toboolean(L, -1);
      lua_pop(L, 1);
    }
    if (!validproxy)
      lj_err_arg(L, 1, LJ_ERR_NOPROXY);
    lua_getmetatable(L, 1);
  }
  lua_setmetatable(L, 2);
  return 1;
}

LJLIB_PUSH("tostring")
LJLIB_CF(print)
{
  ptrdiff_t i, nargs = L->top - L->base;
  cTValue *tv = lj_tab_getstr(tabref(L->env), strV(lj_lib_upvalue(L, 1)));
  int shortcut;
  if (tv && !tvisnil(tv)) {
    copyTV(L, L->top++, tv);
  } else {
    setstrV(L, L->top++, strV(lj_lib_upvalue(L, 1)));
```

```
    lua_gettable(L, LUA_GLOBALSINDEX);
    tv = L->top-1;
  }
  shortcut = (tvisfunc(tv) && funcV(tv)->c.ffid == FF_tostring);
  for (i = 0; i < nargs; i++) {
    cTValue *o = &L->base[i];
    char buf[STRFMT_MAXBUF_NUM];
    const char *str;
    size_t size;
    MSize len;
    if (shortcut && (str = lj_strfmt_wstrnum(buf, o, &len)) != NULL) {
      size = len;
    } else {
      copyTV(L, L->top+1, o);
      copyTV(L, L->top, L->top-1);
      L->top += 2;
      lua_call(L, 1, 1);
      str = lua_tolstring(L, -1, &size);
      if (!str)
        lj_err_caller(L, LJ_ERR_PRTOSTR);
      L->top--;
    }
    if (i)
      putchar('\t');
    fwrite(str, 1, size, stdout);
  }
  putchar('\n');
  return 0;
}

LJLIB_PUSH(top-3)
LJLIB_SET(_VERSION)

#include "lj_libdef.h"

/* -- Coroutine library ---------------------------------------------------- */

#define LJLIB_MODULE_coroutine

LJLIB_CF(coroutine_status)
{
  const char *s;
  lua_State *co;
  if (!(L->top > L->base && tvisthread(L->base)))
    lj_err_arg(L, 1, LJ_ERR_NOCORO);
  co = threadV(L->base);
  if (co == L) s = "running";
  else if (co->status == LUA_YIELD) s = "suspended";
  else if (co->status != 0) s = "dead";
  else if (co->base > tvref(co->stack)+1+LJ_FR2) s = "normal";
  else if (co->top == co->base) s = "dead";
  else s = "suspended";
  lua_pushstring(L, s);
  return 1;
}

LJLIB_CF(coroutine_running)
{
#if LJ_52
  int ismain = lua_pushthread(L);
  setboolV(L->top++, ismain);
  return 2;
#else
  if (lua_pushthread(L))
    setnilV(L->top++);
  return 1;
#endif
}

LJLIB_CF(coroutine_create)
{
  lua_State *L1;
  if (!(L->base < L->top && tvisfunc(L->base)))
    lj_err_argt(L, 1, LUA_TFUNCTION);
  L1 = lua_newthread(L);
  setfuncV(L, L1->top++, funcV(L->base));
```

```
  return 1;
}

LJLIB_ASM(coroutine_yield)
{
  lj_err_caller(L, LJ_ERR_CYIELD);
  return FFH_UNREACHABLE;
}

static int ffh_resume(lua_State *L, lua_State *co, int wrap)
{
  if (co->cframe != NULL || co->status > LUA_YIELD ||
      (co->status == 0 && co->top == co->base)) {
    ErrMsg em = co->cframe ? LJ_ERR_CORUN : LJ_ERR_CODEAD;
    if (wrap) lj_err_caller(L, em);
    setboolV(L->base-1-LJ_FR2, 0);
    setstrV(L, L->base-LJ_FR2, lj_err_str(L, em));
    return FFH_RES(2);
  }
  lj_state_growstack(co, (MSize)(L->top - L->base));
  return FFH_RETRY;
}

LJLIB_ASM(coroutine_resume)
{
  if (!(L->top > L->base && tvisthread(L->base)))
    lj_err_arg(L, 1, LJ_ERR_NOCORO);
  return ffh_resume(L, threadV(L->base), 0);
}

LJLIB_NOREG LJLIB_ASM(coroutine_wrap_aux)
{
  return ffh_resume(L, threadV(lj_lib_upvalue(L, 1)), 1);
}

/* Inline declarations. */
LJ_ASMF void lj_ff_coroutine_wrap_aux(void);
#if !(LJ_TARGET_MIPS && defined(ljamalg_c))
LJ_FUNCA_NORET void LJ_FASTCALL lj_ffh_coroutine_wrap_err(lua_State *L,
                                                          lua_State *co);
#endif

/* Error handler, called from assembler VM. */
void LJ_FASTCALL lj_ffh_coroutine_wrap_err(lua_State *L, lua_State *co)
{
  co->top--; copyTV(L, L->top, co->top); L->top++;
  if (tvisstr(L->top-1))
    lj_err_callermsg(L, strVdata(L->top-1));
  else
    lj_err_run(L);
}

/* Forward declaration. */
static void setpc_wrap_aux(lua_State *L, GCfunc *fn);

LJLIB_CF(coroutine_wrap)
{
  GCfunc *fn;
  lj_cf_coroutine_create(L);
  fn = lj_lib_pushcc(L, lj_ffh_coroutine_wrap_aux, FF_coroutine_wrap_aux, 1);
  setpc_wrap_aux(L, fn);
  return 1;
}

#include "lj_libdef.h"

/* Fix the PC of wrap_aux. Really ugly workaround. */
static void setpc_wrap_aux(lua_State *L, GCfunc *fn)
{
  setmref(fn->c.pc, &L2GG(L)->bcff[lj_lib_init_coroutine[1]+2]);
}

/* ------------------------------------------------------------------------ */

static void newproxy_weaktable(lua_State *L)
{
```

```
644   /* NOBARRIER: The table is new (marked white). */
645   GCtab *t = lj_tab_new(L, 0, 1);
646   settabV(L, L->top++, t);
647   setgcref(t->metatable, obj2gco(t));
648   setstrV(L, lj_tab_setstr(L, t, lj_str_newlit(L, "__mode")),
649             lj_str_newlit(L, "kv"));
650   t->nomm = (uint8_t)(~(1u<<MM_mode));
651 }
652
653 LUALIB_API int luaopen_base(lua_State *L)
654 {
655   /* NOBARRIER: Table and value are the same. */
656   GCtab *env = tabref(L->env);
657   settabV(L, lj_tab_setstr(L, env, lj_str_newlit(L, "_G")), env);
658   lua_pushliteral(L, LUA_VERSION);  /* top-3. */
659   newproxy_weaktable(L);  /* top-2. */
660   LJ_LIB_REG(L, "_G", base);
661   LJ_LIB_REG(L, LUA_COLIBNAME, coroutine);
662   return 2;
663 }
664
```

One Level Up Top Level

src/lj_lib.h - luajit-2.0-src

Data types defined

- RandomState

Macros defined

- FFH_RES
- FFH_RETRY
- FFH_TAILCALL
- FFH_UNREACHABLE
- LIBINIT_ASM
- LIBINIT_ASM_
- LIBINIT_CF
- LIBINIT_COPY
- LIBINIT_END
- LIBINIT_FFID
- LIBINIT_LASTCL
- LIBINIT_LENMASK
- LIBINIT_LUA
- LIBINIT_MAXSTR
- LIBINIT_NUMBER
- LIBINIT_SET
- LIBINIT_STRING
- LIBINIT_TAGMASK
- LJLIB_ASM
- LJLIB_ASM_
- LJLIB_CF
- LJLIB_LUA
- LJLIB_NOREG
- LJLIB_NOREGUV
- LJLIB_PUSH
- LJLIB_REC
- LJLIB_SET

- LJ_LIB_REG
- _LJ_LIB_H
- lj_lib_checkfpu
- lj_lib_checkfpu
- lj_lib_checknumber
- lj_lib_pushcf
- lj_lib_upvalue
- lj_lib_upvalue
- lj_lib_upvalue

Source code

```
1  /*
2  ** Library function support.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_LIB_H
7  #define _LJ_LIB_H
8
9  #include "lj_obj.h"
10
11 /*
12 ** A fallback handler is called by the assembler VM if the fast path fails:
13 **
14 ** - too few arguments:   unrecoverable.
15 ** - wrong argument type:   recoverable, if coercion succeeds.
16 ** - bad argument value:  unrecoverable.
17 ** - stack overflow:        recoverable, if stack reallocation succeeds.
18 ** - extra handling:        recoverable.
19 **
20 ** The unrecoverable cases throw an error with lj_err_arg(), lj_err_argtype(),
21 ** lj_err_caller() or lj_err_callermsg().
22 ** The recoverable cases return 0 or the number of results + 1.
23 ** The assembler VM retries the fast path only if 0 is returned.
24 ** This time the fallback must not be called again or it gets stuck in a loop.
25 */
26
27 /* Return values from fallback handler. */
28 #define FFH_RETRY        0
29 #define FFH_UNREACHABLE        FFH_RETRY
30 #define FFH_RES(n)        ((n)+1)
31 #define FFH_TAILCALL        (-1)
32
33 LJ_FUNC TValue *lj_lib_checkany(lua_State *L, int narg);
34 LJ_FUNC GCstr *lj_lib_checkstr(lua_State *L, int narg);
35 LJ_FUNC GCstr *lj_lib_optstr(lua_State *L, int narg);
36 #if LJ_DUALNUM
37 LJ_FUNC void lj_lib_checknumber(lua_State *L, int narg);
38 #else
39 #define lj_lib_checknumber(L, narg)        lj_lib_checknum((L), (narg))
40 #endif
41 LJ_FUNC lua_Number lj_lib_checknum(lua_State *L, int narg);
42 LJ_FUNC int32_t lj_lib_checkint(lua_State *L, int narg);
43 LJ_FUNC int32_t lj_lib_optint(lua_State *L, int narg, int32_t def);
44 LJ_FUNC GCfunc *lj_lib_checkfunc(lua_State *L, int narg);
45 LJ_FUNC GCtab *lj_lib_checktab(lua_State *L, int narg);
46 LJ_FUNC GCtab *lj_lib_checktabornil(lua_State *L, int narg);
47 LJ_FUNC int lj_lib_checkopt(lua_State *L, int narg, int def, const char *lst);
48
49 /* Avoid including lj_frame.h. */
50 #if LJ_GC64
51 #define lj_lib_upvalue(L, n) \
52   (&gcval(L->base-2)->fn.c.upvalue[(n)-1])
```

```
#elif LJ_FR2
#define lj_lib_upvalue(L, n) \
  (&gcref((L->base-2)->gcr)->fn.c.upvalue[(n)-1])
#else
#define lj_lib_upvalue(L, n) \
  (&gcref((L->base-1)->fr.func)->fn.c.upvalue[(n)-1])
#endif

#if LJ_TARGET_WINDOWS
#define lj_lib_checkfpu(L) \
  do { setnumV(L->top++, (lua_Number)1437217655); \
    if (lua_tointeger(L, -1) != 1437217655) lj_err_caller(L, LJ_ERR_BADFPU); \
    L->top--; } while (0)
#else
#define lj_lib_checkfpu(L)        UNUSED(L)
#endif

LJ_FUNC GCfunc *lj_lib_pushcc(lua_State *L, lua_CFunction f, int id, int n);
#define lj_lib_pushcf(L, fn, id)        (lj_lib_pushcc(L, (fn), (id), 0))

/* Library function declarations. Scanned by buildvm. */
#define LJLIB_CF(name)               static int lj_cf_##name(lua_State *L)
#define LJLIB_ASM(name)               static int lj_ffh_##name(lua_State *L)
#define LJLIB_ASM_(name)
#define LJLIB_LUA(name)
#define LJLIB_SET(name)
#define LJLIB_PUSH(arg)
#define LJLIB_REC(handler)
#define LJLIB_NOREGUV
#define LJLIB_NOREG

#define LJ_LIB_REG(L, regname, name) \
  lj_lib_register(L, regname, lj_lib_init_##name, lj_lib_cf_##name)

LJ_FUNC void lj_lib_register(lua_State *L, const char *libname,
                             const uint8_t *init, const lua_CFunction *cf);
LJ_FUNC void lj_lib_prereg(lua_State *L, const char *name, lua_CFunction f,
                           GCtab *env);
LJ_FUNC int lj_lib_postreg(lua_State *L, lua_CFunction cf, int id,
                           const char *name);

/* Library init data tags. */
#define LIBINIT_LENMASK        0x3f
#define LIBINIT_TAGMASK        0xc0
#define LIBINIT_CF        0x00
#define LIBINIT_ASM        0x40
#define LIBINIT_ASM_        0x80
 #define LIBINIT_STRING        0xc0
 #define LIBINIT_MAXSTR        0x38
 #define LIBINIT_LUA        0xf9
 #define LIBINIT_SET        0xfa
 #define LIBINIT_NUMBER        0xfb
 #define LIBINIT_COPY        0xfc
 #define LIBINIT_LASTCL        0xfd
 #define LIBINIT_FFID        0xfe
 #define LIBINIT_END        0xff

 /* Exported library functions. */

 typedef struct RandomState RandomState;
 LJ_FUNC uint64_t LJ_FASTCALL lj_math_random_step(RandomState *rs);

 #endif
```

One Level Up Top Level

src/lj_lib.c - luajit-2.0-src

Functions defined

Macros defined

- LUA_CORE
- lj_lib_c

Source code

```
/*
** Library function support.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_lib_c
#define LUA_CORE

#include "lauxlib.h"

 #include "lj_obj.h"
 #include "lj_gc.h"
 #include "lj_err.h"
 #include "lj_str.h"
 #include "lj_tab.h"
 #include "lj_func.h"
 #include "lj_bc.h"
 #include "lj_dispatch.h"
```

```c
#include "lj_vm.h"
#include "lj_strscan.h"
#include "lj_strfmt.h"
#include "lj_lex.h"
#include "lj_bcdump.h"
#include "lj_lib.h"

/* -- Library initialization ---------------------------------------------- */

static GCtab *lib_create_table(lua_State *L, const char *libname, int hsize)
{
  if (libname) {
    luaL_findtable(L, LUA_REGISTRYINDEX, "_LOADED", 16);
    lua_getfield(L, -1, libname);
    if (!tvistab(L->top-1)) {
      L->top--;
      if (luaL_findtable(L, LUA_GLOBALSINDEX, libname, hsize) != NULL)
        lj_err_callerv(L, LJ_ERR_BADMODN, libname);
      settabV(L, L->top, tabV(L->top-1));
      L->top++;
      lua_setfield(L, -3, libname);  /* _LOADED[libname] = new table */
    }
    L->top--;
    settabV(L, L->top-1, tabV(L->top));
  } else {
    lua_createtable(L, 0, hsize);
  }
  return tabV(L->top-1);
}

static const uint8_t *lib_read_lfunc(lua_State *L, const uint8_t *p, GCtab *tab)
{
  int len = *p++;
  GCstr *name = lj_str_new(L, (const char *)p, len);
  LexState ls;
  GCproto *pt;
  GCfunc *fn;
  memset(&ls, 0, sizeof(ls));
  ls.L = L;
  ls.p = (const char *)(p+len);
  ls.pe = (const char *)~(uintptr_t)0;
  ls.c = -1;
  ls.level = (BCDUMP_F_STRIP|(LJ_BE*BCDUMP_F_BE));
  ls.chunkname = name;
  pt = lj_bcread_proto(&ls);
  pt->firstline = ~(BCLine)0;
  fn = lj_func_newL_empty(L, pt, tabref(L->env));
  /* NOBARRIER: See below for common barrier. */
  setfuncV(L, lj_tab_setstr(L, tab, name), fn);
  return (const uint8_t *)ls.p;
}

void lj_lib_register(lua_State *L, const char *libname,
                     const uint8_t *p, const lua_CFunction *cf)
{
  GCtab *env = tabref(L->env);
  GCfunc *ofn = NULL;
  int ffid = *p++;
  BCIns *bcff = &L2GG(L)->bcff[*p++];
  GCtab *tab = lib_create_table(L, libname, *p++);
  ptrdiff_t tpos = L->top - L->base;

  /* Avoid barriers further down. */
  lj_gc_anybarriert(L, tab);
  tab->nomm = 0;

  for (;;) {
    uint32_t tag = *p++;
    MSize len = tag & LIBINIT_LENMASK;
    tag &= LIBINIT_TAGMASK;
    if (tag != LIBINIT_STRING) {
      const char *name;
      MSize nuv = (MSize)(L->top - L->base - tpos);
      GCfunc *fn = lj_func_newC(L, nuv, env);
      if (nuv) {
        L->top = L->base + tpos;
```

```
        memcpy(fn->c.upvalue, L->top, sizeof(TValue)*nuv);
      }
      fn->c.ffid = (uint8_t)(ffid++);
      name = (const char *)p;
      p += len;
       if (tag == LIBINIT_CF)
         setmref(fn->c.pc, &G(L)->bc_cfunc_int);
       else
         setmref(fn->c.pc, bcff++);
       if (tag == LIBINIT_ASM_)
         fn->c.f = ofn->c.f;  /* Copy handler from previous function. */
       else
         fn->c.f = *cf++;  /* Get cf or handler from C function table. */
       if (len) {
         /* NOBARRIER: See above for common barrier. */
         setfuncV(L, lj_tab_setstr(L, tab, lj_str_new(L, name, len)), fn);
       }
       ofn = fn;
     } else {
       switch (tag | len) {
       case LIBINIT_LUA:
         p = lib_read_lfunc(L, p, tab);
         break;
       case LIBINIT_SET:
         L->top -= 2;
         if (tvisstr(L->top+1) && strV(L->top+1)->len == 0)
           env = tabV(L->top);
         else  /* NOBARRIER: See above for common barrier. */
           copyTV(L, lj_tab_set(L, tab, L->top+1), L->top);
         break;
       case LIBINIT_NUMBER:
         memcpy(&L->top->n, p, sizeof(double));
         L->top++;
         p += sizeof(double);
         break;
       case LIBINIT_COPY:
         copyTV(L, L->top, L->top - *p++);
         L->top++;
         break;
       case LIBINIT_LASTCL:
         setfuncV(L, L->top++, ofn);
         break;
       case LIBINIT_FFID:
         ffid++;
         break;
       case LIBINIT_END:
         return;
       default:
         setstrV(L, L->top++, lj_str_new(L, (const char *)p, len));
         p += len;
         break;
       }
     }
   }
 }

 /* Push internal function on the stack. */
 GCfunc *lj_lib_pushcc(lua_State *L, lua_CFunction f, int id, int n)
 {
   GCfunc *fn;
   lua_pushcclosure(L, f, n);
   fn = funcV(L->top-1);
   fn->c.ffid = (uint8_t)id;
   setmref(fn->c.pc, &G(L)->bc_cfunc_int);
   return fn;
 }

 void lj_lib_prereg(lua_State *L, const char *name, lua_CFunction f, GCtab *env)
 {
   luaL_findtable(L, LUA_REGISTRYINDEX, "_PRELOAD", 4);
   lua_pushcfunction(L, f);
   /* NOBARRIER: The function is new (marked white). */
   setgcref(funcV(L->top-1)->c.env, obj2gco(env));
   lua_setfield(L, -2, name);
   L->top--;
 }
```

```c

int lj_lib_postreg(lua_State *L, lua_CFunction cf, int id, const char *name)
{
  GCfunc *fn = lj_lib_pushcf(L, cf, id);
  GCtab *t = tabref(curr_func(L)->c.env);  /* Reference to parent table. */
  setfuncV(L, lj_tab_setstr(L, t, lj_str_newz(L, name)), fn);
  lj_gc_anybarriert(L, t);
  setfuncV(L, L->top++, fn);
  return 1;
}

/* -- Type checks ---------------------------------------------------------- */

TValue *lj_lib_checkany(lua_State *L, int narg)
{
  TValue *o = L->base + narg-1;
  if (o >= L->top)
    lj_err_arg(L, narg, LJ_ERR_NOVAL);
  return o;
}

GCstr *lj_lib_checkstr(lua_State *L, int narg)
{
  TValue *o = L->base + narg-1;
  if (o < L->top) {
    if (LJ_LIKELY(tvisstr(o))) {
      return strV(o);
    } else if (tvisnumber(o)) {
      GCstr *s = lj_strfmt_number(L, o);
      setstrV(L, o, s);
      return s;
    }
  }
  lj_err_argt(L, narg, LUA_TSTRING);
  return NULL;  /* unreachable */
}

GCstr *lj_lib_optstr(lua_State *L, int narg)
{
  TValue *o = L->base + narg-1;
  return (o < L->top && !tvisnil(o)) ? lj_lib_checkstr(L, narg) : NULL;
}

#if LJ_DUALNUM
void lj_lib_checknumber(lua_State *L, int narg)
{
  TValue *o = L->base + narg-1;
  if (!(o < L->top && lj_strscan_numberobj(o)))
    lj_err_argt(L, narg, LUA_TNUMBER);
}
#endif

lua_Number lj_lib_checknum(lua_State *L, int narg)
{
  TValue *o = L->base + narg-1;
  if (!(o < L->top &&
        (tvisnumber(o) || (tvisstr(o) && lj_strscan_num(strV(o), o)))))
    lj_err_argt(L, narg, LUA_TNUMBER);
  if (LJ_UNLIKELY(tvisint(o))) {
    lua_Number n = (lua_Number)intV(o);
    setnumV(o, n);
    return n;
  } else {
    return numV(o);
  }
}

int32_t lj_lib_checkint(lua_State *L, int narg)
{
  TValue *o = L->base + narg-1;
  if (!(o < L->top && lj_strscan_numberobj(o)))
    lj_err_argt(L, narg, LUA_TNUMBER);
  if (LJ_LIKELY(tvisint(o))) {
    return intV(o);
  } else {
    int32_t i = lj_num2int(numV(o));
```

```
    if (LJ_DUALNUM) setintV(o, i);
    return i;
  }
}

int32_t lj_lib_optint(lua_State *L, int narg, int32_t def)
{
  TValue *o = L->base + narg-1;
  return (o < L->top && !tvisnil(o)) ? lj_lib_checkint(L, narg) : def;
}

GCfunc *lj_lib_checkfunc(lua_State *L, int narg)
{
  TValue *o = L->base + narg-1;
  if (!(o < L->top && tvisfunc(o)))
    lj_err_argt(L, narg, LUA_TFUNCTION);
  return funcV(o);
}

GCtab *lj_lib_checktab(lua_State *L, int narg)
{
  TValue *o = L->base + narg-1;
  if (!(o < L->top && tvistab(o)))
    lj_err_argt(L, narg, LUA_TTABLE);
  return tabV(o);
}

GCtab *lj_lib_checktabornil(lua_State *L, int narg)
{
  TValue *o = L->base + narg-1;
  if (o < L->top) {
    if (tvistab(o))
      return tabV(o);
    else if (tvisnil(o))
      return NULL;
  }
  lj_err_arg(L, narg, LJ_ERR_NOTABN);
  return NULL;  /* unreachable */
}

int lj_lib_checkopt(lua_State *L, int narg, int def, const char *lst)
{
  GCstr *s = def >= 0 ? lj_lib_optstr(L, narg) : lj_lib_checkstr(L, narg);
  if (s) {
    const char *opt = strdata(s);
    MSize len = s->len;
    int i;
    for (i = 0; *(const uint8_t *)lst; i++) {
      if (*(const uint8_t *)lst == len && memcmp(opt, lst+1, len) == 0)
        return i;
      lst += 1+*(const uint8_t *)lst;
    }
    lj_err_argv(L, narg, LJ_ERR_INVOPTM, opt);
  }
  return def;
}

```

One Level Up Top Level

src/lib_aux.c - luajit-2.0-src

Functions defined

- adjuststack
- emptybuffer
- libsize
- luaL_addlstring
- luaL_addstring
- luaL_addvalue
- luaL_buffinit
- luaL_execresult
- luaL_fileresult
- luaL_findtable
- luaL_gsub
- luaL_newstate
- luaL_newstate
- luaL_openlib
- luaL_prepbuffer
- luaL_pushresult
- luaL_ref
- luaL_register
- luaL_unref
- lua_newstate
- mem_alloc
- panic

Macros defined

- FREELIST_REF
- LUA_LIB
- abs_index
- bufffree
- bufflen
- lib_aux_c

Source code

```c
/*
** Auxiliary library for the Lua/C API.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Major parts taken verbatim or adapted from the Lua interpreter.
** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
*/

#include <errno.h>
 #include <stdarg.h>
 #include <stdio.h>

 #define lib_aux_c
 #define LUA_LIB

 #include "lua.h"
 #include "lauxlib.h"

 #include "lj_obj.h"
 #include "lj_err.h"
 #include "lj_state.h"
 #include "lj_trace.h"
 #include "lj_lib.h"

 #if LJ_TARGET_POSIX
 #include <sys/wait.h>
 #endif

 /* -- I/O error handling ---------------------------------------------- */

 LUALIB_API int luaL_fileresult(lua_State *L, int stat, const char *fname)
 {
   if (stat) {
     setboolV(L->top++, 1);
     return 1;
   } else {
     int en = errno;  /* Lua API calls may change this value. */
     setnilV(L->top++);
     if (fname)
       lua_pushfstring(L, "%s: %s", fname, strerror(en));
     else
       lua_pushfstring(L, "%s", strerror(en));
     setintV(L->top++, en);
     lj_trace_abort(G(L));
     return 3;
   }
 }

 LUALIB_API int luaL_execresult(lua_State *L, int stat)
 {
   if (stat != -1) {
 #if LJ_TARGET_POSIX
     if (WIFSIGNALED(stat)) {
       stat = WTERMSIG(stat);
       setnilV(L->top++);
       lua_pushliteral(L, "signal");
     } else {
       if (WIFEXITED(stat))
         stat = WEXITSTATUS(stat);
       if (stat == 0)
         setboolV(L->top++, 1);
       else
         setnilV(L->top++);
       lua_pushliteral(L, "exit");
     }
 #else
     if (stat == 0)
       setboolV(L->top++, 1);
     else
       setnilV(L->top++);
     lua_pushliteral(L, "exit");
 #endif
     setintV(L->top++, stat);
```

```
    return 3;
  }
  return luaL_fileresult(L, 0, NULL);
}

/* -- Module registration -------------------------------------------- */

LUALIB_API const char *luaL_findtable(lua_State *L, int idx,
                                      const char *fname, int szhint)
{
  const char *e;
  lua_pushvalue(L, idx);
  do {
    e = strchr(fname, '.');
    if (e == NULL) e = fname + strlen(fname);
    lua_pushlstring(L, fname, (size_t)(e - fname));
    lua_rawget(L, -2);
    if (lua_isnil(L, -1)) {  /* no such field? */
      lua_pop(L, 1);  /* remove this nil */
      lua_createtable(L, 0, (*e == '.' ? 1 : szhint)); /* new table for field */
      lua_pushlstring(L, fname, (size_t)(e - fname));
      lua_pushvalue(L, -2);
      lua_settable(L, -4);  /* set new table into field */
    } else if (!lua_istable(L, -1)) {  /* field has a non-table value? */
      lua_pop(L, 2);  /* remove table and value */
      return fname;  /* return problematic part of the name */
     }
     lua_remove(L, -2);  /* remove previous table */
     fname = e + 1;
  } while (*e == '.');
  return NULL;
}

static int libsize(const luaL_Reg *l)
{
  int size = 0;
  for (; l->name; l++) size++;
  return size;
}

LUALIB_API void luaL_openlib(lua_State *L, const char *libname,
                             const luaL_Reg *l, int nup)
{
  lj_lib_checkfpu(L);
  if (libname) {
    int size = libsize(l);
    /* check whether lib already exists */
    luaL_findtable(L, LUA_REGISTRYINDEX, "_LOADED", 16);
    lua_getfield(L, -1, libname);  /* get _LOADED[libname] */
    if (!lua_istable(L, -1)) {  /* not found? */
      lua_pop(L, 1);  /* remove previous result */
      /* try global variable (and create one if it does not exist) */
      if (luaL_findtable(L, LUA_GLOBALSINDEX, libname, size) != NULL)
        lj_err_callerv(L, LJ_ERR_BADMODN, libname);
      lua_pushvalue(L, -1);
      lua_setfield(L, -3, libname);  /* _LOADED[libname] = new table */
    }
    lua_remove(L, -2);  /* remove _LOADED table */
    lua_insert(L, -(nup+1));  /* move library table to below upvalues */
  }
  for (; l->name; l++) {
    int i;
    for (i = 0; i < nup; i++)  /* copy upvalues to the top */
      lua_pushvalue(L, -nup);
    lua_pushcclosure(L, l->func, nup);
    lua_setfield(L, -(nup+2), l->name);
  }
  lua_pop(L, nup);  /* remove upvalues */
}

LUALIB_API void luaL_register(lua_State *L, const char *libname,
                              const luaL_Reg *l)
{
  luaL_openlib(L, libname, l, 0);
}
```

```c
LUALIB_API const char *luaL_gsub(lua_State *L, const char *s,
                                 const char *p, const char *r)
{
  const char *wild;
  size_t l = strlen(p);
  luaL_Buffer b;
  luaL_buffinit(L, &b);
  while ((wild = strstr(s, p)) != NULL) {
    luaL_addlstring(&b, s, (size_t)(wild - s));  /* push prefix */
    luaL_addstring(&b, r);  /* push replacement in place of pattern */
    s = wild + l;  /* continue after `p' */
  }
  luaL_addstring(&b, s);  /* push last suffix */
  luaL_pushresult(&b);
  return lua_tostring(L, -1);
}

/* -- Buffer handling ---------------------------------------------------- */

#define bufflen(B)        ((size_t)((B)->p - (B)->buffer))
#define bufffree(B)        ((size_t)(LUAL_BUFFERSIZE - bufflen(B)))

static int emptybuffer(luaL_Buffer *B)
{
  size_t l = bufflen(B);
  if (l == 0)
    return 0;  /* put nothing on stack */
  lua_pushlstring(B->L, B->buffer, l);
  B->p = B->buffer;
  B->lvl++;
  return 1;
}

static void adjuststack(luaL_Buffer *B)
{
  if (B->lvl > 1) {
    lua_State *L = B->L;
    int toget = 1;  /* number of levels to concat */
    size_t toplen = lua_strlen(L, -1);
    do {
      size_t l = lua_strlen(L, -(toget+1));
      if (!(B->lvl - toget + 1 >= LUA_MINSTACK/2 || toplen > l))
        break;
      toplen += l;
      toget++;
    } while (toget < B->lvl);
    lua_concat(L, toget);
    B->lvl = B->lvl - toget + 1;
  }
}

LUALIB_API char *luaL_prepbuffer(luaL_Buffer *B)
{
  if (emptybuffer(B))
    adjuststack(B);
  return B->buffer;
}

LUALIB_API void luaL_addlstring(luaL_Buffer *B, const char *s, size_t l)
{
  while (l--)
    luaL_addchar(B, *s++);
}

LUALIB_API void luaL_addstring(luaL_Buffer *B, const char *s)
{
  luaL_addlstring(B, s, strlen(s));
}

LUALIB_API void luaL_pushresult(luaL_Buffer *B)
{
  emptybuffer(B);
  lua_concat(B->L, B->lvl);
  B->lvl = 1;
}

```

```c
LUALIB_API void luaL_addvalue(luaL_Buffer *B)
{
  lua_State *L = B->L;
  size_t vl;
  const char *s = lua_tolstring(L, -1, &vl);
  if (vl <= bufffree(B)) {  /* fit into buffer? */
    memcpy(B->p, s, vl);  /* put it there */
    B->p += vl;
    lua_pop(L, 1);  /* remove from stack */
  } else {
    if (emptybuffer(B))
      lua_insert(L, -2);  /* put buffer before new value */
    B->lvl++;  /* add new value into B stack */
    adjuststack(B);
  }
}

LUALIB_API void luaL_buffinit(lua_State *L, luaL_Buffer *B)
{
  B->L = L;
  B->p = B->buffer;
  B->lvl = 0;
}

/* -- Reference management ------------------------------------------- */

#define FREELIST_REF        0

/* Convert a stack index to an absolute index. */
#define abs_index(L, i) \
  ((i) > 0 || (i) <= LUA_REGISTRYINDEX ? (i) : lua_gettop(L) + (i) + 1)

LUALIB_API int luaL_ref(lua_State *L, int t)
{
  int ref;
  t = abs_index(L, t);
  if (lua_isnil(L, -1)) {
    lua_pop(L, 1);  /* remove from stack */
    return LUA_REFNIL;  /* `nil' has a unique fixed reference */
  }
  lua_rawgeti(L, t, FREELIST_REF);  /* get first free element */
  ref = (int)lua_tointeger(L, -1);  /* ref = t[FREELIST_REF] */
  lua_pop(L, 1);  /* remove it from stack */
  if (ref != 0) {  /* any free element? */
    lua_rawgeti(L, t, ref);  /* remove it from list */
    lua_rawseti(L, t, FREELIST_REF);  /* (t[FREELIST_REF] = t[ref]) */
  } else {  /* no free elements */
    ref = (int)lua_objlen(L, t);
    ref++;  /* create new reference */
  }
  lua_rawseti(L, t, ref);
  return ref;
}

LUALIB_API void luaL_unref(lua_State *L, int t, int ref)
{
  if (ref >= 0) {
    t = abs_index(L, t);
    lua_rawgeti(L, t, FREELIST_REF);
    lua_rawseti(L, t, ref);  /* t[ref] = t[FREELIST_REF] */
    lua_pushinteger(L, ref);
    lua_rawseti(L, t, FREELIST_REF);  /* t[FREELIST_REF] = ref */
  }
}

/* -- Default allocator and panic function ------------------------------ */

static int panic(lua_State *L)
{
  const char *s = lua_tostring(L, -1);
  fputs("PANIC: unprotected error in call to Lua API (", stderr);
  fputs(s ? s : "?", stderr);
  fputc(')', stderr); fputc('\n', stderr);
  fflush(stderr);
  return 0;
}
```

```
302
303 #ifdef LUAJIT_USE_SYSMALLOC
304
305 #if LJ_64 && !defined(LUAJIT_USE_VALGRIND)
306 #error "Must use builtin allocator for 64 bit target"
307 #endif
308
309 static void *mem_alloc(void *ud, void *ptr, size_t osize, size_t nsize)
310 {
311   (void)ud;
312   (void)osize;
313   if (nsize == 0) {
314     free(ptr);
315     return NULL;
316   } else {
317     return realloc(ptr, nsize);
318   }
319 }
320
321 LUALIB_API lua_State *luaL_newstate(void)
322 {
323   lua_State *L = lua_newstate(mem_alloc, NULL);
324   if (L) G(L)->panic = panic;
325   return L;
326 }
327
328 #else
329
330 #include "lj_alloc.h"
331
332 LUALIB_API lua_State *luaL_newstate(void)
333 {
334   lua_State *L;
335   void *ud = lj_alloc_create();
336   if (ud == NULL) return NULL;
337 #if LJ_64
338   L = lj_state_newstate(lj_alloc_f, ud);
339 #else
340   L = lua_newstate(lj_alloc_f, ud);
341 #endif
342   if (L) G(L)->panic = panic;
343   return L;
344 }
345
346 #if LJ_64
347 LUA_API lua_State *lua_newstate(lua_Alloc f, void *ud)
348 {
349   UNUSED(f); UNUSED(ud);
350   fputs("Must use luaL_newstate() for 64 bit target\n", stderr);
351   return NULL;
352 }
353 #endif
354
355 #endif
356
```

One Level Up Top Level

src/lj_trace.h - luajit-2.0-src

Data types defined

- TraceError

Macros defined

- TREDEF
- _LJ_TRACE_H
- lj_trace_abort
- lj_trace_abort
- lj_trace_end
- lj_trace_end
- lj_trace_flushall
- lj_trace_freestate
- lj_trace_initstate

Source code

```
1  /*
2  ** Trace management.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_TRACE_H
7  #define _LJ_TRACE_H
8
9  #include "lj_obj.h"
10
11 #if LJ_HASJIT
12 #include "lj_jit.h"
13 #include "lj_dispatch.h"
14
15 /* Trace errors. */
16 typedef enum {
17 #define TREDEF(name, msg)        LJ_TRERR_##name,
18 #include "lj_traceerr.h"
19   LJ_TRERR__MAX
20 } TraceError;
21
22 LJ_FUNC_NORET void lj_trace_err(jit_State *J, TraceError e);
23 LJ_FUNC_NORET void lj_trace_err_info(jit_State *J, TraceError e);
24
25 /* Trace management. */
26 LJ_FUNC void LJ_FASTCALL lj_trace_free(global_State *g, GCtrace *T);
27 LJ_FUNC void lj_trace_reenableproto(GCproto *pt);
28 LJ_FUNC void lj_trace_flushproto(global_State *g, GCproto *pt);
29 LJ_FUNC void lj_trace_flush(jit_State *J, TraceNo traceno);
30 LJ_FUNC int lj_trace_flushall(lua_State *L);
31 LJ_FUNC void lj_trace_initstate(global_State *g);
32 LJ_FUNC void lj_trace_freestate(global_State *g);
33
34 /* Event handling. */
35 LJ_FUNC void lj_trace_ins(jit_State *J, const BCIns *pc);
36 LJ_FUNCA void LJ_FASTCALL lj_trace_hot(jit_State *J, const BCIns *pc);
37 LJ_FUNCA void LJ_FASTCALL lj_trace_stitch(jit_State *J, const BCIns *pc);
38 LJ_FUNCA int LJ_FASTCALL lj_trace_exit(jit_State *J, void *exptr);
```

```
39
40  /* Signal asynchronous abort of trace or end of trace. */
41  #define lj_trace_abort(g)        (G2J(g)->state &= ~LJ_TRACE_ACTIVE)
42  #define lj_trace_end(J)               (J->state = LJ_TRACE_END)
43
44  #else
45
46  #define lj_trace_flushall(L)        (UNUSED(L), 0)
47  #define lj_trace_initstate(g)        UNUSED(g)
48  #define lj_trace_freestate(g)        UNUSED(g)
49  #define lj_trace_abort(g)        UNUSED(g)
50  #define lj_trace_end(J)               UNUSED(J)
51
52  #endif
53
54  #endif
```

One Level Up Top Level

src/lauxlib.h - luajit-2.0-src

Data types defined

- luaL_Buffer
- luaL_Buffer
- luaL_Reg
- luaL_Reg

Macros defined

- LUA_ERRFILE
- LUA_NOREF
- LUA_REFNIL
- lauxlib_h
- luaL_addchar
- luaL_addsize
- luaL_argcheck
- luaL_checkint
- luaL_checklong
- luaL_checkstring
- luaL_dofile
- luaL_dostring
- luaL_getmetatable
- luaL_getn
- luaL_opt
- luaL_optint
- luaL_optlong
- luaL_optstring
- luaL_putchar
- luaL_reg
- luaL_setn
- luaL_typename
- lua_getref
- lua_ref

- lua_unref

Source code

```
/*
** $Id: lauxlib.h,v 1.88.1.1 2007/12/27 13:02:25 roberto Exp $
** Auxiliary functions for building Lua libraries
** See Copyright Notice in lua.h
*/


#ifndef lauxlib_h
#define lauxlib_h


#include <stddef.h>
#include <stdio.h>

#include "lua.h"


#define luaL_getn(L,i)          ((int)lua_objlen(L, i))
#define luaL_setn(L,i,j)        ((void)0)  /* no op! */

/* extra error code for `luaL_load' */
#define LUA_ERRFILE     (LUA_ERRERR+1)

typedef struct luaL_Reg {
  const char *name;
  lua_CFunction func;
} luaL_Reg;

LUALIB_API void (luaL_openlib) (lua_State *L, const char *libname,
                                const luaL_Reg *l, int nup);
LUALIB_API void (luaL_register) (lua_State *L, const char *libname,
                                const luaL_Reg *l);
LUALIB_API int (luaL_getmetafield) (lua_State *L, int obj, const char *e);
LUALIB_API int (luaL_callmeta) (lua_State *L, int obj, const char *e);
LUALIB_API int (luaL_typerror) (lua_State *L, int narg, const char *tname);
LUALIB_API int (luaL_argerror) (lua_State *L, int numarg, const char *extramsg);
LUALIB_API const char *(luaL_checklstring) (lua_State *L, int numArg,
                                                          size_t *l);
LUALIB_API const char *(luaL_optlstring) (lua_State *L, int numArg,
                                          const char *def, size_t *l);
LUALIB_API lua_Number (luaL_checknumber) (lua_State *L, int numArg);
LUALIB_API lua_Number (luaL_optnumber) (lua_State *L, int nArg, lua_Number def);

LUALIB_API lua_Integer (luaL_checkinteger) (lua_State *L, int numArg);
LUALIB_API lua_Integer (luaL_optinteger) (lua_State *L, int nArg,
                                          lua_Integer def);

LUALIB_API void (luaL_checkstack) (lua_State *L, int sz, const char *msg);
LUALIB_API void (luaL_checktype) (lua_State *L, int narg, int t);
LUALIB_API void (luaL_checkany) (lua_State *L, int narg);

LUALIB_API int   (luaL_newmetatable) (lua_State *L, const char *tname);
LUALIB_API void *(luaL_checkudata) (lua_State *L, int ud, const char *tname);

LUALIB_API void (luaL_where) (lua_State *L, int lvl);
LUALIB_API int (luaL_error) (lua_State *L, const char *fmt, ...);

LUALIB_API int (luaL_checkoption) (lua_State *L, int narg, const char *def,
                                   const char *const lst[]);

LUALIB_API int (luaL_ref) (lua_State *L, int t);
LUALIB_API void (luaL_unref) (lua_State *L, int t, int ref);

LUALIB_API int (luaL_loadfile) (lua_State *L, const char *filename);
LUALIB_API int (luaL_loadbuffer) (lua_State *L, const char *buff, size_t sz,
                                  const char *name);
LUALIB_API int (luaL_loadstring) (lua_State *L, const char *s);

LUALIB_API lua_State *(luaL_newstate) (void);

```

```
71
72  LUALIB_API const char *(luaL_gsub) (lua_State *L, const char *s, const char *p,
73                                                    const char *r);
74
75  LUALIB_API const char *(luaL_findtable) (lua_State *L, int idx,
76                                           const char *fname, int szhint);
77
78  /* From Lua 5.2. */
79  LUALIB_API int luaL_fileresult(lua_State *L, int stat, const char *fname);
80  LUALIB_API int luaL_execresult(lua_State *L, int stat);
81  LUALIB_API int (luaL_loadfilex) (lua_State *L, const char *filename,
82                                   const char *mode);
83  LUALIB_API int (luaL_loadbufferx) (lua_State *L, const char *buff, size_t sz,
84                                     const char *name, const char *mode);
85  LUALIB_API void luaL_traceback (lua_State *L, lua_State *L1, const char *msg,
86                                  int level);
87
88
89  /*
90  ** ===============================================================
91  ** some useful macros
92  ** ===============================================================
93  */
94
95  #define luaL_argcheck(L, cond,numarg,extramsg)        \
96                  ((void)((cond) || luaL_argerror(L, (numarg), (extramsg))))
97  #define luaL_checkstring(L,n)       (luaL_checklstring(L, (n), NULL))
98  #define luaL_optstring(L,n,d)       (luaL_optlstring(L, (n), (d), NULL))
99  #define luaL_checkint(L,n)        ((int)luaL_checkinteger(L, (n)))
100  #define luaL_optint(L,n,d)        ((int)luaL_optinteger(L, (n), (d)))
101  #define luaL_checklong(L,n)        ((long)luaL_checkinteger(L, (n)))
102  #define luaL_optlong(L,n,d)        ((long)luaL_optinteger(L, (n), (d)))
103
104  #define luaL_typename(L,i)        lua_typename(L, lua_type(L,(i)))
105
106  #define luaL_dofile(L, fn) \
107          (luaL_loadfile(L, fn) || lua_pcall(L, 0, LUA_MULTRET, 0))
108
109  #define luaL_dostring(L, s) \
110          (luaL_loadstring(L, s) || lua_pcall(L, 0, LUA_MULTRET, 0))
111
112  #define luaL_getmetatable(L,n)        (lua_getfield(L, LUA_REGISTRYINDEX, (n)))
113
114  #define luaL_opt(L,f,n,d)        (lua_isnoneornil(L,(n)) ? (d) : f(L,(n)))
115
116  /*
117  ** {======================================================
118  ** Generic Buffer manipulation
119  ** =======================================================
120  */
121
122
123
124  typedef struct luaL_Buffer {
125    char *p;                       /* current position in buffer */
126    int lvl;  /* number of strings in the stack (level) */
127    lua_State *L;
128    char buffer[LUAL_BUFFERSIZE];
129  } luaL_Buffer;
130
131  #define luaL_addchar(B,c) \
132    ((void)((B)->p < ((B)->buffer+LUAL_BUFFERSIZE) || luaL_prepbuffer(B)), \
133     (*(B)->p++ = (char)(c)))
134
135  /* compatibility only */
136  #define luaL_putchar(B,c)        luaL_addchar(B,c)
137
138  #define luaL_addsize(B,n)        ((B)->p += (n))
139
140  LUALIB_API void (luaL_buffinit) (lua_State *L, luaL_Buffer *B);
141  LUALIB_API char *(luaL_prepbuffer) (luaL_Buffer *B);
142  LUALIB_API void (luaL_addlstring) (luaL_Buffer *B, const char *s, size_t l);
143  LUALIB_API void (luaL_addstring) (luaL_Buffer *B, const char *s);
144  LUALIB_API void (luaL_addvalue) (luaL_Buffer *B);
145  LUALIB_API void (luaL_pushresult) (luaL_Buffer *B);
146
```

```
147
148 /* }====================================================== */
149
150
151 /* compatibility with ref system */
152
153 /* pre-defined references */
154 #define LUA_NOREF       (-2)
155 #define LUA_REFNIL      (-1)
156
157 #define lua_ref(L,lock) ((lock) ? luaL_ref(L, LUA_REGISTRYINDEX) : \
158       (lua_pushstring(L, "unlocked references are obsolete"), lua_error(L), 0))
159
160 #define lua_unref(L,ref)        luaL_unref(L, LUA_REGISTRYINDEX, (ref))
161
162 #define lua_getref(L,ref)       lua_rawgeti(L, LUA_REGISTRYINDEX, (ref))
163
164
165 #define luaL_reg        luaL_Reg
166
167 #endif
```

One Level Up Top Level

src/lj_load.c - luajit-2.0-src

Data types defined

- FileReaderCtx
- FileReaderCtx
- StringReaderCtx
- StringReaderCtx

Functions defined

- cpparser
- luaL_loadbuffer
- luaL_loadbufferx
- luaL_loadfile
- luaL_loadfilex
- luaL_loadstring
- lua_dump
- lua_load
- lua_loadx
- reader_file
- reader_string

Macros defined

- LUA_CORE
- lj_load_c

Source code

```
1  /*
2  ** Load and dump code.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #include <errno.h>
7  #include <stdio.h>
8
9  #define lj_load_c
10  #define LUA_CORE
11
12  #include "lua.h"
13  #include "lauxlib.h"
14
15  #include "lj_obj.h"
16  #include "lj_gc.h"
17  #include "lj_err.h"
18  #include "lj_buf.h"
19  #include "lj_func.h"
```

```c
#include "lj_frame.h"
#include "lj_vm.h"
#include "lj_lex.h"
#include "lj_bcdump.h"
#include "lj_parse.h"

/* -- Load Lua source code and bytecode ---------------------------------- */

static TValue *cpparser(lua_State *L, lua_CFunction dummy, void *ud)
{
  LexState *ls = (LexState *)ud;
  GCproto *pt;
  GCfunc *fn;
  int bc;
  UNUSED(dummy);
  cframe_errfunc(L->cframe) = -1;  /* Inherit error function. */
  bc = lj_lex_setup(L, ls);
  if (ls->mode && !strchr(ls->mode, bc ? 'b' : 't')) {
    setstrV(L, L->top++, lj_err_str(L, LJ_ERR_XMODE));
    lj_err_throw(L, LUA_ERRSYNTAX);
  }
  pt = bc ? lj_bcread(ls) : lj_parse(ls);
  fn = lj_func_newL_empty(L, pt, tabref(L->env));
  /* Don't combine above/below into one statement. */
  setfuncV(L, L->top++, fn);
  return NULL;
}

LUA_API int lua_loadx(lua_State *L, lua_Reader reader, void *data,
                      const char *chunkname, const char *mode)
{
  LexState ls;
  int status;
  ls.rfunc = reader;
  ls.rdata = data;
  ls.chunkarg = chunkname ? chunkname : "?";
  ls.mode = mode;
  lj_buf_init(L, &ls.sb);
  status = lj_vm_cpcall(L, NULL, &ls, cpparser);
  lj_lex_cleanup(L, &ls);
  lj_gc_check(L);
  return status;
}

LUA_API int lua_load(lua_State *L, lua_Reader reader, void *data,
                     const char *chunkname)
{
  return lua_loadx(L, reader, data, chunkname, NULL);
}

typedef struct FileReaderCtx {
  FILE *fp;
  char buf[LUAL_BUFFERSIZE];
} FileReaderCtx;

static const char *reader_file(lua_State *L, void *ud, size_t *size)
{
  FileReaderCtx *ctx = (FileReaderCtx *)ud;
  UNUSED(L);
  if (feof(ctx->fp)) return NULL;
  *size = fread(ctx->buf, 1, sizeof(ctx->buf), ctx->fp);
  return *size > 0 ? ctx->buf : NULL;
}

LUALIB_API int luaL_loadfilex(lua_State *L, const char *filename,
                              const char *mode)
{
  FileReaderCtx ctx;
  int status;
  const char *chunkname;
  if (filename) {
    ctx.fp = fopen(filename, "rb");
    if (ctx.fp == NULL) {
      lua_pushfstring(L, "cannot open %s: %s", filename, strerror(errno));
      return LUA_ERRFILE;
    }
```

```
    chunkname = lua_pushfstring(L, "@%s", filename);
  } else {
    ctx.fp = stdin;
    chunkname = "=stdin";
  }
  status = lua_loadx(L, reader_file, &ctx, chunkname, mode);
  if (ferror(ctx.fp)) {
    L->top -= filename ? 2 : 1;
    lua_pushfstring(L, "cannot read %s: %s", chunkname+1, strerror(errno));
    if (filename)
      fclose(ctx.fp);
    return LUA_ERRFILE;
  }
  if (filename) {
    L->top--;
    copyTV(L, L->top-1, L->top);
    fclose(ctx.fp);
  }
  return status;
}

LUALIB_API int luaL_loadfile(lua_State *L, const char *filename)
{
  return luaL_loadfilex(L, filename, NULL);
}

typedef struct StringReaderCtx {
  const char *str;
  size_t size;
} StringReaderCtx;

static const char *reader_string(lua_State *L, void *ud, size_t *size)
{
  StringReaderCtx *ctx = (StringReaderCtx *)ud;
  UNUSED(L);
  if (ctx->size == 0) return NULL;
  *size = ctx->size;
  ctx->size = 0;
  return ctx->str;
}

LUALIB_API int luaL_loadbufferx(lua_State *L, const char *buf, size_t size,
                                const char *name, const char *mode)
{
  StringReaderCtx ctx;
  ctx.str = buf;
  ctx.size = size;
  return lua_loadx(L, reader_string, &ctx, name, mode);
}

LUALIB_API int luaL_loadbuffer(lua_State *L, const char *buf, size_t size,
                               const char *name)
{
  return luaL_loadbufferx(L, buf, size, name, NULL);
}

LUALIB_API int luaL_loadstring(lua_State *L, const char *s)
{
  return luaL_loadbuffer(L, s, strlen(s), s);
}

/* -- Dump bytecode ------------------------------------------------------ */

LUA_API int lua_dump(lua_State *L, lua_Writer writer, void *data)
{
  cTValue *o = L->top-1;
  api_check(L, L->top > L->base);
  if (tvisfunc(o) && isluafunc(funcV(o)))
    return lj_bcwrite(L, funcproto(funcV(o)), writer, data, 0);
  else
    return 1;
}
```

One Level Up Top Level

src/lj_parse.c - luajit-2.0-src

Global variables defined

- priority

Data types defined

- BinOpr
- BinOpr
- ExpDesc
- ExpDesc
- ExpKind
- FuncScope
- FuncScope
- FuncState
- FuncState
- LHSVarList
- LHSVarList
- VarIndex

Functions defined

- assign_adjust
- assign_hazard
- bcemit_INS
- bcemit_arith
- bcemit_binop
- bcemit_binop_left
- bcemit_branch
- bcemit_branch_f
- bcemit_branch_t
- bcemit_comp
- bcemit_jmp
- bcemit_method
- bcemit_nil
- bcemit_store

- bcemit_unop
- bcopisret
- bcreg_bump
- bcreg_free
- bcreg_reserve
- const_gc
- const_num
- const_str
- err_limit
- err_syntax
- err_token
- expr
- expr_binop
- expr_bracket
- expr_cond
- expr_discharge
- expr_field
- expr_free
- expr_index
- expr_init
- expr_kvalue
- expr_list
- expr_next
- expr_numiszero
- expr_primary
- expr_simple
- expr_str
- expr_table
- expr_toanyreg
- expr_tonextreg
- expr_toreg
- expr_toreg_nobranch
- expr_toval

- expr_unop
- foldarith
- fs_finish
- fs_fixup_bc
- fs_fixup_k
- fs_fixup_line
- fs_fixup_ret
- fs_fixup_uv1
- fs_fixup_uv2
- fs_fixup_var
- fs_init
- fs_prep_line
- fs_prep_var
- fscope_begin
- fscope_end
- fscope_uvmark
- gola_close
- gola_findlabel
- gola_fixup
- gola_new
- gola_patch
- gola_resolve
- invertcond
- jmp_append
- jmp_dropval
- jmp_next
- jmp_novalue
- jmp_patch
- jmp_patchins
- jmp_patchtestreg
- jmp_patchval
- jmp_tohere
- lex_check

- lex_match
- lex_opt
- lex_str
- lj_parse
- lj_parse_keepcdata
- lj_parse_keepstr
- parse_args
- parse_assignment
- parse_block
- parse_body
- parse_break
- parse_call_assign
- parse_chunk
- parse_for
- parse_for_iter
- parse_for_num
- parse_func
- parse_goto
- parse_if
- parse_isend
- parse_label
- parse_local
- parse_params
- parse_repeat
- parse_return
- parse_stmt
- parse_then
- parse_while
- predict_next
- synlevel_begin
- token2binop
- var_add
- var_lookup_

- var_lookup_local
- var_lookup_uv
- var_new
- var_remove

Macros defined

- FSCOPE_BREAK
- FSCOPE_GOLA
- FSCOPE_LOOP
- FSCOPE_NOCLOSE
- FSCOPE_UPVAL
- LJ_MAX_VSTACK
- LUA_CORE
- NAME_BREAK
- UNARY_PRIORITY
- VSTACK_GOTO
- VSTACK_LABEL
- VSTACK_VAR_RW
- bcemit_ABC
- bcemit_AD
- bcemit_AJ
- bcptr
- checkcond
- checklimit
- checklimitgt
- const_pri
- expr_hasjump
- expr_isk
- expr_isk_nojump
- expr_isnumk
- expr_isnumk_nojump
- expr_isstrk
- expr_numberV
- expr_numtv

- fs_fixup_line
- fs_fixup_var
- fs_prep_line
- fs_prep_var
- gola_isgoto
- gola_isgotolabel
- gola_islabel
- lj_parse_c
- synlevel_end
- tvhaskslot
- tvkslot
- var_get
- var_lookup
- var_new_fixed
- var_new_lit

Source code

```
/*
** Lua parser (source code -> bytecode).
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Major portions taken verbatim or adapted from the Lua interpreter.
** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
*/

#define lj_parse_c
 #define LUA_CORE

 #include "lj_obj.h"
 #include "lj_gc.h"
 #include "lj_err.h"
 #include "lj_debug.h"
 #include "lj_buf.h"
 #include "lj_str.h"
 #include "lj_tab.h"
 #include "lj_func.h"
 #include "lj_state.h"
 #include "lj_bc.h"
 #if LJ_HASFFI
 #include "lj_ctype.h"
 #endif
 #include "lj_strfmt.h"
 #include "lj_lex.h"
 #include "lj_parse.h"
 #include "lj_vm.h"
 #include "lj_vmevent.h"

 /* -- Parser structures and definitions ---------------------------------- */

 /* Expression kinds. */
 typedef enum {
   /* Constant expressions must be first and in this order: */
   VKNIL,
   VKFALSE,
   VKTRUE,
```

```
  VKSTR,        /* sval = string value */
  VKNUM,        /* nval = number value */
  VKLAST = VKNUM,
  VKCDATA,        /* nval = cdata value, not treated as a constant expression */
  /* Non-constant expressions follow: */
  VLOCAL,        /* info = local register, aux = vstack index */
  VUPVAL,        /* info = upvalue index, aux = vstack index */
  VGLOBAL,        /* sval = string value */
  VINDEXED,        /* info = table register, aux = index reg/byte/string const */
  VJMP,                /* info = instruction PC */
  VRELOCABLE,        /* info = instruction PC */
  VNONRELOC,        /* info = result register */
  VCALL,        /* info = instruction PC, aux = base */
  VVOID
} ExpKind;

/* Expression descriptor. */
typedef struct ExpDesc {
  union {
    struct {
      uint32_t info;        /* Primary info. */
      uint32_t aux;        /* Secondary info. */
    } s;
    TValue nval;        /* Number value. */
    GCstr *sval;        /* String value. */
  } u;
  ExpKind k;
  BCPos t;                /* True condition jump list. */
  BCPos f;                /* False condition jump list. */
} ExpDesc;

/* Macros for expressions. */
#define expr_hasjump(e)                ((e)->t != (e)->f)

#define expr_isk(e)                ((e)->k <= VKLAST)
#define expr_isk_nojump(e)        (expr_isk(e) && !expr_hasjump(e))
#define expr_isnumk(e)                ((e)->k == VKNUM)
#define expr_isnumk_nojump(e)        (expr_isnumk(e) && !expr_hasjump(e))
#define expr_isstrk(e)                ((e)->k == VKSTR)

#define expr_numtv(e)                check_exp(expr_isnumk((e)), &(e)->u.nval)
#define expr_numberV(e)                numberVnum(expr_numtv((e)))

/* Initialize expression. */
static LJ_AINLINE void expr_init(ExpDesc *e, ExpKind k, uint32_t info)
{
  e->k = k;
  e->u.s.info = info;
  e->f = e->t = NO_JMP;
}

/* Check number constant for +-0. */
static int expr_numiszero(ExpDesc *e)
{
  TValue *o = expr_numtv(e);
  return tvisint(o) ? (intV(o) == 0) : tviszero(o);
}

/* Per-function linked list of scope blocks. */
typedef struct FuncScope {
  struct FuncScope *prev;        /* Link to outer scope. */
   MSize vstart;                        /* Start of block-local variables. */
   uint8_t nactvar;                /* Number of active vars outside the scope. */
   uint8_t flags;                /* Scope flags. */
 } FuncScope;

 #define FSCOPE_LOOP                0x01        /* Scope is a (breakable) loop. */
 #define FSCOPE_BREAK                0x02        /* Break used in scope. */
 #define FSCOPE_GOLA                0x04        /* Goto or label used in scope. */
 #define FSCOPE_UPVAL                0x08        /* Upvalue in scope. */
 #define FSCOPE_NOCLOSE                0x10        /* Do not close upvalues. */

 #define NAME_BREAK                ((GCstr *)(uintptr_t)1)

 /* Index into variable stack. */
 typedef uint16_t VarIndex;
```

```c
#define LJ_MAX_VSTACK		(65536 - LJ_MAX_UPVAL)

/* Variable/goto/label info. */
#define VSTACK_VAR_RW		0x01	/* R/W variable. */
#define VSTACK_GOTO		0x02	/* Pending goto. */
#define VSTACK_LABEL		0x04	/* Label. */

/* Per-function state. */
typedef struct FuncState {
  GCtab *kt;			/* Hash table for constants. */
  LexState *ls;			/* Lexer state. */
  lua_State *L;			/* Lua state. */
  FuncScope *bl;		/* Current scope. */
  struct FuncState *prev;	/* Enclosing function. */
  BCPos pc;			/* Next bytecode position. */
  BCPos lasttarget;		/* Bytecode position of last jump target. */
  BCPos jpc;			/* Pending jump list to next bytecode. */
  BCReg freereg;		/* First free register. */
  BCReg nactvar;		/* Number of active local variables. */
  BCReg nkn, nkgc;		/* Number of lua_Number/GCobj constants */
  BCLine linedefined;		/* First line of the function definition. */
  BCInsLine *bcbase;		/* Base of bytecode stack. */
  BCPos bclim;			/* Limit of bytecode stack. */
  MSize vbase;			/* Base of variable stack for this function. */
  uint8_t flags;		/* Prototype flags. */
  uint8_t numparams;		/* Number of parameters. */
  uint8_t framesize;		/* Fixed frame size. */
  uint8_t nuv;			/* Number of upvalues */
  VarIndex varmap[LJ_MAX_LOCVAR];  /* Map from register to variable idx. */
  VarIndex uvmap[LJ_MAX_UPVAL];	/* Map from upvalue to variable idx. */
  VarIndex uvtmp[LJ_MAX_UPVAL];	/* Temporary upvalue map. */
} FuncState;

/* Binary and unary operators. ORDER OPR */
typedef enum BinOpr {
  OPR_ADD, OPR_SUB, OPR_MUL, OPR_DIV, OPR_MOD, OPR_POW,  /* ORDER ARITH */
  OPR_CONCAT,
  OPR_NE, OPR_EQ,
  OPR_LT, OPR_GE, OPR_LE, OPR_GT,
  OPR_AND, OPR_OR,
  OPR_NOBINOPR
} BinOpr;

LJ_STATIC_ASSERT((int)BC_ISGE-(int)BC_ISLT == (int)OPR_GE-(int)OPR_LT);
LJ_STATIC_ASSERT((int)BC_ISLE-(int)BC_ISLT == (int)OPR_LE-(int)OPR_LT);
LJ_STATIC_ASSERT((int)BC_ISGT-(int)BC_ISLT == (int)OPR_GT-(int)OPR_LT);
LJ_STATIC_ASSERT((int)BC_SUBVV-(int)BC_ADDVV == (int)OPR_SUB-(int)OPR_ADD);
LJ_STATIC_ASSERT((int)BC_MULVV-(int)BC_ADDVV == (int)OPR_MUL-(int)OPR_ADD);
LJ_STATIC_ASSERT((int)BC_DIVVV-(int)BC_ADDVV == (int)OPR_DIV-(int)OPR_ADD);
LJ_STATIC_ASSERT((int)BC_MODVV-(int)BC_ADDVV == (int)OPR_MOD-(int)OPR_ADD);

/* -- Error handling ------------------------------------------------------ */

LJ_NORET LJ_NOINLINE static void err_syntax(LexState *ls, ErrMsg em)
{
  lj_lex_error(ls, ls->tok, em);
}

LJ_NORET LJ_NOINLINE static void err_token(LexState *ls, LexToken tok)
{
  lj_lex_error(ls, ls->tok, LJ_ERR_XTOKEN, lj_lex_token2str(ls, tok));
}

LJ_NORET static void err_limit(FuncState *fs, uint32_t limit, const char *what)
{
  if (fs->linedefined == 0)
    lj_lex_error(fs->ls, 0, LJ_ERR_XLIMM, limit, what);
  else
    lj_lex_error(fs->ls, 0, LJ_ERR_XLIMF, fs->linedefined, limit, what);
}

#define checklimit(fs, v, l, m)		if ((v) >= (l)) err_limit(fs, l, m)
#define checklimitgt(fs, v, l, m)	if ((v) > (l)) err_limit(fs, l, m)
#define checkcond(ls, c, em)		{ if (!(c)) err_syntax(ls, em); }

/* -- Management of constants --------------------------------------------- */
```

```c

/* Return bytecode encoding for primitive constant. */
#define const_pri(e)		check_exp((e)->k <= VKTRUE, (e)->k)

#define tvhaskslot(o)	((o)->u32.hi == 0)
#define tvkslot(o)	((o)->u32.lo)

/* Add a number constant. */
static BCReg const_num(FuncState *fs, ExpDesc *e)
{
  lua_State *L = fs->L;
  TValue *o;
  lua_assert(expr_isnumk(e));
  o = lj_tab_set(L, fs->kt, &e->u.nval);
  if (tvhaskslot(o))
    return tvkslot(o);
  o->u64 = fs->nkn;
  return fs->nkn++;
}

/* Add a GC object constant. */
static BCReg const_gc(FuncState *fs, GCobj *gc, uint32_t itype)
{
  lua_State *L = fs->L;
  TValue key, *o;
  setgcV(L, &key, gc, itype);
  /* NOBARRIER: the key is new or kept alive. */
  o = lj_tab_set(L, fs->kt, &key);
  if (tvhaskslot(o))
    return tvkslot(o);
  o->u64 = fs->nkgc;
  return fs->nkgc++;
}

/* Add a string constant. */
static BCReg const_str(FuncState *fs, ExpDesc *e)
{
  lua_assert(expr_isstrk(e) || e->k == VGLOBAL);
  return const_gc(fs, obj2gco(e->u.sval), LJ_TSTR);
}

/* Anchor string constant to avoid GC. */
GCstr *lj_parse_keepstr(LexState *ls, const char *str, size_t len)
{
  /* NOBARRIER: the key is new or kept alive. */
  lua_State *L = ls->L;
  GCstr *s = lj_str_new(L, str, len);
  TValue *tv = lj_tab_setstr(L, ls->fs->kt, s);
  if (tvisnil(tv)) setboolV(tv, 1);
  lj_gc_check(L);
  return s;
}

#if LJ_HASFFI
/* Anchor cdata to avoid GC. */
void lj_parse_keepcdata(LexState *ls, TValue *tv, GCcdata *cd)
{
  /* NOBARRIER: the key is new or kept alive. */
  lua_State *L = ls->L;
  setcdataV(L, tv, cd);
  setboolV(lj_tab_set(L, ls->fs->kt, tv), 1);
}
#endif

/* -- Jump list handling -------------------------------------------------- */

/* Get next element in jump list. */
static BCPos jmp_next(FuncState *fs, BCPos pc)
{
  ptrdiff_t delta = bc_j(fs->bcbase[pc].ins);
  if ((BCPos)delta == NO_JMP)
    return NO_JMP;
  else
    return (BCPos)(((ptrdiff_t)pc+1)+delta);
}

```

```c
/* Check if any of the instructions on the jump list produce no value. */
static int jmp_novalue(FuncState *fs, BCPos list)
{
  for (; list != NO_JMP; list = jmp_next(fs, list)) {
    BCIns p = fs->bcbase[list >= 1 ? list-1 : list].ins;
    if (!(bc_op(p) == BC_ISTC || bc_op(p) == BC_ISFC || bc_a(p) == NO_REG))
      return 1;
  }
  return 0;
}

/* Patch register of test instructions. */
static int jmp_patchtestreg(FuncState *fs, BCPos pc, BCReg reg)
{
  BCInsLine *ilp = &fs->bcbase[pc >= 1 ? pc-1 : pc];
  BCOp op = bc_op(ilp->ins);
  if (op == BC_ISTC || op == BC_ISFC) {
    if (reg != NO_REG && reg != bc_d(ilp->ins)) {
      setbc_a(&ilp->ins, reg);
    } else {  /* Nothing to store or already in the right register. */
      setbc_op(&ilp->ins, op+(BC_IST-BC_ISTC));
      setbc_a(&ilp->ins, 0);
    }
  } else if (bc_a(ilp->ins) == NO_REG) {
    if (reg == NO_REG) {
      ilp->ins = BCINS_AJ(BC_JMP, bc_a(fs->bcbase[pc].ins), 0);
    } else {
      setbc_a(&ilp->ins, reg);
      if (reg >= bc_a(ilp[1].ins))
        setbc_a(&ilp[1].ins, reg+1);
    }
  } else {
    return 0;  /* Cannot patch other instructions. */
  }
  return 1;
}

/* Drop values for all instructions on jump list. */
static void jmp_dropval(FuncState *fs, BCPos list)
{
  for (; list != NO_JMP; list = jmp_next(fs, list))
    jmp_patchtestreg(fs, list, NO_REG);
}

/* Patch jump instruction to target. */
static void jmp_patchins(FuncState *fs, BCPos pc, BCPos dest)
{
  BCIns *jmp = &fs->bcbase[pc].ins;
  BCPos offset = dest-(pc+1)+BCBIAS_J;
  lua_assert(dest != NO_JMP);
  if (offset > BCMAX_D)
    err_syntax(fs->ls, LJ_ERR_XJUMP);
  setbc_d(jmp, offset);
}

/* Append to jump list. */
static void jmp_append(FuncState *fs, BCPos *l1, BCPos l2)
{
  if (l2 == NO_JMP) {
    return;
  } else if (*l1 == NO_JMP) {
    *l1 = l2;
  } else {
    BCPos list = *l1;
    BCPos next;
    while ((next = jmp_next(fs, list)) != NO_JMP)  /* Find last element. */
      list = next;
    jmp_patchins(fs, list, l2);
  }
}

/* Patch jump list and preserve produced values. */
static void jmp_patchval(FuncState *fs, BCPos list, BCPos vtarget,
                         BCReg reg, BCPos dtarget)
{
  while (list != NO_JMP) {
```

```
    BCPos next = jmp_next(fs, list);
    if (jmp_patchtestreg(fs, list, reg))
      jmp_patchins(fs, list, vtarget);  /* Jump to target with value. */
    else
      jmp_patchins(fs, list, dtarget);  /* Jump to default target. */
    list = next;
  }
}

/* Jump to following instruction. Append to list of pending jumps. */
static void jmp_tohere(FuncState *fs, BCPos list)
{
  fs->lasttarget = fs->pc;
  jmp_append(fs, &fs->jpc, list);
}

/* Patch jump list to target. */
static void jmp_patch(FuncState *fs, BCPos list, BCPos target)
{
  if (target == fs->pc) {
    jmp_tohere(fs, list);
  } else {
    lua_assert(target < fs->pc);
    jmp_patchval(fs, list, target, NO_REG, target);
  }
}

/* -- Bytecode register allocator ---------------------------------------- */

/* Bump frame size. */
static void bcreg_bump(FuncState *fs, BCReg n)
{
  BCReg sz = fs->freereg + n;
  if (sz > fs->framesize) {
    if (sz >= LJ_MAX_SLOTS)
      err_syntax(fs->ls, LJ_ERR_XSLOTS);
    fs->framesize = (uint8_t)sz;
  }
}

/* Reserve registers. */
static void bcreg_reserve(FuncState *fs, BCReg n)
{
  bcreg_bump(fs, n);
  fs->freereg += n;
}

/* Free register. */
static void bcreg_free(FuncState *fs, BCReg reg)
{
  if (reg >= fs->nactvar) {
    fs->freereg--;
    lua_assert(reg == fs->freereg);
  }
}

/* Free register for expression. */
static void expr_free(FuncState *fs, ExpDesc *e)
{
  if (e->k == VNONRELOC)
    bcreg_free(fs, e->u.s.info);
}

/* -- Bytecode emitter ---------------------------------------------------- */

/* Emit bytecode instruction. */
static BCPos bcemit_INS(FuncState *fs, BCIns ins)
{
  BCPos pc = fs->pc;
  LexState *ls = fs->ls;
  jmp_patchval(fs, fs->jpc, pc, NO_REG, pc);
  fs->jpc = NO_JMP;
  if (LJ_UNLIKELY(pc >= fs->bclim)) {
    ptrdiff_t base = fs->bcbase - ls->bcstack;
    checklimit(fs, ls->sizebcstack, LJ_MAX_BCINS, "bytecode instructions");
    lj_mem_growvec(fs->L, ls->bcstack, ls->sizebcstack, LJ_MAX_BCINS,BCInsLine);
```

```
    fs->bclim = (BCPos)(ls->sizebcstack - base);
    fs->bcbase = ls->bcstack + base;
  }
  fs->bcbase[pc].ins = ins;
  fs->bcbase[pc].line = ls->lastline;
  fs->pc = pc+1;
  return pc;
}

#define bcemit_ABC(fs, o, a, b, c)        bcemit_INS(fs, BCINS_ABC(o, a, b, c))
#define bcemit_AD(fs, o, a, d)               bcemit_INS(fs, BCINS_AD(o, a, d))
#define bcemit_AJ(fs, o, a, j)               bcemit_INS(fs, BCINS_AJ(o, a, j))

#define bcptr(fs, e)                          (&(fs)->bcbase[(e)->u.s.info].ins)

/* -- Bytecode emitter for expressions ----------------------------------- */

/* Discharge non-constant expression to any register. */
static void expr_discharge(FuncState *fs, ExpDesc *e)
{
  BCIns ins;
  if (e->k == VUPVAL) {
    ins = BCINS_AD(BC_UGET, 0, e->u.s.info);
  } else if (e->k == VGLOBAL) {
    ins = BCINS_AD(BC_GGET, 0, const_str(fs, e));
  } else if (e->k == VINDEXED) {
    BCReg rc = e->u.s.aux;
    if ((int32_t)rc < 0) {
      ins = BCINS_ABC(BC_TGETS, 0, e->u.s.info, ~rc);
    } else if (rc > BCMAX_C) {
      ins = BCINS_ABC(BC_TGETB, 0, e->u.s.info, rc-(BCMAX_C+1));
    } else {
      bcreg_free(fs, rc);
      ins = BCINS_ABC(BC_TGETV, 0, e->u.s.info, rc);
    }
    bcreg_free(fs, e->u.s.info);
  } else if (e->k == VCALL) {
    e->u.s.info = e->u.s.aux;
    e->k = VNONRELOC;
    return;
  } else if (e->k == VLOCAL) {
    e->k = VNONRELOC;
    return;
  } else {
    return;
  }
  e->u.s.info = bcemit_INS(fs, ins);
  e->k = VRELOCABLE;
}

/* Emit bytecode to set a range of registers to nil. */
static void bcemit_nil(FuncState *fs, BCReg from, BCReg n)
{
  if (fs->pc > fs->lasttarget) {  /* No jumps to current position? */
    BCIns *ip = &fs->bcbase[fs->pc-1].ins;
    BCReg pto, pfrom = bc_a(*ip);
    switch (bc_op(*ip)) {  /* Try to merge with the previous instruction. */
    case BC_KPRI:
      if (bc_d(*ip) != ~LJ_TNIL) break;
      if (from == pfrom) {
        if (n == 1) return;
      } else if (from == pfrom+1) {
        from = pfrom;
        n++;
      } else {
        break;
      }
      *ip = BCINS_AD(BC_KNIL, from, from+n-1);  /* Replace KPRI. */
      return;
    case BC_KNIL:
      pto = bc_d(*ip);
      if (pfrom <= from && from <= pto+1) {  /* Can we connect both ranges? */
        if (from+n-1 > pto)
          setbc_d(ip, from+n-1);  /* Patch previous instruction range. */
        return;
      }
```

```
      break;
    default:
      break;
    }
  }
  /* Emit new instruction or replace old instruction. */
  bcemit_INS(fs, n == 1 ? BCINS_AD(BC_KPRI, from, VKNIL) :
                          BCINS_AD(BC_KNIL, from, from+n-1));
}

/* Discharge an expression to a specific register. Ignore branches. */
static void expr_toreg_nobranch(FuncState *fs, ExpDesc *e, BCReg reg)
{
  BCIns ins;
  expr_discharge(fs, e);
  if (e->k == VKSTR) {
    ins = BCINS_AD(BC_KSTR, reg, const_str(fs, e));
  } else if (e->k == VKNUM) {
#if LJ_DUALNUM
    cTValue *tv = expr_numtv(e);
    if (tvisint(tv) && checki16(intV(tv)))
      ins = BCINS_AD(BC_KSHORT, reg, (BCReg)(uint16_t)intV(tv));
    else
#else
    lua_Number n = expr_numberV(e);
    int32_t k = lj_num2int(n);
    if (checki16(k) && n == (lua_Number)k)
      ins = BCINS_AD(BC_KSHORT, reg, (BCReg)(uint16_t)k);
    else
#endif
      ins = BCINS_AD(BC_KNUM, reg, const_num(fs, e));
#if LJ_HASFFI
  } else if (e->k == VKCDATA) {
    fs->flags |= PROTO_FFI;
    ins = BCINS_AD(BC_KCDATA, reg,
                   const_gc(fs, obj2gco(cdataV(&e->u.nval)), LJ_TCDATA));
#endif
  } else if (e->k == VRELOCABLE) {
    setbc_a(bcptr(fs, e), reg);
    goto noins;
  } else if (e->k == VNONRELOC) {
    if (reg == e->u.s.info)
      goto noins;
    ins = BCINS_AD(BC_MOV, reg, e->u.s.info);
  } else if (e->k == VKNIL) {
    bcemit_nil(fs, reg, 1);
    goto noins;
  } else if (e->k <= VKTRUE) {
    ins = BCINS_AD(BC_KPRI, reg, const_pri(e));
  } else {
    lua_assert(e->k == VVOID || e->k == VJMP);
    return;
  }
  bcemit_INS(fs, ins);
noins:
  e->u.s.info = reg;
  e->k = VNONRELOC;
}

/* Forward declaration. */
static BCPos bcemit_jmp(FuncState *fs);

/* Discharge an expression to a specific register. */
static void expr_toreg(FuncState *fs, ExpDesc *e, BCReg reg)
{
  expr_toreg_nobranch(fs, e, reg);
  if (e->k == VJMP)
    jmp_append(fs, &e->t, e->u.s.info);  /* Add it to the true jump list. */
  if (expr_hasjump(e)) {  /* Discharge expression with branches. */
    BCPos jend, jfalse = NO_JMP, jtrue = NO_JMP;
    if (jmp_novalue(fs, e->t) || jmp_novalue(fs, e->f)) {
      BCPos jval = (e->k == VJMP) ? NO_JMP : bcemit_jmp(fs);
      jfalse = bcemit_AD(fs, BC_KPRI, reg, VKFALSE);
      bcemit_AJ(fs, BC_JMP, fs->freereg, 1);
      jtrue = bcemit_AD(fs, BC_KPRI, reg, VKTRUE);
      jmp_tohere(fs, jval);
```

```
571     }
572     jend = fs->pc;
573     fs->lasttarget = jend;
574     jmp_patchval(fs, e->f, jend, reg, jfalse);
575     jmp_patchval(fs, e->t, jend, reg, jtrue);
576   }
577   e->f = e->t = NO_JMP;
578   e->u.s.info = reg;
579   e->k = VNONRELOC;
580 }
581
582 /* Discharge an expression to the next free register. */
583 static void expr_tonextreg(FuncState *fs, ExpDesc *e)
584 {
585   expr_discharge(fs, e);
586   expr_free(fs, e);
587   bcreg_reserve(fs, 1);
588   expr_toreg(fs, e, fs->freereg - 1);
589 }
590
591 /* Discharge an expression to any register. */
592 static BCReg expr_toanyreg(FuncState *fs, ExpDesc *e)
593 {
594   expr_discharge(fs, e);
595   if (e->k == VNONRELOC) {
596     if (!expr_hasjump(e)) return e->u.s.info;  /* Already in a register. */
597     if (e->u.s.info >= fs->nactvar) {
598       expr_toreg(fs, e, e->u.s.info);  /* Discharge to temp. register. */
599       return e->u.s.info;
600     }
601   }
602   expr_tonextreg(fs, e);  /* Discharge to next register. */
603   return e->u.s.info;
604 }
605
606 /* Partially discharge expression to a value. */
607 static void expr_toval(FuncState *fs, ExpDesc *e)
608 {
609   if (expr_hasjump(e))
610     expr_toanyreg(fs, e);
611   else
612     expr_discharge(fs, e);
613 }
614
615 /* Emit store for LHS expression. */
616 static void bcemit_store(FuncState *fs, ExpDesc *var, ExpDesc *e)
617 {
618   BCIns ins;
619   if (var->k == VLOCAL) {
620     fs->ls->vstack[var->u.s.aux].info |= VSTACK_VAR_RW;
621     expr_free(fs, e);
622     expr_toreg(fs, e, var->u.s.info);
623     return;
624   } else if (var->k == VUPVAL) {
625     fs->ls->vstack[var->u.s.aux].info |= VSTACK_VAR_RW;
626     expr_toval(fs, e);
627     if (e->k <= VKTRUE)
628       ins = BCINS_AD(BC_USETP, var->u.s.info, const_pri(e));
629     else if (e->k == VKSTR)
630       ins = BCINS_AD(BC_USETS, var->u.s.info, const_str(fs, e));
631     else if (e->k == VKNUM)
632       ins = BCINS_AD(BC_USETN, var->u.s.info, const_num(fs, e));
633     else
634       ins = BCINS_AD(BC_USETV, var->u.s.info, expr_toanyreg(fs, e));
635   } else if (var->k == VGLOBAL) {
636     BCReg ra = expr_toanyreg(fs, e);
637     ins = BCINS_AD(BC_GSET, ra, const_str(fs, var));
638   } else {
639     BCReg ra, rc;
640     lua_assert(var->k == VINDEXED);
641     ra = expr_toanyreg(fs, e);
642     rc = var->u.s.aux;
643     if ((int32_t)rc < 0) {
644       ins = BCINS_ABC(BC_TSETS, ra, var->u.s.info, ~rc);
645     } else if (rc > BCMAX_C) {
646       ins = BCINS_ABC(BC_TSETB, ra, var->u.s.info, rc-(BCMAX_C+1));
```

```
    } else {
      /* Free late alloced key reg to avoid assert on free of value reg. */
      /* This can only happen when called from expr_table(). */
      lua_assert(e->k != VNONRELOC || ra < fs->nactvar ||
                 rc < ra || (bcreg_free(fs, rc),1));
      ins = BCINS_ABC(BC_TSETV, ra, var->u.s.info, rc);
    }
  }
  bcemit_INS(fs, ins);
  expr_free(fs, e);
}

/* Emit method lookup expression. */
static void bcemit_method(FuncState *fs, ExpDesc *e, ExpDesc *key)
{
  BCReg idx, func, obj = expr_toanyreg(fs, e);
  expr_free(fs, e);
  func = fs->freereg;
  bcemit_AD(fs, BC_MOV, func+1+LJ_FR2, obj);  /* Copy object to 1st argument. */
  lua_assert(expr_isstrk(key));
  idx = const_str(fs, key);
  if (idx <= BCMAX_C) {
    bcreg_reserve(fs, 2+LJ_FR2);
    bcemit_ABC(fs, BC_TGETS, func, obj, idx);
  } else {
    bcreg_reserve(fs, 3+LJ_FR2);
    bcemit_AD(fs, BC_KSTR, func+2+LJ_FR2, idx);
    bcemit_ABC(fs, BC_TGETV, func, obj, func+2+LJ_FR2);
    fs->freereg--;
  }
  e->u.s.info = func;
  e->k = VNONRELOC;
}

/* -- Bytecode emitter for branches ---------------------------------------- */

/* Emit unconditional branch. */
static BCPos bcemit_jmp(FuncState *fs)
{
  BCPos jpc = fs->jpc;
  BCPos j = fs->pc - 1;
  BCIns *ip = &fs->bcbase[j].ins;
  fs->jpc = NO_JMP;
  if ((int32_t)j >= (int32_t)fs->lasttarget && bc_op(*ip) == BC_UCLO) {
    setbc_j(ip, NO_JMP);
    fs->lasttarget = j+1;
  } else {
    j = bcemit_AJ(fs, BC_JMP, fs->freereg, NO_JMP);
  }
  jmp_append(fs, &j, jpc);
  return j;
}

/* Invert branch condition of bytecode instruction. */
static void invertcond(FuncState *fs, ExpDesc *e)
{
  BCIns *ip = &fs->bcbase[e->u.s.info - 1].ins;
  setbc_op(ip, bc_op(*ip)^1);
}

/* Emit conditional branch. */
static BCPos bcemit_branch(FuncState *fs, ExpDesc *e, int cond)
{
  BCPos pc;
  if (e->k == VRELOCABLE) {
    BCIns *ip = bcptr(fs, e);
    if (bc_op(*ip) == BC_NOT) {
      *ip = BCINS_AD(cond ? BC_ISF : BC_IST, 0, bc_d(*ip));
      return bcemit_jmp(fs);
    }
  }
  if (e->k != VNONRELOC) {
    bcreg_reserve(fs, 1);
    expr_toreg_nobranch(fs, e, fs->freereg-1);
  }
  bcemit_AD(fs, cond ? BC_ISTC : BC_ISFC, NO_REG, e->u.s.info);
```

```
  pc = bcemit_jmp(fs);
  expr_free(fs, e);
  return pc;
}

/* Emit branch on true condition. */
static void bcemit_branch_t(FuncState *fs, ExpDesc *e)
{
  BCPos pc;
  expr_discharge(fs, e);
  if (e->k == VKSTR || e->k == VKNUM || e->k == VKTRUE)
    pc = NO_JMP;  /* Never jump. */
  else if (e->k == VJMP)
    invertcond(fs, e), pc = e->u.s.info;
  else if (e->k == VKFALSE || e->k == VKNIL)
    expr_toreg_nobranch(fs, e, NO_REG), pc = bcemit_jmp(fs);
  else
    pc = bcemit_branch(fs, e, 0);
  jmp_append(fs, &e->f, pc);
  jmp_tohere(fs, e->t);
  e->t = NO_JMP;
}

/* Emit branch on false condition. */
static void bcemit_branch_f(FuncState *fs, ExpDesc *e)
{
  BCPos pc;
  expr_discharge(fs, e);
  if (e->k == VKNIL || e->k == VKFALSE)
    pc = NO_JMP;  /* Never jump. */
  else if (e->k == VJMP)
    pc = e->u.s.info;
  else if (e->k == VKSTR || e->k == VKNUM || e->k == VKTRUE)
    expr_toreg_nobranch(fs, e, NO_REG), pc = bcemit_jmp(fs);
  else
    pc = bcemit_branch(fs, e, 1);
  jmp_append(fs, &e->t, pc);
  jmp_tohere(fs, e->f);
  e->f = NO_JMP;
}

/* -- Bytecode emitter for operators ----------------------------------- */

/* Try constant-folding of arithmetic operators. */
static int foldarith(BinOpr opr, ExpDesc *e1, ExpDesc *e2)
{
  TValue o;
  lua_Number n;
  if (!expr_isnumk_nojump(e1) || !expr_isnumk_nojump(e2)) return 0;
  n = lj_vm_foldarith(expr_numberV(e1), expr_numberV(e2), (int)opr-OPR_ADD);
  setnumV(&o, n);
  if (tvisnan(&o) || tvismzero(&o)) return 0;  /* Avoid NaN and -0 as consts. */
  if (LJ_DUALNUM) {
    int32_t k = lj_num2int(n);
    if ((lua_Number)k == n) {
      setintV(&e1->u.nval, k);
      return 1;
    }
  }
  setnumV(&e1->u.nval, n);
  return 1;
}

/* Emit arithmetic operator. */
static void bcemit_arith(FuncState *fs, BinOpr opr, ExpDesc *e1, ExpDesc *e2)
{
  BCReg rb, rc, t;
  uint32_t op;
  if (foldarith(opr, e1, e2))
    return;
  if (opr == OPR_POW) {
    op = BC_POW;
    rc = expr_toanyreg(fs, e2);
    rb = expr_toanyreg(fs, e1);
  } else {
    op = opr-OPR_ADD+BC_ADDVV;
```

```
    /* Must discharge 2nd operand first since VINDEXED might free regs. */
    expr_toval(fs, e2);
    if (expr_isnumk(e2) && (rc = const_num(fs, e2)) <= BCMAX_C)
      op -= BC_ADDVV-BC_ADDVN;
    else
      rc = expr_toanyreg(fs, e2);
    /* 1st operand discharged by bcemit_binop_left, but need KNUM/KSHORT. */
    lua_assert(expr_isnumk(e1) || e1->k == VNONRELOC);
    expr_toval(fs, e1);
    /* Avoid two consts to satisfy bytecode constraints. */
    if (expr_isnumk(e1) && !expr_isnumk(e2) &&
        (t = const_num(fs, e1)) <= BCMAX_B) {
      rb = rc; rc = t; op -= BC_ADDVV-BC_ADDNV;
    } else {
      rb = expr_toanyreg(fs, e1);
    }
  }
  /* Using expr_free might cause asserts if the order is wrong. */
  if (e1->k == VNONRELOC && e1->u.s.info >= fs->nactvar) fs->freereg--;
  if (e2->k == VNONRELOC && e2->u.s.info >= fs->nactvar) fs->freereg--;
  e1->u.s.info = bcemit_ABC(fs, op, 0, rb, rc);
  e1->k = VRELOCABLE;
}

/* Emit comparison operator. */
static void bcemit_comp(FuncState *fs, BinOpr opr, ExpDesc *e1, ExpDesc *e2)
{
  ExpDesc *eret = e1;
  BCIns ins;
  expr_toval(fs, e1);
  if (opr == OPR_EQ || opr == OPR_NE) {
    BCOp op = opr == OPR_EQ ? BC_ISEQV : BC_ISNEV;
    BCReg ra;
    if (expr_isk(e1)) { e1 = e2; e2 = eret; }  /* Need constant in 2nd arg. */
    ra = expr_toanyreg(fs, e1);  /* First arg must be in a reg. */
    expr_toval(fs, e2);
    switch (e2->k) {
    case VKNIL: case VKFALSE: case VKTRUE:
      ins = BCINS_AD(op+(BC_ISEQP-BC_ISEQV), ra, const_pri(e2));
      break;
    case VKSTR:
      ins = BCINS_AD(op+(BC_ISEQS-BC_ISEQV), ra, const_str(fs, e2));
      break;
    case VKNUM:
      ins = BCINS_AD(op+(BC_ISEQN-BC_ISEQV), ra, const_num(fs, e2));
      break;
    default:
      ins = BCINS_AD(op, ra, expr_toanyreg(fs, e2));
      break;
    }
  } else {
    uint32_t op = opr-OPR_LT+BC_ISLT;
    BCReg ra, rd;
    if ((op-BC_ISLT) & 1) {  /* GT -> LT, GE -> LE */
      e1 = e2; e2 = eret;  /* Swap operands. */
      op = ((op-BC_ISLT)^3)+BC_ISLT;
      expr_toval(fs, e1);
    }
    rd = expr_toanyreg(fs, e2);
    ra = expr_toanyreg(fs, e1);
    ins = BCINS_AD(op, ra, rd);
  }
  /* Using expr_free might cause asserts if the order is wrong. */
  if (e1->k == VNONRELOC && e1->u.s.info >= fs->nactvar) fs->freereg--;
  if (e2->k == VNONRELOC && e2->u.s.info >= fs->nactvar) fs->freereg--;
  bcemit_INS(fs, ins);
  eret->u.s.info = bcemit_jmp(fs);
  eret->k = VJMP;
}

/* Fixup left side of binary operator. */
static void bcemit_binop_left(FuncState *fs, BinOpr op, ExpDesc *e)
{
  if (op == OPR_AND) {
    bcemit_branch_t(fs, e);
  } else if (op == OPR_OR) {
```

```
    bcemit_branch_f(fs, e);
  } else if (op == OPR_CONCAT) {
    expr_tonextreg(fs, e);
  } else if (op == OPR_EQ || op == OPR_NE) {
    if (!expr_isk_nojump(e)) expr_toanyreg(fs, e);
  } else {
    if (!expr_isnumk_nojump(e)) expr_toanyreg(fs, e);
  }
}

/* Emit binary operator. */
static void bcemit_binop(FuncState *fs, BinOpr op, ExpDesc *e1, ExpDesc *e2)
{
  if (op <= OPR_POW) {
    bcemit_arith(fs, op, e1, e2);
  } else if (op == OPR_AND) {
    lua_assert(e1->t == NO_JMP);  /* List must be closed. */
    expr_discharge(fs, e2);
    jmp_append(fs, &e2->f, e1->f);
    *e1 = *e2;
  } else if (op == OPR_OR) {
    lua_assert(e1->f == NO_JMP);  /* List must be closed. */
    expr_discharge(fs, e2);
    jmp_append(fs, &e2->t, e1->t);
    *e1 = *e2;
  } else if (op == OPR_CONCAT) {
    expr_toval(fs, e2);
    if (e2->k == VRELOCABLE && bc_op(*bcptr(fs, e2)) == BC_CAT) {
      lua_assert(e1->u.s.info == bc_b(*bcptr(fs, e2))-1);
      expr_free(fs, e1);
      setbc_b(bcptr(fs, e2), e1->u.s.info);
      e1->u.s.info = e2->u.s.info;
    } else {
      expr_tonextreg(fs, e2);
      expr_free(fs, e2);
      expr_free(fs, e1);
      e1->u.s.info = bcemit_ABC(fs, BC_CAT, 0, e1->u.s.info, e2->u.s.info);
    }
    e1->k = VRELOCABLE;
  } else {
    lua_assert(op == OPR_NE || op == OPR_EQ ||
               op == OPR_LT || op == OPR_GE || op == OPR_LE || op == OPR_GT);
    bcemit_comp(fs, op, e1, e2);
  }
}

/* Emit unary operator. */
static void bcemit_unop(FuncState *fs, BCOp op, ExpDesc *e)
{
  if (op == BC_NOT) {
    /* Swap true and false lists. */
    { BCPos temp = e->f; e->f = e->t; e->t = temp; }
    jmp_dropval(fs, e->f);
    jmp_dropval(fs, e->t);
    expr_discharge(fs, e);
    if (e->k == VKNIL || e->k == VKFALSE) {
      e->k = VKTRUE;
      return;
    } else if (expr_isk(e) || (LJ_HASFFI && e->k == VKCDATA)) {
      e->k = VKFALSE;
      return;
    } else if (e->k == VJMP) {
      invertcond(fs, e);
      return;
    } else if (e->k == VRELOCABLE) {
      bcreg_reserve(fs, 1);
      setbc_a(bcptr(fs, e), fs->freereg-1);
      e->u.s.info = fs->freereg-1;
      e->k = VNONRELOC;
    } else {
      lua_assert(e->k == VNONRELOC);
    }
  } else {
    lua_assert(op == BC_UNM || op == BC_LEN);
    if (op == BC_UNM && !expr_hasjump(e)) {  /* Constant-fold negations. */
#if LJ_HASFFI
```

```
      if (e->k == VKCDATA) {  /* Fold in-place since cdata is not interned. */
        GCcdata *cd = cdataV(&e->u.nval);
        int64_t *p = (int64_t *)cdataptr(cd);
        if (cd->ctypeid == CTID_COMPLEX_DOUBLE)
          p[1] ^= (int64_t)U64x(80000000,00000000);
        else
          *p = -*p;
        return;
      } else
#endif
      if (expr_isnumk(e) && !expr_numiszero(e)) {  /* Avoid folding to -0. */
        TValue *o = expr_numtv(e);
        if (tvisint(o)) {
          int32_t k = intV(o);
          if (k == -k)
            setnumV(o, -(lua_Number)k);
          else
            setintV(o, -k);
          return;
        } else {
          o->u64 ^= U64x(80000000,00000000);
          return;
        }
      }
    }
    expr_toanyreg(fs, e);
  }
  expr_free(fs, e);
  e->u.s.info = bcemit_AD(fs, op, 0, e->u.s.info);
  e->k = VRELOCABLE;
}

/* -- Lexer support ------------------------------------------------------ */

/* Check and consume optional token. */
static int lex_opt(LexState *ls, LexToken tok)
{
  if (ls->tok == tok) {
    lj_lex_next(ls);
    return 1;
  }
  return 0;
}

/* Check and consume token. */
static void lex_check(LexState *ls, LexToken tok)
{
  if (ls->tok != tok)
    err_token(ls, tok);
  lj_lex_next(ls);
}

/* Check for matching token. */
static void lex_match(LexState *ls, LexToken what, LexToken who, BCLine line)
{
  if (!lex_opt(ls, what)) {
    if (line == ls->linenumber) {
      err_token(ls, what);
    } else {
      const char *swhat = lj_lex_token2str(ls, what);
      const char *swho = lj_lex_token2str(ls, who);
      lj_lex_error(ls, ls->tok, LJ_ERR_XMATCH, swhat, swho, line);
    }
  }
}

/* Check for string token. */
static GCstr *lex_str(LexState *ls)
{
  GCstr *s;
  if (ls->tok != TK_name && (LJ_52 || ls->tok != TK_goto))
    err_token(ls, TK_name);
  s = strV(&ls->tokval);
  lj_lex_next(ls);
  return s;
}
```

```
1027
1028 /* -- Variable handling ---------------------------------------------- */
1029
1030 #define var_get(ls, fs, i)        ((ls)->vstack[(fs)->varmap[(i)]])
1031
1032 /* Define a new local variable. */
1033 static void var_new(LexState *ls, BCReg n, GCstr *name)
1034 {
1035   FuncState *fs = ls->fs;
1036   MSize vtop = ls->vtop;
1037   checklimit(fs, fs->nactvar+n, LJ_MAX_LOCVAR, "local variables");
1038   if (LJ_UNLIKELY(vtop >= ls->sizevstack)) {
1039     if (ls->sizevstack >= LJ_MAX_VSTACK)
1040       lj_lex_error(ls, 0, LJ_ERR_XLIMC, LJ_MAX_VSTACK);
1041     lj_mem_growvec(ls->L, ls->vstack, ls->sizevstack, LJ_MAX_VSTACK, VarInfo);
1042   }
1043   lua_assert((uintptr_t)name < VARNAME__MAX ||
1044              lj_tab_getstr(fs->kt, name) != NULL);
1045   /* NOBARRIER: name is anchored in fs->kt and ls->vstack is not a GCobj. */
1046   setgcref(ls->vstack[vtop].name, obj2gco(name));
1047   fs->varmap[fs->nactvar+n] = (uint16_t)vtop;
1048   ls->vtop = vtop+1;
1049 }
1050
1051 #define var_new_lit(ls, n, v) \
1052   var_new(ls, (n), lj_parse_keepstr(ls, "" v, sizeof(v)-1))
1053
1054 #define var_new_fixed(ls, n, vn) \
1055   var_new(ls, (n), (GCstr *)(uintptr_t)(vn))
1056
1057 /* Add local variables. */
1058 static void var_add(LexState *ls, BCReg nvars)
1059 {
1060   FuncState *fs = ls->fs;
1061   BCReg nactvar = fs->nactvar;
1062   while (nvars--) {
1063     VarInfo *v = &var_get(ls, fs, nactvar);
1064     v->startpc = fs->pc;
1065     v->slot = nactvar++;
1066     v->info = 0;
1067   }
1068   fs->nactvar = nactvar;
1069 }
1070
1071 /* Remove local variables. */
1072 static void var_remove(LexState *ls, BCReg tolevel)
1073 {
1074   FuncState *fs = ls->fs;
1075   while (fs->nactvar > tolevel)
1076     var_get(ls, fs, --fs->nactvar).endpc = fs->pc;
1077 }
1078
1079 /* Lookup local variable name. */
1080 static BCReg var_lookup_local(FuncState *fs, GCstr *n)
1081 {
1082   int i;
1083   for (i = fs->nactvar-1; i >= 0; i--) {
1084     if (n == strref(var_get(fs->ls, fs, i).name))
1085       return (BCReg)i;
1086   }
1087   return (BCReg)-1;  /* Not found. */
1088 }
1089
1090 /* Lookup or add upvalue index. */
1091 static MSize var_lookup_uv(FuncState *fs, MSize vidx, ExpDesc *e)
1092 {
1093   MSize i, n = fs->nuv;
1094   for (i = 0; i < n; i++)
1095     if (fs->uvmap[i] == vidx)
1096       return i;  /* Already exists. */
1097   /* Otherwise create a new one. */
1098   checklimit(fs, fs->nuv, LJ_MAX_UPVAL, "upvalues");
1099   lua_assert(e->k == VLOCAL || e->k == VUPVAL);
1100   fs->uvmap[n] = (uint16_t)vidx;
1101   fs->uvtmp[n] = (uint16_t)(e->k == VLOCAL ? vidx : LJ_MAX_VSTACK+e->u.s.info);
1102   fs->nuv = n+1;
```

```
  return n;
}

/* Forward declaration. */
static void fscope_uvmark(FuncState *fs, BCReg level);

/* Recursively lookup variables in enclosing functions. */
static MSize var_lookup_(FuncState *fs, GCstr *name, ExpDesc *e, int first)
{
  if (fs) {
    BCReg reg = var_lookup_local(fs, name);
    if ((int32_t)reg >= 0) {  /* Local in this function? */
      expr_init(e, VLOCAL, reg);
      if (!first)
        fscope_uvmark(fs, reg);  /* Scope now has an upvalue. */
      return (MSize)(e->u.s.aux = (uint32_t)fs->varmap[reg]);
    } else {
      MSize vidx = var_lookup_(fs->prev, name, e, 0);  /* Var in outer func? */
      if ((int32_t)vidx >= 0) {  /* Yes, make it an upvalue here. */
        e->u.s.info = (uint8_t)var_lookup_uv(fs, vidx, e);
        e->k = VUPVAL;
        return vidx;
      }
    }
  } else {  /* Not found in any function, must be a global. */
    expr_init(e, VGLOBAL, 0);
    e->u.sval = name;
  }
  return (MSize)-1;  /* Global. */
}

/* Lookup variable name. */
#define var_lookup(ls, e) \
  var_lookup_((ls)->fs, lex_str(ls), (e), 1)

/* -- Goto an label handling ---------------------------------------------- */

/* Add a new goto or label. */
static MSize gola_new(LexState *ls, GCstr *name, uint8_t info, BCPos pc)
{
  FuncState *fs = ls->fs;
  MSize vtop = ls->vtop;
  if (LJ_UNLIKELY(vtop >= ls->sizevstack)) {
    if (ls->sizevstack >= LJ_MAX_VSTACK)
      lj_lex_error(ls, 0, LJ_ERR_XLIMC, LJ_MAX_VSTACK);
    lj_mem_growvec(ls->L, ls->vstack, ls->sizevstack, LJ_MAX_VSTACK, VarInfo);
  }
  lua_assert(name == NAME_BREAK || lj_tab_getstr(fs->kt, name) != NULL);
  /* NOBARRIER: name is anchored in fs->kt and ls->vstack is not a GCobj. */
  setgcref(ls->vstack[vtop].name, obj2gco(name));
  ls->vstack[vtop].startpc = pc;
  ls->vstack[vtop].slot = (uint8_t)fs->nactvar;
  ls->vstack[vtop].info = info;
  ls->vtop = vtop+1;
  return vtop;
}

#define gola_isgoto(v)               ((v)->info & VSTACK_GOTO)
#define gola_islabel(v)               ((v)->info & VSTACK_LABEL)
#define gola_isgotolabel(v)        ((v)->info & (VSTACK_GOTO|VSTACK_LABEL))

/* Patch goto to jump to label. */
static void gola_patch(LexState *ls, VarInfo *vg, VarInfo *vl)
{
  FuncState *fs = ls->fs;
  BCPos pc = vg->startpc;
  setgcrefnull(vg->name);  /* Invalidate pending goto. */
  setbc_a(&fs->bcbase[pc].ins, vl->slot);
  jmp_patch(fs, pc, vl->startpc);
}

/* Patch goto to close upvalues. */
static void gola_close(LexState *ls, VarInfo *vg)
{
  FuncState *fs = ls->fs;
  BCPos pc = vg->startpc;
```

```
  BCIns *ip = &fs->bcbase[pc].ins;
  lua_assert(gola_isgoto(vg));
  lua_assert(bc_op(*ip) == BC_JMP || bc_op(*ip) == BC_UCLO);
  setbc_a(ip, vg->slot);
  if (bc_op(*ip) == BC_JMP) {
    BCPos next = jmp_next(fs, pc);
    if (next != NO_JMP) jmp_patch(fs, next, pc);  /* Jump to UCLO. */
    setbc_op(ip, BC_UCLO);  /* Turn into UCLO. */
    setbc_j(ip, NO_JMP);
  }
}

/* Resolve pending forward gotos for label. */
static void gola_resolve(LexState *ls, FuncScope *bl, MSize idx)
{
  VarInfo *vg = ls->vstack + bl->vstart;
  VarInfo *vl = ls->vstack + idx;
  for (; vg < vl; vg++)
    if (gcrefeq(vg->name, vl->name) && gola_isgoto(vg)) {
      if (vg->slot < vl->slot) {
        GCstr *name = strref(var_get(ls, ls->fs, vg->slot).name);
        lua_assert((uintptr_t)name >= VARNAME__MAX);
        ls->linenumber = ls->fs->bcbase[vg->startpc].line;
        lua_assert(strref(vg->name) != NAME_BREAK);
        lj_lex_error(ls, 0, LJ_ERR_XGSCOPE,
                     strdata(strref(vg->name)), strdata(name));
      }
      gola_patch(ls, vg, vl);
    }
}

/* Fixup remaining gotos and labels for scope. */
static void gola_fixup(LexState *ls, FuncScope *bl)
{
  VarInfo *v = ls->vstack + bl->vstart;
  VarInfo *ve = ls->vstack + ls->vtop;
  for (; v < ve; v++) {
    GCstr *name = strref(v->name);
    if (name != NULL) {  /* Only consider remaining valid gotos/labels. */
      if (gola_islabel(v)) {
        VarInfo *vg;
        setgcrefnull(v->name);  /* Invalidate label that goes out of scope. */
        for (vg = v+1; vg < ve; vg++)  /* Resolve pending backward gotos. */
          if (strref(vg->name) == name && gola_isgoto(vg)) {
            if ((bl->flags&FSCOPE_UPVAL) && vg->slot > v->slot)
              gola_close(ls, vg);
            gola_patch(ls, vg, v);
          }
      } else if (gola_isgoto(v)) {
        if (bl->prev) {  /* Propagate goto or break to outer scope. */
          bl->prev->flags |= name == NAME_BREAK ? FSCOPE_BREAK : FSCOPE_GOLA;
          v->slot = bl->nactvar;
          if ((bl->flags & FSCOPE_UPVAL))
            gola_close(ls, v);
        } else {  /* No outer scope: undefined goto label or no loop. */
          ls->linenumber = ls->fs->bcbase[v->startpc].line;
          if (name == NAME_BREAK)
            lj_lex_error(ls, 0, LJ_ERR_XBREAK);
          else
            lj_lex_error(ls, 0, LJ_ERR_XLUNDEF, strdata(name));
        }
      }
    }
  }
}

/* Find existing label. */
static VarInfo *gola_findlabel(LexState *ls, GCstr *name)
{
  VarInfo *v = ls->vstack + ls->fs->bl->vstart;
  VarInfo *ve = ls->vstack + ls->vtop;
  for (; v < ve; v++)
    if (strref(v->name) == name && gola_islabel(v))
      return v;
  return NULL;
}
```

```
1255
1256 /* -- Scope handling ------------------------------------------------------ */
1257
1258 /* Begin a scope. */
1259 static void fscope_begin(FuncState *fs, FuncScope *bl, int flags)
1260 {
1261   bl->nactvar = (uint8_t)fs->nactvar;
1262   bl->flags = flags;
1263   bl->vstart = fs->ls->vtop;
1264   bl->prev = fs->bl;
1265   fs->bl = bl;
1266   lua_assert(fs->freereg == fs->nactvar);
1267 }
1268
1269 /* End a scope. */
1270 static void fscope_end(FuncState *fs)
1271 {
1272   FuncScope *bl = fs->bl;
1273   LexState *ls = fs->ls;
1274   fs->bl = bl->prev;
1275   var_remove(ls, bl->nactvar);
1276   fs->freereg = fs->nactvar;
1277   lua_assert(bl->nactvar == fs->nactvar);
1278   if ((bl->flags & (FSCOPE_UPVAL|FSCOPE_NOCLOSE)) == FSCOPE_UPVAL)
1279     bcemit_AJ(fs, BC_UCLO, bl->nactvar, 0);
1280   if ((bl->flags & FSCOPE_BREAK)) {
1281     if ((bl->flags & FSCOPE_LOOP)) {
1282       MSize idx = gola_new(ls, NAME_BREAK, VSTACK_LABEL, fs->pc);
1283       ls->vtop = idx;  /* Drop break label immediately. */
1284       gola_resolve(ls, bl, idx);
1285       return;
1286     }  /* else: need the fixup step to propagate the breaks. */
1287   } else if (!(bl->flags & FSCOPE_GOLA)) {
1288     return;
1289   }
1290   gola_fixup(ls, bl);
1291 }
1292
1293 /* Mark scope as having an upvalue. */
1294 static void fscope_uvmark(FuncState *fs, BCReg level)
1295 {
1296   FuncScope *bl;
1297   for (bl = fs->bl; bl && bl->nactvar > level; bl = bl->prev)
1298     ;
1299   if (bl)
1300     bl->flags |= FSCOPE_UPVAL;
1301 }
1302
1303 /* -- Function state management ------------------------------------------- */
1304
1305 /* Fixup bytecode for prototype. */
1306 static void fs_fixup_bc(FuncState *fs, GCproto *pt, BCIns *bc, MSize n)
1307 {
1308   BCInsLine *base = fs->bcbase;
1309   MSize i;
1310   pt->sizebc = n;
1311   bc[0] = BCINS_AD((fs->flags & PROTO_VARARG) ? BC_FUNCV : BC_FUNCF,
1312                    fs->framesize, 0);
1313   for (i = 1; i < n; i++)
1314     bc[i] = base[i].ins;
1315 }
1316
1317 /* Fixup upvalues for child prototype, step #2. */
1318 static void fs_fixup_uv2(FuncState *fs, GCproto *pt)
1319 {
1320   VarInfo *vstack = fs->ls->vstack;
1321   uint16_t *uv = proto_uv(pt);
1322   MSize i, n = pt->sizeuv;
1323   for (i = 0; i < n; i++) {
1324     VarIndex vidx = uv[i];
1325     if (vidx >= LJ_MAX_VSTACK)
1326       uv[i] = vidx - LJ_MAX_VSTACK;
1327     else if ((vstack[vidx].info & VSTACK_VAR_RW))
1328       uv[i] = vstack[vidx].slot | PROTO_UV_LOCAL;
1329     else
1330       uv[i] = vstack[vidx].slot | PROTO_UV_LOCAL | PROTO_UV_IMMUTABLE;
```

```
  }
}

/* Fixup constants for prototype. */
static void fs_fixup_k(FuncState *fs, GCproto *pt, void *kptr)
{
  GCtab *kt;
  TValue *array;
  Node *node;
  MSize i, hmask;
  checklimitgt(fs, fs->nkn, BCMAX_D+1, "constants");
  checklimitgt(fs, fs->nkgc, BCMAX_D+1, "constants");
  setmref(pt->k, kptr);
  pt->sizekn = fs->nkn;
  pt->sizekgc = fs->nkgc;
  kt = fs->kt;
  array = tvref(kt->array);
  for (i = 0; i < kt->asize; i++)
    if (tvhaskslot(&array[i])) {
      TValue *tv = &((TValue *)kptr)[tvkslot(&array[i])];
      if (LJ_DUALNUM)
        setintV(tv, (int32_t)i);
      else
        setnumV(tv, (lua_Number)i);
    }
  node = noderef(kt->node);
  hmask = kt->hmask;
  for (i = 0; i <= hmask; i++) {
    Node *n = &node[i];
    if (tvhaskslot(&n->val)) {
      ptrdiff_t kidx = (ptrdiff_t)tvkslot(&n->val);
      lua_assert(!tvisint(&n->key));
      if (tvisnum(&n->key)) {
        TValue *tv = &((TValue *)kptr)[kidx];
        if (LJ_DUALNUM) {
          lua_Number nn = numV(&n->key);
          int32_t k = lj_num2int(nn);
          lua_assert(!tvismzero(&n->key));
          if ((lua_Number)k == nn)
            setintV(tv, k);
          else
            *tv = n->key;
        } else {
          *tv = n->key;
        }
      } else {
        GCobj *o = gcV(&n->key);
        setgcref(((GCRef *)kptr)[~kidx], o);
        lj_gc_objbarrier(fs->L, pt, o);
        if (tvisproto(&n->key))
          fs_fixup_uv2(fs, gco2pt(o));
      }
    }
  }
}

/* Fixup upvalues for prototype, step #1. */
static void fs_fixup_uv1(FuncState *fs, GCproto *pt, uint16_t *uv)
{
  setmref(pt->uv, uv);
  pt->sizeuv = fs->nuv;
  memcpy(uv, fs->uvtmp, fs->nuv*sizeof(VarIndex));
}

#ifndef LUAJIT_DISABLE_DEBUGINFO
/* Prepare lineinfo for prototype. */
static size_t fs_prep_line(FuncState *fs, BCLine numline)
{
  return (fs->pc-1) << (numline < 256 ? 0 : numline < 65536 ? 1 : 2);
}

/* Fixup lineinfo for prototype. */
static void fs_fixup_line(FuncState *fs, GCproto *pt,
                          void *lineinfo, BCLine numline)
{
  BCInsLine *base = fs->bcbase + 1;
```

```
  BCLine first = fs->linedefined;
  MSize i = 0, n = fs->pc-1;
  pt->firstline = fs->linedefined;
  pt->numline = numline;
  setmref(pt->lineinfo, lineinfo);
  if (LJ_LIKELY(numline < 256)) {
    uint8_t *li = (uint8_t *)lineinfo;
    do {
      BCLine delta = base[i].line - first;
      lua_assert(delta >= 0 && delta < 256);
      li[i] = (uint8_t)delta;
    } while (++i < n);
  } else if (LJ_LIKELY(numline < 65536)) {
    uint16_t *li = (uint16_t *)lineinfo;
    do {
      BCLine delta = base[i].line - first;
      lua_assert(delta >= 0 && delta < 65536);
      li[i] = (uint16_t)delta;
    } while (++i < n);
  } else {
    uint32_t *li = (uint32_t *)lineinfo;
    do {
      BCLine delta = base[i].line - first;
      lua_assert(delta >= 0);
      li[i] = (uint32_t)delta;
    } while (++i < n);
  }
}

/* Prepare variable info for prototype. */
static size_t fs_prep_var(LexState *ls, FuncState *fs, size_t *ofsvar)
{
  VarInfo *vs =ls->vstack, *ve;
  MSize i, n;
  BCPos lastpc;
  lj_buf_reset(&ls->sb);  /* Copy to temp. string buffer. */
  /* Store upvalue names. */
  for (i = 0, n = fs->nuv; i < n; i++) {
    GCstr *s = strref(vs[fs->uvmap[i]].name);
    MSize len = s->len+1;
    char *p = lj_buf_more(&ls->sb, len);
    p = lj_buf_wmem(p, strdata(s), len);
    setsbufP(&ls->sb, p);
  }
  *ofsvar = sbuflen(&ls->sb);
  lastpc = 0;
  /* Store local variable names and compressed ranges. */
  for (ve = vs + ls->vtop, vs += fs->vbase; vs < ve; vs++) {
    if (!gola_isgotolabel(vs)) {
      GCstr *s = strref(vs->name);
      BCPos startpc;
      char *p;
      if ((uintptr_t)s < VARNAME__MAX) {
        p = lj_buf_more(&ls->sb, 1 + 2*5);
        *p++ = (char)(uintptr_t)s;
      } else {
        MSize len = s->len+1;
        p = lj_buf_more(&ls->sb, len + 2*5);
        p = lj_buf_wmem(p, strdata(s), len);
      }
      startpc = vs->startpc;
      p = lj_strfmt_wuleb128(p, startpc-lastpc);
      p = lj_strfmt_wuleb128(p, vs->endpc-startpc);
      setsbufP(&ls->sb, p);
      lastpc = startpc;
    }
  }
  lj_buf_putb(&ls->sb, '\0');  /* Terminator for varinfo. */
  return sbuflen(&ls->sb);
}

/* Fixup variable info for prototype. */
static void fs_fixup_var(LexState *ls, GCproto *pt, uint8_t *p, size_t ofsvar)
{
  setmref(pt->uvinfo, p);
  setmref(pt->varinfo, (char *)p + ofsvar);
```

```
  memcpy(p, sbufB(&ls->sb), sbuflen(&ls->sb));  /* Copy from temp. buffer. */
}
#else

/* Initialize with empty debug info, if disabled. */
#define fs_prep_line(fs, numline)               (UNUSED(numline), 0)
#define fs_fixup_line(fs, pt, li, numline) \
  pt->firstline = pt->numline = 0, setmref((pt)->lineinfo, NULL)
#define fs_prep_var(ls, fs, ofsvar)               (UNUSED(ofsvar), 0)
#define fs_fixup_var(ls, pt, p, ofsvar) \
  setmref((pt)->uvinfo, NULL), setmref((pt)->varinfo, NULL)

#endif

/* Check if bytecode op returns. */
static int bcopisret(BCOp op)
{
  switch (op) {
  case BC_CALLMT: case BC_CALLT:
  case BC_RETM: case BC_RET: case BC_RET0: case BC_RET1:
    return 1;
  default:
    return 0;
  }
}

/* Fixup return instruction for prototype. */
static void fs_fixup_ret(FuncState *fs)
{
  BCPos lastpc = fs->pc;
  if (lastpc <= fs->lasttarget || !bcopisret(bc_op(fs->bcbase[lastpc-1].ins))) {
    if ((fs->bl->flags & FSCOPE_UPVAL))
      bcemit_AJ(fs, BC_UCLO, 0, 0);
    bcemit_AD(fs, BC_RET0, 0, 1);  /* Need final return. */
  }
  fs->bl->flags |= FSCOPE_NOCLOSE;  /* Handled above. */
  fscope_end(fs);
  lua_assert(fs->bl == NULL);
  /* May need to fixup returns encoded before first function was created. */
  if (fs->flags & PROTO_FIXUP_RETURN) {
    BCPos pc;
    for (pc = 1; pc < lastpc; pc++) {
      BCIns ins = fs->bcbase[pc].ins;
      BCPos offset;
      switch (bc_op(ins)) {
      case BC_CALLMT: case BC_CALLT:
      case BC_RETM: case BC_RET: case BC_RET0: case BC_RET1:
        offset = bcemit_INS(fs, ins);  /* Copy original instruction. */
        fs->bcbase[offset].line = fs->bcbase[pc].line;
        offset = offset-(pc+1)+BCBIAS_J;
        if (offset > BCMAX_D)
          err_syntax(fs->ls, LJ_ERR_XFIXUP);
        /* Replace with UCLO plus branch. */
        fs->bcbase[pc].ins = BCINS_AD(BC_UCLO, 0, offset);
        break;
      case BC_UCLO:
        return;  /* We're done. */
      default:
        break;
      }
    }
  }
}

/* Finish a FuncState and return the new prototype. */
static GCproto *fs_finish(LexState *ls, BCLine line)
{
  lua_State *L = ls->L;
  FuncState *fs = ls->fs;
  BCLine numline = line - fs->linedefined;
  size_t sizept, ofsk, ofsuv, ofsli, ofsdbg, ofsvar;
  GCproto *pt;

  /* Apply final fixups. */
  fs_fixup_ret(fs);

```

```
  /* Calculate total size of prototype including all colocated arrays. */
  sizept = sizeof(GCproto) + fs->pc*sizeof(BCIns) + fs->nkgc*sizeof(GCRef);
  sizept = (sizept + sizeof(TValue)-1) & ~(sizeof(TValue)-1);
  ofsk = sizept; sizept += fs->nkn*sizeof(TValue);
  ofsuv = sizept; sizept += ((fs->nuv+1)&~1)*2;
  ofsli = sizept; sizept += fs_prep_line(fs, numline);
  ofsdbg = sizept; sizept += fs_prep_var(ls, fs, &ofsvar);

  /* Allocate prototype and initialize its fields. */
  pt = (GCproto *)lj_mem_newgco(L, (MSize)sizept);
  pt->gct = ~LJ_TPROTO;
  pt->sizept = (MSize)sizept;
  pt->trace = 0;
  pt->flags = (uint8_t)(fs->flags & ~(PROTO_HAS_RETURN|PROTO_FIXUP_RETURN));
  pt->numparams = fs->numparams;
  pt->framesize = fs->framesize;
  setgcref(pt->chunkname, obj2gco(ls->chunkname));

  /* Close potentially uninitialized gap between bc and kgc. */
  *(uint32_t *)((char *)pt + ofsk - sizeof(GCRef)*(fs->nkgc+1)) = 0;
  fs_fixup_bc(fs, pt, (BCIns *)((char *)pt + sizeof(GCproto)), fs->pc);
  fs_fixup_k(fs, pt, (void *)((char *)pt + ofsk));
  fs_fixup_uv1(fs, pt, (uint16_t *)((char *)pt + ofsuv));
  fs_fixup_line(fs, pt, (void *)((char *)pt + ofsli), numline);
  fs_fixup_var(ls, pt, (uint8_t *)((char *)pt + ofsdbg), ofsvar);

  lj_vmevent_send(L, BC,
    setprotoV(L, L->top++, pt);
  );

  L->top--;  /* Pop table of constants. */
  ls->vtop = fs->vbase;  /* Reset variable stack. */
  ls->fs = fs->prev;
  lua_assert(ls->fs != NULL || ls->tok == TK_eof);
  return pt;
}

/* Initialize a new FuncState. */
static void fs_init(LexState *ls, FuncState *fs)
{
  lua_State *L = ls->L;
  fs->prev = ls->fs; ls->fs = fs;  /* Append to list. */
  fs->ls = ls;
  fs->vbase = ls->vtop;
  fs->L = L;
  fs->pc = 0;
  fs->lasttarget = 0;
  fs->jpc = NO_JMP;
  fs->freereg = 0;
  fs->nkgc = 0;
  fs->nkn = 0;
  fs->nactvar = 0;
  fs->nuv = 0;
  fs->bl = NULL;
  fs->flags = 0;
  fs->framesize = 1;  /* Minimum frame size. */
  fs->kt = lj_tab_new(L, 0, 0);
  /* Anchor table of constants in stack to avoid being collected. */
  settabV(L, L->top, fs->kt);
  incr_top(L);
}

/* -- Expressions -------------------------------------------------------- */

/* Forward declaration. */
static void expr(LexState *ls, ExpDesc *v);

/* Return string expression. */
static void expr_str(LexState *ls, ExpDesc *e)
{
  expr_init(e, VKSTR, 0);
  e->u.sval = lex_str(ls);
}

/* Return index expression. */
static void expr_index(FuncState *fs, ExpDesc *t, ExpDesc *e)
```

```
{
  /* Already called: expr_toval(fs, e). */
  t->k = VINDEXED;
  if (expr_isnumk(e)) {
#if LJ_DUALNUM
    if (tvisint(expr_numtv(e))) {
      int32_t k = intV(expr_numtv(e));
      if (checku8(k)) {
        t->u.s.aux = BCMAX_C+1+(uint32_t)k;  /* 256..511: const byte key */
        return;
      }
    }
#else
    lua_Number n = expr_numberV(e);
    int32_t k = lj_num2int(n);
    if (checku8(k) && n == (lua_Number)k) {
      t->u.s.aux = BCMAX_C+1+(uint32_t)k;  /* 256..511: const byte key */
      return;
    }
#endif
  } else if (expr_isstrk(e)) {
    BCReg idx = const_str(fs, e);
    if (idx <= BCMAX_C) {
      t->u.s.aux = ~idx;  /* -256..-1: const string key */
      return;
    }
  }
  t->u.s.aux = expr_toanyreg(fs, e);  /* 0..255: register */
}

/* Parse index expression with named field. */
static void expr_field(LexState *ls, ExpDesc *v)
{
  FuncState *fs = ls->fs;
  ExpDesc key;
  expr_toanyreg(fs, v);
  lj_lex_next(ls);  /* Skip dot or colon. */
  expr_str(ls, &key);
  expr_index(fs, v, &key);
}

/* Parse index expression with brackets. */
static void expr_bracket(LexState *ls, ExpDesc *v)
{
  lj_lex_next(ls);  /* Skip '['. */
  expr(ls, v);
  expr_toval(ls->fs, v);
  lex_check(ls, ']');
}

/* Get value of constant expression. */
static void expr_kvalue(TValue *v, ExpDesc *e)
{
  if (e->k <= VKTRUE) {
    setpriV(v, ~(uint32_t)e->k);
  } else if (e->k == VKSTR) {
    setgcVraw(v, obj2gco(e->u.sval), LJ_TSTR);
  } else {
    lua_assert(tvisnumber(expr_numtv(e)));
    *v = *expr_numtv(e);
  }
}

/* Parse table constructor expression. */
static void expr_table(LexState *ls, ExpDesc *e)
{
  FuncState *fs = ls->fs;
  BCLine line = ls->linenumber;
  GCtab *t = NULL;
  int vcall = 0, needarr = 0, fixt = 0;
  uint32_t narr = 1;  /* First array index. */
  uint32_t nhash = 0;  /* Number of hash entries. */
  BCReg freg = fs->freereg;
  BCPos pc = bcemit_AD(fs, BC_TNEW, freg, 0);
  expr_init(e, VNONRELOC, freg);
  bcreg_reserve(fs, 1);
```

```
  freg++;
  lex_check(ls, '{');
  while (ls->tok != '}') {
    ExpDesc key, val;
    vcall = 0;
    if (ls->tok == '[') {
      expr_bracket(ls, &key);  /* Already calls expr_toval. */
      if (!expr_isk(&key)) expr_index(fs, e, &key);
      if (expr_isnumk(&key) && expr_numiszero(&key)) needarr = 1; else nhash++;
      lex_check(ls, '=');
    } else if ((ls->tok == TK_name || (!LJ_52 && ls->tok == TK_goto)) &&
               lj_lex_lookahead(ls) == '=') {
      expr_str(ls, &key);
      lex_check(ls, '=');
      nhash++;
    } else {
      expr_init(&key, VKNUM, 0);
      setintV(&key.u.nval, (int)narr);
      narr++;
      needarr = vcall = 1;
    }
    expr(ls, &val);
    if (expr_isk(&key) && key.k != VKNIL &&
        (key.k == VKSTR || expr_isk_nojump(&val))) {
      TValue k, *v;
      if (!t) {  /* Create template table on demand. */
        BCReg kidx;
        t = lj_tab_new(fs->L, needarr ? narr : 0, hsize2hbits(nhash));
        kidx = const_gc(fs, obj2gco(t), LJ_TTAB);
        fs->bcbase[pc].ins = BCINS_AD(BC_TDUP, freg-1, kidx);
      }
      vcall = 0;
      expr_kvalue(&k, &key);
      v = lj_tab_set(fs->L, t, &k);
      lj_gc_anybarriert(fs->L, t);
      if (expr_isk_nojump(&val)) {  /* Add const key/value to template table. */
        expr_kvalue(v, &val);
      } else {  /* Otherwise create dummy string key (avoids lj_tab_newkey). */
        settabV(fs->L, v, t);  /* Preserve key with table itself as value. */
        fixt = 1;   /* Fix this later, after all resizes. */
        goto nonconst;
      }
    } else {
    nonconst:
      if (val.k != VCALL) { expr_toanyreg(fs, &val); vcall = 0; }
      if (expr_isk(&key)) expr_index(fs, e, &key);
      bcemit_store(fs, e, &val);
    }
    fs->freereg = freg;
    if (!lex_opt(ls, ',') && !lex_opt(ls, ';')) break;
  }
  lex_match(ls, '}', '{', line);
  if (vcall) {
    BCInsLine *ilp = &fs->bcbase[fs->pc-1];
    ExpDesc en;
    lua_assert(bc_a(ilp->ins) == freg &&
               bc_op(ilp->ins) == (narr > 256 ? BC_TSETV : BC_TSETB));
    expr_init(&en, VKNUM, 0);
    en.u.nval.u32.lo = narr-1;
    en.u.nval.u32.hi = 0x43300000;  /* Biased integer to avoid denormals. */
    if (narr > 256) { fs->pc--; ilp--; }
    ilp->ins = BCINS_AD(BC_TSETM, freg, const_num(fs, &en));
    setbc_b(&ilp[-1].ins, 0);
  }
  if (pc == fs->pc-1) {  /* Make expr relocable if possible. */
    e->u.s.info = pc;
    fs->freereg--;
    e->k = VRELOCABLE;
  } else {
    e->k = VNONRELOC;  /* May have been changed by expr_index. */
  }
  if (!t) {  /* Construct TNEW RD: hhhhhaaaaaaaaaaa. */
    BCIns *ip = &fs->bcbase[pc].ins;
    if (!needarr) narr = 0;
    else if (narr < 3) narr = 3;
    else if (narr > 0x7ff) narr = 0x7ff;
```

```
    setbc_d(ip, narr|(hsize2hbits(nhash)<<11));
  } else {
    if (needarr && t->asize < narr)
      lj_tab_reasize(fs->L, t, narr-1);
    if (fixt) {  /* Fix value for dummy keys in template table. */
      Node *node = noderef(t->node);
      uint32_t i, hmask = t->hmask;
      for (i = 0; i <= hmask; i++) {
        Node *n = &node[i];
        if (tvistab(&n->val)) {
          lua_assert(tabV(&n->val) == t);
          setnilV(&n->val);  /* Turn value into nil. */
        }
      }
    }
    lj_gc_check(fs->L);
  }
}

/* Parse function parameters. */
static BCReg parse_params(LexState *ls, int needself)
{
  FuncState *fs = ls->fs;
  BCReg nparams = 0;
  lex_check(ls, '(');
  if (needself)
    var_new_lit(ls, nparams++, "self");
  if (ls->tok != ')') {
    do {
      if (ls->tok == TK_name || (!LJ_52 && ls->tok == TK_goto)) {
        var_new(ls, nparams++, lex_str(ls));
      } else if (ls->tok == TK_dots) {
        lj_lex_next(ls);
        fs->flags |= PROTO_VARARG;
        break;
      } else {
        err_syntax(ls, LJ_ERR_XPARAM);
      }
    } while (lex_opt(ls, ','));
  }
  var_add(ls, nparams);
  lua_assert(fs->nactvar == nparams);
  bcreg_reserve(fs, nparams);
  lex_check(ls, ')');
  return nparams;
}

/* Forward declaration. */
static void parse_chunk(LexState *ls);

/* Parse body of a function. */
static void parse_body(LexState *ls, ExpDesc *e, int needself, BCLine line)
{
  FuncState fs, *pfs = ls->fs;
  FuncScope bl;
  GCproto *pt;
  ptrdiff_t oldbase = pfs->bcbase - ls->bcstack;
  fs_init(ls, &fs);
  fscope_begin(&fs, &bl, 0);
  fs.linedefined = line;
  fs.numparams = (uint8_t)parse_params(ls, needself);
  fs.bcbase = pfs->bcbase + pfs->pc;
  fs.bclim = pfs->bclim - pfs->pc;
  bcemit_AD(&fs, BC_FUNCF, 0, 0);  /* Placeholder. */
  parse_chunk(ls);
  if (ls->tok != TK_end) lex_match(ls, TK_end, TK_function, line);
  pt = fs_finish(ls, (ls->lastline = ls->linenumber));
  pfs->bcbase = ls->bcstack + oldbase;  /* May have been reallocated. */
  pfs->bclim = (BCPos)(ls->sizebcstack - oldbase);
  /* Store new prototype in the constant array of the parent. */
  expr_init(e, VRELOCABLE,
            bcemit_AD(pfs, BC_FNEW, 0, const_gc(pfs, obj2gco(pt), LJ_TPROTO)));
#if LJ_HASFFI
  pfs->flags |= (fs.flags & PROTO_FFI);
#endif
  if (!(pfs->flags & PROTO_CHILD)) {
```

```
    if (pfs->flags & PROTO_HAS_RETURN)
      pfs->flags |= PROTO_FIXUP_RETURN;
    pfs->flags |= PROTO_CHILD;
  }
  lj_lex_next(ls);
}

/* Parse expression list. Last expression is left open. */
static BCReg expr_list(LexState *ls, ExpDesc *v)
{
  BCReg n = 1;
  expr(ls, v);
  while (lex_opt(ls, ',')) {
    expr_tonextreg(ls->fs, v);
    expr(ls, v);
    n++;
  }
  return n;
}

/* Parse function argument list. */
static void parse_args(LexState *ls, ExpDesc *e)
{
  FuncState *fs = ls->fs;
  ExpDesc args;
  BCIns ins;
  BCReg base;
  BCLine line = ls->linenumber;
  if (ls->tok == '(') {
#if !LJ_52
    if (line != ls->lastline)
      err_syntax(ls, LJ_ERR_XAMBIG);
#endif
    lj_lex_next(ls);
    if (ls->tok == ')') {  /* f(). */
      args.k = VVOID;
    } else {
      expr_list(ls, &args);
      if (args.k == VCALL)  /* f(a, b, g()) or f(a, b, ...). */
        setbc_b(bcptr(fs, &args), 0);  /* Pass on multiple results. */
    }
    lex_match(ls, ')', '(', line);
  } else if (ls->tok == '{') {
    expr_table(ls, &args);
  } else if (ls->tok == TK_string) {
    expr_init(&args, VKSTR, 0);
    args.u.sval = strV(&ls->tokval);
    lj_lex_next(ls);
  } else {
    err_syntax(ls, LJ_ERR_XFUNARG);
    return;  /* Silence compiler. */
  }
  lua_assert(e->k == VNONRELOC);
  base = e->u.s.info;  /* Base register for call. */
  if (args.k == VCALL) {
    ins = BCINS_ABC(BC_CALLM, base, 2, args.u.s.aux - base - 1 - LJ_FR2);
  } else {
    if (args.k != VVOID)
      expr_tonextreg(fs, &args);
    ins = BCINS_ABC(BC_CALL, base, 2, fs->freereg - base - LJ_FR2);
  }
  expr_init(e, VCALL, bcemit_INS(fs, ins));
  e->u.s.aux = base;
  fs->bcbase[fs->pc - 1].line = line;
  fs->freereg = base+1;  /* Leave one result by default. */
}

/* Parse primary expression. */
static void expr_primary(LexState *ls, ExpDesc *v)
{
  FuncState *fs = ls->fs;
  /* Parse prefix expression. */
  if (ls->tok == '(') {
    BCLine line = ls->linenumber;
    lj_lex_next(ls);
    expr(ls, v);
```

```
    lex_match(ls, ')', '(', line);
    expr_discharge(ls->fs, v);
  } else if (ls->tok == TK_name || (!LJ_52 && ls->tok == TK_goto)) {
    var_lookup(ls, v);
  } else {
    err_syntax(ls, LJ_ERR_XSYMBOL);
  }
  for (;;) {  /* Parse multiple expression suffixes. */
    if (ls->tok == '.') {
      expr_field(ls, v);
    } else if (ls->tok == '[') {
      ExpDesc key;
      expr_toanyreg(fs, v);
      expr_bracket(ls, &key);
      expr_index(fs, v, &key);
    } else if (ls->tok == ':') {
      ExpDesc key;
      lj_lex_next(ls);
      expr_str(ls, &key);
      bcemit_method(fs, v, &key);
      parse_args(ls, v);
    } else if (ls->tok == '(' || ls->tok == TK_string || ls->tok == '{') {
      expr_tonextreg(fs, v);
      if (LJ_FR2) bcreg_reserve(fs, 1);
      parse_args(ls, v);
    } else {
      break;
    }
  }
}

/* Parse simple expression. */
static void expr_simple(LexState *ls, ExpDesc *v)
{
  switch (ls->tok) {
  case TK_number:
    expr_init(v, (LJ_HASFFI && tviscdata(&ls->tokval)) ? VKCDATA : VKNUM, 0);
    copyTV(ls->L, &v->u.nval, &ls->tokval);
    break;
  case TK_string:
    expr_init(v, VKSTR, 0);
    v->u.sval = strV(&ls->tokval);
    break;
  case TK_nil:
    expr_init(v, VKNIL, 0);
    break;
  case TK_true:
    expr_init(v, VKTRUE, 0);
    break;
  case TK_false:
    expr_init(v, VKFALSE, 0);
    break;
  case TK_dots: {  /* Vararg. */
    FuncState *fs = ls->fs;
    BCReg base;
    checkcond(ls, fs->flags & PROTO_VARARG, LJ_ERR_XDOTS);
    bcreg_reserve(fs, 1);
    base = fs->freereg-1;
    expr_init(v, VCALL, bcemit_ABC(fs, BC_VARG, base, 2, fs->numparams));
    v->u.s.aux = base;
    break;
  }
  case '{':  /* Table constructor. */
    expr_table(ls, v);
    return;
  case TK_function:
    lj_lex_next(ls);
    parse_body(ls, v, 0, ls->linenumber);
    return;
  default:
    expr_primary(ls, v);
    return;
  }
  lj_lex_next(ls);
}

```

```
2015 /* Manage syntactic levels to avoid blowing up the stack. */
2016 static void synlevel_begin(LexState *ls)
2017 {
2018   if (++ls->level >= LJ_MAX_XLEVEL)
2019     lj_lex_error(ls, 0, LJ_ERR_XLEVELS);
2020 }
2021
2022 #define synlevel_end(ls)        ((ls)->level--)
2023
2024 /* Convert token to binary operator. */
2025 static BinOpr token2binop(LexToken tok)
2026 {
2027   switch (tok) {
2028   case '+':        return OPR_ADD;
2029   case '-':        return OPR_SUB;
2030   case '*':        return OPR_MUL;
2031   case '/':        return OPR_DIV;
2032   case '%':        return OPR_MOD;
2033   case '^':        return OPR_POW;
2034   case TK_concat: return OPR_CONCAT;
2035   case TK_ne:        return OPR_NE;
2036   case TK_eq:        return OPR_EQ;
2037   case '<':        return OPR_LT;
2038   case TK_le:        return OPR_LE;
2039   case '>':        return OPR_GT;
2040   case TK_ge:        return OPR_GE;
2041   case TK_and:        return OPR_AND;
2042   case TK_or:        return OPR_OR;
2043   default:        return OPR_NOBINOPR;
2044   }
2045 }
2046
2047 /* Priorities for each binary operator. ORDER OPR. */
2048 static const struct {
2049   uint8_t left;                /* Left priority. */
2050   uint8_t right;        /* Right priority. */
2051 } priority[] = {
2052   {6,6}, {6,6}, {7,7}, {7,7}, {7,7},        /* ADD SUB MUL DIV MOD */
2053   {10,9}, {5,4},                          /* POW CONCAT (right associative) */
2054   {3,3}, {3,3},                                 /* EQ NE */
2055   {3,3}, {3,3}, {3,3}, {3,3},                /* LT GE GT LE */
2056   {2,2}, {1,1}                                 /* AND OR */
2057 };
2058
2059 #define UNARY_PRIORITY                8  /* Priority for unary operators. */
2060
2061 /* Forward declaration. */
2062 static BinOpr expr_binop(LexState *ls, ExpDesc *v, uint32_t limit);
2063
2064 /* Parse unary expression. */
2065 static void expr_unop(LexState *ls, ExpDesc *v)
2066 {
2067   BCOp op;
2068   if (ls->tok == TK_not) {
2069     op = BC_NOT;
2070   } else if (ls->tok == '-') {
2071     op = BC_UNM;
2072   } else if (ls->tok == '#') {
2073     op = BC_LEN;
2074   } else {
2075     expr_simple(ls, v);
2076     return;
2077   }
2078   lj_lex_next(ls);
2079   expr_binop(ls, v, UNARY_PRIORITY);
2080   bcemit_unop(ls->fs, op, v);
2081 }
2082
2083 /* Parse binary expressions with priority higher than the limit. */
2084 static BinOpr expr_binop(LexState *ls, ExpDesc *v, uint32_t limit)
2085 {
2086   BinOpr op;
2087   synlevel_begin(ls);
2088   expr_unop(ls, v);
2089   op = token2binop(ls->tok);
2090   while (op != OPR_NOBINOPR && priority[op].left > limit) {
```

```
2091      ExpDesc v2;
2092      BinOpr nextop;
2093      lj_lex_next(ls);
2094      bcemit_binop_left(ls->fs, op, v);
2095      /* Parse binary expression with higher priority. */
2096      nextop = expr_binop(ls, &v2, priority[op].right);
2097      bcemit_binop(ls->fs, op, v, &v2);
2098      op = nextop;
2099    }
2100    synlevel_end(ls);
2101    return op;  /* Return unconsumed binary operator (if any). */
2102  }
2103
2104  /* Parse expression. */
2105  static void expr(LexState *ls, ExpDesc *v)
2106  {
2107    expr_binop(ls, v, 0);  /* Priority 0: parse whole expression. */
2108  }
2109
2110  /* Assign expression to the next register. */
2111  static void expr_next(LexState *ls)
2112  {
2113    ExpDesc e;
2114    expr(ls, &e);
2115    expr_tonextreg(ls->fs, &e);
2116  }
2117
2118  /* Parse conditional expression. */
2119  static BCPos expr_cond(LexState *ls)
2120  {
2121    ExpDesc v;
2122    expr(ls, &v);
2123    if (v.k == VKNIL) v.k = VKFALSE;
2124    bcemit_branch_t(ls->fs, &v);
2125    return v.f;
2126  }
2127
2128  /* -- Assignments ----------------------------------------------------- */
2129
2130  /* List of LHS variables. */
2131  typedef struct LHSVarList {
2132    ExpDesc v;                          /* LHS variable. */
2133    struct LHSVarList *prev;        /* Link to previous LHS variable. */
2134  } LHSVarList;
2135
2136  /* Eliminate write-after-read hazards for local variable assignment. */
2137  static void assign_hazard(LexState *ls, LHSVarList *lh, const ExpDesc *v)
2138  {
2139    FuncState *fs = ls->fs;
2140    BCReg reg = v->u.s.info;  /* Check against this variable. */
2141    BCReg tmp = fs->freereg;  /* Rename to this temp. register (if needed). */
2142    int hazard = 0;
2143    for (; lh; lh = lh->prev) {
2144      if (lh->v.k == VINDEXED) {
2145        if (lh->v.u.s.info == reg) {  /* t[i], t = 1, 2 */
2146          hazard = 1;
2147          lh->v.u.s.info = tmp;
2148        }
2149        if (lh->v.u.s.aux == reg) {  /* t[i], i = 1, 2 */
2150          hazard = 1;
2151          lh->v.u.s.aux = tmp;
2152        }
2153      }
2154    }
2155    if (hazard) {
2156      bcemit_AD(fs, BC_MOV, tmp, reg);  /* Rename conflicting variable. */
2157      bcreg_reserve(fs, 1);
2158    }
2159  }
2160
2161  /* Adjust LHS/RHS of an assignment. */
2162  static void assign_adjust(LexState *ls, BCReg nvars, BCReg nexps, ExpDesc *e)
2163  {
2164    FuncState *fs = ls->fs;
2165    int32_t extra = (int32_t)nvars - (int32_t)nexps;
2166    if (e->k == VCALL) {
```

```
    extra++;  /* Compensate for the VCALL itself. */
    if (extra < 0) extra = 0;
    setbc_b(bcptr(fs, e), extra+1);  /* Fixup call results. */
    if (extra > 1) bcreg_reserve(fs, (BCReg)extra-1);
  } else {
    if (e->k != VVOID)
      expr_tonextreg(fs, e);  /* Close last expression. */
    if (extra > 0) {  /* Leftover LHS are set to nil. */
      BCReg reg = fs->freereg;
      bcreg_reserve(fs, (BCReg)extra);
      bcemit_nil(fs, reg, (BCReg)extra);
    }
  }
}

/* Recursively parse assignment statement. */
static void parse_assignment(LexState *ls, LHSVarList *lh, BCReg nvars)
{
  ExpDesc e;
  checkcond(ls, VLOCAL <= lh->v.k && lh->v.k <= VINDEXED, LJ_ERR_XSYNTAX);
  if (lex_opt(ls, ',')) {  /* Collect LHS list and recurse upwards. */
    LHSVarList vl;
    vl.prev = lh;
    expr_primary(ls, &vl.v);
    if (vl.v.k == VLOCAL)
      assign_hazard(ls, lh, &vl.v);
    checklimit(ls->fs, ls->level + nvars, LJ_MAX_XLEVEL, "variable names");
    parse_assignment(ls, &vl, nvars+1);
  } else {  /* Parse RHS. */
    BCReg nexps;
    lex_check(ls, '=');
    nexps = expr_list(ls, &e);
    if (nexps == nvars) {
      if (e.k == VCALL) {
        if (bc_op(*bcptr(ls->fs, &e)) == BC_VARG) {  /* Vararg assignment. */
          ls->fs->freereg--;
          e.k = VRELOCABLE;
        } else {  /* Multiple call results. */
          e.u.s.info = e.u.s.aux;  /* Base of call is not relocatable. */
          e.k = VNONRELOC;
        }
      }
      bcemit_store(ls->fs, &lh->v, &e);
      return;
    }
    assign_adjust(ls, nvars, nexps, &e);
    if (nexps > nvars)
      ls->fs->freereg -= nexps - nvars;  /* Drop leftover regs. */
  }
  /* Assign RHS to LHS and recurse downwards. */
  expr_init(&e, VNONRELOC, ls->fs->freereg-1);
  bcemit_store(ls->fs, &lh->v, &e);
}

/* Parse call statement or assignment. */
static void parse_call_assign(LexState *ls)
{
  FuncState *fs = ls->fs;
  LHSVarList vl;
  expr_primary(ls, &vl.v);
  if (vl.v.k == VCALL) {  /* Function call statement. */
    setbc_b(bcptr(fs, &vl.v), 1);  /* No results. */
  } else {  /* Start of an assignment. */
    vl.prev = NULL;
    parse_assignment(ls, &vl, 1);
  }
}

/* Parse 'local' statement. */
static void parse_local(LexState *ls)
{
  if (lex_opt(ls, TK_function)) {  /* Local function declaration. */
    ExpDesc v, b;
    FuncState *fs = ls->fs;
    var_new(ls, 0, lex_str(ls));
    expr_init(&v, VLOCAL, fs->freereg);
```

```
    v.u.s.aux = fs->varmap[fs->freereg];
    bcreg_reserve(fs, 1);
    var_add(ls, 1);
    parse_body(ls, &b, 0, ls->linenumber);
    /* bcemit_store(fs, &v, &b) without setting VSTACK_VAR_RW. */
    expr_free(fs, &b);
    expr_toreg(fs, &b, v.u.s.info);
    /* The upvalue is in scope, but the local is only valid after the store. */
    var_get(ls, fs, fs->nactvar - 1).startpc = fs->pc;
  } else {  /* Local variable declaration. */
    ExpDesc e;
    BCReg nexps, nvars = 0;
    do {  /* Collect LHS. */
      var_new(ls, nvars++, lex_str(ls));
    } while (lex_opt(ls, ','));
    if (lex_opt(ls, '=')) {  /* Optional RHS. */
      nexps = expr_list(ls, &e);
    } else {  /* Or implicitly set to nil. */
      e.k = VVOID;
      nexps = 0;
    }
    assign_adjust(ls, nvars, nexps, &e);
    var_add(ls, nvars);
  }
}

/* Parse 'function' statement. */
static void parse_func(LexState *ls, BCLine line)
{
  FuncState *fs;
  ExpDesc v, b;
  int needself = 0;
  lj_lex_next(ls);  /* Skip 'function'. */
  /* Parse function name. */
  var_lookup(ls, &v);
  while (ls->tok == '.')  /* Multiple dot-separated fields. */
    expr_field(ls, &v);
  if (ls->tok == ':') {  /* Optional colon to signify method call. */
    needself = 1;
    expr_field(ls, &v);
  }
  parse_body(ls, &b, needself, line);
  fs = ls->fs;
  bcemit_store(fs, &v, &b);
  fs->bcbase[fs->pc - 1].line = line;  /* Set line for the store. */
}

/* -- Control transfer statements ---------------------------------------- */

/* Check for end of block. */
static int parse_isend(LexToken tok)
{
  switch (tok) {
  case TK_else: case TK_elseif: case TK_end: case TK_until: case TK_eof:
    return 1;
  default:
    return 0;
  }
}

/* Parse 'return' statement. */
static void parse_return(LexState *ls)
{
  BCIns ins;
  FuncState *fs = ls->fs;
  lj_lex_next(ls);  /* Skip 'return'. */
  fs->flags |= PROTO_HAS_RETURN;
  if (parse_isend(ls->tok) || ls->tok == ';') {  /* Bare return. */
    ins = BCINS_AD(BC_RET0, 0, 1);
  } else {  /* Return with one or more values. */
    ExpDesc e;  /* Receives the _last_ expression in the list. */
    BCReg nret = expr_list(ls, &e);
    if (nret == 1) {  /* Return one result. */
      if (e.k == VCALL) {  /* Check for tail call. */
        BCIns *ip = bcptr(fs, &e);
        /* It doesn't pay off to add BC_VARGT just for 'return ...'. */
```

```
2319          if (bc_op(*ip) == BC_VARG) goto notailcall;
2320          fs->pc--;
2321          ins = BCINS_AD(bc_op(*ip)-BC_CALL+BC_CALLT, bc_a(*ip), bc_c(*ip));
2322        } else {  /* Can return the result from any register. */
2323          ins = BCINS_AD(BC_RET1, expr_toanyreg(fs, &e), 2);
2324        }
2325      } else {
2326        if (e.k == VCALL) {  /* Append all results from a call. */
2327        notailcall:
2328          setbc_b(bcptr(fs, &e), 0);
2329          ins = BCINS_AD(BC_RETM, fs->nactvar, e.u.s.aux - fs->nactvar);
2330        } else {
2331          expr_tonextreg(fs, &e);  /* Force contiguous registers. */
2332          ins = BCINS_AD(BC_RET, fs->nactvar, nret+1);
2333        }
2334      }
2335    }
2336    if (fs->flags & PROTO_CHILD)
2337      bcemit_AJ(fs, BC_UCLO, 0, 0);  /* May need to close upvalues first. */
2338    bcemit_INS(fs, ins);
2339  }
2340
2341  /* Parse 'break' statement. */
2342  static void parse_break(LexState *ls)
2343  {
2344    ls->fs->bl->flags |= FSCOPE_BREAK;
2345    gola_new(ls, NAME_BREAK, VSTACK_GOTO, bcemit_jmp(ls->fs));
2346  }
2347
2348  /* Parse 'goto' statement. */
2349  static void parse_goto(LexState *ls)
2350  {
2351    FuncState *fs = ls->fs;
2352    GCstr *name = lex_str(ls);
2353    VarInfo *vl = gola_findlabel(ls, name);
2354    if (vl)  /* Treat backwards goto within same scope like a loop. */
2355      bcemit_AJ(fs, BC_LOOP, vl->slot, -1);  /* No BC range check. */
2356    fs->bl->flags |= FSCOPE_GOLA;
2357    gola_new(ls, name, VSTACK_GOTO, bcemit_jmp(fs));
2358  }
2359
2360  /* Parse label. */
2361  static void parse_label(LexState *ls)
2362  {
2363    FuncState *fs = ls->fs;
2364    GCstr *name;
2365    MSize idx;
2366    fs->lasttarget = fs->pc;
2367    fs->bl->flags |= FSCOPE_GOLA;
2368    lj_lex_next(ls);  /* Skip '::'. */
2369    name = lex_str(ls);
2370    if (gola_findlabel(ls, name))
2371      lj_lex_error(ls, 0, LJ_ERR_XLDUP, strdata(name));
2372    idx = gola_new(ls, name, VSTACK_LABEL, fs->pc);
2373    lex_check(ls, TK_label);
2374    /* Recursively parse trailing statements: labels and ';' (Lua 5.2 only). */
2375    for (;;) {
2376      if (ls->tok == TK_label) {
2377        synlevel_begin(ls);
2378        parse_label(ls);
2379        synlevel_end(ls);
2380      } else if (LJ_52 && ls->tok == ';') {
2381        lj_lex_next(ls);
2382      } else {
2383        break;
2384      }
2385    }
2386    /* Trailing label is considered to be outside of scope. */
2387    if (parse_isend(ls->tok) && ls->tok != TK_until)
2388      ls->vstack[idx].slot = fs->bl->nactvar;
2389    gola_resolve(ls, fs->bl, idx);
2390  }
2391
2392  /* -- Blocks, loops and conditional statements ---------------------------- */
2393
2394  /* Parse a block. */
```

```
2395 static void parse_block(LexState *ls)
2396 {
2397   FuncState *fs = ls->fs;
2398   FuncScope bl;
2399   fscope_begin(fs, &bl, 0);
2400   parse_chunk(ls);
2401   fscope_end(fs);
2402 }
2403
2404 /* Parse 'while' statement. */
2405 static void parse_while(LexState *ls, BCLine line)
2406 {
2407   FuncState *fs = ls->fs;
2408   BCPos start, loop, condexit;
2409   FuncScope bl;
2410   lj_lex_next(ls);  /* Skip 'while'. */
2411   start = fs->lasttarget = fs->pc;
2412   condexit = expr_cond(ls);
2413   fscope_begin(fs, &bl, FSCOPE_LOOP);
2414   lex_check(ls, TK_do);
2415   loop = bcemit_AD(fs, BC_LOOP, fs->nactvar, 0);
2416   parse_block(ls);
2417   jmp_patch(fs, bcemit_jmp(fs), start);
2418   lex_match(ls, TK_end, TK_while, line);
2419   fscope_end(fs);
2420   jmp_tohere(fs, condexit);
2421   jmp_patchins(fs, loop, fs->pc);
2422 }
2423
2424 /* Parse 'repeat' statement. */
2425 static void parse_repeat(LexState *ls, BCLine line)
2426 {
2427   FuncState *fs = ls->fs;
2428   BCPos loop = fs->lasttarget = fs->pc;
2429   BCPos condexit;
2430   FuncScope bl1, bl2;
2431   fscope_begin(fs, &bl1, FSCOPE_LOOP);  /* Breakable loop scope. */
2432   fscope_begin(fs, &bl2, 0);  /* Inner scope. */
2433   lj_lex_next(ls);  /* Skip 'repeat'. */
2434   bcemit_AD(fs, BC_LOOP, fs->nactvar, 0);
2435   parse_chunk(ls);
2436   lex_match(ls, TK_until, TK_repeat, line);
2437   condexit = expr_cond(ls);  /* Parse condition (still inside inner scope). */
2438   if (!(bl2.flags & FSCOPE_UPVAL)) {  /* No upvalues? Just end inner scope. */
2439     fscope_end(fs);
2440   } else {  /* Otherwise generate: cond: UCLO+JMP out, !cond: UCLO+JMP loop. */
2441     parse_break(ls);  /* Break from loop and close upvalues. */
2442     jmp_tohere(fs, condexit);
2443     fscope_end(fs);  /* End inner scope and close upvalues. */
2444     condexit = bcemit_jmp(fs);
2445   }
2446   jmp_patch(fs, condexit, loop);  /* Jump backwards if !cond. */
2447   jmp_patchins(fs, loop, fs->pc);
2448   fscope_end(fs);  /* End loop scope. */
2449 }
2450
2451 /* Parse numeric 'for'. */
2452 static void parse_for_num(LexState *ls, GCstr *varname, BCLine line)
2453 {
2454   FuncState *fs = ls->fs;
2455   BCReg base = fs->freereg;
2456   FuncScope bl;
2457   BCPos loop, loopend;
2458   /* Hidden control variables. */
2459   var_new_fixed(ls, FORL_IDX, VARNAME_FOR_IDX);
2460   var_new_fixed(ls, FORL_STOP, VARNAME_FOR_STOP);
2461   var_new_fixed(ls, FORL_STEP, VARNAME_FOR_STEP);
2462   /* Visible copy of index variable. */
2463   var_new(ls, FORL_EXT, varname);
2464   lex_check(ls, '=');
2465   expr_next(ls);
2466   lex_check(ls, ',');
2467   expr_next(ls);
2468   if (lex_opt(ls, ',')) {
2469     expr_next(ls);
2470   } else {
```

```
    bcemit_AD(fs, BC_KSHORT, fs->freereg, 1);  /* Default step is 1. */
    bcreg_reserve(fs, 1);
  }
  var_add(ls, 3);  /* Hidden control variables. */
  lex_check(ls, TK_do);
  loop = bcemit_AJ(fs, BC_FORI, base, NO_JMP);
  fscope_begin(fs, &bl, 0);  /* Scope for visible variables. */
  var_add(ls, 1);
  bcreg_reserve(fs, 1);
  parse_block(ls);
  fscope_end(fs);
  /* Perform loop inversion. Loop control instructions are at the end. */
  loopend = bcemit_AJ(fs, BC_FORL, base, NO_JMP);
  fs->bcbase[loopend].line = line;  /* Fix line for control ins. */
  jmp_patchins(fs, loopend, loop+1);
  jmp_patchins(fs, loop, fs->pc);
}

/* Try to predict whether the iterator is next() and specialize the bytecode.
** Detecting next() and pairs() by name is simplistic, but quite effective.
** The interpreter backs off if the check for the closure fails at runtime.
*/
static int predict_next(LexState *ls, FuncState *fs, BCPos pc)
{
  BCIns ins = fs->bcbase[pc].ins;
  GCstr *name;
  cTValue *o;
  switch (bc_op(ins)) {
  case BC_MOV:
    name = gco2str(gcref(var_get(ls, fs, bc_d(ins)).name));
    break;
  case BC_UGET:
    name = gco2str(gcref(ls->vstack[fs->uvmap[bc_d(ins)]].name));
    break;
  case BC_GGET:
    /* There's no inverse index (yet), so lookup the strings. */
    o = lj_tab_getstr(fs->kt, lj_str_newlit(ls->L, "pairs"));
    if (o && tvhaskslot(o) && tvkslot(o) == bc_d(ins))
      return 1;
    o = lj_tab_getstr(fs->kt, lj_str_newlit(ls->L, "next"));
    if (o && tvhaskslot(o) && tvkslot(o) == bc_d(ins))
      return 1;
    return 0;
  default:
    return 0;
  }
  return (name->len == 5 && !strcmp(strdata(name), "pairs")) ||
         (name->len == 4 && !strcmp(strdata(name), "next"));
}

/* Parse 'for' iterator. */
static void parse_for_iter(LexState *ls, GCstr *indexname)
{
  FuncState *fs = ls->fs;
  ExpDesc e;
  BCReg nvars = 0;
  BCLine line;
  BCReg base = fs->freereg + 3;
  BCPos loop, loopend, exprpc = fs->pc;
  FuncScope bl;
  int isnext;
  /* Hidden control variables. */
  var_new_fixed(ls, nvars++, VARNAME_FOR_GEN);
  var_new_fixed(ls, nvars++, VARNAME_FOR_STATE);
  var_new_fixed(ls, nvars++, VARNAME_FOR_CTL);
  /* Visible variables returned from iterator. */
  var_new(ls, nvars++, indexname);
  while (lex_opt(ls, ','))
    var_new(ls, nvars++, lex_str(ls));
  lex_check(ls, TK_in);
  line = ls->linenumber;
  assign_adjust(ls, 3, expr_list(ls, &e), &e);
  /* The iterator needs another 3 [4] slots (func [pc] | state ctl). */
  bcreg_bump(fs, 3+LJ_FR2);
  isnext = (nvars <= 5 && predict_next(ls, fs, exprpc));
  var_add(ls, 3);  /* Hidden control variables. */
```

```
  lex_check(ls, TK_do);
  loop = bcemit_AJ(fs, isnext ? BC_ISNEXT : BC_JMP, base, NO_JMP);
  fscope_begin(fs, &bl, 0);  /* Scope for visible variables. */
  var_add(ls, nvars-3);
  bcreg_reserve(fs, nvars-3);
  parse_block(ls);
  fscope_end(fs);
  /* Perform loop inversion. Loop control instructions are at the end. */
  jmp_patchins(fs, loop, fs->pc);
  bcemit_ABC(fs, isnext ? BC_ITERN : BC_ITERC, base, nvars-3+1, 2+1);
  loopend = bcemit_AJ(fs, BC_ITERL, base, NO_JMP);
  fs->bcbase[loopend-1].line = line;  /* Fix line for control ins. */
  fs->bcbase[loopend].line = line;
  jmp_patchins(fs, loopend, loop+1);
}

/* Parse 'for' statement. */
static void parse_for(LexState *ls, BCLine line)
{
  FuncState *fs = ls->fs;
  GCstr *varname;
  FuncScope bl;
  fscope_begin(fs, &bl, FSCOPE_LOOP);
  lj_lex_next(ls);  /* Skip 'for'. */
  varname = lex_str(ls);  /* Get first variable name. */
  if (ls->tok == '=')
    parse_for_num(ls, varname, line);
  else if (ls->tok == ',' || ls->tok == TK_in)
    parse_for_iter(ls, varname);
  else
    err_syntax(ls, LJ_ERR_XFOR);
  lex_match(ls, TK_end, TK_for, line);
  fscope_end(fs);  /* Resolve break list. */
}

/* Parse condition and 'then' block. */
static BCPos parse_then(LexState *ls)
{
  BCPos condexit;
  lj_lex_next(ls);  /* Skip 'if' or 'elseif'. */
  condexit = expr_cond(ls);
  lex_check(ls, TK_then);
  parse_block(ls);
  return condexit;
}

/* Parse 'if' statement. */
static void parse_if(LexState *ls, BCLine line)
{
  FuncState *fs = ls->fs;
  BCPos flist;
  BCPos escapelist = NO_JMP;
  flist = parse_then(ls);
  while (ls->tok == TK_elseif) {  /* Parse multiple 'elseif' blocks. */
    jmp_append(fs, &escapelist, bcemit_jmp(fs));
    jmp_tohere(fs, flist);
    flist = parse_then(ls);
  }
  if (ls->tok == TK_else) {  /* Parse optional 'else' block. */
    jmp_append(fs, &escapelist, bcemit_jmp(fs));
    jmp_tohere(fs, flist);
    lj_lex_next(ls);  /* Skip 'else'. */
    parse_block(ls);
  } else {
    jmp_append(fs, &escapelist, flist);
  }
  jmp_tohere(fs, escapelist);
  lex_match(ls, TK_end, TK_if, line);
}

/* -- Parse statements ---------------------------------------------------- */

/* Parse a statement. Returns 1 if it must be the last one in a chunk. */
static int parse_stmt(LexState *ls)
{
  BCLine line = ls->linenumber;
```

```
2623   switch (ls->tok) {
2624   case TK_if:
2625     parse_if(ls, line);
2626     break;
2627   case TK_while:
2628     parse_while(ls, line);
2629     break;
2630   case TK_do:
2631     lj_lex_next(ls);
2632     parse_block(ls);
2633     lex_match(ls, TK_end, TK_do, line);
2634     break;
2635   case TK_for:
2636     parse_for(ls, line);
2637     break;
2638   case TK_repeat:
2639     parse_repeat(ls, line);
2640     break;
2641   case TK_function:
2642     parse_func(ls, line);
2643     break;
2644   case TK_local:
2645     lj_lex_next(ls);
2646     parse_local(ls);
2647     break;
2648   case TK_return:
2649     parse_return(ls);
2650     return 1;  /* Must be last. */
2651   case TK_break:
2652     lj_lex_next(ls);
2653     parse_break(ls);
2654     return !LJ_52;  /* Must be last in Lua 5.1. */
2655 #if LJ_52
2656   case ';':
2657     lj_lex_next(ls);
2658     break;
2659 #endif
2660   case TK_label:
2661     parse_label(ls);
2662     break;
2663   case TK_goto:
2664     if (LJ_52 || lj_lex_lookahead(ls) == TK_name) {
2665       lj_lex_next(ls);
2666       parse_goto(ls);
2667       break;
2668     }  /* else: fallthrough */
2669   default:
2670     parse_call_assign(ls);
2671     break;
2672   }
2673   return 0;
2674 }
2675
2676 /* A chunk is a list of statements optionally separated by semicolons. */
2677 static void parse_chunk(LexState *ls)
2678 {
2679   int islast = 0;
2680   synlevel_begin(ls);
2681   while (!islast && !parse_isend(ls->tok)) {
2682     islast = parse_stmt(ls);
2683     lex_opt(ls, ';');
2684     lua_assert(ls->fs->framesize >= ls->fs->freereg &&
2685                ls->fs->freereg >= ls->fs->nactvar);
2686     ls->fs->freereg = ls->fs->nactvar;  /* Free registers after each stmt. */
2687   }
2688   synlevel_end(ls);
2689 }
2690
2691 /* Entry point of bytecode parser. */
2692 GCproto *lj_parse(LexState *ls)
2693 {
2694   FuncState fs;
2695   FuncScope bl;
2696   GCproto *pt;
2697   lua_State *L = ls->L;
2698 #ifdef LUAJIT_DISABLE_DEBUGINFO
```

```
2699    ls->chunkname = lj_str_newlit(L, "=");
2700  #else
2701    ls->chunkname = lj_str_newz(L, ls->chunkarg);
2702  #endif
2703    setstrV(L, L->top, ls->chunkname);  /* Anchor chunkname string. */
2704    incr_top(L);
2705    ls->level = 0;
2706    fs_init(ls, &fs);
2707    fs.linedefined = 0;
2708    fs.numparams = 0;
2709    fs.bcbase = NULL;
2710    fs.bclim = 0;
2711    fs.flags |= PROTO_VARARG;  /* Main chunk is always a vararg func. */
2712    fscope_begin(&fs, &bl, 0);
2713    bcemit_AD(&fs, BC_FUNCV, 0, 0);  /* Placeholder. */
2714    lj_lex_next(ls);  /* Read-ahead first token. */
2715    parse_chunk(ls);
2716    if (ls->tok != TK_eof)
2717      err_token(ls, TK_eof);
2718    pt = fs_finish(ls, ls->linenumber);
2719    L->top--;  /* Drop chunkname. */
2720    lua_assert(fs.prev == NULL);
2721    lua_assert(ls->fs == NULL);
2722    lua_assert(pt->sizeuv == 0);
2723    return pt;
2724  }
2725
```

One Level Up Top Level

src/lj_bc.h - luajit-2.0-src

Global variables defined

- lj_bc_mode
- lj_bc_ofs

Data types defined

- BCMode
- BCOp

Functions defined

- bc_isret

Macros defined

- BCBIAS_J
- BCDEF
- BCENUM
- BCENUM
- BCINS_ABC
- BCINS_AD
- BCINS_AJ
- BCMAX_A
- BCMAX_B
- BCMAX_C
- BCMAX_D
- BCMODE
- BCMODE_FF
- BCM___
- FF_next_N
- NO_JMP
- NO_REG
- _LJ_BC_H
- bc_a
- bc_b

- bc_c
- bc_d
- bc_j
- bc_op
- bcmode_a
- bcmode_b
- bcmode_c
- bcmode_d
- bcmode_hasd
- bcmode_mm
- setbc_a
- setbc_b
- setbc_byte
- setbc_c
- setbc_d
- setbc_j
- setbc_op

Source code

```
/*
** Bytecode instruction format.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_BC_H
#define _LJ_BC_H

#include "lj_def.h"
 #include "lj_arch.h"

 /* Bytecode instruction format, 32 bit wide, fields of 8 or 16 bit:
 **
 ** +----+----+----+----+
 ** | B  | C  | A  | OP | Format ABC
 ** +----+----+----+----+
 ** |    D    | A  | OP | Format AD
 ** +--------------------
 ** MSB               LSB
 **
 ** In-memory instructions are always stored in host byte order.
 */

 /* Operand ranges and related constants. */
 #define BCMAX_A                0xff
 #define BCMAX_B                0xff
 #define BCMAX_C                0xff
 #define BCMAX_D                0xffff
 #define BCBIAS_J        0x8000
 #define NO_REG                BCMAX_A
 #define NO_JMP                (~(BCPos)0)

 /* Macros to get instruction fields. */
 #define bc_op(i)        ((BCOp)((i)&0xff))
```

```
#define bc_a(i)                 ((BCReg)(((i)>>8)&0xff))
#define bc_b(i)                 ((BCReg)((i)>>24))
#define bc_c(i)                 ((BCReg)(((i)>>16)&0xff))
#define bc_d(i)                 ((BCReg)((i)>>16))
#define bc_j(i)                 ((ptrdiff_t)bc_d(i)-BCBIAS_J)

/* Macros to set instruction fields. */
#define setbc_byte(p, x, ofs) \
  ((uint8_t *)(p))[LJ_ENDIAN_SELECT(ofs, 3-ofs)] = (uint8_t)(x)
#define setbc_op(p, x)          setbc_byte(p, (x), 0)
#define setbc_a(p, x)         setbc_byte(p, (x), 1)
#define setbc_b(p, x)         setbc_byte(p, (x), 3)
#define setbc_c(p, x)         setbc_byte(p, (x), 2)
#define setbc_d(p, x) \
  ((uint16_t *)(p))[LJ_ENDIAN_SELECT(1, 0)] = (uint16_t)(x)
#define setbc_j(p, x)         setbc_d(p, (BCPos)((int32_t)(x)+BCBIAS_J))

/* Macros to compose instructions. */
#define BCINS_ABC(o, a, b, c) \
  (((BCIns)(o))|((BCIns)(a)<<8)|((BCIns)(b)<<24)|((BCIns)(c)<<16))
#define BCINS_AD(o, a, d) \
  (((BCIns)(o))|((BCIns)(a)<<8)|((BCIns)(d)<<16))
#define BCINS_AJ(o, a, j)         BCINS_AD(o, a, (BCPos)((int32_t)(j)+BCBIAS_J))

/* Bytecode instruction definition. Order matters, see below.
**
** (name, filler, Amode, Bmode, Cmode or Dmode, metamethod)
**
** The opcode name suffixes specify the type for RB/RC or RD:
** V = variable slot
** S = string const
** N = number const
** P = primitive type (~itype)
** B = unsigned byte literal
** M = multiple args/results
*/
#define BCDEF(_) \
  /* Comparison ops. ORDER OPR. */ \
  _(ISLT,         var,         ___,         var,         lt) \
  _(ISGE,         var,         ___,         var,         lt) \
  _(ISLE,         var,         ___,         var,         le) \
  _(ISGT,         var,         ___,         var,         le) \
  \
  _(ISEQV,         var,         ___,         var,         eq) \
  _(ISNEV,         var,         ___,         var,         eq) \
  _(ISEQS,         var,         ___,         str,         eq) \
  _(ISNES,         var,         ___,         str,         eq) \
  _(ISEQN,         var,         ___,         num,         eq) \
  _(ISNEN,         var,         ___,         num,         eq) \
  _(ISEQP,         var,         ___,         pri,         eq) \
  _(ISNEP,         var,         ___,         pri,         eq) \
  \
  /* Unary test and copy ops. */ \
  _(ISTC,         dst,         ___,         var,         ___) \
  _(ISFC,         dst,         ___,         var,         ___) \
  _(IST,          ___,         ___,         var,         ___) \
  _(ISF,          ___,         ___,         var,         ___) \
  _(ISTYPE,         var,         ___,         lit,         ___) \
  _(ISNUM,         var,         ___,         lit,         ___) \
  \
  /* Unary ops. */ \
  _(MOV,         dst,         ___,         var,         ___) \
  _(NOT,         dst,         ___,         var,         ___) \
  _(UNM,         dst,         ___,         var,         unm) \
  _(LEN,         dst,         ___,         var,         len) \
  \
  /* Binary ops. ORDER OPR. VV last, POW must be next. */ \
  _(ADDVN,         dst,         var,         num,         add) \
  _(SUBVN,         dst,         var,         num,         sub) \
  _(MULVN,         dst,         var,         num,         mul) \
  _(DIVVN,         dst,         var,         num,         div) \
  _(MODVN,         dst,         var,         num,         mod) \
  \
  _(ADDNV,         dst,         var,         num,         add) \
  _(SUBNV,         dst,         var,         num,         sub) \
  _(MULNV,         dst,         var,         num,         mul) \
```

```
  _(DIVNV,        dst,        var,        num,        div) \
  _(MODNV,        dst,        var,        num,        mod) \
  \
  _(ADDVV,        dst,        var,        var,        add) \
  _(SUBVV,        dst,        var,        var,        sub) \
  _(MULVV,        dst,        var,        var,        mul) \
  _(DIVVV,        dst,        var,        var,        div) \
  _(MODVV,        dst,        var,        var,        mod) \
  \
  _(POW,        dst,        var,        var,        pow) \
  _(CAT,        dst,        rbase,        rbase,        concat) \
  \
  /* Constant ops. */ \
  _(KSTR,        dst,        ___,        str,        ___) \
  _(KCDATA,        dst,        ___,        cdata,        ___) \
  _(KSHORT,        dst,        ___,        lits,        ___) \
  _(KNUM,        dst,        ___,        num,        ___) \
  _(KPRI,        dst,        ___,        pri,        ___) \
  _(KNIL,        base,        ___,        base,        ___) \
  \
  /* Upvalue and function ops. */ \
  _(UGET,        dst,        ___,        uv,        ___) \
  _(USETV,        uv,        ___,        var,        ___) \
  _(USETS,        uv,        ___,        str,        ___) \
  _(USETN,        uv,        ___,        num,        ___) \
  _(USETP,        uv,        ___,        pri,        ___) \
  _(UCLO,        rbase,        ___,        jump,        ___) \
  _(FNEW,        dst,        ___,        func,        gc) \
  \
  /* Table ops. */ \
  _(TNEW,        dst,        ___,        lit,        gc) \
  _(TDUP,        dst,        ___,        tab,        gc) \
  _(GGET,        dst,        ___,        str,        index) \
  _(GSET,        var,        ___,        str,        newindex) \
  _(TGETV,        dst,        var,        var,        index) \
  _(TGETS,        dst,        var,        str,        index) \
  _(TGETB,        dst,        var,        lit,        index) \
  _(TGETR,        dst,        var,        var,        index) \
  _(TSETV,        var,        var,        var,        newindex) \
  _(TSETS,        var,        var,        str,        newindex) \
  _(TSETB,        var,        var,        lit,        newindex) \
  _(TSETM,        base,        ___,        num,        newindex) \
  _(TSETR,        var,        var,        var,        newindex) \
  \
  /* Calls and vararg handling. T = tail call. */ \
  _(CALLM,        base,        lit,        lit,        call) \
  _(CALL,        base,        lit,        lit,        call) \
  _(CALLMT,        base,        ___,        lit,        call) \
  _(CALLT,        base,        ___,        lit,        call) \
  _(ITERC,        base,        lit,        lit,        call) \
  _(ITERN,        base,        lit,        lit,        call) \
  _(VARG,        base,        lit,        lit,        ___) \
  _(ISNEXT,        base,        ___,        jump,        ___) \
  \
  /* Returns. */ \
  _(RETM,        base,        ___,        lit,        ___) \
  _(RET,        rbase,        ___,        lit,        ___) \
  _(RET0,        rbase,        ___,        lit,        ___) \
  _(RET1,        rbase,        ___,        lit,        ___) \
  \
  /* Loops and branches. I/J = interp/JIT, I/C/L = init/call/loop. */ \
  _(FORI,        base,        ___,        jump,        ___) \
  _(JFORI,        base,        ___,        jump,        ___) \
  \
  _(FORL,        base,        ___,        jump,        ___) \
  _(IFORL,        base,        ___,        jump,        ___) \
  _(JFORL,        base,        ___,        lit,        ___) \
  \
  _(ITERL,        base,        ___,        jump,        ___) \
  _(IITERL,        base,        ___,        jump,        ___) \
  _(JITERL,        base,        ___,        lit,        ___) \
  \
  _(LOOP,        rbase,        ___,        jump,        ___) \
  _(ILOOP,        rbase,        ___,        jump,        ___) \
  _(JLOOP,        rbase,        ___,        lit,        ___) \
  \
```

```
  _(JMP,        rbase,        ___,        jump,        ___) \
  \
  /* Function headers. I/J = interp/JIT, F/V/C = fixarg/vararg/C func. */ \
  _(FUNCF,        rbase,        ___,        ___,        ___) \
  _(IFUNCF,        rbase,        ___,        ___,        ___) \
  _(JFUNCF,        rbase,        ___,        lit,        ___) \
  _(FUNCV,        rbase,        ___,        ___,        ___) \
  _(IFUNCV,        rbase,        ___,        ___,        ___) \
  _(JFUNCV,        rbase,        ___,        lit,        ___) \
  _(FUNCC,        rbase,        ___,        ___,        ___) \
  _(FUNCCW,        rbase,        ___,        ___,        ___)

/* Bytecode opcode numbers. */
typedef enum {
#define BCENUM(name, ma, mb, mc, mt)        BC_##name,
BCDEF(BCENUM)
#undef BCENUM
  BC__MAX
} BCOp;

LJ_STATIC_ASSERT((int)BC_ISEQV+1 == (int)BC_ISNEV);
LJ_STATIC_ASSERT(((int)BC_ISEQV^1) == (int)BC_ISNEV);
LJ_STATIC_ASSERT(((int)BC_ISEQS^1) == (int)BC_ISNES);
LJ_STATIC_ASSERT(((int)BC_ISEQN^1) == (int)BC_ISNEN);
LJ_STATIC_ASSERT(((int)BC_ISEQP^1) == (int)BC_ISNEP);
LJ_STATIC_ASSERT(((int)BC_ISLT^1) == (int)BC_ISGE);
LJ_STATIC_ASSERT(((int)BC_ISLE^1) == (int)BC_ISGT);
LJ_STATIC_ASSERT(((int)BC_ISLT^3) == (int)BC_ISGT);
LJ_STATIC_ASSERT((int)BC_IST-(int)BC_ISTC == (int)BC_ISF-(int)BC_ISFC);
LJ_STATIC_ASSERT((int)BC_CALLT-(int)BC_CALL == (int)BC_CALLMT-(int)BC_CALLM);
LJ_STATIC_ASSERT((int)BC_CALLMT + 1 == (int)BC_CALLT);
LJ_STATIC_ASSERT((int)BC_RETM + 1 == (int)BC_RET);
LJ_STATIC_ASSERT((int)BC_FORL + 1 == (int)BC_IFORL);
LJ_STATIC_ASSERT((int)BC_FORL + 2 == (int)BC_JFORL);
LJ_STATIC_ASSERT((int)BC_ITERL + 1 == (int)BC_IITERL);
LJ_STATIC_ASSERT((int)BC_ITERL + 2 == (int)BC_JITERL);
LJ_STATIC_ASSERT((int)BC_LOOP + 1 == (int)BC_ILOOP);
LJ_STATIC_ASSERT((int)BC_LOOP + 2 == (int)BC_JLOOP);
LJ_STATIC_ASSERT((int)BC_FUNCF + 1 == (int)BC_IFUNCF);
LJ_STATIC_ASSERT((int)BC_FUNCF + 2 == (int)BC_JFUNCF);
LJ_STATIC_ASSERT((int)BC_FUNCV + 1 == (int)BC_IFUNCV);
LJ_STATIC_ASSERT((int)BC_FUNCV + 2 == (int)BC_JFUNCV);

/* This solves a circular dependency problem, change as needed. */
#define FF_next_N        4

/* Stack slots used by FORI/FORL, relative to operand A. */
enum {
  FORL_IDX, FORL_STOP, FORL_STEP, FORL_EXT
};

/* Bytecode operand modes. ORDER BCMode */
typedef enum {
  BCMnone, BCMdst, BCMbase, BCMvar, BCMrbase, BCMuv,  /* Mode A must be <= 7 */
  BCMlit, BCMlits, BCMpri, BCMnum, BCMstr, BCMtab, BCMfunc, BCMjump, BCMcdata,
  BCM_max
} BCMode;
#define BCM___                BCMnone

#define bcmode_a(op)        ((BCMode)(lj_bc_mode[op] & 7))
#define bcmode_b(op)        ((BCMode)((lj_bc_mode[op]>>3) & 15))
#define bcmode_c(op)        ((BCMode)((lj_bc_mode[op]>>7) & 15))
#define bcmode_d(op)        bcmode_c(op)
#define bcmode_hasd(op)        ((lj_bc_mode[op] & (15<<3)) == (BCMnone<<3))
#define bcmode_mm(op)        ((MMS)(lj_bc_mode[op]>>11))

#define BCMODE(name, ma, mb, mc, mm) \
  (BCM##ma|(BCM##mb<<3)|(BCM##mc<<7)|(MM_##mm<<11)),
#define BCMODE_FF        0

static LJ_AINLINE int bc_isret(BCOp op)
{
  return (op == BC_RETM || op == BC_RET || op == BC_RET0 || op == BC_RET1);
}

LJ_DATA const uint16_t lj_bc_mode[];
```

```
263 LJ_DATA const uint16_t lj_bc_ofs[];
264
265 #endif
```

One Level Up Top Level

src/lj_lex.h - luajit-2.0-src

Data types defined

- BCInsLine
- BCInsLine
- LexChar
- LexState
- LexState
- LexToken
- VarInfo
- VarInfo

Macros defined

- TKDEF
- TKENUM1
- TKENUM1
- TKENUM2
- TKENUM2
- _LJ_LEX_H

Source code

```
1  /*
2  ** Lexical analyzer.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_LEX_H
7  #define _LJ_LEX_H
8
9  #include <stdarg.h>
10
11  #include "lj_obj.h"
12  #include "lj_err.h"
13
14  /* Lua lexer tokens. */
15  #define TKDEF(_, __) \
16    _(and) _(break) _(do) _(else) _(elseif) _(end) _(false) \
17    _(for) _(function) _(goto) _(if) _(in) _(local) _(nil) _(not) _(or) \
18    _(repeat) _(return) _(then) _(true) _(until) _(while) \
19    __(concat, ..) __(dots, ...) __(eq, ==) __(ge, >=) __(le, <=) __(ne, ~=) \
20    __(label, ::) __(number, <number>) __(name, <name>) __(string, <string>) \
21    __(eof, <eof>)
22
23  enum {
24    TK_OFS = 256,
25  #define TKENUM1(name)                TK_##name,
26  #define TKENUM2(name, sym)        TK_##name,
27  TKDEF(TKENUM1, TKENUM2)
28  #undef TKENUM1
29  #undef TKENUM2
```

```
  TK_RESERVED = TK_while - TK_OFS
};

typedef int LexChar;        /* Lexical character. Unsigned ext. from char. */
typedef int LexToken;        /* Lexical token. */

/* Combined bytecode ins/line. Only used during bytecode generation. */
typedef struct BCInsLine {
  BCIns ins;                /* Bytecode instruction. */
  BCLine line;                /* Line number for this bytecode. */
} BCInsLine;

/* Info for local variables. Only used during bytecode generation. */
typedef struct VarInfo {
  GCRef name;                /* Local variable name or goto/label name. */
  BCPos startpc;        /* First point where the local variable is active. */
  BCPos endpc;                /* First point where the local variable is dead. */
  uint8_t slot;                /* Variable slot. */
  uint8_t info;                /* Variable/goto/label info. */
} VarInfo;

/* Lua lexer state. */
typedef struct LexState {
  struct FuncState *fs;        /* Current FuncState. Defined in lj_parse.c. */
  struct lua_State *L;        /* Lua state. */
  TValue tokval;        /* Current token value. */
  TValue lookaheadval;        /* Lookahead token value. */
  const char *p;        /* Current position in input buffer. */
  const char *pe;        /* End of input buffer. */
  LexChar c;                /* Current character. */
  LexToken tok;                /* Current token. */
  LexToken lookahead;        /* Lookahead token. */
  SBuf sb;                /* String buffer for tokens. */
  lua_Reader rfunc;        /* Reader callback. */
  void *rdata;                /* Reader callback data. */
  BCLine linenumber;        /* Input line counter. */
  BCLine lastline;        /* Line of last token. */
  GCstr *chunkname;        /* Current chunk name (interned string). */
  const char *chunkarg;        /* Chunk name argument. */
  const char *mode;        /* Allow loading bytecode (b) and/or source text (t). */
  VarInfo *vstack;        /* Stack for names and extents of local variables. */
  MSize sizevstack;        /* Size of variable stack. */
  MSize vtop;                /* Top of variable stack. */
  BCInsLine *bcstack;        /* Stack for bytecode instructions/line numbers. */
  MSize sizebcstack;        /* Size of bytecode stack. */
  uint32_t level;        /* Syntactical nesting level. */
} LexState;

LJ_FUNC int lj_lex_setup(lua_State *L, LexState *ls);
LJ_FUNC void lj_lex_cleanup(lua_State *L, LexState *ls);
LJ_FUNC void lj_lex_next(LexState *ls);
LJ_FUNC LexToken lj_lex_lookahead(LexState *ls);
LJ_FUNC const char *lj_lex_token2str(LexState *ls, LexToken tok);
LJ_FUNC_NORET void lj_lex_error(LexState *ls, LexToken tok, ErrMsg em, ...);
LJ_FUNC void lj_lex_init(lua_State *L);

#endif
```

One Level Up Top Level

src/lj_lex.c - luajit-2.0-src

Global variables defined

- tokennames

Functions defined

- lex_longstring
- lex_more
- lex_newline
- lex_next
- lex_number
- lex_save
- lex_savenext
- lex_scan
- lex_skipeq
- lex_string
- lj_lex_cleanup
- lj_lex_error
- lj_lex_init
- lj_lex_lookahead
- lj_lex_next
- lj_lex_setup
- lj_lex_token2str

Macros defined

- LEX_EOF
- LUA_CORE
- TKSTR1
- TKSTR1
- TKSTR2
- TKSTR2
- lex_iseol
- lj_lex_c

Source code

```c
/*
** Lexical analyzer.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Major portions taken verbatim or adapted from the Lua interpreter.
** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
*/

#define lj_lex_c
 #define LUA_CORE

 #include "lj_obj.h"
 #include "lj_gc.h"
 #include "lj_err.h"
 #include "lj_buf.h"
 #include "lj_str.h"
 #if LJ_HASFFI
 #include "lj_tab.h"
 #include "lj_ctype.h"
 #include "lj_cdata.h"
 #include "lualib.h"
 #endif
 #include "lj_state.h"
 #include "lj_lex.h"
 #include "lj_parse.h"
 #include "lj_char.h"
 #include "lj_strscan.h"
 #include "lj_strfmt.h"

 /* Lua lexer token names. */
 static const char *const tokennames[] = {
 #define TKSTR1(name)                #name,
 #define TKSTR2(name, sym)        #sym,
 TKDEF(TKSTR1, TKSTR2)
 #undef TKSTR1
 #undef TKSTR2
   NULL
 };

 /* -- Buffer handling ----------------------------------------------------- */

 #define LEX_EOF                        (-1)
 #define lex_iseol(ls)                (ls->c == '\n' || ls->c == '\r')

 /* Get more input from reader. */
 static LJ_NOINLINE LexChar lex_more(LexState *ls)
 {
   size_t sz;
   const char *p = ls->rfunc(ls->L, ls->rdata, &sz);
   if (p == NULL || sz == 0) return LEX_EOF;
   ls->pe = p + sz;
   ls->p = p + 1;
   return (LexChar)(uint8_t)p[0];
 }

 /* Get next character. */
 static LJ_AINLINE LexChar lex_next(LexState *ls)
 {
   return (ls->c = ls->p < ls->pe ? (LexChar)(uint8_t)*ls->p++ : lex_more(ls));
 }

 /* Save character. */
 static LJ_AINLINE void lex_save(LexState *ls, LexChar c)
 {
   lj_buf_putb(&ls->sb, c);
 }

 /* Save previous character and get next character. */
 static LJ_AINLINE LexChar lex_savenext(LexState *ls)
 {
   lex_save(ls, ls->c);
   return lex_next(ls);
 }

 /* Skip line break. Handles "\n", "\r", "\r\n" or "\n\r". */
 static void lex_newline(LexState *ls)
```

```
{
  LexChar old = ls->c;
  lua_assert(lex_iseol(ls));
  lex_next(ls);  /* Skip "\n" or "\r". */
  if (lex_iseol(ls) && ls->c != old) lex_next(ls);  /* Skip "\n\r" or "\r\n". */
  if (++ls->linenumber >= LJ_MAX_LINE)
    lj_lex_error(ls, ls->tok, LJ_ERR_XLINES);
}

/* -- Scanner for terminals ----------------------------------------------- */

/* Parse a number literal. */
static void lex_number(LexState *ls, TValue *tv)
{
  StrScanFmt fmt;
  LexChar c, xp = 'e';
  lua_assert(lj_char_isdigit(ls->c));
  if ((c = ls->c) == '0' && (lex_savenext(ls) | 0x20) == 'x')
    xp = 'p';
  while (lj_char_isident(ls->c) || ls->c == '.' ||
         ((ls->c == '-' || ls->c == '+') && (c | 0x20) == xp)) {
    c = ls->c;
    lex_savenext(ls);
  }
  lex_save(ls, '\0');
  fmt = lj_strscan_scan((const uint8_t *)sbufB(&ls->sb), tv,
          (LJ_DUALNUM ? STRSCAN_OPT_TOINT : STRSCAN_OPT_TONUM) |
          (LJ_HASFFI ? (STRSCAN_OPT_LL|STRSCAN_OPT_IMAG) : 0));
  if (LJ_DUALNUM && fmt == STRSCAN_INT) {
    setitype(tv, LJ_TISNUM);
  } else if (fmt == STRSCAN_NUM) {
    /* Already in correct format. */
#if LJ_HASFFI
  } else if (fmt != STRSCAN_ERROR) {
    lua_State *L = ls->L;
    GCcdata *cd;
    lua_assert(fmt == STRSCAN_I64 || fmt == STRSCAN_U64 || fmt == STRSCAN_IMAG);
    if (!ctype_ctsG(G(L))) {
      ptrdiff_t oldtop = savestack(L, L->top);
      luaopen_ffi(L);  /* Load FFI library on-demand. */
      L->top = restorestack(L, oldtop);
    }
    if (fmt == STRSCAN_IMAG) {
      cd = lj_cdata_new_(L, CTID_COMPLEX_DOUBLE, 2*sizeof(double));
      ((double *)cdataptr(cd))[0] = 0;
      ((double *)cdataptr(cd))[1] = numV(tv);
    } else {
      cd = lj_cdata_new_(L, fmt==STRSCAN_I64 ? CTID_INT64 : CTID_UINT64, 8);
      *(uint64_t *)cdataptr(cd) = tv->u64;
    }
    lj_parse_keepcdata(ls, tv, cd);
#endif
  } else {
    lua_assert(fmt == STRSCAN_ERROR);
    lj_lex_error(ls, TK_number, LJ_ERR_XNUMBER);
  }
}

/* Skip equal signs for "[=...=[" and "]=...=]" and return their count. */
static int lex_skipeq(LexState *ls)
{
  int count = 0;
  LexChar s = ls->c;
  lua_assert(s == '[' || s == ']');
  while (lex_savenext(ls) == '=')
    count++;
  return (ls->c == s) ? count : (-count) - 1;
}

/* Parse a long string or long comment (tv set to NULL). */
static void lex_longstring(LexState *ls, TValue *tv, int sep)
{
  lex_savenext(ls);  /* Skip second '['. */
  if (lex_iseol(ls))  /* Skip initial newline. */
    lex_newline(ls);
  for (;;) {
```

```
    switch (ls->c) {
    case LEX_EOF:
      lj_lex_error(ls, TK_eof, tv ? LJ_ERR_XLSTR : LJ_ERR_XLCOM);
      break;
    case ']':
      if (lex_skipeq(ls) == sep) {
        lex_savenext(ls);  /* Skip second ']'. */
        goto endloop;
      }
      break;
    case '\n':
    case '\r':
      lex_save(ls, '\n');
      lex_newline(ls);
      if (!tv) lj_buf_reset(&ls->sb);  /* Don't waste space for comments. */
      break;
    default:
      lex_savenext(ls);
      break;
    }
  } endloop:
  if (tv) {
    GCstr *str = lj_parse_keepstr(ls, sbufB(&ls->sb) + (2 + (MSize)sep),
                                      sbuflen(&ls->sb) - 2*(2 + (MSize)sep));
    setstrV(ls->L, tv, str);
  }
}

/* Parse a string. */
static void lex_string(LexState *ls, TValue *tv)
{
  LexChar delim = ls->c;  /* Delimiter is '\'' or '"'. */
  lex_savenext(ls);
  while (ls->c != delim) {
    switch (ls->c) {
    case LEX_EOF:
      lj_lex_error(ls, TK_eof, LJ_ERR_XSTR);
      continue;
    case '\n':
    case '\r':
      lj_lex_error(ls, TK_string, LJ_ERR_XSTR);
      continue;
    case '\\': {
      LexChar c = lex_next(ls);  /* Skip the '\\'. */
      switch (c) {
      case 'a': c = '\a'; break;
      case 'b': c = '\b'; break;
      case 'f': c = '\f'; break;
      case 'n': c = '\n'; break;
      case 'r': c = '\r'; break;
      case 't': c = '\t'; break;
      case 'v': c = '\v'; break;
      case 'x':  /* Hexadecimal escape '\xXX'. */
        c = (lex_next(ls) & 15u) << 4;
        if (!lj_char_isdigit(ls->c)) {
          if (!lj_char_isxdigit(ls->c)) goto err_xesc;
          c += 9 << 4;
        }
        c += (lex_next(ls) & 15u);
        if (!lj_char_isdigit(ls->c)) {
          if (!lj_char_isxdigit(ls->c)) goto err_xesc;
          c += 9;
        }
        break;
      case 'z':  /* Skip whitespace. */
        lex_next(ls);
        while (lj_char_isspace(ls->c))
          if (lex_iseol(ls)) lex_newline(ls); else lex_next(ls);
        continue;
      case '\n': case '\r': lex_save(ls, '\n'); lex_newline(ls); continue;
      case '\\': case '\"': case '\'': break;
      case LEX_EOF: continue;
      default:
        if (!lj_char_isdigit(c))
          goto err_xesc;
        c -= '0';  /* Decimal escape '\ddd'. */
```

```
        if (lj_char_isdigit(lex_next(ls))) {
          c = c*10 + (ls->c - '0');
          if (lj_char_isdigit(lex_next(ls))) {
            c = c*10 + (ls->c - '0');
            if (c > 255) {
            err_xesc:
              lj_lex_error(ls, TK_string, LJ_ERR_XESC);
            }
            lex_next(ls);
          }
        }
        lex_save(ls, c);
        continue;
      }
      lex_save(ls, c);
      lex_next(ls);
      continue;
      }
    default:
      lex_savenext(ls);
      break;
    }
  }
  lex_savenext(ls);  /* Skip trailing delimiter. */
  setstrV(ls->L, tv,
          lj_parse_keepstr(ls, sbufB(&ls->sb)+1, sbuflen(&ls->sb)-2));
}

/* -- Main lexical scanner ---------------------------------------------- */

/* Get next lexical token. */
static LexToken lex_scan(LexState *ls, TValue *tv)
{
  lj_buf_reset(&ls->sb);
  for (;;) {
    if (lj_char_isident(ls->c)) {
      GCstr *s;
      if (lj_char_isdigit(ls->c)) {  /* Numeric literal. */
        lex_number(ls, tv);
        return TK_number;
      }
      /* Identifier or reserved word. */
      do {
        lex_savenext(ls);
      } while (lj_char_isident(ls->c));
      s = lj_parse_keepstr(ls, sbufB(&ls->sb), sbuflen(&ls->sb));
      setstrV(ls->L, tv, s);
      if (s->reserved > 0)  /* Reserved word? */
        return TK_OFS + s->reserved;
      return TK_name;
    }
    switch (ls->c) {
    case '\n':
    case '\r':
      lex_newline(ls);
      continue;
    case ' ':
    case '\t':
    case '\v':
    case '\f':
      lex_next(ls);
      continue;
    case '-':
      lex_next(ls);
      if (ls->c != '-') return '-';
      lex_next(ls);
      if (ls->c == '[') {  /* Long comment "--[=*[...]=*]". */
        int sep = lex_skipeq(ls);
        lj_buf_reset(&ls->sb);  /* `lex_skipeq' may dirty the buffer */
        if (sep >= 0) {
          lex_longstring(ls, NULL, sep);
          lj_buf_reset(&ls->sb);
          continue;
        }
      }
      /* Short comment "--.*\n". */
```

```
      while (!lex_iseol(ls) && ls->c != LEX_EOF)
        lex_next(ls);
      continue;
    case '[': {
      int sep = lex_skipeq(ls);
      if (sep >= 0) {
        lex_longstring(ls, tv, sep);
        return TK_string;
      } else if (sep == -1) {
        return '[';
      } else {
        lj_lex_error(ls, TK_string, LJ_ERR_XLDELIM);
        continue;
      }
      }
    case '=':
      lex_next(ls);
      if (ls->c != '=') return '='; else { lex_next(ls); return TK_eq; }
    case '<':
      lex_next(ls);
      if (ls->c != '=') return '<'; else { lex_next(ls); return TK_le; }
    case '>':
      lex_next(ls);
      if (ls->c != '=') return '>'; else { lex_next(ls); return TK_ge; }
    case '~':
      lex_next(ls);
      if (ls->c != '=') return '~'; else { lex_next(ls); return TK_ne; }
    case ':':
      lex_next(ls);
      if (ls->c != ':') return ':'; else { lex_next(ls); return TK_label; }
    case '"':
    case '\'':
      lex_string(ls, tv);
      return TK_string;
    case '.':
      if (lex_savenext(ls) == '.') {
        lex_next(ls);
        if (ls->c == '.') {
          lex_next(ls);
          return TK_dots;   /* ... */
        }
        return TK_concat;   /* .. */
      } else if (!lj_char_isdigit(ls->c)) {
        return '.';
      } else {
        lex_number(ls, tv);
        return TK_number;
      }
    case LEX_EOF:
      return TK_eof;
    default: {
      LexChar c = ls->c;
      lex_next(ls);
      return c;  /* Single-char tokens (+ - / ...). */
    }
    }
  }
}

/* -- Lexer API ---------------------------------------------------------- */

/* Setup lexer state. */
int lj_lex_setup(lua_State *L, LexState *ls)
{
  int header = 0;
  ls->L = L;
  ls->fs = NULL;
  ls->pe = ls->p = NULL;
  ls->vstack = NULL;
  ls->sizevstack = 0;
  ls->vtop = 0;
  ls->bcstack = NULL;
  ls->sizebcstack = 0;
  ls->tok = 0;
  ls->lookahead = TK_eof;  /* No look-ahead token. */
  ls->linenumber = 1;
```

```
  ls->lastline = 1;
  lex_next(ls);  /* Read-ahead first char. */
  if (ls->c == 0xef && ls->p + 2 <= ls->pe && (uint8_t)ls->p[0] == 0xbb &&
      (uint8_t)ls->p[1] == 0xbf) {  /* Skip UTF-8 BOM (if buffered). */
    ls->p += 2;
    lex_next(ls);
    header = 1;
  }
  if (ls->c == '#') {  /* Skip POSIX #! header line. */
    do {
      lex_next(ls);
      if (ls->c == LEX_EOF) return 0;
    } while (!lex_iseol(ls));
    lex_newline(ls);
    header = 1;
  }
  if (ls->c == LUA_SIGNATURE[0]) {  /* Bytecode dump. */
    if (header) {
      /*
      ** Loading bytecode with an extra header is disabled for security
      ** reasons. This may circumvent the usual check for bytecode vs.
      ** Lua code by looking at the first char. Since this is a potential
      ** security violation no attempt is made to echo the chunkname either.
      */
      setstrV(L, L->top++, lj_err_str(L, LJ_ERR_BCBAD));
      lj_err_throw(L, LUA_ERRSYNTAX);
    }
    return 1;
  }
  return 0;
}

/* Cleanup lexer state. */
void lj_lex_cleanup(lua_State *L, LexState *ls)
{
  global_State *g = G(L);
  lj_mem_freevec(g, ls->bcstack, ls->sizebcstack, BCInsLine);
  lj_mem_freevec(g, ls->vstack, ls->sizevstack, VarInfo);
  lj_buf_free(g, &ls->sb);
}

/* Return next lexical token. */
void lj_lex_next(LexState *ls)
{
  ls->lastline = ls->linenumber;
  if (LJ_LIKELY(ls->lookahead == TK_eof)) {  /* No lookahead token? */
    ls->tok = lex_scan(ls, &ls->tokval);  /* Get next token. */
  } else {  /* Otherwise return lookahead token. */
    ls->tok = ls->lookahead;
    ls->lookahead = TK_eof;
    ls->tokval = ls->lookaheadval;
  }
}

/* Look ahead for the next token. */
LexToken lj_lex_lookahead(LexState *ls)
{
  lua_assert(ls->lookahead == TK_eof);
  ls->lookahead = lex_scan(ls, &ls->lookaheadval);
  return ls->lookahead;
}

/* Convert token to string. */
const char *lj_lex_token2str(LexState *ls, LexToken tok)
{
  if (tok > TK_OFS)
    return tokennames[tok-TK_OFS-1];
  else if (!lj_char_iscntrl(tok))
    return lj_strfmt_pushf(ls->L, "%c", tok);
  else
    return lj_strfmt_pushf(ls->L, "char(%d)", tok);
}

/* Lexer error. */
void lj_lex_error(LexState *ls, LexToken tok, ErrMsg em, ...)
{
```

```
457   const char *tokstr;
458   va_list argp;
459   if (tok == 0) {
460     tokstr = NULL;
461   } else if (tok == TK_name || tok == TK_string || tok == TK_number) {
462     lex_save(ls, '\0');
463     tokstr = sbufB(&ls->sb);
464   } else {
465     tokstr = lj_lex_token2str(ls, tok);
466   }
467   va_start(argp, em);
468   lj_err_lex(ls->L, ls->chunkname, tokstr, ls->linenumber, em, argp);
469   va_end(argp);
470 }
471
472 /* Initialize strings for reserved words. */
473 void lj_lex_init(lua_State *L)
474 {
475   uint32_t i;
476   for (i = 0; i < TK_RESERVED; i++) {
477     GCstr *s = lj_str_newz(L, tokennames[i]);
478     fixstring(s);  /* Reserved words are never collected. */
479     s->reserved = (uint8_t)(i+1);
480   }
481 }
482
```

One Level Up Top Level

src/lj_buf.h - luajit-2.0-src

Functions defined

- lj_buf_free
- lj_buf_init
- lj_buf_more
- lj_buf_need
- lj_buf_putb
- lj_buf_reset
- lj_buf_str
- lj_buf_tmp_
- lj_buf_wmem

Macros defined

- _LJ_BUF_H
- sbufB
- sbufE
- sbufL
- sbufP
- sbufleft
- sbuflen
- sbufsz
- setsbufL
- setsbufP

Source code

```
1  /*
2  ** Buffer handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_BUF_H
7  #define _LJ_BUF_H
8
9  #include "lj_obj.h"
10 #include "lj_gc.h"
11 #include "lj_str.h"
12
13 /* Resizable string buffers. Struct definition in lj_obj.h. */
14 #define sbufB(sb)        (mref((sb)->b, char))
15 #define sbufP(sb)        (mref((sb)->p, char))
16 #define sbufE(sb)        (mref((sb)->e, char))
17 #define sbufL(sb)        (mref((sb)->L, lua_State))
18 #define sbufsz(sb)        ((MSize)(sbufE((sb)) - sbufB((sb))))
```

```
#define sbuflen(sb)        ((MSize)(sbufP((sb)) - sbufB((sb))))
#define sbufleft(sb)        ((MSize)(sbufE((sb)) - sbufP((sb))))
#define setsbufP(sb, q)        (setmref((sb)->p, (q)))
#define setsbufL(sb, l)        (setmref((sb)->L, (l)))

/* Buffer management */
LJ_FUNC char *LJ_FASTCALL lj_buf_need2(SBuf *sb, MSize sz);
LJ_FUNC char *LJ_FASTCALL lj_buf_more2(SBuf *sb, MSize sz);
LJ_FUNC void LJ_FASTCALL lj_buf_shrink(lua_State *L, SBuf *sb);
LJ_FUNC char * LJ_FASTCALL lj_buf_tmp(lua_State *L, MSize sz);

static LJ_AINLINE void lj_buf_init(lua_State *L, SBuf *sb)
{
  setsbufL(sb, L);
  setmref(sb->p, NULL); setmref(sb->e, NULL); setmref(sb->b, NULL);
}

static LJ_AINLINE void lj_buf_reset(SBuf *sb)
{
  setmrefr(sb->p, sb->b);
}

static LJ_AINLINE SBuf *lj_buf_tmp_(lua_State *L)
{
  SBuf *sb = &G(L)->tmpbuf;
  setsbufL(sb, L);
  lj_buf_reset(sb);
  return sb;
}

static LJ_AINLINE void lj_buf_free(global_State *g, SBuf *sb)
{
  lj_mem_free(g, sbufB(sb), sbufsz(sb));
}

static LJ_AINLINE char *lj_buf_need(SBuf *sb, MSize sz)
{
  if (LJ_UNLIKELY(sz > sbufsz(sb)))
    return lj_buf_need2(sb, sz);
  return sbufB(sb);
}

static LJ_AINLINE char *lj_buf_more(SBuf *sb, MSize sz)
{
  if (LJ_UNLIKELY(sz > sbufleft(sb)))
    return lj_buf_more2(sb, sz);
  return sbufP(sb);
}

/* Low-level buffer put operations */
LJ_FUNC SBuf *lj_buf_putmem(SBuf *sb, const void *q, MSize len);
#if LJ_HASJIT
LJ_FUNC SBuf * LJ_FASTCALL lj_buf_putchar(SBuf *sb, int c);
#endif
LJ_FUNC SBuf * LJ_FASTCALL lj_buf_putstr(SBuf *sb, GCstr *s);

static LJ_AINLINE char *lj_buf_wmem(char *p, const void *q, MSize len)
{
  return (char *)memcpy(p, q, len) + len;
}

static LJ_AINLINE void lj_buf_putb(SBuf *sb, int c)
{
  char *p = lj_buf_more(sb, 1);
  *p++ = (char)c;
  setsbufP(sb, p);
}

/* High-level buffer put operations */
LJ_FUNCA SBuf * LJ_FASTCALL lj_buf_putstr_reverse(SBuf *sb, GCstr *s);
LJ_FUNCA SBuf * LJ_FASTCALL lj_buf_putstr_lower(SBuf *sb, GCstr *s);
LJ_FUNCA SBuf * LJ_FASTCALL lj_buf_putstr_upper(SBuf *sb, GCstr *s);
LJ_FUNC SBuf *lj_buf_putstr_rep(SBuf *sb, GCstr *s, int32_t rep);
LJ_FUNC SBuf *lj_buf_puttab(SBuf *sb, GCtab *t, GCstr *sep,
                            int32_t i, int32_t e);

```

```
95  /* Miscellaneous buffer operations */
96  LJ_FUNCA GCstr * LJ_FASTCALL lj_buf_tostr(SBuf *sb);
97  LJ_FUNC GCstr *lj_buf_cat2str(lua_State *L, GCstr *s1, GCstr *s2);
98  LJ_FUNC uint32_t LJ_FASTCALL lj_buf_ruleb128(const char **pp);
99
100 static LJ_AINLINE GCstr *lj_buf_str(lua_State *L, SBuf *sb)
101 {
102   return lj_str_new(L, sbufB(sb), sbuflen(sb));
103 }
104
105 #endif
```

One Level Up Top Level

src/lj_buf.c - luajit-2.0-src

Functions defined

Macros defined

Source code

```
1  /*
2  ** Buffer handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lj_buf_c
7  #define LUA_CORE
8
9  #include "lj_obj.h"
10  #include "lj_gc.h"
11  #include "lj_err.h"
12  #include "lj_buf.h"
13  #include "lj_str.h"
14  #include "lj_tab.h"
15  #include "lj_strfmt.h"
16
17  /* -- Buffer management ---------------------------------------------- */
18
19  static void buf_grow(SBuf *sb, MSize sz)
20  {
```

```
  MSize osz = sbufsz(sb), len = sbuflen(sb), nsz = osz;
  char *b;
  if (nsz < LJ_MIN_SBUF) nsz = LJ_MIN_SBUF;
  while (nsz < sz) nsz += nsz;
  b = (char *)lj_mem_realloc(sbufL(sb), sbufB(sb), osz, nsz);
  setmref(sb->b, b);
  setmref(sb->p, b + len);
  setmref(sb->e, b + nsz);
}

LJ_NOINLINE char *LJ_FASTCALL lj_buf_need2(SBuf *sb, MSize sz)
{
  lua_assert(sz > sbufsz(sb));
  if (LJ_UNLIKELY(sz > LJ_MAX_BUF))
    lj_err_mem(sbufL(sb));
  buf_grow(sb, sz);
  return sbufB(sb);
}

LJ_NOINLINE char *LJ_FASTCALL lj_buf_more2(SBuf *sb, MSize sz)
{
  MSize len = sbuflen(sb);
  lua_assert(sz > sbufleft(sb));
  if (LJ_UNLIKELY(sz > LJ_MAX_BUF || len + sz > LJ_MAX_BUF))
    lj_err_mem(sbufL(sb));
  buf_grow(sb, len + sz);
  return sbufP(sb);
}

void LJ_FASTCALL lj_buf_shrink(lua_State *L, SBuf *sb)
{
  char *b = sbufB(sb);
  MSize osz = (MSize)(sbufE(sb) - b);
  if (osz > 2*LJ_MIN_SBUF) {
    MSize n = (MSize)(sbufP(sb) - b);
    b = lj_mem_realloc(L, b, osz, (osz >> 1));
    setmref(sb->b, b);
    setmref(sb->p, b + n);
    setmref(sb->e, b + (osz >> 1));
  }
}

char * LJ_FASTCALL lj_buf_tmp(lua_State *L, MSize sz)
{
  SBuf *sb = &G(L)->tmpbuf;
  setsbufL(sb, L);
  return lj_buf_need(sb, sz);
}

/* -- Low-level buffer put operations ----------------------------------- */

SBuf *lj_buf_putmem(SBuf *sb, const void *q, MSize len)
{
  char *p = lj_buf_more(sb, len);
  p = lj_buf_wmem(p, q, len);
  setsbufP(sb, p);
  return sb;
}

#if LJ_HASJIT
SBuf * LJ_FASTCALL lj_buf_putchar(SBuf *sb, int c)
{
  char *p = lj_buf_more(sb, 1);
  *p++ = (char)c;
  setsbufP(sb, p);
  return sb;
}
#endif

SBuf * LJ_FASTCALL lj_buf_putstr(SBuf *sb, GCstr *s)
{
  MSize len = s->len;
  char *p = lj_buf_more(sb, len);
  p = lj_buf_wmem(p, strdata(s), len);
  setsbufP(sb, p);
  return sb;
```

```
}

/* -- High-level buffer put operations ----------------------------------- */

SBuf * LJ_FASTCALL lj_buf_putstr_reverse(SBuf *sb, GCstr *s)
{
  MSize len = s->len;
  char *p = lj_buf_more(sb, len), *e = p+len;
  const char *q = strdata(s)+len-1;
  while (p < e)
    *p++ = *q--;
  setsbufP(sb, p);
  return sb;
}

SBuf * LJ_FASTCALL lj_buf_putstr_lower(SBuf *sb, GCstr *s)
{
  MSize len = s->len;
  char *p = lj_buf_more(sb, len), *e = p+len;
  const char *q = strdata(s);
  for (; p < e; p++, q++) {
    uint32_t c = *(unsigned char *)q;
#if LJ_TARGET_PPC
    *p = c + ((c >= 'A' && c <= 'Z') << 5);
#else
    if (c >= 'A' && c <= 'Z') c += 0x20;
    *p = c;
#endif
  }
  setsbufP(sb, p);
  return sb;
}

SBuf * LJ_FASTCALL lj_buf_putstr_upper(SBuf *sb, GCstr *s)
{
  MSize len = s->len;
  char *p = lj_buf_more(sb, len), *e = p+len;
  const char *q = strdata(s);
  for (; p < e; p++, q++) {
    uint32_t c = *(unsigned char *)q;
#if LJ_TARGET_PPC
    *p = c - ((c >= 'a' && c <= 'z') << 5);
#else
    if (c >= 'a' && c <= 'z') c -= 0x20;
    *p = c;
#endif
  }
  setsbufP(sb, p);
  return sb;
}

SBuf *lj_buf_putstr_rep(SBuf *sb, GCstr *s, int32_t rep)
{
  MSize len = s->len;
  if (rep > 0 && len) {
    uint64_t tlen = (uint64_t)rep * len;
    char *p;
    if (LJ_UNLIKELY(tlen > LJ_MAX_STR))
      lj_err_mem(sbufL(sb));
    p = lj_buf_more(sb, (MSize)tlen);
    if (len == 1) {  /* Optimize a common case. */
      uint32_t c = strdata(s)[0];
      do { *p++ = c; } while (--rep > 0);
    } else {
      const char *e = strdata(s) + len;
      do {
        const char *q = strdata(s);
        do { *p++ = *q++; } while (q < e);
      } while (--rep > 0);
    }
    setsbufP(sb, p);
  }
  return sb;
}

SBuf *lj_buf_puttab(SBuf *sb, GCtab *t, GCstr *sep, int32_t i, int32_t e)
```

```
{
  MSize seplen = sep ? sep->len : 0;
  if (i <= e) {
    for (;;) {
      cTValue *o = lj_tab_getint(t, i);
      char *p;
      if (!o) {
      badtype:  /* Error: bad element type. */
        setsbufP(sb, (void *)(intptr_t)i);  /* Store failing index. */
        return NULL;
      } else if (tvisstr(o)) {
        MSize len = strV(o)->len;
        p = lj_buf_wmem(lj_buf_more(sb, len + seplen), strVdata(o), len);
      } else if (tvisint(o)) {
        p = lj_strfmt_wint(lj_buf_more(sb, STRFMT_MAXBUF_INT+seplen), intV(o));
      } else if (tvisnum(o)) {
        p = lj_strfmt_wnum(lj_buf_more(sb, STRFMT_MAXBUF_NUM+seplen), o);
      } else {
        goto badtype;
      }
      if (i++ == e) {
        setsbufP(sb, p);
        break;
      }
      if (seplen) p = lj_buf_wmem(p, strdata(sep), seplen);
      setsbufP(sb, p);
    }
  }
  return sb;
}

/* -- Miscellaneous buffer operations ------------------------------------ */

GCstr * LJ_FASTCALL lj_buf_tostr(SBuf *sb)
{
  return lj_str_new(sbufL(sb), sbufB(sb), sbuflen(sb));
}

/* Concatenate two strings. */
GCstr *lj_buf_cat2str(lua_State *L, GCstr *s1, GCstr *s2)
{
  MSize len1 = s1->len, len2 = s2->len;
  char *buf = lj_buf_tmp(L, len1 + len2);
  memcpy(buf, strdata(s1), len1);
  memcpy(buf+len1, strdata(s2), len2);
  return lj_str_new(L, buf, len1 + len2);
}

/* Read ULEB128 from buffer. */
uint32_t LJ_FASTCALL lj_buf_ruleb128(const char **pp)
{
  const uint8_t *p = (const uint8_t *)*pp;
  uint32_t v = *p++;
  if (LJ_UNLIKELY(v >= 0x80)) {
    int sh = 0;
    v &= 0x7f;
    do { v |= ((*p & 0x7f) << (sh += 7)); } while (*p++ >= 0x80);
  }
  *pp = (const char *)p;
  return v;
}

```

One Level Up Top Level

src/lj_strfmt.c - luajit-2.0-src

Global variables defined

- strfmt_map

Functions defined

- lj_strfmt_char
- lj_strfmt_int
- lj_strfmt_num
- lj_strfmt_number
- lj_strfmt_obj
- lj_strfmt_parse
- lj_strfmt_pushf
- lj_strfmt_pushvf
- lj_strfmt_putfchar
- lj_strfmt_putfnum
- lj_strfmt_putfnum_int
- lj_strfmt_putfnum_uint
- lj_strfmt_putfstr
- lj_strfmt_putfxint
- lj_strfmt_putint
- lj_strfmt_putnum
- lj_strfmt_putptr
- lj_strfmt_putquoted
- lj_strfmt_wint
- lj_strfmt_wnum
- lj_strfmt_wptr
- lj_strfmt_wstrnum
- lj_strfmt_wuleb128

Macros defined

- LUA_CORE
- STRFMT_FMTNUMBUF
- WINT_R

- WINT_R
- lj_strfmt_c

Source code

```
/*
** String formatting.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#include <stdio.h>

#define lj_strfmt_c
#define LUA_CORE

 #include "lj_obj.h"
 #include "lj_buf.h"
 #include "lj_str.h"
 #include "lj_state.h"
 #include "lj_char.h"
 #include "lj_strfmt.h"

 /* -- Format parser -------------------------------------------------------- */

 static const uint8_t strfmt_map[('x'-'A')+1] = {
   STRFMT_A,0,0,0,STRFMT_E,0,STRFMT_G,0,0,0,0,0,0,
   0,0,0,0,0,0,0,0,0,STRFMT_X,0,0,
   0,0,0,0,0,0,
   STRFMT_A,0,STRFMT_C,STRFMT_D,STRFMT_E,STRFMT_F,STRFMT_G,0,STRFMT_I,0,0,0,0,
   0,STRFMT_O,STRFMT_P,STRFMT_Q,0,STRFMT_S,0,STRFMT_U,0,0,STRFMT_X
 };

 SFormat LJ_FASTCALL lj_strfmt_parse(FormatState *fs)
 {
   const uint8_t *p = fs->p, *e = fs->e;
   fs->str = (const char *)p;
   for (; p < e; p++) {
     if (*p == '%') {  /* Escape char? */
       if (p[1] == '%') {  /* '%%'? */
         fs->p = ++p+1;
         goto retlit;
       } else {
         SFormat sf = 0;
         uint32_t c;
         if (p != (const uint8_t *)fs->str)
           break;
         for (p++; (uint32_t)*p - ' ' <= (uint32_t)('0' - ' '); p++) {
           /* Parse flags. */
           if (*p == '-') sf |= STRFMT_F_LEFT;
           else if (*p == '+') sf |= STRFMT_F_PLUS;
           else if (*p == '0') sf |= STRFMT_F_ZERO;
           else if (*p == ' ') sf |= STRFMT_F_SPACE;
           else if (*p == '#') sf |= STRFMT_F_ALT;
           else break;
         }
         if ((uint32_t)*p - '0' < 10) {  /* Parse width. */
           uint32_t width = (uint32_t)*p++ - '0';
           if ((uint32_t)*p - '0' < 10)
             width = (uint32_t)*p++ - '0' + width*10;
           sf |= (width << STRFMT_SH_WIDTH);
         }
         if (*p == '.') {  /* Parse precision. */
           uint32_t prec = 0;
           p++;
           if ((uint32_t)*p - '0' < 10) {
             prec = (uint32_t)*p++ - '0';
             if ((uint32_t)*p - '0' < 10)
               prec = (uint32_t)*p++ - '0' + prec*10;
           }
           sf |= ((prec+1) << STRFMT_SH_PREC);
         }
         /* Parse conversion. */
         c = (uint32_t)*p - 'A';
```

```
        if (LJ_LIKELY(c <= (uint32_t)('x' - 'A'))) {
          uint32_t sx = strfmt_map[c];
          if (sx) {
            fs->p = p+1;
            return (sf | sx | ((c & 0x20) ? 0 : STRFMT_F_UPPER));
          }
        }
        /* Return error location. */
        if (*p >= 32) p++;
        fs->len = (MSize)(p - (const uint8_t *)fs->str);
        fs->p = fs->e;
        return STRFMT_ERR;
      }
    }
  }
  fs->p = p;
retlit:
  fs->len = (MSize)(p - (const uint8_t *)fs->str);
  return fs->len ? STRFMT_LIT : STRFMT_EOF;
}

/* -- Raw conversions ------------------------------------------------------ */

/* Write number to bufer. */
char * LJ_FASTCALL lj_strfmt_wnum(char *p, cTValue *o)
{
  if (LJ_LIKELY((o->u32.hi << 1) < 0xffe00000)) {  /* Finite? */
#if __BIONIC__
    if (tvismzero(o)) { *p++ = '-'; *p++ = '0'; return p; }
#endif
    return p + lua_number2str(p, o->n);
  } else if (((o->u32.hi & 0x000fffff) | o->u32.lo) != 0) {
    *p++ = 'n'; *p++ = 'a'; *p++ = 'n';
  } else if ((o->u32.hi & 0x80000000) == 0) {
    *p++ = 'i'; *p++ = 'n'; *p++ = 'f';
  } else {
    *p++ = '-'; *p++ = 'i'; *p++ = 'n'; *p++ = 'f';
  }
  return p;
}

#define WINT_R(x, sh, sc) \
  { uint32_t d = (x*(((1<<sh)+sc-1)/sc))>>sh; x -= d*sc; *p++ = (char)('0'+d); }

/* Write integer to buffer. */
char * LJ_FASTCALL lj_strfmt_wint(char *p, int32_t k)
{
  uint32_t u = (uint32_t)k;
  if (k < 0) { u = (uint32_t)-k; *p++ = '-'; }
  if (u < 10000) {
    if (u < 10) goto dig1; if (u < 100) goto dig2; if (u < 1000) goto dig3;
  } else {
    uint32_t v = u / 10000; u -= v * 10000;
    if (v < 10000) {
      if (v < 10) goto dig5; if (v < 100) goto dig6; if (v < 1000) goto dig7;
    } else {
      uint32_t w = v / 10000; v -= w * 10000;
      if (w >= 10) WINT_R(w, 10, 10)
      *p++ = (char)('0'+w);
    }
    WINT_R(v, 23, 1000)
    dig7: WINT_R(v, 12, 100)
    dig6: WINT_R(v, 10, 10)
    dig5: *p++ = (char)('0'+v);
  }
  WINT_R(u, 23, 1000)
  dig3: WINT_R(u, 12, 100)
  dig2: WINT_R(u, 10, 10)
  dig1: *p++ = (char)('0'+u);
  return p;
}
#undef WINT_R

/* Write pointer to buffer. */
char * LJ_FASTCALL lj_strfmt_wptr(char *p, const void *v)
{
```

```
  ptrdiff_t x = (ptrdiff_t)v;
  MSize i, n = STRFMT_MAXBUF_PTR;
  if (x == 0) {
    *p++ = 'N'; *p++ = 'U'; *p++ = 'L'; *p++ = 'L';
    return p;
  }
#if LJ_64
  /* Shorten output for 64 bit pointers. */
  n = 2+2*4+((x >> 32) ? 2+2*(lj_fls((uint32_t)(x >> 32))>>3) : 0);
#endif
  p[0] = '0';
  p[1] = 'x';
  for (i = n-1; i >= 2; i--, x >>= 4)
    p[i] = "0123456789abcdef"[(x & 15)];
  return p+n;
}

/* Write ULEB128 to buffer. */
char * LJ_FASTCALL lj_strfmt_wuleb128(char *p, uint32_t v)
{
  for (; v >= 0x80; v >>= 7)
    *p++ = (char)((v & 0x7f) | 0x80);
  *p++ = (char)v;
  return p;
}

/* Return string or write number to buffer and return pointer to start. */
const char *lj_strfmt_wstrnum(char *buf, cTValue *o, MSize *lenp)
{
  if (tvisstr(o)) {
    *lenp = strV(o)->len;
    return strVdata(o);
  } else if (tvisint(o)) {
    *lenp = (MSize)(lj_strfmt_wint(buf, intV(o)) - buf);
    return buf;
  } else if (tvisnum(o)) {
    *lenp = (MSize)(lj_strfmt_wnum(buf, o) - buf);
    return buf;
  } else {
    return NULL;
  }
}

/* -- Unformatted conversions to buffer ----------------------------------- */

/* Add integer to buffer. */
SBuf * LJ_FASTCALL lj_strfmt_putint(SBuf *sb, int32_t k)
{
  setsbufP(sb, lj_strfmt_wint(lj_buf_more(sb, STRFMT_MAXBUF_INT), k));
  return sb;
}

#if LJ_HASJIT
/* Add number to buffer. */
SBuf * LJ_FASTCALL lj_strfmt_putnum(SBuf *sb, cTValue *o)
{
  setsbufP(sb, lj_strfmt_wnum(lj_buf_more(sb, STRFMT_MAXBUF_NUM), o));
  return sb;
}
#endif

SBuf * LJ_FASTCALL lj_strfmt_putptr(SBuf *sb, const void *v)
{
  setsbufP(sb, lj_strfmt_wptr(lj_buf_more(sb, STRFMT_MAXBUF_PTR), v));
  return sb;
}

/* Add quoted string to buffer. */
SBuf * LJ_FASTCALL lj_strfmt_putquoted(SBuf *sb, GCstr *str)
{
  const char *s = strdata(str);
  MSize len = str->len;
  lj_buf_putb(sb, '"');
  while (len--) {
    uint32_t c = (uint32_t)(uint8_t)*s++;
    char *p = lj_buf_more(sb, 4);
```

```
    if (c == '"' || c == '\\' || c == '\n') {
      *p++ = '\\';
    } else if (lj_char_iscntrl(c)) {  /* This can only be 0-31 or 127. */
      uint32_t d;
      *p++ = '\\';
      if (c >= 100 || lj_char_isdigit((uint8_t)*s)) {
        *p++ = (char)('0'+(c >= 100)); if (c >= 100) c -= 100;
        goto tens;
      } else if (c >= 10) {
      tens:
        d = (c * 205) >> 11; c -= d * 10; *p++ = (char)('0'+d);
      }
      c += '0';
    }
    *p++ = (char)c;
    setsbufP(sb, p);
  }
  lj_buf_putb(sb, '"');
  return sb;
}

/* -- Formatted conversions to buffer ------------------------------------ */

/* Add formatted char to buffer. */
SBuf *lj_strfmt_putfchar(SBuf *sb, SFormat sf, int32_t c)
{
  MSize width = STRFMT_WIDTH(sf);
  char *p = lj_buf_more(sb, width > 1 ? width : 1);
  if ((sf & STRFMT_F_LEFT)) *p++ = (char)c;
  while (width-- > 1) *p++ = ' ';
  if (!(sf & STRFMT_F_LEFT)) *p++ = (char)c;
  setsbufP(sb, p);
  return sb;
}

/* Add formatted string to buffer. */
SBuf *lj_strfmt_putfstr(SBuf *sb, SFormat sf, GCstr *str)
{
  MSize len = str->len <= STRFMT_PREC(sf) ? str->len : STRFMT_PREC(sf);
  MSize width = STRFMT_WIDTH(sf);
  char *p = lj_buf_more(sb, width > len ? width : len);
  if ((sf & STRFMT_F_LEFT)) p = lj_buf_wmem(p, strdata(str), len);
  while (width-- > len) *p++ = ' ';
  if (!(sf & STRFMT_F_LEFT)) p = lj_buf_wmem(p, strdata(str), len);
  setsbufP(sb, p);
  return sb;
}

/* Add formatted signed/unsigned integer to buffer. */
SBuf *lj_strfmt_putfxint(SBuf *sb, SFormat sf, uint64_t k)
{
  char buf[STRFMT_MAXBUF_XINT], *q = buf + sizeof(buf), *p;
#ifdef LUA_USE_ASSERT
  char *ps;
#endif
  MSize prefix = 0, len, prec, pprec, width, need;

  /* Figure out signed prefixes. */
  if (STRFMT_TYPE(sf) == STRFMT_INT) {
    if ((int64_t)k < 0) {
      k = (uint64_t)-(int64_t)k;
      prefix = 256 + '-';
    } else if ((sf & STRFMT_F_PLUS)) {
      prefix = 256 + '+';
    } else if ((sf & STRFMT_F_SPACE)) {
      prefix = 256 + ' ';
    }
  }

  /* Convert number and store to fixed-size buffer in reverse order. */
  prec = STRFMT_PREC(sf);
  if ((int32_t)prec >= 0) sf &= ~STRFMT_F_ZERO;
  if (k == 0) {  /* Special-case zero argument. */
    if (prec != 0 ||
        (sf & (STRFMT_T_OCT|STRFMT_F_ALT)) == (STRFMT_T_OCT|STRFMT_F_ALT))
      *--q = '0';
```

```
  } else if (!(sf & (STRFMT_T_HEX|STRFMT_T_OCT))) {  /* Decimal. */
    uint32_t k2;
    while ((k >> 32)) { *--q = (char)('0' + k % 10); k /= 10; }
    k2 = (uint32_t)k;
    do { *--q = (char)('0' + k2 % 10); k2 /= 10; } while (k2);
  } else if ((sf & STRFMT_T_HEX)) {  /* Hex. */
    const char *hexdig = (sf & STRFMT_F_UPPER) ? "0123456789ABCDEF" :
                                                 "0123456789abcdef";
    do { *--q = hexdig[(k & 15)]; k >>= 4; } while (k);
    if ((sf & STRFMT_F_ALT)) prefix = 512 + ((sf & STRFMT_F_UPPER) ? 'X' : 'x');
  } else {  /* Octal. */
    do { *--q = (char)('0' + (uint32_t)(k & 7)); k >>= 3; } while (k);
    if ((sf & STRFMT_F_ALT)) *--q = '0';
  }

  /* Calculate sizes. */
  len = (MSize)(buf + sizeof(buf) - q);
  if ((int32_t)len >= (int32_t)prec) prec = len;
  width = STRFMT_WIDTH(sf);
  pprec = prec + (prefix >> 8);
  need = width > pprec ? width : pprec;
  p = lj_buf_more(sb, need);
#ifdef LUA_USE_ASSERT
  ps = p;
#endif

  /* Format number with leading/trailing whitespace and zeros. */
  if ((sf & (STRFMT_F_LEFT|STRFMT_F_ZERO)) == 0)
    while (width-- > pprec) *p++ = ' ';
  if (prefix) {
    if ((char)prefix >= 'X') *p++ = '0';
    *p++ = (char)prefix;
  }
  if ((sf & (STRFMT_F_LEFT|STRFMT_F_ZERO)) == STRFMT_F_ZERO)
    while (width-- > pprec) *p++ = '0';
  while (prec-- > len) *p++ = '0';
  while (q < buf + sizeof(buf)) *p++ = *q++;  /* Add number itself. */
  if ((sf & STRFMT_F_LEFT))
    while (width-- > pprec) *p++ = ' ';

  lua_assert(need == (MSize)(p - ps));
  setsbufP(sb, p);
  return sb;
}

/* Add number formatted as signed integer to buffer. */
SBuf *lj_strfmt_putfnum_int(SBuf *sb, SFormat sf, lua_Number n)
{
  int64_t k = (int64_t)n;
  if (checki32(k) && sf == STRFMT_INT)
    return lj_strfmt_putint(sb, (int32_t)k);  /* Shortcut for plain %d. */
  else
    return lj_strfmt_putfxint(sb, sf, (uint64_t)k);
}

/* Add number formatted as unsigned integer to buffer. */
SBuf *lj_strfmt_putfnum_uint(SBuf *sb, SFormat sf, lua_Number n)
{
  int64_t k;
  if (n >= 9223372036854775808.0)
    k = (int64_t)(n - 18446744073709551616.0);
  else
    k = (int64_t)n;
  return lj_strfmt_putfxint(sb, sf, (uint64_t)k);
}

/* Max. sprintf buffer size needed. At least #string.format("%.99f", -1e308). */
#define STRFMT_FMTNUMBUF        512

/* Add formatted floating-point number to buffer. */
SBuf *lj_strfmt_putfnum(SBuf *sb, SFormat sf, lua_Number n)
{
  TValue tv;
  tv.n = n;
  if (LJ_UNLIKELY((tv.u32.hi << 1) >= 0xffe00000)) {
    /* Canonicalize output of non-finite values. */
```

```
    MSize width = STRFMT_WIDTH(sf), len = 3;
    int prefix = 0, ch = (sf & STRFMT_F_UPPER) ? 0x202020 : 0;
    char *p;
    if (((tv.u32.hi & 0x000fffff) | tv.u32.lo) != 0) {
      ch ^= ('n' << 16) | ('a' << 8) | 'n';
      if ((sf & STRFMT_F_SPACE)) prefix = ' ';
    } else {
      ch ^= ('i' << 16) | ('n' << 8) | 'f';
      if ((tv.u32.hi & 0x80000000)) prefix = '-';
      else if ((sf & STRFMT_F_PLUS)) prefix = '+';
      else if ((sf & STRFMT_F_SPACE)) prefix = ' ';
    }
    if (prefix) len = 4;
    p = lj_buf_more(sb, width > len ? width : len);
    if (!(sf & STRFMT_F_LEFT)) while (width-- > len) *p++ = ' ';
    if (prefix) *p++ = prefix;
    *p++ = (char)(ch >> 16); *p++ = (char)(ch >> 8); *p++ = (char)ch;
    if ((sf & STRFMT_F_LEFT)) while (width-- > len) *p++ = ' ';
    setsbufP(sb, p);
  } else {  /* Delegate to sprintf() for now. */
    uint8_t width = (uint8_t)STRFMT_WIDTH(sf), prec = (uint8_t)STRFMT_PREC(sf);
    char fmt[1+5+2+3+1+1], *p = fmt;
    *p++ = '%';
    if ((sf & STRFMT_F_LEFT)) *p++ = '-';
    if ((sf & STRFMT_F_PLUS)) *p++ = '+';
    if ((sf & STRFMT_F_ZERO)) *p++ = '0';
    if ((sf & STRFMT_F_SPACE)) *p++ = ' ';
    if ((sf & STRFMT_F_ALT)) *p++ = '#';
    if (width) {
      uint8_t x = width / 10, y = width % 10;
      if (x) *p++ = '0' + x;
      *p++ = '0' + y;
    }
    if (prec != 255) {
      uint8_t x = prec / 10, y = prec % 10;
      *p++ = '.';
      if (x) *p++ = '0' + x;
      *p++ = '0' + y;
    }
    *p++ = (0x67666561 >> (STRFMT_FP(sf)<<3)) ^ ((sf & STRFMT_F_UPPER)?0x20:0);
    *p = '\0';
    p = lj_buf_more(sb, STRFMT_FMTNUMBUF);
    setsbufP(sb, p + sprintf(p, fmt, n));
  }
  return sb;
}

/* -- Conversions to strings ---------------------------------------------- */

/* Convert integer to string. */
GCstr * LJ_FASTCALL lj_strfmt_int(lua_State *L, int32_t k)
{
  char buf[STRFMT_MAXBUF_INT];
  MSize len = (MSize)(lj_strfmt_wint(buf, k) - buf);
  return lj_str_new(L, buf, len);
}

/* Convert number to string. */
GCstr * LJ_FASTCALL lj_strfmt_num(lua_State *L, cTValue *o)
{
  char buf[STRFMT_MAXBUF_NUM];
  MSize len = (MSize)(lj_strfmt_wnum(buf, o) - buf);
  return lj_str_new(L, buf, len);
}

/* Convert integer or number to string. */
GCstr * LJ_FASTCALL lj_strfmt_number(lua_State *L, cTValue *o)
{
  return tvisint(o) ? lj_strfmt_int(L, intV(o)) : lj_strfmt_num(L, o);
}

#if LJ_HASJIT
/* Convert char value to string. */
GCstr * LJ_FASTCALL lj_strfmt_char(lua_State *L, int c)
{
  char buf[1];
```

```
  buf[0] = c;
  return lj_str_new(L, buf, 1);
}
#endif

/* Raw conversion of object to string. */
GCstr * LJ_FASTCALL lj_strfmt_obj(lua_State *L, cTValue *o)
{
  if (tvisstr(o)) {
    return strV(o);
  } else if (tvisnumber(o)) {
    return lj_strfmt_number(L, o);
  } else if (tvisnil(o)) {
    return lj_str_newlit(L, "nil");
  } else if (tvisfalse(o)) {
    return lj_str_newlit(L, "false");
  } else if (tvistrue(o)) {
    return lj_str_newlit(L, "true");
  } else {
    char buf[8+2+2+16], *p = buf;
    p = lj_buf_wmem(p, lj_typename(o), (MSize)strlen(lj_typename(o)));
    *p++ = ':'; *p++ = ' ';
    if (tvisfunc(o) && isffunc(funcV(o))) {
      p = lj_buf_wmem(p, "builtin#", 8);
      p = lj_strfmt_wint(p, funcV(o)->c.ffid);
    } else {
      p = lj_strfmt_wptr(p, lj_obj_ptr(o));
    }
    return lj_str_new(L, buf, (size_t)(p - buf));
  }
}

/* -- Internal string formatting ----------------------------------------- */

/*
** These functions are only used for lua_pushfstring(), lua_pushvfstring()
** and for internal string formatting (e.g. error messages). Caveat: unlike
** string.format(), only a limited subset of formats and flags are supported!
**
** LuaJIT has support for a couple more formats than Lua 5.1/5.2:
** - %d %u %o %x with full formatting, 32 bit integers only.
** - %f and other FP formats are really %.14g.
** - %s %c %p without formatting.
*/

/* Push formatted message as a string object to Lua stack. va_list variant. */
const char *lj_strfmt_pushvf(lua_State *L, const char *fmt, va_list argp)
{
  SBuf *sb = lj_buf_tmp_(L);
  FormatState fs;
  SFormat sf;
  GCstr *str;
  lj_strfmt_init(&fs, fmt, (MSize)strlen(fmt));
  while ((sf = lj_strfmt_parse(&fs)) != STRFMT_EOF) {
    switch (STRFMT_TYPE(sf)) {
    case STRFMT_LIT:
      lj_buf_putmem(sb, fs.str, fs.len);
      break;
    case STRFMT_INT:
      lj_strfmt_putfxint(sb, sf, va_arg(argp, int32_t));
      break;
    case STRFMT_UINT:
      lj_strfmt_putfxint(sb, sf, va_arg(argp, uint32_t));
      break;
    case STRFMT_NUM: {
      TValue tv;
      tv.n = va_arg(argp, lua_Number);
      setsbufP(sb, lj_strfmt_wnum(lj_buf_more(sb, STRFMT_MAXBUF_NUM), &tv));
      break;
      }
    case STRFMT_STR: {
      const char *s = va_arg(argp, char *);
      if (s == NULL) s = "(null)";
      lj_buf_putmem(sb, s, (MSize)strlen(s));
      break;
      }
```

```
525     case STRFMT_CHAR:
526       lj_buf_putb(sb, va_arg(argp, int));
527       break;
528     case STRFMT_PTR:
529       lj_strfmt_putptr(sb, va_arg(argp, void *));
530       break;
531     case STRFMT_ERR:
532     default:
533       lj_buf_putb(sb, '?');
534       lua_assert(0);
535       break;
536     }
537   }
538   str = lj_buf_str(L, sb);
539   setstrV(L, L->top, str);
540   incr_top(L);
541   return strdata(str);
542 }
543
544 /* Push formatted message as a string object to Lua stack. Vararg variant. */
545 const char *lj_strfmt_pushf(lua_State *L, const char *fmt, ...)
546 {
547   const char *msg;
548   va_list argp;
549   va_start(argp, fmt);
550   msg = lj_strfmt_pushvf(L, fmt, argp);
551   va_end(argp);
552   return msg;
553 }
554
```

One Level Up Top Level

src/lj_strfmt.h - luajit-2.0-src

Data types defined

- FormatState
- FormatState
- FormatType
- FormatType
- SFormat

Functions defined

- lj_strfmt_init

Macros defined

- STRFMT_A
- STRFMT_C
- STRFMT_D
- STRFMT_E
- STRFMT_F
- STRFMT_FP
- STRFMT_F_ALT
- STRFMT_F_LEFT
- STRFMT_F_PLUS
- STRFMT_F_SPACE
- STRFMT_F_UPPER
- STRFMT_F_ZERO
- STRFMT_G
- STRFMT_I
- STRFMT_MAXBUF_INT
- STRFMT_MAXBUF_NUM
- STRFMT_MAXBUF_PTR
- STRFMT_MAXBUF_XINT
- STRFMT_O
- STRFMT_P
- STRFMT_PREC

- STRFMT_Q
- STRFMT_S
- STRFMT_SH_PREC
- STRFMT_SH_WIDTH
- STRFMT_TYPE
- STRFMT_T_FP_A
- STRFMT_T_FP_E
- STRFMT_T_FP_F
- STRFMT_T_FP_G
- STRFMT_T_HEX
- STRFMT_T_OCT
- STRFMT_T_QUOTED
- STRFMT_U
- STRFMT_WIDTH
- STRFMT_X
- _LJ_STRFMT_H

Source code

```
1  /*
2  ** String formatting.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_STRFMT_H
7  #define _LJ_STRFMT_H
8
9  #include "lj_obj.h"
10
11  typedef uint32_t SFormat;  /* Format indicator. */
12
13  /* Format parser state. */
14  typedef struct FormatState {
15    const uint8_t *p;        /* Current format string pointer. */
16    const uint8_t *e;        /* End of format string. */
17    const char *str;        /* Returned literal string. */
18    MSize len;                /* Size of literal string. */
19  } FormatState;
20
21  /* Format types (max. 16). */
22  typedef enum FormatType {
23    STRFMT_EOF, STRFMT_ERR, STRFMT_LIT,
24    STRFMT_INT, STRFMT_UINT, STRFMT_NUM, STRFMT_STR, STRFMT_CHAR, STRFMT_PTR
25  } FormatType;
26
27  /* Format subtypes (bits are reused). */
28  #define STRFMT_T_HEX        0x0010        /* STRFMT_UINT */
29  #define STRFMT_T_OCT        0x0020        /* STRFMT_UINT */
30  #define STRFMT_T_FP_A        0x0000        /* STRFMT_NUM */
31  #define STRFMT_T_FP_E        0x0010        /* STRFMT_NUM */
32  #define STRFMT_T_FP_F        0x0020        /* STRFMT_NUM */
33  #define STRFMT_T_FP_G        0x0030        /* STRFMT_NUM */
34  #define STRFMT_T_QUOTED        0x0010        /* STRFMT_STR */
35
36  /* Format flags. */
```

```
#define STRFMT_F_LEFT        0x0100
#define STRFMT_F_PLUS        0x0200
#define STRFMT_F_ZERO        0x0400
#define STRFMT_F_SPACE        0x0800
#define STRFMT_F_ALT        0x1000
#define STRFMT_F_UPPER        0x2000

/* Format indicator fields. */
#define STRFMT_SH_WIDTH        16
#define STRFMT_SH_PREC        24

#define STRFMT_TYPE(sf)                ((FormatType)((sf) & 15))
#define STRFMT_WIDTH(sf)        (((sf) >> STRFMT_SH_WIDTH) & 255u)
#define STRFMT_PREC(sf)                ((((sf) >> STRFMT_SH_PREC) & 255u) - 1u)
#define STRFMT_FP(sf)                (((sf) >> 4) & 3)

/* Formats for conversion characters. */
#define STRFMT_A        (STRFMT_NUM|STRFMT_T_FP_A)
#define STRFMT_C        (STRFMT_CHAR)
#define STRFMT_D        (STRFMT_INT)
#define STRFMT_E        (STRFMT_NUM|STRFMT_T_FP_E)
#define STRFMT_F        (STRFMT_NUM|STRFMT_T_FP_F)
#define STRFMT_G        (STRFMT_NUM|STRFMT_T_FP_G)
#define STRFMT_I        STRFMT_D
#define STRFMT_O        (STRFMT_UINT|STRFMT_T_OCT)
#define STRFMT_P        (STRFMT_PTR)
#define STRFMT_Q        (STRFMT_STR|STRFMT_T_QUOTED)
#define STRFMT_S        (STRFMT_STR)
#define STRFMT_U        (STRFMT_UINT)
#define STRFMT_X        (STRFMT_UINT|STRFMT_T_HEX)

/* Maximum buffer sizes for conversions. */
#define STRFMT_MAXBUF_XINT        (1+22)  /* '0' prefix + uint64_t in octal. */
#define STRFMT_MAXBUF_INT        (1+10)  /* Sign + int32_t in decimal. */
#define STRFMT_MAXBUF_NUM        LUAI_MAXNUMBER2STR
#define STRFMT_MAXBUF_PTR        (2+2*sizeof(ptrdiff_t))  /* "0x" + hex ptr. */

/* Format parser. */
LJ_FUNC SFormat LJ_FASTCALL lj_strfmt_parse(FormatState *fs);

static LJ_AINLINE void lj_strfmt_init(FormatState *fs, const char *p, MSize len)
{
  fs->p = (const uint8_t *)p;
  fs->e = (const uint8_t *)p + len;
  lua_assert(*fs->e == 0);  /* Must be NUL-terminated (may have NULs inside). */
}

/* Raw conversions. */
LJ_FUNC char * LJ_FASTCALL lj_strfmt_wint(char *p, int32_t k);
LJ_FUNC char * LJ_FASTCALL lj_strfmt_wnum(char *p, cTValue *o);
LJ_FUNC char * LJ_FASTCALL lj_strfmt_wptr(char *p, const void *v);
LJ_FUNC char * LJ_FASTCALL lj_strfmt_wuleb128(char *p, uint32_t v);
LJ_FUNC const char *lj_strfmt_wstrnum(char *buf, cTValue *o, MSize *lenp);

/* Unformatted conversions to buffer. */
LJ_FUNC SBuf * LJ_FASTCALL lj_strfmt_putint(SBuf *sb, int32_t k);
#if LJ_HASJIT
LJ_FUNC SBuf * LJ_FASTCALL lj_strfmt_putnum(SBuf *sb, cTValue *o);
#endif
LJ_FUNC SBuf * LJ_FASTCALL lj_strfmt_putptr(SBuf *sb, const void *v);
LJ_FUNC SBuf * LJ_FASTCALL lj_strfmt_putquoted(SBuf *sb, GCstr *str);

/* Formatted conversions to buffer. */
LJ_FUNC SBuf *lj_strfmt_putfxint(SBuf *sb, SFormat sf, uint64_t k);
LJ_FUNC SBuf *lj_strfmt_putfnum_int(SBuf *sb, SFormat sf, lua_Number n);
LJ_FUNC SBuf *lj_strfmt_putfnum_uint(SBuf *sb, SFormat sf, lua_Number n);
LJ_FUNC SBuf *lj_strfmt_putfnum(SBuf *sb, SFormat, lua_Number n);
LJ_FUNC SBuf *lj_strfmt_putfchar(SBuf *sb, SFormat, int32_t c);
LJ_FUNC SBuf *lj_strfmt_putfstr(SBuf *sb, SFormat, GCstr *str);

/* Conversions to strings. */
LJ_FUNC GCstr * LJ_FASTCALL lj_strfmt_int(lua_State *L, int32_t k);
LJ_FUNCA GCstr * LJ_FASTCALL lj_strfmt_num(lua_State *L, cTValue *o);
LJ_FUNCA GCstr * LJ_FASTCALL lj_strfmt_number(lua_State *L, cTValue *o);
#if LJ_HASJIT
LJ_FUNC GCstr * LJ_FASTCALL lj_strfmt_char(lua_State *L, int c);
```

```
#endif
LJ_FUNC GCstr * LJ_FASTCALL lj_strfmt_obj(lua_State *L, cTValue *o);

/* Internal string formatting. */
LJ_FUNC const char *lj_strfmt_pushvf(lua_State *L, const char *fmt,
                                     va_list argp);
LJ_FUNC const char *lj_strfmt_pushf(lua_State *L, const char *fmt, ...)
#ifdef __GNUC__
  __attribute__ ((format (printf, 2, 3)))
#endif
  ;

#endif
```

One Level Up Top Level

src/lj_str.h - luajit-2.0-src

Macros defined

- _LJ_STR_H
- lj_str_newlit
- lj_str_newz

Source code

```
/*
** String handling.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_STR_H
#define _LJ_STR_H

#include <stdarg.h>

#include "lj_obj.h"

/* String helpers. */
LJ_FUNC int32_t LJ_FASTCALL lj_str_cmp(GCstr *a, GCstr *b);
LJ_FUNC const char *lj_str_find(const char *s, const char *f,
                                MSize slen, MSize flen);
LJ_FUNC int lj_str_haspattern(GCstr *s);

/* String interning. */
LJ_FUNC void lj_str_resize(lua_State *L, MSize newmask);
LJ_FUNCA GCstr *lj_str_new(lua_State *L, const char *str, size_t len);
LJ_FUNC void LJ_FASTCALL lj_str_free(global_State *g, GCstr *s);

#define lj_str_newz(L, s)        (lj_str_new(L, s, strlen(s)))
#define lj_str_newlit(L, s)        (lj_str_new(L, "" s, sizeof(s)-1))

#endif
```

src/host/buildvm_lib.c - luajit-2.0-src

Global variables defined

- ffasmfunc
- ffid
- funcname
- libdef_handlers
- modname
- modnamelen
- modstate
- obuf
- optr
- recffid
- regfunc

Data types defined

- LibDefFunc
- LibDefHandler
- LibDefHandler

Functions defined

- emit_lib
- find_ffofs
- find_rec
- libdef_endmodule
- libdef_fixupbc
- libdef_func
- libdef_lua
- libdef_module
- libdef_name
- libdef_push
- libdef_rec
- libdef_regfunc
- libdef_set

- libdef_uleb128
- memcpy_endian

Source code

```
/*
** LuaJIT VM builder: library definition compiler.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#include "buildvm.h"
#include "lj_obj.h"
#include "lj_bc.h"
#include "lj_lib.h"
 #include "buildvm_libbc.h"

 /* Context for library definitions. */
 static uint8_t obuf[8192];
 static uint8_t *optr;
 static char modname[80];
 static size_t modnamelen;
 static char funcname[80];
 static int modstate, regfunc;
 static int ffid, recffid, ffasmfunc;

 enum {
   REGFUNC_OK,
   REGFUNC_NOREG,
   REGFUNC_NOREGUV
 };

 static void libdef_name(const char *p, int kind)
 {
   size_t n = strlen(p);
   if (kind != LIBINIT_STRING) {
     if (n > modnamelen && p[modnamelen] == '_' &&
         !strncmp(p, modname, modnamelen)) {
       p += modnamelen+1;
       n -= modnamelen+1;
     }
   }
   if (n > LIBINIT_MAXSTR) {
     fprintf(stderr, "Error: string too long: '%s'\n",  p);
     exit(1);
   }
   if (optr+1+n+2 > obuf+sizeof(obuf)) {  /* +2 for caller. */
     fprintf(stderr, "Error: output buffer overflow\n");
     exit(1);
   }
   *optr++ = (uint8_t)(n | kind);
   memcpy(optr, p, n);
   optr += n;
 }

 static void libdef_endmodule(BuildCtx *ctx)
 {
   if (modstate != 0) {
     char line[80];
     const uint8_t *p;
     int n;
     if (modstate == 1)
       fprintf(ctx->fp, "  (lua_CFunction)0");
     fprintf(ctx->fp, "\n};\n");
     fprintf(ctx->fp, "static const uint8_t %s%s[] = {\n",
             LABEL_PREFIX_LIBINIT, modname);
     line[0] = '\0';
     for (n = 0, p = obuf; p < optr; p++) {
       n += sprintf(line+n, "%d,", *p);
       if (n >= 75) {
         fprintf(ctx->fp, "%s\n", line);
         n = 0;
         line[0] = '\0';
       }
```

```
    }
    fprintf(ctx->fp, "%s%d\n};\n#endif\n\n", line, LIBINIT_END);
  }
}

static void libdef_module(BuildCtx *ctx, char *p, int arg)
{
  UNUSED(arg);
  if (ctx->mode == BUILD_libdef) {
    libdef_endmodule(ctx);
    optr = obuf;
    *optr++ = (uint8_t)ffid;
    *optr++ = (uint8_t)ffasmfunc;
    *optr++ = 0;  /* Hash table size. */
    modstate = 1;
    fprintf(ctx->fp, "#ifdef %sMODULE_%s\n", LIBDEF_PREFIX, p);
    fprintf(ctx->fp, "#undef %sMODULE_%s\n", LIBDEF_PREFIX, p);
    fprintf(ctx->fp, "static const lua_CFunction %s%s[] = {\n",
            LABEL_PREFIX_LIBCF, p);
  }
  modnamelen = strlen(p);
  if (modnamelen > sizeof(modname)-1) {
    fprintf(stderr, "Error: module name too long: '%s'\n", p);
    exit(1);
  }
  strcpy(modname, p);
}

static int find_ffofs(BuildCtx *ctx, const char *name)
{
  int i;
   for (i = 0; i < ctx->nglob; i++) {
     const char *gl = ctx->globnames[i];
     if (gl[0] == 'f' && gl[1] == 'f' && gl[2] == '_' && !strcmp(gl+3, name)) {
       return (int)((uint8_t *)ctx->glob[i] - ctx->code);
     }
   }
   fprintf(stderr, "Error: undefined fast function %s%s\n",
           LABEL_PREFIX_FF, name);
  exit(1);
 }

 static void libdef_func(BuildCtx *ctx, char *p, int arg)
 {
   if (arg != LIBINIT_CF)
     ffasmfunc++;
   if (ctx->mode == BUILD_libdef) {
     if (modstate == 0) {
       fprintf(stderr, "Error: no module for function definition %s\n", p);
       exit(1);
     }
     if (regfunc == REGFUNC_NOREG) {
       if (optr+1 > obuf+sizeof(obuf)) {
         fprintf(stderr, "Error: output buffer overflow\n");
         exit(1);
       }
       *optr++ = LIBINIT_FFID;
     } else {
       if (arg != LIBINIT_ASM_) {
         if (modstate != 1) fprintf(ctx->fp, ",\n");
         modstate = 2;
         fprintf(ctx->fp, "  %s%s", arg ? LABEL_PREFIX_FFH : LABEL_PREFIX_CF, p);
       }
       if (regfunc != REGFUNC_NOREGUV) obuf[2]++;  /* Bump hash table size. */
       libdef_name(regfunc == REGFUNC_NOREGUV ? "" : p, arg);
     }
   } else if (ctx->mode == BUILD_ffdef) {
     fprintf(ctx->fp, "FFDEF(%s)\n", p);
   } else if (ctx->mode == BUILD_recdef) {
     if (strlen(p) > sizeof(funcname)-1) {
       fprintf(stderr, "Error: function name too long: '%s'\n", p);
       exit(1);
     }
     strcpy(funcname, p);
   } else if (ctx->mode == BUILD_vmdef) {
     int i;
```

```
    for (i = 1; p[i] && modname[i-1]; i++)
      if (p[i] == '_') p[i] = '.';
    fprintf(ctx->fp, "\"%s\",\n", p);
  } else if (ctx->mode == BUILD_bcdef) {
    if (arg != LIBINIT_CF)
      fprintf(ctx->fp, ",\n%d", find_ffofs(ctx, p));
  }
  ffid++;
  regfunc = REGFUNC_OK;
}

static uint8_t *libdef_uleb128(uint8_t *p, uint32_t *vv)
{
  uint32_t v = *p++;
  if (v >= 0x80) {
    int sh = 0; v &= 0x7f;
    do { v |= ((*p & 0x7f) << (sh += 7)); } while (*p++ >= 0x80);
  }
  *vv = v;
  return p;
}

static void libdef_fixupbc(uint8_t *p)
{
  uint32_t i, sizebc;
  p += 4;
  p = libdef_uleb128(p, &sizebc);
  p = libdef_uleb128(p, &sizebc);
  p = libdef_uleb128(p, &sizebc);
  for (i = 0; i < sizebc; i++, p += 4) {
    uint8_t op = p[libbc_endian ? 3 : 0];
    uint8_t ra = p[libbc_endian ? 2 : 1];
    uint8_t rc = p[libbc_endian ? 1 : 2];
    uint8_t rb = p[libbc_endian ? 0 : 3];
    if (!LJ_DUALNUM && op == BC_ISTYPE && rc == ~LJ_TNUMX+1) {
      op = BC_ISNUM; rc++;
    }
    p[LJ_ENDIAN_SELECT(0, 3)] = op;
    p[LJ_ENDIAN_SELECT(1, 2)] = ra;
    p[LJ_ENDIAN_SELECT(2, 1)] = rc;
    p[LJ_ENDIAN_SELECT(3, 0)] = rb;
  }
}

static void libdef_lua(BuildCtx *ctx, char *p, int arg)
{
  UNUSED(arg);
  if (ctx->mode == BUILD_libdef) {
    int i;
    for (i = 0; libbc_map[i].name != NULL; i++) {
      if (!strcmp(libbc_map[i].name, p)) {
        int ofs = libbc_map[i].ofs;
        int len = libbc_map[i+1].ofs - ofs;
        obuf[2]++;  /* Bump hash table size. */
        *optr++ = LIBINIT_LUA;
        libdef_name(p, 0);
        memcpy(optr, libbc_code + ofs, len);
        libdef_fixupbc(optr);
        optr += len;
        return;
      }
    }
    fprintf(stderr, "Error: missing libbc definition for %s\n", p);
    exit(1);
  }
}

static uint32_t find_rec(char *name)
{
  char *p = (char *)obuf;
  uint32_t n;
  for (n = 2; *p; n++) {
    if (strcmp(p, name) == 0)
      return n;
    p += strlen(p)+1;
  }
```

```
  if (p+strlen(name)+1 >= (char *)obuf+sizeof(obuf)) {
    fprintf(stderr, "Error: output buffer overflow\n");
    exit(1);
  }
  strcpy(p, name);
  return n;
}

static void libdef_rec(BuildCtx *ctx, char *p, int arg)
{
  UNUSED(arg);
  if (ctx->mode == BUILD_recdef) {
    char *q;
    uint32_t n;
    for (; recffid+1 < ffid; recffid++)
      fprintf(ctx->fp, ",\n0");
    recffid = ffid;
    if (*p == '.') p = funcname;
    q = strchr(p, ' ');
    if (q) *q++ = '\0';
    n = find_rec(p);
    if (q)
      fprintf(ctx->fp, ",\n0x%02x00+(%s)", n, q);
    else
      fprintf(ctx->fp, ",\n0x%02x00", n);
  }
}

static void memcpy_endian(void *dst, void *src, size_t n)
{
  union { uint8_t b; uint32_t u; } host_endian;
  host_endian.u = 1;
  if (host_endian.b == LJ_ENDIAN_SELECT(1, 0)) {
    memcpy(dst, src, n);
  } else {
    size_t i;
    for (i = 0; i < n; i++)
      ((uint8_t *)dst)[i] = ((uint8_t *)src)[n-i-1];
  }
}

static void libdef_push(BuildCtx *ctx, char *p, int arg)
{
  UNUSED(arg);
  if (ctx->mode == BUILD_libdef) {
    int len = (int)strlen(p);
    if (*p == '"') {
      if (len > 1 && p[len-1] == '"') {
        p[len-1] = '\0';
        libdef_name(p+1, LIBINIT_STRING);
        return;
      }
    } else if (*p >= '0' && *p <= '9') {
      char *ep;
      double d = strtod(p, &ep);
      if (*ep == '\0') {
        if (optr+1+sizeof(double) > obuf+sizeof(obuf)) {
          fprintf(stderr, "Error: output buffer overflow\n");
          exit(1);
        }
        *optr++ = LIBINIT_NUMBER;
        memcpy_endian(optr, &d, sizeof(double));
        optr += sizeof(double);
        return;
      }
    } else if (!strcmp(p, "lastcl")) {
      if (optr+1 > obuf+sizeof(obuf)) {
        fprintf(stderr, "Error: output buffer overflow\n");
        exit(1);
      }
      *optr++ = LIBINIT_LASTCL;
      return;
    } else if (len > 4 && !strncmp(p, "top-", 4)) {
      if (optr+2 > obuf+sizeof(obuf)) {
        fprintf(stderr, "Error: output buffer overflow\n");
        exit(1);
```

```
      }
      *optr++ = LIBINIT_COPY;
      *optr++ = (uint8_t)atoi(p+4);
      return;
    }
    fprintf(stderr, "Error: bad value for %sPUSH(%s)\n", LIBDEF_PREFIX, p);
    exit(1);
  }
}

static void libdef_set(BuildCtx *ctx, char *p, int arg)
{
  UNUSED(arg);
  if (ctx->mode == BUILD_libdef) {
    if (p[0] == '!' && p[1] == '\0') p[0] = '\0';  /* Set env. */
    libdef_name(p, LIBINIT_STRING);
    *optr++ = LIBINIT_SET;
    obuf[2]++;  /* Bump hash table size. */
  }
}

static void libdef_regfunc(BuildCtx *ctx, char *p, int arg)
{
  UNUSED(ctx); UNUSED(p);
  regfunc = arg;
}

typedef void (*LibDefFunc)(BuildCtx *ctx, char *p, int arg);

typedef struct LibDefHandler {
  const char *suffix;
  const char *stop;
  const LibDefFunc func;
  const int arg;
} LibDefHandler;

static const LibDefHandler libdef_handlers[] = {
  { "MODULE_",        " \t\r\n",       libdef_module,              0 },
  { "CF(",         ")",              libdef_func,             LIBINIT_CF },
  { "ASM(",         ")",              libdef_func,              LIBINIT_ASM },
  { "ASM_(",         ")",              libdef_func,              LIBINIT_ASM_ },
  { "LUA(",         ")",              libdef_lua,              0 },
  { "REC(",         ")",              libdef_rec,              0 },
  { "PUSH(",         ")",              libdef_push,              0 },
  { "SET(",         ")",              libdef_set,              0 },
  { "NOREGUV",        NULL,             libdef_regfunc,                REGFUNC_NOREGUV },
  { "NOREG",        NULL,             libdef_regfunc,               REGFUNC_NOREG },
  { NULL,        NULL,             (LibDefFunc)0,              0 }
};

/* Emit C source code for library function definitions. */
void emit_lib(BuildCtx *ctx)
{
  const char *fname;

  if (ctx->mode == BUILD_ffdef || ctx->mode == BUILD_libdef ||
      ctx->mode == BUILD_recdef)
    fprintf(ctx->fp, "/* This is a generated file. DO NOT EDIT! */\n\n");
  else if (ctx->mode == BUILD_vmdef)
    fprintf(ctx->fp, "ffnames = {\n[0]=\"Lua\",\n\"C\",\n");
  if (ctx->mode == BUILD_recdef)
    fprintf(ctx->fp, "static const uint16_t recff_idmap[] = {\n0,\n0x0100");
  recffid = ffid = FF_C+1;
  ffasmfunc = 0;

  while ((fname = *ctx->args++)) {
    char buf[256];  /* We don't care about analyzing lines longer than that. */
    FILE *fp;
    if (fname[0] == '-' && fname[1] == '\0') {
      fp = stdin;
    } else {
      fp = fopen(fname, "r");
      if (!fp) {
        fprintf(stderr, "Error: cannot open input file '%s': %s\n",
                fname, strerror(errno));
        exit(1);
```

```
      }
    }
    modstate = 0;
    regfunc = REGFUNC_OK;
    while (fgets(buf, sizeof(buf), fp) != NULL) {
      char *p;
      /* Simplistic pre-processor. Only handles top-level #if/#endif. */
      if (buf[0] == '#' && buf[1] == 'i' && buf[2] == 'f') {
        int ok = 1;
        if (!strcmp(buf, "#if LJ_52\n"))
          ok = LJ_52;
        else if (!strcmp(buf, "#if LJ_HASJIT\n"))
          ok = LJ_HASJIT;
        else if (!strcmp(buf, "#if LJ_HASFFI\n"))
          ok = LJ_HASFFI;
        if (!ok) {
          int lvl = 1;
          while (fgets(buf, sizeof(buf), fp) != NULL) {
            if (buf[0] == '#' && buf[1] == 'e' && buf[2] == 'n') {
              if (--lvl == 0) break;
            } else if (buf[0] == '#' && buf[1] == 'i' && buf[2] == 'f') {
              lvl++;
            }
          }
          continue;
        }
      }
      for (p = buf; (p = strstr(p, LIBDEF_PREFIX)) != NULL; ) {
        const LibDefHandler *ldh;
        p += sizeof(LIBDEF_PREFIX)-1;
        for (ldh = libdef_handlers; ldh->suffix != NULL; ldh++) {
          size_t n, len = strlen(ldh->suffix);
          if (!strncmp(p, ldh->suffix, len)) {
            p += len;
            n = ldh->stop ? strcspn(p, ldh->stop) : 0;
            if (!p[n]) break;
            p[n] = '\0';
            ldh->func(ctx, p, ldh->arg);
            p += n+1;
            break;
          }
        }
        if (ldh->suffix == NULL) {
          buf[strlen(buf)-1] = '\0';
          fprintf(stderr, "Error: unknown library definition tag %s%s\n",
                  LIBDEF_PREFIX, p);
          exit(1);
        }
      }
    }
    fclose(fp);
    if (ctx->mode == BUILD_libdef) {
      libdef_endmodule(ctx);
    }
  }

  if (ctx->mode == BUILD_ffdef) {
    fprintf(ctx->fp, "\n#undef FFDEF\n\n");
    fprintf(ctx->fp,
      "#ifndef FF_NUM_ASMFUNC\n#define FF_NUM_ASMFUNC %d\n#endif\n\n",
      ffasmfunc);
  } else if (ctx->mode == BUILD_vmdef) {
    fprintf(ctx->fp, "},\n\n");
  } else if (ctx->mode == BUILD_bcdef) {
    int i;
    fprintf(ctx->fp, "\n};\n\n");
    fprintf(ctx->fp, "LJ_DATADEF const uint16_t lj_bc_mode[] = {\n");
    fprintf(ctx->fp, "BCDEF(BCMODE)\n");
    for (i = ffasmfunc-1; i > 0; i--)
      fprintf(ctx->fp, "BCMODE_FF,\n");
    fprintf(ctx->fp, "BCMODE_FF\n};\n\n");
  } else if (ctx->mode == BUILD_recdef) {
    char *p = (char *)obuf;
    fprintf(ctx->fp, "\n};\n\n");
    fprintf(ctx->fp, "static const RecordFunc recff_func[] = {\n"
            "recff_nyi,\n"
```

```
449             "recff_c");
450     while (*p) {
451       fprintf(ctx->fp, ",\nrecff_%s", p);
452       p += strlen(p)+1;
453     }
454     fprintf(ctx->fp, "\n};\n\n");
455   }
456 }
457
```

One Level Up Top Level

src/host/buildvm_arch.h - luajit-2.0-src

Global variables defined

- build_actionlist
- extnames
- globnames

Functions defined

- build_backend
- build_ins
- build_subroutines
- emit_asm_debug

Macros defined

- BSZPTR
- DASM_MAXSECTION
- DASM_SECTION_CODE_OP
- DASM_SECTION_CODE_SUB
- DISPATCH_GL
- DISPATCH_J
- Dt1
- Dt10
- Dt2
- Dt3
- Dt4
- Dt5
- Dt6
- Dt7
- Dt8
- Dt9
- DtA
- DtB
- DtC
- DtD

- DtE
- DtF
- PC2PROTO
- REG_RA
- REG_SP
- SZPTR
- TV2MARKOFS
- TV2MARKOFS

Source code

```
/*
** This file has been pre-processed with DynASM.
** http://luajit.org/dynasm.html
** DynASM version 1.3.0, DynASM x64 version 1.3.0
** DO NOT EDIT! The original file is in "vm_x86.dasc".
*/

#line 1 "vm_x86.dasc"
//|// Low-level VM code for x86 CPUs.
 //|// Bytecode interpreter, fast functions and helper functions.
 //|// Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
 //|
 //|.if P64
 //|.arch x64
 #if DASM_VERSION != 10300
 #error "Version mismatch between DynASM and included encoding engine"
 #endif
 #line 7 "vm_x86.dasc"
 //|.else
 //|.arch x86
 //|.endif
 //|.section code_op, code_sub
 #define DASM_SECTION_CODE_OP        0
 #define DASM_SECTION_CODE_SUB       1
 #define DASM_MAXSECTION             2
 #line 11 "vm_x86.dasc"
 //|
 //|.actionlist build_actionlist
 static const unsigned char build_actionlist[14864] = {
   254,1,248,10,252,247,195,237,15,132,244,11,131,227,252,248,41,218,72,141,
   76,25,252,248,139,90,252,252,199,68,10,4,237,248,12,131,192,1,15,132,244,
   13,137,68,36,4,252,247,195,237,15,132,244,14,248,15,129,252,243,239,252,247,
   195,237,15,133,244,10,65,199,134,233,237,131,227,252,248,41,211,252,247,219,
   131,232,1,15,132,244,248,248,1,255,72,139,44,10,72,137,106,252,248,131,194,
   8,131,232,1,15,133,244,1,248,2,139,108,36,24,137,157,233,248,3,139,68,36,
   4,139,76,36,16,248,4,57,193,15,133,244,252,248,5,131,252,234,8,137,149,233,
   248,16,72,139,76,36,32,72,137,141,233,49,192,248,17,72,131,196,40,65,94,65,
   95,91,93,195,248,6,15,130,244,253,59,149,233,15,135,244,254,199,66,252,252,
   237,255,131,194,8,131,192,1,252,233,244,4,248,7,133,201,15,132,244,5,41,193,
   141,20,202,252,233,244,5,248,8,137,149,233,137,68,36,4,137,206,137,252,239,
   232,251,1,0,139,149,233,252,233,244,3,248,13,176,235,252,233,244,18,248,19,
   137,252,240,72,137,252,252,248,18,139,108,36,24,139,173,233,199,133,233,237,
   255,252,233,244,17,248,20,139,124,36,24,137,198,72,131,196,40,65,94,65,95,
   91,93,252,233,251,1,1,248,21,72,129,231,239,72,137,252,252,248,22,139,108,
   36,24,72,199,193,252,248,252,255,252,255,252,255,184,237,139,149,233,68,139,
   181,233,65,129,198,239,139,90,252,252,199,66,252,252,237,65,199,134,233,237,
   252,233,244,12,248,23,190,237,252,233,244,248,248,24,255,131,232,8,252,233,
   244,247,248,25,141,68,194,252,248,248,1,15,182,139,233,131,195,4,137,149,
   233,137,133,233,137,92,36,28,137,206,248,2,137,252,239,232,251,1,0,139,149,
   233,139,133,233,139,106,252,248,41,208,193,232,3,131,192,1,139,157,233,139,
   11,15,182,252,233,15,182,205,131,195,4,65,252,255,36,252,238,248,26,85,83,
   65,87,65,86,72,131,252,236,40,137,252,253,137,124,36,24,137,252,241,187,237,
   49,192,76,141,188,253,36,233,68,139,181,233,65,129,198,239,137,68,36,28,72,
   137,68,36,32,137,68,36,16,137,68,36,20,76,137,189,233,56,133,233,15,132,244,
```

```
248,255,65,137,174,233,65,199,134,233,237,136,133,233,139,149,233,139,133,
233,41,200,193,232,3,131,192,1,41,209,139,90,252,252,137,68,36,4,252,247,
195,237,15,132,244,14,252,233,244,15,248,27,85,83,65,87,65,86,72,131,252,
236,40,187,237,137,76,36,20,252,233,244,247,248,28,85,83,65,87,65,86,72,131,
252,236,40,187,237,248,1,137,84,36,16,137,252,253,137,124,36,24,137,252,241,
68,139,181,233,76,139,189,233,255,76,137,124,36,32,137,108,36,28,65,129,198,
239,72,137,165,233,248,2,65,137,174,233,65,199,134,233,237,139,149,233,1,
203,41,211,139,133,233,41,200,193,232,3,131,192,1,248,29,139,105,252,248,
129,121,253,252,252,239,15,133,244,30,248,31,137,202,137,90,252,252,139,157,
233,139,11,15,182,252,233,15,182,205,131,195,4,65,252,255,36,252,238,248,
32,85,83,65,87,65,86,72,131,252,236,40,137,252,253,137,124,36,24,137,108,
36,28,68,139,189,233,68,43,189,233,68,139,181,233,199,68,36,20,0,0,0,0,68,
137,124,36,16,65,129,198,239,76,139,189,233,255,76,137,124,36,32,72,137,165,
233,65,137,174,233,252,255,209,133,192,15,132,244,16,137,193,187,237,252,
233,244,2,248,11,1,209,131,227,252,248,137,213,41,218,199,68,193,252,252,
237,137,200,139,93,252,244,72,99,77,252,240,131,252,249,1,15,134,244,247,
76,141,61,245,76,1,252,249,68,139,122,252,248,69,139,191,233,69,139,191,233,
252,255,225,248,1,15,132,244,33,41,213,193,252,237,3,141,69,252,255,252,233,
244,34,248,35,255,15,182,75,252,255,131,252,237,16,141,12,202,41,252,233,
15,132,244,36,252,247,217,193,252,233,3,139,124,36,24,137,151,233,137,202,
72,139,8,72,137,77,0,137,252,238,252,233,244,37,248,38,137,4,36,199,68,36,
4,237,72,141,4,36,128,123,252,252,235,15,133,244,247,65,141,142,233,137,41,
199,65,4,237,137,205,252,233,244,248,248,39,15,182,67,252,254,199,68,36,4,
237,137,4,36,72,141,4,36,252,233,244,247,248,40,15,182,67,252,254,141,4,194,
248,1,255,15,182,107,252,255,141,44,252,234,248,2,139,124,36,24,137,151,233,
137,252,238,72,137,194,137,252,253,137,92,36,28,232,251,1,2,139,149,233,133,
192,15,132,244,249,248,36,15,182,75,252,253,72,139,40,72,137,44,202,139,3,
15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,248,3,139,
141,233,137,89,252,244,141,153,233,41,211,139,105,252,248,184,237,252,233,
244,31,248,41,137,252,239,137,213,137,198,232,251,1,3,15,182,75,252,253,137,
252,234,133,192,15,133,244,42,199,68,202,4,237,252,233,244,43,248,44,137,
4,36,199,68,36,4,237,255,72,141,4,36,128,123,252,252,235,15,133,244,247,65,
141,142,233,137,41,199,65,4,237,137,205,252,233,244,248,248,45,15,182,67,
252,254,199,68,36,4,237,137,4,36,72,141,4,36,252,233,244,247,248,46,15,182,
67,252,254,141,4,194,248,1,15,182,107,252,255,141,44,252,234,248,2,139,124,
36,24,137,151,233,137,252,238,72,137,194,137,252,253,137,92,36,28,232,251,
1,4,139,149,233,133,192,15,132,244,249,15,182,75,252,253,72,139,44,202,72,
137,40,248,47,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,
36,252,238,248,3,255,139,141,233,137,89,252,244,15,182,67,252,253,72,139,
44,194,72,137,105,16,141,153,233,41,211,139,105,252,248,184,237,252,233,244,
31,248,48,139,124,36,24,137,252,238,137,151,233,137,213,137,194,137,92,36,
28,232,251,1,5,15,182,75,252,253,137,252,234,252,233,244,49,248,50,139,108,
36,24,137,149,233,141,52,202,141,20,194,137,252,239,15,182,75,252,252,137,
92,36,28,232,251,1,6,248,3,139,149,233,131,252,248,1,15,135,244,51,248,4,
141,91,4,15,130,244,252,248,5,15,183,67,252,254,141,156,253,131,233,248,6,
255,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,
248,52,131,195,4,129,120,253,4,239,15,130,244,5,252,233,244,6,248,53,129,
120,253,4,239,252,233,244,4,248,54,131,252,235,4,137,206,137,252,233,139,
108,36,24,137,149,233,137,194,137,252,239,137,92,36,28,232,251,1,7,252,233,
244,3,248,55,131,252,235,4,139,108,36,24,137,149,233,137,252,239,139,115,
252,252,137,92,36,28,232,251,1,8,252,233,244,3,248,56,139,108,36,24,137,149,
233,255,137,206,15,183,83,252,254,137,252,239,137,92,36,28,232,251,1,9,139,
149,233,252,233,244,6,248,57,15,182,107,252,255,248,58,65,141,4,199,252,233,
244,247,248,59,15,182,67,252,254,248,60,65,141,4,199,141,44,252,234,149,252,
233,244,248,248,61,141,4,194,137,197,252,233,244,248,248,62,15,182,107,252,
255,248,63,141,4,194,248,1,141,44,252,234,248,2,141,12,202,68,15,182,67,252,
252,137,206,137,193,139,124,36,24,137,151,233,137,252,234,137,252,253,137,
92,36,28,232,251,1,10,139,149,233,255,133,192,15,132,244,47,248,51,137,193,
41,208,137,89,252,244,141,152,233,184,237,252,233,244,29,248,64,139,108,36,
24,137,149,233,141,52,194,137,252,239,137,92,36,28,232,251,1,11,139,149,233,
255,133,192,15,133,244,51,15,183,67,252,254,139,60,194,252,233,244,65,255,
252,233,244,51,255,248,66,141,76,202,8,248,30,137,76,36,4,137,4,36,131,252,
233,8,139,108,36,24,137,149,233,137,206,141,20,193,137,252,239,137,92,36,
28,232,251,1,12,139,149,233,139,76,36,4,139,4,36,139,105,252,248,131,192,
1,65,57,215,15,132,244,67,137,202,137,90,252,252,139,157,233,139,11,15,182,
252,233,15,182,205,131,195,4,65,252,255,36,252,238,248,68,139,108,36,24,137,
149,233,137,206,137,252,239,137,92,36,28,232,251,1,13,139,149,233,139,67,
252,252,15,182,204,15,182,232,193,232,16,65,252,255,164,253,252,238,233,248,
69,129,252,248,239,15,130,244,70,139,106,4,129,252,253,239,15,131,244,70,
139,90,252,252,137,68,36,4,137,106,252,252,139,42,137,106,252,248,131,232,
2,15,132,244,248,255,137,209,248,1,131,193,8,72,139,41,72,137,105,252,248,
131,232,1,15,133,244,1,248,2,139,68,36,4,252,233,244,71,248,72,129,252,248,
239,15,130,244,70,139,106,4,137,252,233,193,252,249,15,131,252,249,252,254,
15,132,244,249,184,237,252,247,213,57,232,15,71,197,248,2,139,106,252,248,
139,132,253,197,233,139,90,252,252,199,66,252,252,237,137,66,252,248,252,
233,244,73,248,3,184,237,255,252,233,244,2,248,74,129,252,248,239,15,130,
```

```
244,70,139,106,4,139,90,252,252,129,252,253,239,15,133,244,252,248,1,139,
42,139,173,233,248,2,133,252,237,199,66,252,252,237,15,132,244,73,65,139,
134,233,199,66,252,252,237,137,106,252,248,139,141,233,35,136,233,105,201,
239,255,3,141,233,248,3,129,185,233,239,15,133,244,250,57,129,233,15,132,
244,251,248,4,139,137,233,133,201,15,133,244,3,252,233,244,73,248,5,139,105,
4,129,252,253,239,15,132,244,73,139,1,137,106,252,252,137,66,252,248,252,
233,244,73,248,6,255,129,252,253,239,15,132,244,1,129,252,253,239,15,135,
244,254,129,252,253,239,15,134,244,253,189,237,252,233,244,254,248,7,189,
237,248,8,252,247,213,65,139,172,253,174,233,252,233,244,2,248,75,129,252,
248,239,255,15,130,244,70,129,122,253,4,239,15,133,244,70,139,42,131,189,
233,0,15,133,244,70,129,122,253,12,239,15,133,244,70,139,66,8,137,133,233,
139,90,252,252,199,66,252,252,237,137,106,252,248,252,246,133,233,235,15,
132,244,247,128,165,233,235,65,139,134,233,255,65,137,174,233,137,133,233,
248,1,252,233,244,73,248,76,129,252,248,239,15,130,244,70,129,122,253,4,239,
15,133,244,70,137,213,139,50,141,82,8,139,124,36,24,232,251,1,14,137,252,
234,72,139,40,139,90,252,252,72,137,106,252,248,252,233,244,73,248,77,129,
252,248,239,15,133,244,70,129,122,253,4,239,15,133,244,247,255,139,42,252,
233,244,78,248,1,15,135,244,70,252,242,15,16,2,252,233,244,79,248,80,129,
252,248,239,15,130,244,70,139,90,252,252,129,122,253,4,239,15,133,244,249,
139,2,248,2,199,66,252,252,237,137,66,252,248,252,233,244,73,248,3,129,122,
253,4,239,255,15,135,244,70,65,131,190,233,0,15,133,244,70,65,139,174,233,
65,59,174,233,15,130,244,247,232,244,81,248,1,139,108,36,24,137,149,233,137,
92,36,28,137,214,137,252,239,232,251,1,15,139,149,233,252,233,244,2,248,82,
129,252,248,239,15,130,244,70,255,15,132,244,248,248,1,129,122,253,4,239,
15,133,244,70,139,108,36,24,137,149,233,137,149,233,139,90,252,252,139,50,
141,82,8,137,252,239,137,92,36,28,232,251,1,16,139,149,233,133,192,15,132,
244,249,72,139,106,8,72,139,66,16,72,137,106,252,248,72,137,2,248,83,184,
237,252,233,244,84,248,2,199,66,12,237,252,233,244,1,248,3,255,199,66,252,
252,237,252,233,244,73,248,85,129,252,248,239,15,130,244,70,139,42,129,122,
253,4,239,15,133,244,70,255,131,189,233,0,15,133,244,70,255,139,106,252,248,
139,133,233,139,90,252,252,199,66,252,252,237,137,66,252,248,199,66,12,237,
184,237,252,233,244,84,248,86,129,252,248,239,15,130,244,70,129,122,253,4,
239,15,133,244,70,129,122,253,12,239,15,133,244,70,139,90,252,252,139,66,
8,131,192,1,199,66,252,252,237,137,66,252,248,139,42,59,133,233,15,131,244,
248,255,193,224,3,3,133,233,248,1,129,120,253,4,239,15,132,244,87,72,139,
40,72,137,42,252,233,244,83,248,2,131,189,233,0,15,132,244,87,137,252,239,
137,213,137,198,232,251,1,3,137,252,234,133,192,15,133,244,1,248,87,184,237,
252,233,244,84,248,88,129,252,248,239,15,130,244,70,255,139,106,252,248,139,
133,233,139,90,252,252,199,66,252,252,237,137,66,252,248,199,66,12,237,199,
66,8,0,0,0,0,184,237,252,233,244,84,248,89,129,252,248,239,15,130,244,70,
141,74,8,131,232,1,187,237,248,1,65,15,182,174,233,193,252,237,235,131,229,
1,1,252,235,252,233,244,29,248,90,129,252,248,239,15,130,244,70,255,129,122,
253,12,239,15,133,244,70,139,106,4,137,106,12,199,66,4,237,139,42,139,90,
8,137,106,8,137,26,141,74,16,131,232,2,187,237,252,233,244,1,248,91,129,252,
248,239,15,130,244,70,139,42,139,90,252,252,137,92,36,28,137,44,36,129,122,
253,4,239,15,133,244,70,72,131,189,233,0,15,133,244,70,128,189,233,235,15,
135,244,70,255,139,141,233,15,132,244,247,59,141,233,15,132,244,70,248,1,
141,92,193,252,240,59,157,233,15,135,244,70,137,157,233,139,108,36,24,137,
149,233,131,194,8,137,149,233,141,108,194,232,72,41,221,57,203,15,132,244,
249,248,2,72,139,4,43,72,137,67,252,248,131,252,235,8,57,203,15,133,244,2,
248,3,137,206,139,60,36,232,244,26,255,139,108,36,24,139,28,36,139,149,233,
65,137,174,233,65,199,134,233,237,129,252,248,239,15,135,244,254,248,4,139,
139,233,68,139,187,233,137,139,233,68,137,252,251,41,203,15,132,244,252,141,
4,26,193,252,235,3,59,133,233,15,135,244,255,137,213,72,41,205,248,5,72,139,
1,72,137,4,41,131,193,8,68,57,252,249,15,133,244,5,248,6,255,141,67,2,199,
66,252,252,237,248,7,139,92,36,28,137,68,36,4,72,199,193,252,248,252,255,
252,255,252,255,252,247,195,237,15,132,244,14,252,233,244,15,248,8,199,66,
252,252,237,139,139,233,131,252,233,8,137,139,233,72,139,1,72,137,2,184,237,
252,233,244,7,248,9,139,12,36,68,137,185,233,137,222,137,252,239,232,251,
1,0,139,28,36,139,149,233,252,233,244,4,248,92,255,139,106,252,248,139,173,
233,139,90,252,252,137,92,36,28,137,44,36,72,131,189,233,0,15,133,244,70,
128,189,233,235,15,135,244,70,139,141,233,15,132,244,247,59,141,233,15,132,
244,70,248,1,141,92,193,252,248,59,157,233,15,135,244,70,137,157,233,139,
108,36,24,137,149,233,255,137,149,233,141,108,194,252,240,72,41,221,57,203,
15,132,244,249,248,2,72,139,4,43,72,137,67,252,248,131,252,235,8,57,203,15,
133,244,2,248,3,137,206,139,60,36,232,244,26,139,108,36,24,139,28,36,139,
149,233,65,137,174,233,65,199,134,233,237,129,252,248,239,15,135,244,254,
248,4,139,139,233,68,139,187,233,137,139,233,255,68,137,252,251,41,203,15,
132,244,252,141,4,26,193,252,235,3,59,133,233,15,135,244,255,137,213,72,41,
205,248,5,72,139,1,72,137,4,41,131,193,8,68,57,252,249,15,133,244,5,248,6,
141,67,1,248,7,139,92,36,28,137,68,36,4,49,201,252,247,195,237,15,132,244,
14,252,233,244,15,248,8,137,222,137,252,239,232,251,1,17,248,9,139,12,36,
68,137,185,233,137,222,137,252,239,232,251,1,0,139,28,36,139,149,233,252,
233,244,4,248,93,255,139,108,36,24,72,252,247,133,233,237,15,132,244,70,137,
149,233,141,68,194,252,248,137,133,233,49,192,72,137,133,233,176,235,136,
133,233,252,233,244,17,248,94,139,90,252,252,221,90,252,248,252,233,244,73,
```

248,95,129,252,248,239,15,130,244,70,129,122,253,4,239,15,133,244,248,255,
139,42,131,252,253,0,15,137,244,78,252,247,221,15,136,244,247,248,96,248,
78,139,90,252,252,199,66,252,252,237,137,106,252,248,252,233,244,73,248,1,
139,90,252,252,199,66,252,252,0,0,224,65,199,66,252,248,0,0,0,0,252,233,244,
73,248,2,15,135,244,70,252,242,15,16,2,72,184,237,237,102,72,15,110,200,15,
84,193,248,79,139,90,252,252,252,242,15,17,66,252,248,248,73,184,237,248,
84,255,137,68,36,4,248,71,252,247,195,237,15,133,244,253,248,5,56,67,252,
255,15,135,244,252,15,182,75,252,253,72,252,247,209,141,20,202,139,3,15,182,
204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,248,6,199,68,194,
252,244,237,131,192,1,252,233,244,5,248,7,72,199,193,252,248,252,255,252,
255,252,255,252,233,244,15,248,97,129,122,253,4,239,15,133,244,247,139,42,
252,233,244,78,248,1,255,15,135,244,70,252,242,15,16,2,232,244,98,252,242,
15,44,232,129,252,253,0,0,0,128,15,133,244,78,252,242,15,42,205,102,15,46,
193,15,138,244,79,15,132,244,78,252,233,244,79,248,99,129,122,253,4,239,15,
133,244,247,139,42,252,233,244,78,248,1,15,135,244,70,255,252,242,15,16,2,
232,244,100,252,242,15,44,232,129,252,253,0,0,0,128,15,133,244,78,252,242,
15,42,205,102,15,46,193,15,138,244,79,15,132,244,78,252,233,244,79,248,101,
129,252,248,239,15,130,244,70,129,122,253,4,239,15,131,244,70,252,242,15,
81,2,252,233,244,79,248,102,255,129,252,248,239,15,133,244,70,129,122,253,
4,239,15,131,244,70,252,242,15,16,2,137,213,232,251,1,18,137,252,234,252,
233,244,79,248,103,129,252,248,239,15,130,244,70,129,122,253,4,239,15,131,
244,70,252,242,15,16,2,137,213,232,251,1,19,137,252,234,252,233,244,79,248,
104,129,252,248,239,15,130,244,70,255,129,122,253,4,239,15,131,244,70,252,
242,15,16,2,137,213,232,251,1,20,137,252,234,252,233,244,79,248,105,129,252,
248,239,15,130,244,70,129,122,253,4,239,15,131,244,70,252,242,15,16,2,137,
213,232,251,1,21,137,252,234,252,233,244,79,248,106,129,252,248,239,15,130,
244,70,129,122,253,4,239,15,131,244,70,255,252,242,15,16,2,137,213,232,251,
1,22,137,252,234,252,233,244,79,248,107,129,252,248,239,15,130,244,70,129,
122,253,4,239,15,131,244,70,252,242,15,16,2,137,213,232,251,1,23,137,252,
234,252,233,244,79,248,108,129,252,248,239,15,130,244,70,129,122,253,4,239,
15,131,244,70,252,242,15,16,2,137,213,232,251,1,24,137,252,234,252,233,244,
79,248,109,255,129,252,248,239,15,130,244,70,129,122,253,4,239,15,131,244,
70,252,242,15,16,2,137,213,232,251,1,25,137,252,234,252,233,244,79,248,110,
129,252,248,239,15,130,244,70,129,122,253,4,239,15,131,244,70,252,242,15,
16,2,137,213,232,251,1,26,137,252,234,252,233,244,79,248,111,129,252,248,
239,15,130,244,70,255,129,122,253,4,239,15,131,244,70,252,242,15,16,2,137,
213,232,251,1,27,137,252,234,252,233,244,79,248,112,129,252,248,239,15,130,
244,70,129,122,253,4,239,15,131,244,70,252,242,15,16,2,137,213,232,251,1,
28,137,252,234,252,233,244,79,248,113,129,252,248,239,15,130,244,70,129,122,
253,4,239,15,131,244,70,255,252,242,15,16,2,137,213,232,251,1,29,137,252,
234,252,233,244,79,248,114,129,252,248,239,15,130,244,70,129,122,253,4,239,
15,131,244,70,129,122,253,12,239,15,131,244,70,252,242,15,16,2,252,242,15,
16,74,8,137,213,232,251,1,30,137,252,234,252,233,244,79,248,115,129,252,248,
239,15,130,244,70,129,122,253,4,239,15,131,244,70,255,129,122,253,12,239,
15,131,244,70,252,242,15,16,2,252,242,15,16,74,8,137,213,232,251,1,31,137,
252,234,252,233,244,79,248,116,129,252,248,239,15,130,244,70,129,122,253,
4,239,15,131,244,70,129,122,253,12,239,15,131,244,70,252,242,15,16,2,252,
242,15,16,74,8,137,213,232,251,1,32,137,252,234,252,233,244,79,248,117,129,
252,248,239,15,130,244,70,255,129,122,253,4,239,15,131,244,70,129,122,253,
12,239,15,131,244,70,221,66,8,221,2,217,252,253,221,217,252,233,244,94,248,
118,129,252,248,239,15,130,244,70,139,106,4,129,252,253,239,15,131,244,70,
139,90,252,252,139,2,137,106,252,252,137,66,252,248,209,229,129,252,253,0,
0,224,252,255,15,131,244,249,9,232,15,132,244,249,184,252,254,3,0,0,129,252,
253,0,0,32,0,15,130,244,250,248,1,255,193,252,237,21,41,197,252,242,15,42,
197,139,106,252,252,129,229,252,255,252,255,15,128,129,205,0,0,224,63,137,
106,252,252,248,2,252,242,15,17,2,184,237,252,233,244,84,248,3,15,87,192,
252,233,244,2,248,4,252,242,15,16,2,72,189,237,237,102,72,15,110,205,252,
242,15,89,193,252,242,15,17,66,252,248,139,106,252,252,184,52,4,0,0,209,229,
252,233,244,1,248,119,129,252,248,239,15,130,244,70,129,122,253,4,239,15,
131,244,70,252,242,15,16,2,139,106,4,139,90,252,252,209,229,129,252,253,0,
0,224,252,255,15,132,244,250,255,15,40,224,232,244,120,252,242,15,92,224,
248,1,252,242,15,17,66,252,248,252,242,15,17,34,139,66,252,252,139,106,4,
49,232,15,136,244,249,248,2,184,237,252,233,244,84,248,3,129,252,245,0,0,
0,128,137,106,4,252,233,244,2,248,4,15,87,228,252,233,244,1,248,121,185,2,
0,0,0,129,122,253,4,239,15,133,244,250,139,42,248,1,255,57,193,15,131,244,
78,129,124,253,202,252,252,239,15,133,244,249,59,108,202,252,248,15,79,108,
202,252,248,131,193,1,252,233,244,1,248,3,15,135,244,70,252,242,15,42,197,
252,233,244,252,248,4,15,135,244,70,252,242,15,16,2,248,5,57,193,15,131,244,
79,129,124,253,202,252,252,239,15,130,244,252,255,15,135,244,70,252,242,15,
42,76,202,252,248,252,233,244,253,248,6,252,242,15,16,76,202,252,248,248,
7,252,242,15,93,193,131,193,1,252,233,244,5,248,122,185,2,0,0,0,129,122,253,
4,239,15,133,244,250,139,42,248,1,57,193,15,131,244,78,129,124,253,202,252,
252,239,15,133,244,249,59,108,202,252,248,15,76,108,202,252,248,131,193,1,
252,233,244,1,248,3,255,15,135,244,70,252,242,15,42,197,252,233,244,252,248,
4,15,135,244,70,252,242,15,16,2,248,5,57,193,15,131,244,79,129,124,253,202,
252,252,239,15,130,244,252,15,135,244,70,252,242,15,42,76,202,252,248,252,

```
233,244,253,248,6,252,242,15,16,76,202,252,248,248,7,252,242,15,95,193,131,
193,1,252,233,244,5,248,123,255,129,252,248,239,15,133,244,70,129,122,253,
4,239,15,133,244,70,139,42,139,90,252,252,131,189,233,1,15,130,244,87,15,
182,173,233,252,233,244,78,248,124,65,139,174,233,65,59,174,233,15,130,244,
247,232,244,81,248,1,255,129,252,248,239,15,133,244,70,129,122,253,4,239,
15,133,244,70,139,42,129,252,253,252,255,0,0,0,15,135,244,70,137,108,36,4,
199,68,36,8,1,0,0,0,72,141,68,36,4,248,125,139,108,36,24,137,149,233,139,
84,36,8,72,137,198,137,252,239,137,92,36,28,232,251,1,33,248,126,139,149,
233,139,90,252,252,199,66,252,252,237,137,66,252,248,252,233,244,73,248,127,
65,139,174,233,65,59,174,233,15,130,244,247,255,232,244,81,248,1,199,68,36,
4,252,255,252,255,252,255,252,255,129,252,248,239,15,130,244,70,15,134,244,
247,129,122,253,20,239,15,133,244,70,139,106,16,137,108,36,4,248,1,129,122,
253,4,239,15,133,244,70,129,122,253,12,239,15,133,244,70,139,42,137,108,36,
8,139,173,233,139,74,8,139,68,36,4,57,197,15,130,244,251,248,2,255,133,201,
15,142,244,253,248,3,139,108,36,8,41,200,15,140,244,128,141,172,253,13,233,
131,192,1,248,4,137,68,36,8,137,232,252,233,244,125,248,5,15,140,244,252,
141,68,40,1,252,233,244,2,248,6,137,232,252,233,244,2,248,7,15,132,244,254,
255,1,252,233,131,193,1,15,143,244,3,248,8,185,1,0,0,0,252,233,244,3,248,
128,49,192,252,233,244,4,248,129,129,252,248,239,15,130,244,70,65,139,174,
233,65,59,174,233,15,130,244,247,232,244,81,248,1,129,122,253,4,239,255,15,
133,244,70,139,108,36,24,65,141,190,233,137,149,233,139,50,139,135,233,137,
175,233,137,135,233,137,92,36,28,232,251,1,34,137,199,232,251,1,35,252,233,
244,126,248,130,129,252,248,239,15,130,244,70,65,139,174,233,65,59,174,233,
15,130,244,247,232,244,81,248,1,255,129,122,253,4,239,15,133,244,70,139,108,
36,24,65,141,190,233,137,149,233,139,50,139,135,233,137,175,233,137,135,233,
137,92,36,28,232,251,1,36,137,199,232,251,1,35,252,233,244,126,248,131,129,
252,248,239,15,130,244,70,65,139,174,233,65,59,174,233,15,130,244,247,232,
244,81,248,1,255,129,122,253,4,239,15,133,244,70,139,108,36,24,65,141,190,
233,137,149,233,139,50,139,135,233,137,175,233,137,135,233,137,92,36,28,232,
251,1,37,137,199,232,251,1,35,252,233,244,126,248,132,129,252,248,239,15,
130,244,70,129,122,253,4,239,15,133,244,247,139,42,252,233,244,96,248,1,255,
15,135,244,70,252,242,15,16,2,72,189,237,237,102,72,15,110,205,252,242,15,
88,193,102,15,126,197,248,2,252,233,244,96,248,133,129,252,248,239,15,130,
244,70,72,189,237,237,102,72,15,110,205,129,122,253,4,239,15,133,244,247,
139,42,252,233,244,248,248,1,15,135,244,70,255,252,242,15,16,2,252,242,15,
88,193,102,15,126,197,248,2,137,68,36,4,141,68,194,252,240,248,1,57,208,15,
134,244,96,129,120,253,4,239,15,133,244,248,35,40,131,232,8,252,233,244,1,
248,2,15,135,244,134,252,242,15,16,0,252,242,15,88,193,102,15,126,193,33,
205,131,232,8,252,233,244,1,248,135,129,252,248,239,15,130,244,70,72,189,
237,237,255,102,72,15,110,205,129,122,253,4,239,15,133,244,247,139,42,252,
233,244,248,248,1,15,135,244,70,252,242,15,16,2,252,242,15,88,193,102,15,
126,197,248,2,137,68,36,4,141,68,194,252,240,248,1,57,208,15,134,244,96,129,
120,253,4,239,15,133,244,248,11,40,131,232,8,252,233,244,1,248,2,15,135,244,
134,255,252,242,15,16,0,252,242,15,88,193,102,15,126,193,9,205,131,232,8,
252,233,244,1,248,136,129,252,248,239,15,130,244,70,72,189,237,237,102,72,
15,110,205,129,122,253,4,239,15,133,244,247,139,42,252,233,244,248,248,1,
15,135,244,70,252,242,15,16,2,252,242,15,88,193,102,15,126,197,248,2,137,
68,36,4,141,68,194,252,240,248,1,57,208,15,134,244,96,255,129,120,253,4,239,
15,133,244,248,51,40,131,232,8,252,233,244,1,248,2,15,135,244,134,252,242,
15,16,0,252,242,15,88,193,102,15,126,193,49,205,131,232,8,252,233,244,1,248,
137,129,252,248,239,15,130,244,70,129,122,253,4,239,15,133,244,247,139,42,
252,233,244,248,248,1,255,15,135,244,70,252,242,15,16,2,72,189,237,237,102,
72,15,110,205,252,242,15,88,193,102,15,126,197,248,2,15,205,252,233,244,96,
248,138,129,252,248,239,15,130,244,70,129,122,253,4,239,15,133,244,247,139,
42,252,233,244,248,248,1,15,135,244,70,252,242,15,16,2,72,189,237,237,255,
102,72,15,110,205,252,242,15,88,193,102,15,126,197,248,2,252,247,213,252,
233,244,96,248,134,139,68,36,4,252,233,244,70,248,139,129,252,248,239,15,
130,244,70,129,122,253,4,239,15,133,244,247,139,42,252,233,244,248,248,1,
15,135,244,70,252,242,15,16,2,72,189,237,237,255,102,72,15,110,205,252,242,
15,88,193,102,15,126,197,248,2,129,122,253,12,239,15,133,244,70,139,74,8,
211,229,252,233,244,96,248,140,129,252,248,239,15,130,244,70,129,122,253,
4,239,15,133,244,247,139,42,252,233,244,248,248,1,15,135,244,70,252,242,15,
16,2,72,189,237,237,255,102,72,15,110,205,252,242,15,88,193,102,15,126,197,
248,2,129,122,253,12,239,15,133,244,70,139,74,8,211,252,237,252,233,244,96,
248,141,129,252,248,239,15,130,244,70,129,122,253,4,239,15,133,244,247,139,
42,252,233,244,248,248,1,15,135,244,70,252,242,15,16,2,72,189,237,237,255,
102,72,15,110,205,252,242,15,88,193,102,15,126,197,248,2,129,122,253,12,239,
15,133,244,70,139,74,8,211,252,253,252,233,244,96,248,142,129,252,248,239,
15,130,244,70,129,122,253,4,239,15,133,244,247,139,42,252,233,244,248,248,
1,15,135,244,70,252,242,15,16,2,72,189,237,237,255,102,72,15,110,205,252,
242,15,88,193,102,15,126,197,248,2,129,122,253,12,239,15,133,244,70,139,74,
8,211,197,252,233,244,96,248,143,129,252,248,239,15,130,244,70,129,122,253,
4,239,15,133,244,247,139,42,252,233,244,248,248,1,15,135,244,70,252,242,15,
16,2,72,189,237,237,255,102,72,15,110,205,252,242,15,88,193,102,15,126,197,
248,2,129,122,253,12,239,15,133,244,70,139,74,8,211,205,252,233,244,96,248,
144,184,237,252,233,244,70,248,145,184,237,248,70,139,108,36,24,139,90,252,
```

252,137,92,36,28,137,149,233,141,68,194,252,248,141,136,233,137,133,233,139,
66,252,248,59,141,233,15,135,244,251,137,252,239,252,255,144,233,255,139,
149,233,133,192,15,143,244,84,248,1,139,141,233,41,209,193,252,233,3,133,
192,141,65,1,139,106,252,248,15,133,244,34,139,157,233,139,11,15,182,252,
233,15,182,205,131,195,4,65,252,255,36,252,238,248,34,137,209,252,247,195,
237,15,133,244,249,15,182,107,252,253,72,252,247,213,141,20,252,234,252,233,
244,29,248,3,137,221,131,229,252,248,41,252,234,252,233,244,29,248,5,190,
237,137,252,239,232,251,1,0,139,149,233,255,49,192,252,233,244,1,248,81,93,
72,137,108,36,8,139,108,36,24,137,92,36,28,137,149,233,141,68,194,252,248,
137,252,239,137,133,233,232,251,1,38,139,149,233,139,133,233,41,208,193,232,
3,131,192,1,72,139,108,36,8,85,195,248,146,65,15,182,134,233,168,235,15,133,
244,251,168,235,15,133,244,247,168,235,15,132,244,247,65,252,255,142,233,
252,233,244,247,248,147,255,65,15,182,134,233,168,235,15,133,244,251,252,
233,244,247,248,148,65,15,182,134,233,168,235,15,133,244,251,168,235,15,132,
244,251,65,252,255,142,233,15,132,244,247,168,235,15,132,244,251,248,1,255,
139,108,36,24,137,149,233,137,222,137,252,239,232,251,1,39,248,3,139,149,
233,248,4,15,182,75,252,253,248,5,15,182,107,252,252,15,183,67,252,254,65,
252,255,164,253,252,238,233,248,149,131,195,4,139,77,232,137,76,36,4,252,
233,244,4,248,150,139,106,252,248,139,173,233,15,182,133,233,141,4,194,139,
108,36,24,137,149,233,137,133,233,137,222,65,141,190,233,73,137,174,233,137,
92,36,28,232,251,1,40,252,233,244,3,248,151,137,92,36,28,252,233,244,247,
248,152,255,137,92,36,28,131,203,1,248,1,141,68,194,252,248,139,108,36,24,
137,149,233,137,133,233,137,222,137,252,239,232,251,1,41,199,68,36,28,0,0,
0,0,131,227,252,254,139,149,233,72,137,193,139,133,233,41,208,72,137,205,
15,182,75,252,253,193,232,3,131,192,1,252,255,229,248,153,139,77,232,137,
12,36,68,137,116,36,8,68,139,116,36,4,15,182,75,252,253,141,12,202,65,131,
252,238,1,15,132,244,248,248,1,72,139,40,72,137,41,131,192,8,131,193,8,65,
131,252,238,1,15,133,244,1,248,2,15,182,67,252,253,15,182,107,252,255,1,232,
141,68,194,252,248,248,3,57,200,15,135,244,255,68,139,116,36,8,139,44,36,
65,139,142,233,139,4,169,133,192,15,132,244,47,15,183,128,233,57,232,15,132,
244,47,255,133,192,15,133,245,65,137,174,233,139,108,36,24,137,149,233,137,
222,65,141,190,233,73,137,174,233,232,251,1,42,139,149,233,252,233,244,47,
248,9,199,65,4,237,131,193,8,252,233,244,3,248,154,255,139,108,36,24,137,
149,233,137,222,137,252,239,232,251,1,43,139,149,233,131,252,235,4,252,233,
244,47,255,248,155,65,85,65,84,65,83,65,82,65,81,65,80,87,86,85,72,141,108,
36,88,85,83,82,81,80,15,182,69,252,248,138,101,252,240,76,137,125,252,248,
76,137,117,252,240,68,139,117,0,65,139,142,233,65,199,134,233,237,65,137,
134,233,65,137,142,233,72,129,252,236,239,72,131,197,128,252,242,68,15,17,
125,252,248,252,242,68,15,17,117,252,240,252,242,68,15,17,109,232,252,242,
68,15,17,101,224,252,242,68,15,17,93,216,252,242,68,15,17,85,208,252,242,
68,15,17,77,200,252,242,68,15,17,69,192,252,242,15,17,125,184,252,242,15,
17,117,176,252,242,15,17,109,168,252,242,15,17,101,160,252,242,15,17,93,152,
252,242,15,17,85,144,252,242,15,17,77,136,252,242,15,17,69,128,65,139,174,
233,65,139,150,233,73,137,174,233,137,149,233,72,137,230,65,141,190,233,65,
199,134,233,0,0,0,0,232,251,1,44,72,139,141,233,72,129,225,239,72,137,204,
137,169,233,139,149,233,139,153,233,252,233,244,247,248,156,255,72,131,196,
16,248,1,76,139,108,36,8,76,139,36,36,255,139,124,36,24,137,151,233,137,197,
137,28,36,137,218,65,139,182,233,232,251,1,45,139,28,36,137,232,139,108,36,
24,139,149,233,255,133,192,15,136,244,255,139,108,36,24,137,68,36,4,68,139,
122,252,248,69,139,191,233,69,139,191,233,137,149,233,65,199,134,233,0,0,
0,0,65,199,134,233,237,139,3,15,182,204,15,182,232,131,195,4,193,232,16,129,
252,253,239,15,130,244,249,129,252,253,239,15,131,244,250,248,2,139,68,36,
4,248,3,65,252,255,36,252,238,248,4,139,66,252,252,169,237,15,133,244,2,255,
15,182,64,252,253,72,252,247,208,68,139,124,194,252,248,69,139,191,233,69,
139,191,233,252,233,244,2,248,9,252,247,216,137,252,239,137,198,232,251,1,
1,248,157,248,98,72,184,237,237,102,72,15,110,208,72,184,237,237,102,72,15,
110,216,15,40,200,102,15,84,202,102,15,46,217,15,134,244,247,102,15,85,208,
252,242,15,88,203,252,242,15,92,203,102,15,86,202,72,184,237,237,102,72,15,
110,208,252,242,15,194,193,1,102,15,84,194,252,242,15,92,200,15,40,193,248,
1,195,248,158,248,100,72,184,237,237,255,102,72,15,110,208,72,184,237,237,
102,72,15,110,216,15,40,200,102,15,84,202,102,15,46,217,15,134,244,247,102,
15,85,208,252,242,15,88,203,252,242,15,92,203,102,15,86,202,72,184,237,237,
102,72,15,110,208,252,242,15,194,193,6,102,15,84,194,252,242,15,92,200,15,
40,193,248,1,195,248,159,248,120,72,184,237,237,102,72,15,110,208,72,184,
237,237,102,72,15,110,216,15,40,200,102,15,84,202,102,15,46,217,15,134,244,
247,102,15,85,208,15,40,193,252,242,15,88,203,252,242,15,92,203,72,184,237,
237,102,72,15,110,216,252,242,15,194,193,1,102,15,84,195,252,242,15,92,200,
102,15,86,202,15,40,193,248,1,195,248,160,15,40,232,252,242,15,94,193,72,
184,237,237,255,102,72,15,110,208,72,184,237,237,102,72,15,110,216,15,40,
224,102,15,84,226,102,15,46,220,15,134,244,247,102,15,85,208,252,242,15,88,
227,252,242,15,92,227,102,15,86,226,72,184,237,237,102,72,15,110,208,252,
242,15,194,196,1,102,15,84,194,252,242,15,92,224,15,40,197,252,242,15,89,
204,252,242,15,92,193,195,248,1,252,242,15,89,200,15,40,197,252,242,15,92,
193,195,248,161,131,252,248,1,15,142,244,252,248,1,169,1,0,0,0,15,133,244,
248,252,242,15,89,192,209,232,252,233,244,1,248,2,209,232,15,132,244,251,
15,40,200,248,3,252,242,15,89,192,209,232,15,132,244,250,255,15,131,244,3,

252,242,15,89,200,252,233,244,3,248,4,252,242,15,89,193,248,5,195,248,6,15,
132,244,5,15,130,244,253,252,247,216,232,244,1,72,184,237,237,102,72,15,110,
200,252,242,15,94,200,15,40,193,195,248,7,72,184,237,237,102,72,15,110,192,
195,248,162,137,252,248,83,15,162,137,6,137,94,4,137,78,8,137,86,12,91,195,
248,163,255,204,255,204,248,164,83,65,87,65,86,72,131,252,236,40,68,141,181,
233,139,157,233,15,183,192,137,131,233,72,137,187,233,72,137,179,233,72,137,
147,233,72,137,139,233,252,242,15,17,131,233,252,242,15,17,139,233,252,242,
15,17,147,233,252,242,15,17,155,233,72,141,132,253,36,233,76,137,131,233,
76,137,139,233,252,242,15,17,163,233,252,242,15,17,171,233,252,242,15,17,
179,233,252,242,15,17,187,233,72,137,131,233,255,72,137,230,137,92,36,28,
137,223,232,251,1,46,65,199,134,233,237,139,144,233,139,128,233,41,208,139,
106,252,248,193,232,3,131,192,1,139,157,233,139,11,15,182,252,233,15,182,
205,131,195,4,65,252,255,36,252,238,248,33,139,76,36,24,65,139,158,233,72,
137,139,233,137,145,233,137,169,233,137,223,137,198,232,251,1,47,72,139,131,
233,252,242,15,16,131,233,252,233,244,17,248,165,85,72,137,229,83,72,137,
252,251,139,131,233,72,41,196,15,182,139,233,131,252,233,1,15,136,244,248,
248,1,255,72,139,132,253,203,233,72,137,132,253,204,233,131,252,233,1,15,
137,244,1,248,2,15,182,131,233,72,139,187,233,72,139,179,233,72,139,147,233,
72,139,139,233,76,139,131,233,76,139,139,233,133,192,15,132,244,251,15,40,
131,233,15,40,139,233,15,40,147,233,15,40,155,233,131,252,248,4,15,134,244,
251,255,15,40,163,233,15,40,171,233,15,40,179,233,15,40,187,233,248,5,252,
255,147,233,72,137,131,233,15,41,131,233,72,137,147,233,15,41,139,233,72,
139,93,252,248,201,195,255,249,255,129,124,253,202,4,239,15,133,244,253,129,
124,253,194,4,239,15,133,244,254,139,44,202,131,195,4,59,44,194,255,15,141,
244,255,255,15,140,244,255,255,15,143,244,255,255,15,142,244,255,255,248,
6,15,183,67,252,254,141,156,253,131,233,248,9,139,3,15,182,204,15,182,232,
131,195,4,193,232,16,65,252,255,36,252,238,248,7,15,135,244,50,129,124,253,
194,4,239,15,130,244,247,15,133,244,50,252,242,15,42,4,194,252,233,244,248,
248,8,15,135,244,50,252,242,15,42,12,202,252,242,15,16,4,194,131,195,4,102,
15,46,193,255,15,134,244,9,255,15,135,244,9,255,15,130,244,9,255,15,131,244,
9,255,252,233,244,6,248,1,252,242,15,16,4,194,248,2,131,195,4,102,15,46,4,
202,248,3,255,252,233,244,6,255,139,108,194,4,131,195,4,129,252,253,239,15,
133,244,253,129,124,253,202,4,239,15,133,244,254,139,44,194,59,44,202,255,
15,133,244,255,255,15,132,244,255,255,15,183,67,252,254,141,156,253,131,233,
248,9,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,
238,248,7,15,135,244,251,129,124,253,202,4,239,15,130,244,247,15,133,244,
251,252,242,15,42,4,202,252,233,244,248,248,8,15,135,244,251,252,242,15,42,
4,194,102,15,46,4,202,252,233,244,250,248,1,252,242,15,16,4,202,248,2,102,
15,46,4,194,248,4,255,15,138,244,248,15,133,244,248,255,15,138,244,248,15,
132,244,247,255,248,1,15,183,67,252,254,141,156,253,131,233,248,2,255,248,
2,15,183,67,252,254,141,156,253,131,233,248,1,255,252,233,244,9,255,139,3,
15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,255,248,
5,129,252,253,239,15,132,244,55,129,124,253,202,4,239,15,132,244,55,57,108,
202,4,15,133,244,2,129,252,253,239,15,131,244,1,139,12,202,139,4,194,57,193,
15,132,244,1,129,252,253,239,15,135,244,2,129,252,253,239,15,130,244,2,255,
139,169,233,133,252,237,15,132,244,2,252,246,133,233,235,15,133,244,2,255,
49,252,237,255,189,1,0,0,0,255,252,233,244,54,255,248,3,129,252,253,239,255,
15,133,244,9,255,252,233,244,55,255,72,252,247,208,139,108,202,4,131,195,
4,129,252,253,239,15,133,244,249,139,12,202,65,59,12,135,255,139,108,202,
4,131,195,4,129,252,253,239,15,133,244,253,65,129,124,253,199,4,239,15,133,
244,254,65,139,44,199,59,44,202,255,15,183,67,252,254,141,156,253,131,233,
248,9,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,
238,248,7,15,135,244,249,65,129,124,253,199,4,239,15,130,244,247,252,242,
65,15,42,4,199,252,233,244,248,248,8,252,242,15,42,4,202,102,65,15,46,4,199,
252,233,244,250,248,1,252,242,65,15,16,4,199,248,2,102,15,46,4,202,248,4,
255,72,252,247,208,139,108,202,4,131,195,4,57,197,255,15,133,244,249,15,183,
67,252,254,141,156,253,131,233,248,2,139,3,15,182,204,15,182,232,131,195,
4,193,232,16,65,252,255,36,252,238,248,3,129,252,253,239,15,133,244,2,252,
233,244,55,255,15,132,244,248,129,252,253,239,15,132,244,55,15,183,67,252,
254,141,156,253,131,233,248,2,139,3,15,182,204,15,182,232,131,195,4,193,232,
16,65,252,255,36,252,238,255,139,108,194,4,131,195,4,129,252,253,239,255,
15,131,244,247,255,137,108,202,4,139,44,194,137,44,202,255,15,183,67,252,
254,141,156,253,131,233,248,1,139,3,15,182,204,15,182,232,131,195,4,193,232,
16,65,252,255,36,252,238,255,3,68,202,4,15,133,244,56,139,3,15,182,204,15,
182,232,131,195,4,193,232,16,65,252,255,36,252,238,255,129,124,253,202,4,
239,15,131,244,56,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,
255,36,252,238,255,72,139,44,194,72,137,44,202,139,3,15,182,204,15,182,232,
131,195,4,193,232,16,65,252,255,36,252,238,255,49,252,237,129,124,253,194,
4,239,129,213,239,137,108,202,4,139,3,15,182,204,15,182,232,131,195,4,193,
232,16,65,252,255,36,252,238,255,129,124,253,194,4,239,15,133,244,251,139,
44,194,252,247,221,15,128,244,250,199,68,202,4,237,137,44,202,248,9,139,3,
15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,248,4,199,
68,202,4,0,0,224,65,199,4,202,0,0,0,0,252,233,244,9,248,5,15,135,244,61,252,
242,15,16,4,194,72,184,237,237,102,72,15,110,200,15,87,193,252,242,15,17,
4,202,252,233,244,9,255,129,124,253,194,4,239,15,133,244,248,139,4,194,139,
128,233,248,1,199,68,202,4,237,137,4,202,139,3,15,182,204,15,182,232,131,

195,4,193,232,16,65,252,255,36,252,238,248,2,129,124,253,194,4,239,15,133,
244,64,139,60,194,255,139,175,233,131,252,253,0,15,133,244,255,248,3,255,
248,65,137,213,232,251,1,48,137,252,234,15,182,75,252,253,252,233,244,1,255,
248,9,252,246,133,233,235,15,133,244,3,252,233,244,64,255,15,182,252,236,
15,182,192,255,129,124,253,252,234,4,239,15,133,244,58,65,129,124,253,199,
4,239,15,133,244,58,139,44,252,234,65,3,44,199,15,128,244,57,255,129,124,
253,252,234,4,239,15,133,244,60,65,129,124,253,199,4,239,15,133,244,60,65,
139,4,199,3,4,252,234,15,128,244,59,255,129,124,253,252,234,4,239,15,133,
244,63,129,124,253,194,4,239,15,133,244,63,139,44,252,234,3,44,194,15,128,
244,62,255,199,68,202,4,237,255,137,4,202,255,129,124,253,252,234,4,239,15,
133,244,58,65,129,124,253,199,4,239,15,133,244,58,139,44,252,234,65,43,44,
199,15,128,244,57,255,129,124,253,252,234,4,239,15,133,244,60,65,129,124,
253,199,4,239,15,133,244,60,65,139,4,199,43,4,252,234,15,128,244,59,255,129,
124,253,252,234,4,239,15,133,244,63,129,124,253,194,4,239,15,133,244,63,139,
44,252,234,43,44,194,15,128,244,62,255,129,124,253,252,234,4,239,15,133,244,
58,65,129,124,253,199,4,239,15,133,244,58,139,44,252,234,65,15,175,44,199,
15,128,244,57,255,129,124,253,252,234,4,239,15,133,244,60,65,129,124,253,
199,4,239,15,133,244,60,65,139,4,199,15,175,4,252,234,15,128,244,59,255,129,
124,253,252,234,4,239,15,133,244,63,129,124,253,194,4,239,15,133,244,63,139,
44,252,234,15,175,44,194,15,128,244,62,255,129,124,253,252,234,4,239,15,131,
244,58,65,129,124,253,199,4,239,15,131,244,58,252,242,15,16,4,252,234,252,
242,65,15,94,4,199,255,129,124,253,252,234,4,239,15,131,244,60,65,129,124,
253,199,4,239,15,131,244,60,252,242,65,15,16,4,199,252,242,15,94,4,252,234,
255,129,124,253,252,234,4,239,15,131,244,63,129,124,253,194,4,239,15,131,
244,63,252,242,15,16,4,252,234,252,242,15,94,4,194,255,252,242,15,17,4,202,
139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,255,
129,124,253,252,234,4,239,15,131,244,58,65,129,124,253,199,4,239,15,131,244,
58,252,242,15,16,4,252,234,252,242,65,15,16,12,199,255,129,124,253,252,234,
4,239,15,131,244,60,65,129,124,253,199,4,239,15,131,244,60,252,242,65,15,
16,4,199,252,242,15,16,12,252,234,255,129,124,253,252,234,4,239,15,131,244,
63,129,124,253,194,4,239,15,131,244,63,252,242,15,16,4,252,234,252,242,15,
16,12,194,255,248,166,232,244,160,252,242,15,17,4,202,139,3,15,182,204,15,
182,232,131,195,4,193,232,16,65,252,255,36,252,238,255,252,233,244,166,255,
137,213,232,251,1,30,15,182,75,252,253,137,252,234,252,242,15,17,4,202,139,
3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,255,15,
182,252,236,15,182,192,139,124,36,24,137,151,233,141,52,194,137,194,41,252,
234,248,37,137,252,253,137,92,36,28,232,251,1,49,139,149,233,133,192,15,133,
244,51,15,182,107,252,255,15,182,75,252,253,72,139,4,252,234,72,137,4,202,
139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,255,
72,252,247,208,65,139,4,135,199,68,202,4,237,137,4,202,139,3,15,182,204,15,
182,232,131,195,4,193,232,16,65,252,255,36,252,238,255,15,191,192,199,68,
202,4,237,137,4,202,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,
255,36,252,238,255,252,242,65,15,16,4,199,252,242,15,17,4,202,139,3,15,182,
204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,255,72,252,247,
208,137,68,202,4,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,
255,36,252,238,255,141,76,202,12,141,68,194,4,189,237,137,105,252,248,248,
1,137,41,131,193,8,57,193,15,134,244,1,139,3,15,182,204,15,182,232,131,195,
4,193,232,16,65,252,255,36,252,238,255,139,106,252,248,139,172,253,133,233,
139,173,233,72,139,69,0,72,137,4,202,139,3,15,182,204,15,182,232,131,195,
4,193,232,16,65,252,255,36,252,238,255,139,106,252,248,139,172,253,141,233,
128,189,233,0,139,173,233,139,12,194,139,68,194,4,137,77,0,137,69,4,15,132,
244,247,252,246,133,233,235,15,133,244,248,248,1,139,3,15,182,204,15,182,
232,131,195,4,193,232,16,65,252,255,36,252,238,248,2,129,232,239,129,252,
248,239,15,134,244,1,252,246,129,233,235,15,132,244,1,137,252,238,137,213,
65,141,190,233,255,232,251,1,50,137,252,234,252,233,244,1,255,72,252,247,
208,139,106,252,248,139,172,253,141,233,65,139,12,135,139,133,233,137,8,199,
64,4,237,252,246,133,233,235,15,133,244,248,248,1,139,3,15,182,204,15,182,
232,131,195,4,193,232,16,65,252,255,36,252,238,248,2,252,246,129,233,235,
15,132,244,1,128,189,233,0,15,132,244,1,137,213,137,198,65,141,190,233,232,
251,1,50,137,252,234,252,233,244,1,255,139,106,252,248,252,242,65,15,16,4,
199,139,172,253,141,233,139,141,233,252,242,15,17,1,139,3,15,182,204,15,182,
232,131,195,4,193,232,16,65,252,255,36,252,238,255,72,252,247,208,139,106,
252,248,139,172,253,141,233,139,141,233,137,65,4,139,3,15,182,204,15,182,
232,131,195,4,193,232,16,65,252,255,36,252,238,255,141,156,253,131,233,139,
108,36,24,131,189,233,0,15,132,244,247,137,149,233,141,52,202,137,252,239,
232,251,1,51,139,149,233,248,1,139,3,15,182,204,15,182,232,131,195,4,193,
232,16,65,252,255,36,252,238,255,72,252,247,208,139,108,36,24,137,149,233,
139,82,252,248,65,139,52,135,137,252,239,137,92,36,28,232,251,1,52,139,149,
233,15,182,75,252,253,137,4,202,199,68,202,4,237,139,3,15,182,204,15,182,
232,131,195,4,193,232,16,65,252,255,36,252,238,255,139,108,36,24,137,149,
233,65,139,142,233,65,59,142,233,137,92,36,28,15,131,244,251,248,1,137,194,
37,252,255,7,0,0,193,252,234,11,61,252,255,7,0,0,15,132,244,249,248,2,137,
252,239,137,198,232,251,1,53,139,149,233,15,182,75,252,253,137,4,202,199,
68,202,4,237,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,
36,252,238,248,3,184,1,8,0,0,252,233,244,2,248,5,137,252,239,232,251,1,54,
15,183,67,252,254,252,233,244,1,255,72,252,247,208,139,108,36,24,65,139,142,

233,137,92,36,28,65,59,142,233,137,149,233,15,131,244,249,248,2,65,139,52,
135,137,252,239,232,251,1,55,139,149,233,15,182,75,252,253,137,4,202,199,
68,202,4,237,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,
36,252,238,248,3,137,252,239,232,251,1,54,15,183,67,252,254,72,252,247,208,
252,233,244,2,255,72,252,247,208,139,106,252,248,139,173,233,65,139,4,135,
252,233,244,167,255,72,252,247,208,139,106,252,248,139,173,233,65,139,4,135,
252,233,244,168,255,15,182,252,236,15,182,192,129,124,253,252,234,4,239,15,
133,244,40,139,44,252,234,129,124,253,194,4,239,15,133,244,251,139,4,194,
59,133,233,15,131,244,40,193,224,3,3,133,233,129,120,253,4,239,15,132,244,
248,72,139,40,72,137,44,202,248,1,139,3,15,182,204,15,182,232,131,195,4,193,
232,16,65,252,255,36,252,238,248,2,131,189,233,0,15,132,244,249,139,141,233,
252,246,129,233,235,255,15,132,244,40,15,182,75,252,253,248,3,199,68,202,
4,237,252,233,244,1,248,5,129,124,253,194,4,239,15,133,244,40,139,4,194,252,
233,244,167,255,15,182,252,236,15,182,192,72,252,247,208,65,139,4,135,129,
124,253,252,234,4,239,15,133,244,38,139,44,252,234,248,167,139,141,233,35,
136,233,105,201,239,3,141,233,248,1,129,185,233,239,15,133,244,250,57,129,
233,15,133,244,250,129,121,253,4,239,15,132,244,251,15,182,67,252,253,72,
139,41,72,137,44,194,248,2,255,139,3,15,182,204,15,182,232,131,195,4,193,
232,16,65,252,255,36,252,238,248,3,15,182,67,252,253,199,68,194,4,237,252,
233,244,2,248,4,139,137,233,133,201,15,133,244,1,248,5,139,141,233,133,201,
15,132,244,3,252,246,129,233,235,15,133,244,3,252,233,244,38,255,15,182,252,
236,15,182,192,129,124,253,252,234,4,239,15,133,244,39,139,44,252,234,59,
133,233,15,131,244,39,193,224,3,3,133,233,129,120,253,4,239,15,132,244,248,
72,139,40,72,137,44,202,248,1,139,3,15,182,204,15,182,232,131,195,4,193,232,
16,65,252,255,36,252,238,248,2,131,189,233,0,15,132,244,249,139,141,233,252,
246,129,233,235,15,132,244,39,255,15,182,75,252,253,248,3,199,68,202,4,237,
252,233,244,1,255,15,182,252,236,15,182,192,139,44,252,234,139,4,194,59,133,
233,15,131,244,41,193,224,3,3,133,233,248,42,72,139,40,72,137,44,202,248,
43,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,
255,15,182,252,236,15,182,192,129,124,253,252,234,4,239,15,133,244,46,139,
44,252,234,129,124,253,194,4,239,15,133,244,251,139,4,194,59,133,233,15,131,
244,46,193,224,3,3,133,233,129,120,253,4,239,15,132,244,249,248,1,252,246,
133,233,235,15,133,244,253,248,2,72,139,44,202,72,137,40,139,3,15,182,204,
15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,248,3,255,131,189,233,
0,15,132,244,1,139,141,233,252,246,129,233,235,15,132,244,46,15,182,75,252,
253,252,233,244,1,248,5,129,124,253,194,4,239,15,133,244,46,139,4,194,252,
233,244,168,248,7,128,165,233,235,65,139,142,233,255,65,137,174,233,137,141,
233,15,182,75,252,253,252,233,244,2,255,15,182,252,236,15,182,192,72,252,
247,208,65,139,4,135,129,124,253,252,234,4,239,15,133,244,44,139,44,252,234,
248,168,139,141,233,35,136,233,105,201,239,198,133,233,0,3,141,233,248,1,
129,185,233,239,15,133,244,251,57,129,233,15,133,244,251,129,121,253,4,239,
15,132,244,250,248,2,255,252,246,133,233,235,15,133,244,253,248,3,15,182,
67,252,253,72,139,44,194,72,137,41,139,3,15,182,204,15,182,232,131,195,4,
193,232,16,65,252,255,36,252,238,248,4,131,189,233,0,15,132,244,2,137,12,
36,139,141,233,252,246,129,233,235,15,132,244,44,139,12,36,252,233,244,2,
248,5,139,137,233,133,201,15,133,244,1,255,139,141,233,133,201,15,132,244,
252,252,246,129,233,235,15,132,244,44,248,6,137,4,36,199,68,36,4,237,137,
108,36,8,139,124,36,24,137,151,233,72,141,20,36,137,252,238,137,252,253,137,
92,36,28,232,251,1,56,139,149,233,139,108,36,8,137,193,252,233,244,2,248,
7,128,165,233,235,65,139,134,233,65,137,174,233,137,133,233,252,233,244,3,
255,15,182,252,236,15,182,192,129,124,253,252,234,4,239,15,133,244,45,139,
44,252,234,59,133,233,15,131,244,45,193,224,3,3,133,233,129,120,253,4,239,
15,132,244,249,248,1,252,246,133,233,235,15,133,244,253,248,2,72,139,12,202,
72,137,8,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,
238,248,3,131,189,233,0,15,132,244,1,255,139,141,233,252,246,129,233,235,
15,132,244,45,15,182,75,252,253,252,233,244,1,248,7,128,165,233,235,65,139,
142,233,65,137,174,233,137,141,233,15,182,75,252,253,252,233,244,2,255,15,
182,252,236,15,182,192,139,44,252,234,139,4,194,252,246,133,233,235,15,133,
244,253,248,2,59,133,233,15,131,244,48,193,224,3,3,133,233,248,49,72,139,
44,202,72,137,40,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,
255,36,252,238,248,7,128,165,233,235,65,139,142,233,65,137,174,233,137,141,
233,15,182,75,252,253,252,233,244,2,255,68,137,60,36,69,139,60,199,248,1,
141,12,202,139,105,252,248,252,246,133,233,235,15,133,244,253,248,2,139,68,
36,4,131,232,1,15,132,244,250,68,1,252,248,59,133,233,15,135,244,251,68,41,
252,248,65,193,231,3,68,3,189,233,248,3,72,139,41,131,193,8,73,137,47,65,
131,199,8,131,232,1,15,133,244,3,248,4,68,139,60,36,139,3,15,182,204,15,182,
232,131,195,4,193,232,16,65,252,255,36,252,238,248,5,139,124,36,24,137,151,
233,137,252,238,137,194,137,252,253,137,92,36,28,232,251,1,57,139,149,233,
15,182,75,252,253,252,233,244,1,248,7,255,128,165,233,235,65,139,134,233,
65,137,174,233,137,133,233,252,233,244,2,255,3,68,36,4,255,129,124,253,202,
4,239,139,44,202,15,133,244,66,141,84,202,8,137,90,252,252,139,157,233,139,
11,15,182,252,233,15,182,205,131,195,4,65,252,255,36,252,238,255,141,76,202,
8,65,137,215,139,105,252,248,129,121,253,252,252,239,15,133,244,30,248,67,
139,90,252,252,252,247,195,237,15,133,244,253,248,1,137,106,252,248,137,68,
36,4,131,232,1,15,132,244,249,248,2,72,139,41,131,193,8,73,137,47,65,131,
199,8,131,232,1,15,133,244,2,139,106,252,248,248,3,139,68,36,4,128,189,233,

1,15,135,244,251,248,4,139,157,233,139,11,15,182,252,233,15,182,205,131,195,
4,65,252,255,36,252,238,248,5,255,252,247,195,237,15,133,244,4,15,182,75,
252,253,72,252,247,209,68,139,124,202,252,248,69,139,191,233,69,139,191,233,
252,233,244,4,248,7,129,252,235,239,252,247,195,237,15,133,244,254,41,218,
65,137,215,139,90,252,252,252,233,244,1,248,8,129,195,239,252,233,244,1,255,
141,76,202,8,72,139,105,232,72,139,65,252,240,72,137,41,72,137,65,8,139,105,
224,139,65,228,137,105,252,248,137,65,252,252,129,252,248,239,184,237,15,
133,244,30,137,202,137,90,252,252,139,157,233,139,11,15,182,252,233,15,182,
205,131,195,4,65,252,255,36,252,238,255,68,137,60,36,68,137,116,36,4,139,
108,202,252,240,139,68,202,252,248,68,139,181,233,131,195,4,68,139,189,233,
248,1,68,57,252,240,15,131,244,251,65,129,124,253,199,4,239,15,132,244,250,
199,68,202,4,237,137,4,202,73,139,44,199,72,137,108,202,8,131,192,1,137,68,
202,252,248,248,2,15,183,67,252,254,141,156,253,131,233,248,3,68,139,116,
36,4,68,139,60,36,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,
255,36,252,238,248,4,131,192,1,252,233,244,1,248,5,68,41,252,240,248,6,59,
133,233,15,135,244,3,255,68,105,252,248,239,68,3,189,233,65,129,191,233,239,
15,132,244,253,70,141,116,48,1,73,139,175,233,73,139,135,233,72,137,44,202,
72,137,68,202,8,68,137,116,202,252,248,252,233,244,2,248,7,131,192,1,252,
233,244,6,255,129,124,253,202,252,236,239,15,133,244,251,139,108,202,232,
129,124,253,202,252,244,239,15,133,244,251,129,124,253,202,252,252,239,15,
133,244,251,128,189,233,235,15,133,244,251,141,156,253,131,233,199,68,202,
252,248,0,0,0,0,199,68,202,252,252,252,255,127,252,254,252,255,248,1,139,
3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,248,5,
198,67,252,252,235,141,156,253,131,233,198,3,235,252,233,244,1,255,15,182,
252,236,15,182,192,68,137,60,36,68,141,188,253,194,233,141,12,202,68,43,122,
252,252,133,252,237,15,132,244,251,141,108,252,233,252,248,65,57,215,15,131,
244,248,248,1,73,139,71,252,248,65,131,199,8,72,137,1,131,193,8,57,252,233,
15,131,244,249,65,57,215,15,130,244,1,248,2,199,65,4,237,131,193,8,57,252,
233,15,130,244,2,248,3,68,139,60,36,139,3,15,182,204,15,182,232,131,195,4,
193,232,16,65,252,255,36,252,238,248,5,199,68,36,4,1,0,0,0,137,208,68,41,
252,248,15,134,244,3,137,197,193,252,237,3,131,197,1,137,108,36,4,139,108,
36,24,1,200,59,133,233,15,135,244,253,248,6,255,73,139,71,252,248,65,131,
199,8,72,137,1,131,193,8,65,57,215,15,130,244,6,252,233,244,3,248,7,137,149,
233,137,141,233,137,92,36,28,65,41,215,139,116,36,4,131,252,238,1,137,252,
239,232,251,1,0,139,149,233,139,141,233,65,1,215,252,233,244,6,255,193,225,
3,255,248,1,139,90,252,252,137,68,36,4,252,247,195,237,15,133,244,253,255,
248,14,65,137,215,131,232,1,15,132,244,249,248,2,73,139,44,15,73,137,111,
252,248,65,131,199,8,131,232,1,15,133,244,2,248,3,139,68,36,4,15,182,107,
252,255,248,5,57,197,15,135,244,252,255,72,139,44,10,72,137,106,252,248,255,
248,5,56,67,252,255,15,135,244,252,255,15,182,75,252,253,72,252,247,209,141,
20,202,68,139,122,252,248,69,139,191,233,69,139,191,233,139,3,15,182,204,
15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,248,6,255,65,199,71,
252,252,237,65,131,199,8,255,199,68,194,252,244,237,255,131,192,1,252,233,
244,5,248,7,141,171,233,252,247,197,237,15,133,244,15,41,252,234,255,1,252,
233,255,137,221,209,252,237,129,229,239,102,65,129,172,253,46,233,238,15,
130,244,150,255,141,12,202,255,129,121,253,4,239,15,133,244,255,255,129,121,
253,12,239,15,133,244,68,129,121,253,20,239,15,133,244,68,139,41,131,121,
16,0,15,140,244,251,255,129,121,253,12,239,15,133,244,163,129,121,253,20,
239,15,133,244,163,255,139,105,16,133,252,237,15,136,244,251,3,41,15,128,
244,247,137,41,255,59,105,8,199,65,28,237,137,105,24,255,15,142,244,253,248,
1,248,6,141,156,253,131,233,255,141,156,253,131,233,15,183,67,252,254,15,
142,245,248,1,248,6,255,15,143,244,253,248,6,141,156,253,131,233,248,1,255,
248,7,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,252,255,36,252,
238,248,5,255,3,41,15,128,244,1,137,41,255,15,141,244,7,255,141,156,253,131,
233,15,183,67,252,254,15,141,245,255,15,140,244,7,255,252,233,244,6,248,9,
255,129,121,253,4,239,255,15,131,244,68,129,121,253,12,239,15,131,244,68,
255,129,121,253,12,239,15,131,244,163,129,121,253,20,239,15,131,244,163,255,
139,105,20,255,129,252,253,239,15,131,244,68,255,252,242,15,16,1,252,242,
15,16,73,8,255,252,242,15,88,65,16,252,242,15,17,1,133,252,237,15,136,244,
249,255,15,140,244,249,255,102,15,46,200,248,1,252,242,15,17,65,24,255,15,
131,244,7,255,141,156,253,131,233,15,183,67,252,254,15,131,245,255,15,130,
244,7,255,252,233,244,6,248,3,102,15,46,193,252,233,244,1,255,141,12,202,
139,105,4,129,252,253,239,15,132,244,247,255,137,105,252,252,139,41,137,105,
252,248,252,233,245,255,141,156,253,131,233,139,1,137,105,252,252,137,65,
252,248,255,139,108,36,24,137,149,233,137,4,36,137,218,137,198,137,252,239,
232,251,1,58,139,4,36,139,149,233,255,65,139,142,233,139,4,129,72,139,128,
233,139,108,36,24,65,137,150,233,65,137,174,233,76,137,36,36,76,137,108,36,
8,72,131,252,236,16,252,255,224,255,141,156,253,131,233,139,3,15,182,204,
15,182,232,131,195,4,193,232,16,65,252,255,36,252,238,255,137,221,209,252,
237,129,229,239,102,65,129,172,253,46,233,238,15,130,244,152,255,68,139,187,
233,139,108,36,24,141,12,202,59,141,233,15,135,244,25,15,182,139,233,57,200,
15,134,244,249,248,2,255,15,183,67,252,254,252,233,245,255,248,3,199,68,194,
252,252,237,131,192,1,57,200,15,134,244,3,252,233,244,2,255,141,44,197,237,
141,4,194,68,139,122,252,248,137,104,252,252,68,137,120,252,248,139,108,36,
24,141,12,200,59,141,233,15,135,244,24,137,209,137,194,15,182,171,233,133,
252,237,15,132,244,248,248,1,131,193,8,57,209,15,131,244,249,68,139,121,252,

```
  248,68,137,56,68,139,121,252,252,68,137,120,4,131,192,8,199,65,252,252,237,
  131,252,237,1,15,133,244,1,248,2,255,68,139,187,233,139,3,15,182,204,15,182,
  232,131,195,4,193,232,16,65,252,255,36,252,238,255,248,3,199,64,4,237,131,
  192,8,131,252,237,1,15,133,244,3,252,233,244,2,255,139,106,252,248,76,139,
  189,233,139,108,36,24,141,68,194,252,248,137,149,233,141,136,233,59,141,233,
  137,133,233,255,137,252,239,255,76,137,252,254,137,252,239,255,15,135,244,
  23,65,199,134,233,237,255,65,252,255,215,255,65,252,255,150,233,255,139,149,
  233,65,137,174,233,65,199,134,233,237,141,12,194,252,247,217,3,141,233,139,
  90,252,252,252,233,244,12,255,254,0
};

#line 13 "vm_x86.dasc"
//|.globals GLOB_
enum {
  GLOB_vm_returnp,
  GLOB_cont_dispatch,
  GLOB_vm_returnc,
  GLOB_vm_unwind_yield,
  GLOB_BC_RET_Z,
  GLOB_vm_return,
  GLOB_vm_leave_cp,
  GLOB_vm_leave_unw,
  GLOB_vm_unwind_c_eh,
  GLOB_vm_unwind_c,
  GLOB_vm_unwind_rethrow,
  GLOB_vm_unwind_ff,
  GLOB_vm_unwind_ff_eh,
  GLOB_vm_growstack_c,
  GLOB_vm_growstack_v,
  GLOB_vm_growstack_f,
  GLOB_vm_resume,
  GLOB_vm_pcall,
  GLOB_vm_call,
  GLOB_vm_call_dispatch,
  GLOB_vmeta_call,
  GLOB_vm_call_dispatch_f,
  GLOB_vm_cpcall,
  GLOB_cont_ffi_callback,
  GLOB_vm_call_tail,
  GLOB_cont_cat,
  GLOB_cont_ra,
  GLOB_BC_CAT_Z,
  GLOB_vmeta_tgets,
  GLOB_vmeta_tgetb,
  GLOB_vmeta_tgetv,
  GLOB_vmeta_tgetr,
  GLOB_BC_TGETR_Z,
  GLOB_BC_TGETR2_Z,
  GLOB_vmeta_tsets,
  GLOB_vmeta_tsetb,
  GLOB_vmeta_tsetv,
  GLOB_cont_nop,
  GLOB_vmeta_tsetr,
  GLOB_BC_TSETR_Z,
  GLOB_vmeta_comp,
  GLOB_vmeta_binop,
  GLOB_cont_condt,
  GLOB_cont_condf,
  GLOB_vmeta_equal,
  GLOB_vmeta_equal_cd,
  GLOB_vmeta_istype,
  GLOB_vmeta_arith_vno,
  GLOB_vmeta_arith_vn,
  GLOB_vmeta_arith_nvo,
  GLOB_vmeta_arith_nv,
  GLOB_vmeta_unm,
  GLOB_vmeta_arith_vvo,
  GLOB_vmeta_arith_vv,
  GLOB_vmeta_len,
  GLOB_BC_LEN_Z,
  GLOB_vmeta_call_ra,
  GLOB_BC_CALLT_Z,
  GLOB_vmeta_for,
  GLOB_ff_assert,
  GLOB_fff_fallback,
  GLOB_fff_res_,
```

```
  GLOB_ff_type,
  GLOB_fff_res1,
  GLOB_ff_getmetatable,
  GLOB_ff_setmetatable,
  GLOB_ff_rawget,
  GLOB_ff_tonumber,
  GLOB_fff_resi,
  GLOB_fff_resxmm0,
  GLOB_ff_tostring,
  GLOB_fff_gcstep,
  GLOB_ff_next,
  GLOB_fff_res2,
  GLOB_fff_res,
  GLOB_ff_pairs,
  GLOB_ff_ipairs_aux,
  GLOB_fff_res0,
  GLOB_ff_ipairs,
  GLOB_ff_pcall,
  GLOB_ff_xpcall,
  GLOB_ff_coroutine_resume,
  GLOB_ff_coroutine_wrap_aux,
  GLOB_ff_coroutine_yield,
  GLOB_fff_resn,
  GLOB_ff_math_abs,
  GLOB_fff_resbit,
  GLOB_ff_math_floor,
  GLOB_vm_floor_sse,
  GLOB_ff_math_ceil,
  GLOB_vm_ceil_sse,
  GLOB_ff_math_sqrt,
  GLOB_ff_math_log,
  GLOB_ff_math_log10,
  GLOB_ff_math_exp,
  GLOB_ff_math_sin,
  GLOB_ff_math_cos,
  GLOB_ff_math_tan,
  GLOB_ff_math_asin,
  GLOB_ff_math_acos,
  GLOB_ff_math_atan,
  GLOB_ff_math_sinh,
  GLOB_ff_math_cosh,
  GLOB_ff_math_tanh,
  GLOB_ff_math_pow,
  GLOB_ff_math_atan2,
  GLOB_ff_math_fmod,
  GLOB_ff_math_ldexp,
  GLOB_ff_math_frexp,
  GLOB_ff_math_modf,
  GLOB_vm_trunc_sse,
  GLOB_ff_math_min,
  GLOB_ff_math_max,
  GLOB_ff_string_byte,
  GLOB_ff_string_char,
  GLOB_fff_newstr,
  GLOB_fff_resstr,
  GLOB_ff_string_sub,
  GLOB_fff_emptystr,
  GLOB_ff_string_reverse,
  GLOB_ff_string_lower,
  GLOB_ff_string_upper,
  GLOB_ff_bit_tobit,
  GLOB_ff_bit_band,
  GLOB_fff_fallback_bit_op,
  GLOB_ff_bit_bor,
  GLOB_ff_bit_bxor,
  GLOB_ff_bit_bswap,
  GLOB_ff_bit_bnot,
  GLOB_ff_bit_lshift,
  GLOB_ff_bit_rshift,
  GLOB_ff_bit_arshift,
  GLOB_ff_bit_rol,
  GLOB_ff_bit_ror,
  GLOB_fff_fallback_2,
  GLOB_fff_fallback_1,
  GLOB_vm_record,
  GLOB_vm_rethook,
```

```
  GLOB_vm_inshook,
  GLOB_cont_hook,
  GLOB_vm_hotloop,
  GLOB_vm_callhook,
  GLOB_vm_hotcall,
  GLOB_cont_stitch,
  GLOB_vm_profhook,
  GLOB_vm_exit_handler,
  GLOB_vm_exit_interp,
  GLOB_vm_floor,
  GLOB_vm_ceil,
  GLOB_vm_trunc,
  GLOB_vm_mod,
  GLOB_vm_powi_sse,
  GLOB_vm_cpuid,
  GLOB_assert_bad_for_arg_type,
  GLOB_vm_ffi_callback,
  GLOB_vm_ffi_call,
  GLOB_BC_MODVN_Z,
  GLOB_BC_TGETS_Z,
  GLOB_BC_TSETS_Z,
  GLOB__MAX
};
#line 14 "vm_x86.dasc"
//|.globalnames globnames
static const char *const globnames[] = {
  "vm_returnp",
  "cont_dispatch",
  "vm_returnc",
  "vm_unwind_yield",
  "BC_RET_Z",
  "vm_return",
  "vm_leave_cp",
  "vm_leave_unw",
  "vm_unwind_c_eh",
  "vm_unwind_c@8",
  "vm_unwind_rethrow",
  "vm_unwind_ff@4",
  "vm_unwind_ff_eh",
  "vm_growstack_c",
  "vm_growstack_v",
  "vm_growstack_f",
  "vm_resume",
  "vm_pcall",
  "vm_call",
  "vm_call_dispatch",
  "vmeta_call",
  "vm_call_dispatch_f",
  "vm_cpcall",
  "cont_ffi_callback",
  "vm_call_tail",
  "cont_cat",
  "cont_ra",
  "BC_CAT_Z",
  "vmeta_tgets",
  "vmeta_tgetb",
  "vmeta_tgetv",
  "vmeta_tgetr",
  "BC_TGETR_Z",
  "BC_TGETR2_Z",
  "vmeta_tsets",
  "vmeta_tsetb",
  "vmeta_tsetv",
  "cont_nop",
  "vmeta_tsetr",
  "BC_TSETR_Z",
  "vmeta_comp",
  "vmeta_binop",
  "cont_condt",
  "cont_condf",
  "vmeta_equal",
  "vmeta_equal_cd",
  "vmeta_istype",
  "vmeta_arith_vno",
  "vmeta_arith_vn",
  "vmeta_arith_nvo",
```

```
  "vmeta_arith_nv",
  "vmeta_unm",
  "vmeta_arith_vvo",
  "vmeta_arith_vv",
  "vmeta_len",
  "BC_LEN_Z",
  "vmeta_call_ra",
  "BC_CALLT_Z",
  "vmeta_for",
  "ff_assert",
  "fff_fallback",
  "fff_res_",
  "ff_type",
  "fff_res1",
  "ff_getmetatable",
  "ff_setmetatable",
  "ff_rawget",
  "ff_tonumber",
  "fff_resi",
  "fff_resxmm0",
  "ff_tostring",
  "fff_gcstep",
  "ff_next",
  "fff_res2",
  "fff_res",
  "ff_pairs",
  "ff_ipairs_aux",
  "fff_res0",
  "ff_ipairs",
  "ff_pcall",
  "ff_xpcall",
  "ff_coroutine_resume",
  "ff_coroutine_wrap_aux",
   "ff_coroutine_yield",
   "fff_resn",
   "ff_math_abs",
   "fff_resbit",
   "ff_math_floor",
   "vm_floor_sse",
   "ff_math_ceil",
   "vm_ceil_sse",
   "ff_math_sqrt",
   "ff_math_log",
   "ff_math_log10",
   "ff_math_exp",
   "ff_math_sin",
   "ff_math_cos",
   "ff_math_tan",
   "ff_math_asin",
   "ff_math_acos",
   "ff_math_atan",
   "ff_math_sinh",
   "ff_math_cosh",
   "ff_math_tanh",
   "ff_math_pow",
   "ff_math_atan2",
   "ff_math_fmod",
   "ff_math_ldexp",
   "ff_math_frexp",
   "ff_math_modf",
   "vm_trunc_sse",
   "ff_math_min",
   "ff_math_max",
   "ff_string_byte",
   "ff_string_char",
   "fff_newstr",
   "fff_resstr",
   "ff_string_sub",
   "fff_emptystr",
   "ff_string_reverse",
   "ff_string_lower",
   "ff_string_upper",
   "ff_bit_tobit",
   "ff_bit_band",
   "fff_fallback_bit_op",
   "ff_bit_bor",
```

```
  "ff_bit_bxor",
  "ff_bit_bswap",
  "ff_bit_bnot",
  "ff_bit_lshift",
  "ff_bit_rshift",
  "ff_bit_arshift",
  "ff_bit_rol",
  "ff_bit_ror",
  "fff_fallback_2",
  "fff_fallback_1",
  "vm_record",
  "vm_rethook",
  "vm_inshook",
  "cont_hook",
  "vm_hotloop",
  "vm_callhook",
  "vm_hotcall",
  "cont_stitch",
  "vm_profhook",
  "vm_exit_handler",
  "vm_exit_interp",
  "vm_floor",
  "vm_ceil",
  "vm_trunc",
  "vm_mod",
  "vm_powi_sse",
  "vm_cpuid",
  "assert_bad_for_arg_type",
  "vm_ffi_callback",
  "vm_ffi_call@4",
  "BC_MODVN_Z",
  "BC_TGETS_Z",
  "BC_TSETS_Z",
  (const char *)0
};
#line 15 "vm_x86.dasc"
//|.externnames extnames
static const char *const extnames[] = {
  "lj_state_growstack@8",
  "lj_err_throw@8",
  "lj_meta_tget",
  "lj_tab_getinth@8",
  "lj_meta_tset",
  "lj_tab_setinth",
  "lj_meta_comp",
  "lj_meta_equal",
  "lj_meta_equal_cd@8",
  "lj_meta_istype",
  "lj_meta_arith",
  "lj_meta_len@8",
  "lj_meta_call",
  "lj_meta_for@8",
  "lj_tab_get",
  "lj_strfmt_number@8",
  "lj_tab_next",
  "lj_ffh_coroutine_wrap_err@8",
  "log",
  "log10",
  "exp",
  "sin",
  "cos",
  "tan",
  "asin",
  "acos",
  "atan",
  "sinh",
  "cosh",
  "tanh",
  "pow",
  "atan2",
  "fmod",
  "lj_str_new",
  "lj_buf_putstr_reverse@8",
  "lj_buf_tostr@4",
  "lj_buf_putstr_lower@8",
  "lj_buf_putstr_upper@8",
```

```
  "lj_gc_step@4",
  "lj_dispatch_ins@8",
  "lj_trace_hot@8",
  "lj_dispatch_call@8",
  "lj_dispatch_stitch@8",
  "lj_dispatch_profile@8",
  "lj_trace_exit@8",
  "lj_log_trace_direct_exit@8",
  "lj_ccallback_enter@8",
  "lj_ccallback_leave@8",
  "lj_tab_len@4",
  "lj_meta_cat",
  "lj_gc_barrieruv@8",
  "lj_func_closeuv@8",
  "lj_func_newL_gc",
  "lj_tab_new",
  "lj_gc_step_fixtop@4",
  "lj_tab_dup@8",
  "lj_tab_newkey",
  "lj_tab_reasize",
  "lj_log_trace_entry@8",
  (const char *)0
};
#line 16 "vm_x86.dasc"
//|
//|//-----------------------------------------------------------------------
//|
//|.if P64
//|.define X64, 1
//|.if WIN
//|.define X64WIN, 1
//|.endif
//|.endif
//|
//|// Fixed register assignments for the interpreter.
//|// This is very fragile and has many dependencies. Caveat emptor.
//|.define BASE,		edx		// Not C callee-save, refetched anyway.
//|.if not X64
//|.define KBASE,		edi		// Must be C callee-save.
//|.define KBASEa,	KBASE
//|.define PC,		esi		// Must be C callee-save.
//|.define PCa,		PC
//|.define DISPATCH,	ebx		// Must be C callee-save.
//|.elif X64WIN
//|.define KBASE,		edi		// Must be C callee-save.
//|.define KBASEa,	rdi
//|.define PC,		esi		// Must be C callee-save.
//|.define PCa,		rsi
//|.define DISPATCH,	ebx		// Must be C callee-save.
//|.else
//|.define KBASE,		r15d		// Must be C callee-save.
//|.define KBASEa,	r15
//|.define PC,		ebx		// Must be C callee-save.
//|.define PCa,		rbx
//|.define DISPATCH,	r14d		// Must be C callee-save.
//|.endif
//|
//|.define RA,		ecx
//|.define RAH,		ch
//|.define RAL,		cl
//|.define RB,		ebp		// Must be ebp (C callee-save).
//|.define RC,		eax		// Must be eax.
//|.define RCW,		ax
//|.define RCH,		ah
//|.define RCL,		al
//|.define OP,		RB
//|.define RD,		RC
//|.define RDW,		RCW
//|.define RDL,		RCL
//|.if X64
//|.define RAa, rcx
//|.define RBa, rbp
//|.define RCa, rax
//|.define RDa, rax
//|.else
//|.define RAa, RA
```

```
1195 //|.define RBa, RB
1196 //|.define RCa, RC
1197 //|.define RDa, RD
1198 //|.endif
1199 //|
1200 //|.if not X64
1201 //|.define FCARG1,        ecx               // x86 fastcall arguments.
1202 //|.define FCARG2,        edx
1203 //|.elif X64WIN
1204 //|.define CARG1,               rcx               // x64/WIN64 C call arguments.
1205 //|.define CARG2,               rdx
1206 //|.define CARG3,               r8
1207 //|.define CARG4,               r9
1208 //|.define CARG1d,        ecx
1209 //|.define CARG2d,        edx
1210 //|.define CARG3d,        r8d
1211 //|.define CARG4d,        r9d
1212 //|.define FCARG1,        CARG1d               // Upwards compatible to x86 fastcall.
1213 //|.define FCARG2,        CARG2d
1214 //|.else
1215 //|.define CARG1,               rdi               // x64/POSIX C call arguments.
1216 //|.define CARG2,               rsi
1217 //|.define CARG3,               rdx
1218 //|.define CARG4,               rcx
1219 //|.define CARG5,               r8
1220 //|.define CARG6,               r9
1221 //|.define CARG1d,        edi
1222 //|.define CARG2d,        esi
1223 //|.define CARG3d,        edx
1224 //|.define CARG4d,        ecx
1225 //|.define CARG5d,        r8d
1226 //|.define CARG6d,        r9d
1227 //|.define FCARG1,        CARG1d               // Simulate x86 fastcall.
1228 //|.define FCARG2,        CARG2d
1229 //|.endif
1230 //|
1231 //|// Type definitions. Some of these are only used for documentation.
1232 //|.type L,               lua_State
1233 #define Dt1(_V) (int)(ptrdiff_t)&(((lua_State *)0)_V)
1234 #line 106 "vm_x86.dasc"
1235 //|.type GL,              global_State
1236 #define Dt2(_V) (int)(ptrdiff_t)&(((global_State *)0)_V)
1237 #line 107 "vm_x86.dasc"
1238 //|.type TVALUE,              TValue
1239 #define Dt3(_V) (int)(ptrdiff_t)&(((TValue *)0)_V)
1240 #line 108 "vm_x86.dasc"
1241 //|.type GCOBJ,              GCobj
1242 #define Dt4(_V) (int)(ptrdiff_t)&(((GCobj *)0)_V)
1243 #line 109 "vm_x86.dasc"
1244 //|.type STR,               GCstr
1245 #define Dt5(_V) (int)(ptrdiff_t)&(((GCstr *)0)_V)
1246 #line 110 "vm_x86.dasc"
1247 //|.type TAB,               GCtab
1248 #define Dt6(_V) (int)(ptrdiff_t)&(((GCtab *)0)_V)
1249 #line 111 "vm_x86.dasc"
1250 //|.type LFUNC,              GCfuncL
1251 #define Dt7(_V) (int)(ptrdiff_t)&(((GCfuncL *)0)_V)
1252 #line 112 "vm_x86.dasc"
1253 //|.type CFUNC,              GCfuncC
1254 #define Dt8(_V) (int)(ptrdiff_t)&(((GCfuncC *)0)_V)
1255 #line 113 "vm_x86.dasc"
1256 //|.type PROTO,              GCproto
1257 #define Dt9(_V) (int)(ptrdiff_t)&(((GCproto *)0)_V)
1258 #line 114 "vm_x86.dasc"
1259 //|.type UPVAL,              GCupval
1260 #define DtA(_V) (int)(ptrdiff_t)&(((GCupval *)0)_V)
1261 #line 115 "vm_x86.dasc"
1262 //|.type NODE,              Node
1263 #define DtB(_V) (int)(ptrdiff_t)&(((Node *)0)_V)
1264 #line 116 "vm_x86.dasc"
1265 //|.type NARGS,              int
1266 #define DtC(_V) (int)(ptrdiff_t)&(((int *)0)_V)
1267 #line 117 "vm_x86.dasc"
1268 //|.type TRACE,              GCtrace
1269 #define DtD(_V) (int)(ptrdiff_t)&(((GCtrace *)0)_V)
1270 #line 118 "vm_x86.dasc"
```

```
//|.type SBUF,                SBuf
#define DtE(_V) (int)(ptrdiff_t)&(((SBuf *)0)_V)
#line 119 "vm_x86.dasc"
//|
//|// Stack layout while in interpreter. Must match with lj_frame.h.
//|//-----------------------------------------------------------------------
//|.if not X64                // x86 stack layout.
//|
//|.define CFRAME_SPACE,        aword*7                       // Delta for esp (see <--).
//|.macro saveregs_
//|  push edi; push esi; push ebx
//|  sub esp, CFRAME_SPACE
//|.endmacro
//|.macro saveregs
//|  push ebp; saveregs_
//|.endmacro
//|.macro restoreregs
//|  add esp, CFRAME_SPACE
//|  pop ebx; pop esi; pop edi; pop ebp
//|.endmacro
//|
//|.define SAVE_ERRF,        aword [esp+aword*15]        // vm_pcall/vm_cpcall only.
//|.define SAVE_NRES,        aword [esp+aword*14]
//|.define SAVE_CFRAME,        aword [esp+aword*13]
//|.define SAVE_L,        aword [esp+aword*12]
//|//----- 16 byte aligned, ^^^ arguments from C caller
//|.define SAVE_RET,        aword [esp+aword*11]        //<-- esp entering interpreter.
//|.define SAVE_R4,        aword [esp+aword*10]
//|.define SAVE_R3,        aword [esp+aword*9]
//|.define SAVE_R2,        aword [esp+aword*8]
//|//----- 16 byte aligned
//|.define SAVE_R1,        aword [esp+aword*7]        //<-- esp after register saves.
//|.define SAVE_PC,        aword [esp+aword*6]
//|.define TMP2,                aword [esp+aword*5]
//|.define TMP1,                aword [esp+aword*4]
//|//----- 16 byte aligned
//|.define ARG4,                aword [esp+aword*3]
//|.define ARG3,                aword [esp+aword*2]
//|.define ARG2,                aword [esp+aword*1]
//|.define ARG1,                aword [esp]                //<-- esp while in interpreter.
//|//----- 16 byte aligned, ^^^ arguments for C callee
//|
//|// FPARGx overlaps ARGx and ARG(x+1) on x86.
//|.define FPARG3,        qword [esp+qword*1]
//|.define FPARG1,        qword [esp]
//|// TMPQ overlaps TMP1/TMP2. ARG5/MULTRES overlap TMP1/TMP2 (and TMPQ).
//|.define TMPQ,                qword [esp+aword*4]
//|.define TMP3,                ARG4
//|.define ARG5,                TMP1
//|.define TMPa,                TMP1
//|.define MULTRES,        TMP2
//|
//|// Arguments for vm_call and vm_pcall.
//|.define INARG_BASE,        SAVE_CFRAME                // Overwritten by SAVE_CFRAME!
//|
//|// Arguments for vm_cpcall.
//|.define INARG_CP_CALL,        SAVE_ERRF
//|.define INARG_CP_UD,        SAVE_NRES
//|.define INARG_CP_FUNC,        SAVE_CFRAME
//|
//|//-----------------------------------------------------------------------
//|.elif X64WIN                // x64/Windows stack layout
//|
//|.define CFRAME_SPACE,        aword*5                       // Delta for rsp (see <--).
//|.macro saveregs_
//|  push rdi; push rsi; push rbx
//|  sub rsp, CFRAME_SPACE
//|.endmacro
//|.macro saveregs
//|  push rbp; saveregs_
//|.endmacro
//|.macro restoreregs
//|  add rsp, CFRAME_SPACE
//|  pop rbx; pop rsi; pop rdi; pop rbp
//|.endmacro
//|
```

```
//|.define SAVE_CFRAME,        aword [rsp+aword*13]
//|.define SAVE_PC,         dword [rsp+dword*25]
//|.define SAVE_L,         dword [rsp+dword*24]
//|.define SAVE_ERRF,         dword [rsp+dword*23]
//|.define SAVE_NRES,         dword [rsp+dword*22]
//|.define TMP2,                dword [rsp+dword*21]
//|.define TMP1,                dword [rsp+dword*20]
//|//----- 16 byte aligned, ^^^ 32 byte register save area, owned by interpreter
//|.define SAVE_RET,         aword [rsp+aword*9]        //<-- rsp entering interpreter.
//|.define SAVE_R4,         aword [rsp+aword*8]
//|.define SAVE_R3,         aword [rsp+aword*7]
//|.define SAVE_R2,         aword [rsp+aword*6]
//|.define SAVE_R1,         aword [rsp+aword*5]        //<-- rsp after register saves.
//|.define ARG5,                aword [rsp+aword*4]
//|.define CSAVE_4,         aword [rsp+aword*3]
//|.define CSAVE_3,         aword [rsp+aword*2]
//|.define CSAVE_2,         aword [rsp+aword*1]
//|.define CSAVE_1,         aword [rsp]                //<-- rsp while in interpreter.
//|//----- 16 byte aligned, ^^^ 32 byte register save area, owned by callee
//|
//|// TMPQ overlaps TMP1/TMP2. MULTRES overlaps TMP2 (and TMPQ).
//|.define TMPQ,                qword [rsp+aword*10]
//|.define MULTRES,         TMP2
//|.define TMPa,                ARG5
//|.define ARG5d,                dword [rsp+aword*4]
//|.define TMP3,                ARG5d
//|
//|//-----------------------------------------------------------------------
//|.else                        // x64/POSIX stack layout
//|
//|.define CFRAME_SPACE,        aword*5                        // Delta for rsp (see <--).
//|.macro saveregs_
//|  push rbx; push r15; push r14
//|  sub rsp, CFRAME_SPACE
//|.endmacro
//|.macro saveregs
//|  push rbp; saveregs_
//|.endmacro
//|.macro restoreregs
//|  add rsp, CFRAME_SPACE
//|  pop r14; pop r15; pop rbx; pop rbp
//|.endmacro
//|
//|//----- 16 byte aligned,
//|.define SAVE_RET,         aword [rsp+aword*9]        //<-- rsp entering interpreter.
//|.define SAVE_R4,         aword [rsp+aword*8]
//|.define SAVE_R3,         aword [rsp+aword*7]
//|.define SAVE_R2,         aword [rsp+aword*6]
//|.define SAVE_R1,         aword [rsp+aword*5]        //<-- rsp after register saves.
//|.define SAVE_CFRAME,        aword [rsp+aword*4]
//|.define SAVE_PC,         dword [rsp+dword*7]
//|.define SAVE_L,         dword [rsp+dword*6]
//|.define SAVE_ERRF,         dword [rsp+dword*5]
//|.define SAVE_NRES,         dword [rsp+dword*4]
//|.define TMPa,                aword [rsp+aword*1]
//|.define TMP2,                dword [rsp+dword*1]
//|.define TMP1,                dword [rsp]                //<-- rsp while in interpreter.
//|//----- 16 byte aligned
//|
//|// TMPQ overlaps TMP1/TMP2. MULTRES overlaps TMP2 (and TMPQ).
//|.define TMPQ,                qword [rsp]
//|.define TMP3,                dword [rsp+aword*1]
//|.define MULTRES,         TMP2
//|
//|.endif
//|
//|//-----------------------------------------------------------------------
//|
//|// Instruction headers.
//|.macro ins_A; .endmacro
//|.macro ins_AD; .endmacro
//|.macro ins_AJ; .endmacro
//|.macro ins_ABC; movzx RB, RCH; movzx RC, RCL; .endmacro
//|.macro ins_AB_; movzx RB, RCH; .endmacro
//|.macro ins_A_C; movzx RC, RCL; .endmacro
//|.macro ins_AND; not RDa; .endmacro
```

```
//|
//|// Instruction decode+dispatch. Carefully tuned (nope, lodsd is not faster).
//|.macro ins_NEXT
//|  mov RC, [PC]
//|  movzx RA, RCH
//|  movzx OP, RCL
//|  add PC, 4
//|  shr RC, 16
//|.if X64
//|  jmp aword [DISPATCH+OP*8]
//|.else
//|  jmp aword [DISPATCH+OP*4]
//|.endif
//|.endmacro
//|
//|// Instruction footer.
//|.if 1
//|  // Replicated dispatch. Less unpredictable branches, but higher I-Cache use.
//|  .define ins_next, ins_NEXT
//|  .define ins_next_, ins_NEXT
//|.else
//|  // Common dispatch. Lower I-Cache use, only one (very) unpredictable branch.
//|  // Affects only certain kinds of benchmarks (and only with -j off).
//|  // Around 10%-30% slower on Core2, a lot more slower on P4.
//|  .macro ins_next
//|    jmp ->ins_next
//|  .endmacro
//|  .macro ins_next_
//|  ->ins_next:
//|    ins_NEXT
//|  .endmacro
//|.endif
//|
//|// Call decode and dispatch.
//|.macro ins_callt
//|  // BASE = new base, RB = LFUNC, RD = nargs+1, [BASE-4] = PC
//|  mov PC, LFUNC:RB->pc
//|  mov RA, [PC]
//|  movzx OP, RAL
//|  movzx RA, RAH
//|  add PC, 4
//|.if X64
//|  jmp aword [DISPATCH+OP*8]
//|.else
//|  jmp aword [DISPATCH+OP*4]
//|.endif
//|.endmacro
//|
//|.macro ins_call
//|  // BASE = new base, RB = LFUNC, RD = nargs+1
//|  mov [BASE-4], PC
//|  ins_callt
//|.endmacro
//|
//|//-----------------------------------------------------------------------
//|
//|// Macros to test operand types.
//|.macro checktp, reg, tp;  cmp dword [BASE+reg*8+4], tp; .endmacro
//|.macro checknum, reg, target; checktp reg, LJ_TISNUM; jae target; .endmacro
//|.macro checkint, reg, target; checktp reg, LJ_TISNUM; jne target; .endmacro
//|.macro checkstr, reg, target; checktp reg, LJ_TSTR; jne target; .endmacro
//|.macro checktab, reg, target; checktp reg, LJ_TTAB; jne target; .endmacro
//|
//|// These operands must be used with movzx.
//|.define PC_OP, byte [PC-4]
//|.define PC_RA, byte [PC-3]
//|.define PC_RB, byte [PC-1]
//|.define PC_RC, byte [PC-2]
//|.define PC_RD, word [PC-2]
//|
//|.macro branchPC, reg
//|  lea PC, [PC+reg*4-BCBIAS_J*4]
//|.endmacro
//|
//|// Assumes DISPATCH is relative to GL.
#define DISPATCH_GL(field)        (GG_DISP2G + (int)offsetof(global_State, field))
```

```
#define DISPATCH_J(field)        (GG_DISP2J + (int)offsetof(jit_State, field))
//|
#define PC2PROTO(field)  ((int)offsetof(GCproto, field)-(int)sizeof(GCproto))
//|
//|// Decrement hashed hotcount and trigger trace recorder if zero.
//|.macro hotloop, reg
//|  mov reg, PC
//|  shr reg, 1
//|  and reg, HOTCOUNT_PCMASK
//|  sub word [DISPATCH+reg+GG_DISP2HOT], HOTCOUNT_LOOP
//|  jb ->vm_hotloop
//|.endmacro
//|
//|.macro hotcall, reg
//|  mov reg, PC
//|  shr reg, 1
//|  and reg, HOTCOUNT_PCMASK
//|  sub word [DISPATCH+reg+GG_DISP2HOT], HOTCOUNT_CALL
//|  jb ->vm_hotcall
//|.endmacro
//|
//|// Set current VM state.
//|.macro set_vmstate, st
//|  mov dword [DISPATCH+DISPATCH_GL(vmstate)], ~LJ_VMST_..st
//|.endmacro
//|
//|// x87 compares.
//|.macro fcomparepp                      // Compare and pop st0 >< st1.
//|  fucomip st1
//|  fpop
//|.endmacro
//|
//|.macro fpop1; fstp st1; .endmacro
//|
//|// Synthesize SSE FP constants.
//|.macro sseconst_abs, reg, tmp                // Synthesize abs mask.
//|.if X64
//|  mov64 tmp, U64x(7fffffff,ffffffff); movd reg, tmp
//|.else
//|  pxor reg, reg; pcmpeqd reg, reg; psrlq reg, 1
//|.endif
//|.endmacro
//|
//|.macro sseconst_hi, reg, tmp, val        // Synthesize hi-32 bit const.
//|.if X64
//|  mov64 tmp, U64x(val,00000000); movd reg, tmp
//|.else
//|  mov tmp, 0x .. val; movd reg, tmp; pshufd reg, reg, 0x51
//|.endif
//|.endmacro
//|
//|.macro sseconst_sign, reg, tmp               // Synthesize sign mask.
//|  sseconst_hi reg, tmp, 80000000
//|.endmacro
//|.macro sseconst_1, reg, tmp                // Synthesize 1.0.
//|  sseconst_hi reg, tmp, 3ff00000
//|.endmacro
//|.macro sseconst_m1, reg, tmp                // Synthesize -1.0.
//|  sseconst_hi reg, tmp, bff00000
//|.endmacro
//|.macro sseconst_2p52, reg, tmp               // Synthesize 2^52.
//|  sseconst_hi reg, tmp, 43300000
//|.endmacro
//|.macro sseconst_tobit, reg, tmp        // Synthesize 2^52 + 2^51.
//|  sseconst_hi reg, tmp, 43380000
//|.endmacro
//|
//|// Move table write barrier back. Overwrites reg.
//|.macro barrierback, tab, reg
//|  and byte tab->marked, (uint8_t)~LJ_GC_BLACK        // black2gray(tab)
//|  mov reg, [DISPATCH+DISPATCH_GL(gc.grayagain)]
//|  mov [DISPATCH+DISPATCH_GL(gc.grayagain)], tab
//|  mov tab->gclist, reg
//|.endmacro
//|
//|//-----------------------------------------------------------------------
```

```
1575
1576 /* Generate subroutines used by opcodes and other parts of the VM. */
1577 /* The .code_sub section should be last to help static branch prediction. */
1578 static void build_subroutines(BuildCtx *ctx)
1579 {
1580   //|.code_sub
1581   dasm_put(Dst, 0);
1582 #line 426 "vm_x86.dasc"
1583   //|
1584   //|//-----------------------------------------------------------------------
1585   //|//-- Return handling ----------------------------------------------------
1586   //|//-----------------------------------------------------------------------
1587   //|
1588   //|->vm_returnp:
1589   //|  test PC, FRAME_P
1590   //|  jz ->cont_dispatch
1591   //|
1592   //|  // Return from pcall or xpcall fast func.
1593   //|  and PC, -8
1594   //|  sub BASE, PC                        // Restore caller base.
1595   //|  lea RAa, [RA+PC-8]                     // Rebase RA and prepend one result.
1596   //|  mov PC, [BASE-4]                      // Fetch PC of previous frame.
1597   //|  // Prepending may overwrite the pcall frame, so do it at the end.
1598   //|  mov dword [BASE+RA+4], LJ_TTRUE        // Prepend true to results.
1599   //|
1600   //|->vm_returnc:
1601   //|  add RD, 1                              // RD = nresults+1
1602   //|  jz ->vm_unwind_yield
1603   //|  mov MULTRES, RD
1604   //|  test PC, FRAME_TYPE
1605   //|  jz ->BC_RET_Z                       // Handle regular return to Lua.
1606   //|
1607   //|->vm_return:
1608   //|  // BASE = base, RA = resultofs, RD = nresults+1 (= MULTRES), PC = return
1609   //|  xor PC, FRAME_C
1610   //|  test PC, FRAME_TYPE
1611   //|  jnz ->vm_returnp
1612   //|
1613   //|  // Return to C.
1614   //|  set_vmstate C
1615   //|  and PC, -8
1616   //|  sub PC, BASE
1617   //|  neg PC                                 // Previous base = BASE - delta.
1618   //|
1619   //|  sub RD, 1
1620   //|  jz >2
1621   //|1:  // Move results down.
1622   //|.if X64
1623   //|  mov RBa, [BASE+RA]
1624   dasm_put(Dst, 2, FRAME_P, LJ_TTRUE, FRAME_TYPE, FRAME_C, FRAME_TYPE, DISPATCH_GL(vmstate), ~LJ_VMST_C);
1625 #line 467 "vm_x86.dasc"
1626   //|  mov [BASE-8], RBa
1627   //|.else
1628   //|  mov RB, [BASE+RA]
1629   //|  mov [BASE-8], RB
1630   //|  mov RB, [BASE+RA+4]
1631   //|  mov [BASE-4], RB
1632   //|.endif
1633   //|  add BASE, 8
1634   //|  sub RD, 1
1635   //|  jnz <1
1636   //|2:
1637   //|  mov L:RB, SAVE_L
1638   //|  mov L:RB->base, PC
1639   //|3:
1640   //|  mov RD, MULTRES
1641   //|  mov RA, SAVE_NRES                      // RA = wanted nresults+1
1642   //|4:
1643   //|  cmp RA, RD
1644   //|  jne >6                                 // More/less results wanted?
1645   //|5:
1646   //|  sub BASE, 8
1647   //|  mov L:RB->top, BASE
1648   //|
1649   //|->vm_leave_cp:
1650   //|  mov RAa, SAVE_CFRAME                // Restore previous C frame.
```

```
1651   //|  mov L:RB->cframe, RAa
1652   //|  xor eax, eax                        // Ok return status for vm_pcall.
1653   //|
1654   //|->vm_leave_unw:
1655   //|  restoreregs
1656   //|  ret
1657   //|
1658   //|6:
1659   //|  jb >7                               // Less results wanted?
1660   //|  // More results wanted. Check stack size and fill up results with nil.
1661   //|  cmp BASE, L:RB->maxstack
1662   //|  ja >8
1663   //|  mov dword [BASE-4], LJ_TNIL
1664   //|  add BASE, 8
1665   dasm_put(Dst, 92, Dt1(->base), Dt1(->top), Dt1(->cframe), Dt1(->maxstack), LJ_TNIL);
1666 #line 506 "vm_x86.dasc"
1667   //|  add RD, 1
1668   //|  jmp <4
1669   //|
1670   //|7:  // Less results wanted.
1671   //|  test RA, RA
1672   //|  jz <5                               // But check for LUA_MULTRET+1.
1673   //|  sub RA, RD                               // Negative result!
1674   //|  lea BASE, [BASE+RA*8]               // Correct top.
1675   //|  jmp <5
1676   //|
1677   //|8:  // Corner case: need to grow stack for filling up results.
1678   //|  // This can happen if:
1679   //|  // - A C function grows the stack (a lot).
1680   //|  // - The GC shrinks the stack in between.
1681   //|  // - A return back from a lua_call() with (high) nresults adjustment.
1682   //|  mov L:RB->top, BASE                 // Save current top held in BASE (yes).
1683   //|  mov MULTRES, RD                          // Need to fill only remainder with nil.
1684   //|  mov FCARG2, RA
1685   //|  mov FCARG1, L:RB
1686   //|  call extern lj_state_growstack@8         // (lua_State *L, int n)
1687   //|  mov BASE, L:RB->top                 // Need the (realloced) L->top in BASE.
1688   //|  jmp <3
1689   //|
1690   //|->vm_unwind_yield:
1691   //|  mov al, LUA_YIELD
1692   //|  jmp ->vm_unwind_c_eh
1693   //|
1694   //|->vm_unwind_c@8:                        // Unwind C stack, return from vm_pcall.
1695   //|  // (void *cframe, int errcode)
1696   //|.if X64
1697   //|  mov eax, CARG2d                         // Error return status for vm_pcall.
1698   //|  mov rsp, CARG1
1699   //|.else
1700   //|  mov eax, FCARG2                         // Error return status for vm_pcall.
1701   //|  mov esp, FCARG1
1702   //|.endif
1703   //|->vm_unwind_c_eh:                        // Landing pad for external unwinder.
1704   //|  mov L:RB, SAVE_L
1705   //|  mov GL:RB, L:RB->glref
1706   //|  mov dword GL:RB->vmstate, ~LJ_VMST_C
1707   //|  jmp ->vm_leave_unw
1708   dasm_put(Dst, 192, Dt1(->top), Dt1(->top), LUA_YIELD, Dt1(->glref), Dt2(->vmstate), ~LJ_VMST_C);
1709 #line 547 "vm_x86.dasc"
1710   //|
1711   //|->vm_unwind_rethrow:
1712   //|.if X64 and not X64WIN
1713   //|  mov FCARG1, SAVE_L
1714   //|  mov FCARG2, eax
1715   //|  restoreregs
1716   //|  jmp extern lj_err_throw@8              // (lua_State *L, int errcode)
1717   //|.endif
1718   //|
1719   //|->vm_unwind_ff@4:                        // Unwind C stack, return from ff pcall.
1720   //|  // (void *cframe)
1721   //|.if X64
1722   //|  and CARG1, CFRAME_RAWMASK
1723   //|  mov rsp, CARG1
1724   //|.else
1725   //|  and FCARG1, CFRAME_RAWMASK
1726   //|  mov esp, FCARG1
```

```
1727    //|.endif
1728    //|->vm_unwind_ff_eh:                       // Landing pad for external unwinder.
1729    //|  mov L:RB, SAVE_L
1730    //|  mov RAa, -8                          // Results start at BASE+RA = BASE-8.
1731    //|  mov RD, 1+1                          // Really 1+2 results, incr. later.
1732    //|  mov BASE, L:RB->base
1733    //|  mov DISPATCH, L:RB->glref               // Setup pointer to dispatch table.
1734    //|  add DISPATCH, GG_G2DISP
1735    //|  mov PC, [BASE-4]                       // Fetch PC of previous frame.
1736    //|  mov dword [BASE-4], LJ_TFALSE        // Prepend false to error message.
1737    //|  set_vmstate INTERP
1738    //|  jmp ->vm_returnc                       // Increments RD/MULTRES and returns.
1739    //|
1740    //|//-----------------------------------------------------------------------
1741    //|//-- Grow stack for calls -----------------------------------------------
1742    //|//-----------------------------------------------------------------------
1743    //|
1744    //|->vm_growstack_c:                        // Grow stack for C function.
1745    //|  mov FCARG2, LUA_MINSTACK
1746    //|  jmp >2
1747    //|
1748    //|->vm_growstack_v:                        // Grow stack for vararg Lua function.
1749    //|  sub RD, 8
1750    dasm_put(Dst, 275, CFRAME_RAWMASK, 1+1, Dt1(->base), Dt1(->glref), GG_G2DISP, LJ_TFALSE,
DISPATCH_GL(vmstate), ~LJ_VMST_INTERP, LUA_MINSTACK);
1751  #line 587 "vm_x86.dasc"
1752    //|  jmp >1
1753    //|
1754    //|->vm_growstack_f:                        // Grow stack for fixarg Lua function.
1755    //|  // BASE = new base, RD = nargs+1, RB = L, PC = first PC
1756    //|  lea RD, [BASE+NARGS:RD*8-8]
1757    //|1:
1758    //|  movzx RA, byte [PC-4+PC2PROTO(framesize)]
1759    //|  add PC, 4                               // Must point after first instruction.
1760    //|  mov L:RB->base, BASE
1761    //|  mov L:RB->top, RD
1762    //|  mov SAVE_PC, PC
1763    //|  mov FCARG2, RA
1764    //|2:
1765    //|  // RB = L, L->base = new base, L->top = top
1766    //|  mov FCARG1, L:RB
1767    //|  call extern lj_state_growstack@8        // (lua_State *L, int n)
1768    //|  mov BASE, L:RB->base
1769    //|  mov RD, L:RB->top
1770    //|  mov LFUNC:RB, [BASE-8]
1771    //|  sub RD, BASE
1772    //|  shr RD, 3
1773    //|  add NARGS:RD, 1
1774    //|  // BASE = new base, RB = LFUNC, RD = nargs+1
1775    //|  ins_callt                               // Just retry the call.
1776    //|
1777    //|//-----------------------------------------------------------------------
1778    //|//-- Entry points into the assembler VM ---------------------------------
1779    //|//-----------------------------------------------------------------------
1780    //|
1781    //|->vm_resume:                              // Setup C frame and resume thread.
1782    //|  // (lua_State *L, TValue *base, int nres1 = 0, ptrdiff_t ef = 0)
1783    //|  saveregs
1784    //|.if X64
1785    //|  mov L:RB, CARG1d                        // Caveat: CARG1d may be RA.
1786    //|  mov SAVE_L, CARG1d
1787    //|  mov RA, CARG2d
1788    //|.else
1789    //|  mov L:RB, SAVE_L
1790    //|  mov RA, INARG_BASE                        // Caveat: overlaps SAVE_CFRAME!
1791    //|.endif
1792    //|  mov PC, FRAME_CP
1793    //|  xor RD, RD
1794    //|  lea KBASEa, [esp+CFRAME_RESUME]
1795    //|  mov DISPATCH, L:RB->glref               // Setup pointer to dispatch table.
1796    //|  add DISPATCH, GG_G2DISP
1797    //|  mov SAVE_PC, RD                        // Any value outside of bytecode is ok.
1798    //|  mov SAVE_CFRAME, RDa
1799    //|.if X64
1800    //|  mov SAVE_NRES, RD
1801    //|  mov SAVE_ERRF, RD
```

```
1802    //|.endif
1803    //|  mov L:RB->cframe, KBASEa
1804    //|  cmp byte L:RB->status, RDL
1805    //|  je >2                                // Initial resume (like a call).
1806    //|
1807    //|  // Resume after yield (like a return).
1808    //|  mov [DISPATCH+DISPATCH_GL(cur_L)], L:RB
1809    dasm_put(Dst, 371, -4+PC2PROTO(framesize), Dt1(->base), Dt1(->top), Dt1(->base), Dt1(->top), Dt7(->pc),
FRAME_CP, CFRAME_RESUME, Dt1(->glref), GG_G2DISP, Dt1(->cframe), Dt1(->status));
1810  #line 644 "vm_x86.dasc"
1811    //|  set_vmstate INTERP
1812    //|  mov byte L:RB->status, RDL
1813    //|  mov BASE, L:RB->base
1814    //|  mov RD, L:RB->top
1815    //|  sub RD, RA
1816    //|  shr RD, 3
1817    //|  add RD, 1                                // RD = nresults+1
1818    //|  sub RA, BASE                          // RA = resultofs
1819    //|  mov PC, [BASE-4]
1820    //|  mov MULTRES, RD
1821    //|  test PC, FRAME_TYPE
1822    //|  jz ->BC_RET_Z
1823    //|  jmp ->vm_return
1824    //|
1825    //|->vm_pcall:                               // Setup protected C frame and enter VM.
1826    //|  // (lua_State *L, TValue *base, int nres1, ptrdiff_t ef)
1827    //|  saveregs
1828    //|  mov PC, FRAME_CP
1829    //|.if X64
1830    //|  mov SAVE_ERRF, CARG4d
1831    //|.endif
1832    //|  jmp >1
1833    //|
1834    //|->vm_call:                                // Setup C frame and enter VM.
1835    //|  // (lua_State *L, TValue *base, int nres1)
1836    //|  saveregs
1837    //|  mov PC, FRAME_C
1838    //|
1839    //|1:  // Entry point for vm_pcall above (PC = ftype).
1840    //|.if X64
1841    //|  mov SAVE_NRES, CARG3d
1842    //|  mov L:RB, CARG1d                       // Caveat: CARG1d may be RA.
1843    //|  mov SAVE_L, CARG1d
1844    //|  mov RA, CARG2d
1845    //|.else
1846    //|  mov L:RB, SAVE_L
1847    //|  mov RA, INARG_BASE                       // Caveat: overlaps SAVE_CFRAME!
1848    //|.endif
1849    //|
1850    //|  mov DISPATCH, L:RB->glref                // Setup pointer to dispatch table.
1851    //|  mov KBASEa, L:RB->cframe                // Add our C frame to cframe chain.
1852    //|  mov SAVE_CFRAME, KBASEa
1853    dasm_put(Dst, 524, DISPATCH_GL(cur_L), DISPATCH_GL(vmstate), ~LJ_VMST_INTERP, Dt1(->status), Dt1(->base),
Dt1(->top), FRAME_TYPE, FRAME_CP, FRAME_C, Dt1(->glref), Dt1(->cframe));
1854  #line 686 "vm_x86.dasc"
1855    //|  mov SAVE_PC, L:RB                        // Any value outside of bytecode is ok.
1856    //|  add DISPATCH, GG_G2DISP
1857    //|.if X64
1858    //|  mov L:RB->cframe, rsp
1859    //|.else
1860    //|  mov L:RB->cframe, esp
1861    //|.endif
1862    //|
1863    //|2:  // Entry point for vm_resume/vm_cpcall (RA = base, RB = L, PC = ftype).
1864    //|  mov [DISPATCH+DISPATCH_GL(cur_L)], L:RB
1865    //|  set_vmstate INTERP
1866    //|  mov BASE, L:RB->base                  // BASE = old base (used in vmeta_call).
1867    //|  add PC, RA
1868    //|  sub PC, BASE                          // PC = frame delta + frame type
1869    //|
1870    //|  mov RD, L:RB->top
1871    //|  sub RD, RA
1872    //|  shr NARGS:RD, 3
1873    //|  add NARGS:RD, 1                          // RD = nargs+1
1874    //|
1875    //|->vm_call_dispatch:
```

```
1876    //|  mov LFUNC:RB, [RA-8]
1877    //|  cmp dword [RA-4], LJ_TFUNC
1878    //|  jne ->vmeta_call                          // Ensure KBASE defined and != BASE.
1879    //|
1880    //|->vm_call_dispatch_f:
1881    //|  mov BASE, RA
1882    //|  ins_call
1883    //|  // BASE = new base, RB = func, RD = nargs+1, PC = caller PC
1884    //|
1885    //|->vm_cpcall:                                // Setup protected C frame, call C.
1886    //|  // (lua_State *L, lua_CFunction func, void *ud, lua_CPFunction cp)
1887    //|  saveregs
1888    //|.if X64
1889    //|  mov L:RB, CARG1d                          // Caveat: CARG1d may be RA.
1890    //|  mov SAVE_L, CARG1d
1891    //|.else
1892    //|  mov L:RB, SAVE_L
1893    //|  // Caveat: INARG_CP_* and SAVE_CFRAME/SAVE_NRES/SAVE_ERRF overlap!
1894    //|  mov RC, INARG_CP_UD                  // Get args before they are overwritten.
1895    //|  mov RA, INARG_CP_FUNC
1896    //|  mov BASE, INARG_CP_CALL
1897    //|.endif
1898    //|  mov SAVE_PC, L:RB                          // Any value outside of bytecode is ok.
1899    //|
1900    //|  mov KBASE, L:RB->stack                // Compute -savestack(L, L->top).
1901    //|  sub KBASE, L:RB->top
1902    //|   mov DISPATCH, L:RB->glref                // Setup pointer to dispatch table.
1903    //|  mov SAVE_ERRF, 0                         // No error function.
1904    //|  mov SAVE_NRES, KBASE                 // Neg. delta means cframe w/o frame.
1905    //|   add DISPATCH, GG_G2DISP
1906    //|  // Handler may change cframe_nres(L->cframe) or cframe_errfunc(L->cframe).
1907    //|
1908    //|.if X64
1909    //|  mov KBASEa, L:RB->cframe                  // Add our C frame to cframe chain.
1910    //|  mov SAVE_CFRAME, KBASEa
1911    dasm_put(Dst, 635, GG_G2DISP, Dt1(->cframe), DISPATCH_GL(cur_L), DISPATCH_GL(vmstate), ~LJ_VMST_INTERP,
Dt1(->base), Dt1(->top), LJ_TFUNC, Dt7(->pc), Dt1(->stack), Dt1(->top), Dt1(->glref), GG_G2DISP, Dt1(->cframe));
1912  #line 742 "vm_x86.dasc"
1913    //|  mov L:RB->cframe, rsp
1914    //|  mov [DISPATCH+DISPATCH_GL(cur_L)], L:RB
1915    //|
1916    //|  call CARG4                          // (lua_State *L, lua_CFunction func, void *ud)
1917    //|.else
1918    //|  mov ARG3, RC                          // Have to copy args downwards.
1919    //|  mov ARG2, RA
1920    //|  mov ARG1, L:RB
1921    //|
1922    //|  mov KBASE, L:RB->cframe                  // Add our C frame to cframe chain.
1923    //|  mov SAVE_CFRAME, KBASE
1924    //|  mov L:RB->cframe, esp
1925    //|  mov [DISPATCH+DISPATCH_GL(cur_L)], L:RB
1926    //|
1927    //|  call BASE                           // (lua_State *L, lua_CFunction func, void *ud)
1928    //|.endif
1929    //|  // TValue * (new base) or NULL returned in eax (RC).
1930    //|  test RC, RC
1931    //|  jz ->vm_leave_cp                          // No base? Just remove C frame.
1932    //|  mov RA, RC
1933    //|  mov PC, FRAME_CP
1934    //|  jmp <2                                 // Else continue with the call.
1935    //|
1936    //|//-----------------------------------------------------------------------
1937    //|//-- Metamethod handling -------------------------------------------------
1938    //|//-----------------------------------------------------------------------
1939    //|
1940    //|//-- Continuation dispatch ----------------------------------------------
1941    //|
1942    //|->cont_dispatch:
1943    //|  // BASE = meta base, RA = resultofs, RD = nresults+1 (also in MULTRES)
1944    //|  add RA, BASE
1945    //|  and PC, -8
1946    //|  mov RB, BASE
1947    //|  sub BASE, PC                          // Restore caller BASE.
1948    //|  mov dword [RA+RD*8-4], LJ_TNIL        // Ensure one valid arg.
1949    //|  mov RC, RA                                // ... in [RC]
1950    //|  mov PC, [RB-12]                        // Restore PC from [cont|PC].
```

```
  //|.if X64
  //|  movsxd RAa, dword [RB-16]               // May be negative on WIN64 with debug.
  //|.if FFI
  //|  cmp RA, 1
  //|  jbe >1
  //|.endif
  //|  lea KBASEa, qword [=>0]
  //|  add RAa, KBASEa
  //|.else
  //|  mov RA, dword [RB-16]
  //|.if FFI
  //|  cmp RA, 1
  //|  jbe >1
  //|.endif
  //|.endif
  //|  mov LFUNC:KBASE, [BASE-8]
  //|  mov KBASE, LFUNC:KBASE->pc
  //|  mov KBASE, [KBASE+PC2PROTO(k)]
  //|  // BASE = base, RC = result, RB = meta base
  //|  jmp RAa                                 // Jump to continuation.
  //|
  //|.if FFI
  //|1:
  //|  je ->cont_ffi_callback                // cont = 1: return from FFI callback.
  //|  // cont = 0: Tail call from C function.
  //|  sub RB, BASE
  //|  shr RB, 3
  //|  lea RD, [RB-1]
  //|  jmp ->vm_call_tail
  //|.endif
  //|
  //|->cont_cat:                               // BASE = base, RC = result, RB = mbase
  //|  movzx RA, PC_RB
  dasm_put(Dst, 784, Dt1(->cframe), DISPATCH_GL(cur_L), FRAME_CP, LJ_TNIL, 0, Dt7(->pc), PC2PROTO(k));
#line 813 "vm_x86.dasc"
  //|  sub RB, 16
  //|  lea RA, [BASE+RA*8]
  //|  sub RA, RB
  //|  je ->cont_ra
  //|  neg RA
  //|  shr RA, 3
  //|.if X64WIN
  //|  mov CARG3d, RA
  //|  mov L:CARG1d, SAVE_L
  //|  mov L:CARG1d->base, BASE
  //|  mov RCa, [RC]
  //|  mov [RB], RCa
  //|  mov CARG2d, RB
  //|.elif X64
  //|  mov L:CARG1d, SAVE_L
  //|  mov L:CARG1d->base, BASE
  //|  mov CARG3d, RA
  //|  mov RAa, [RC]
  //|  mov [RB], RAa
  //|  mov CARG2d, RB
  //|.else
  //|  mov ARG3, RA
  //|  mov RA, [RC+4]
  //|  mov RC, [RC]
  //|  mov [RB+4], RA
  //|  mov [RB], RC
  //|  mov ARG2, RB
  //|.endif
  //|  jmp ->BC_CAT_Z
  //|
  //|//-- Table indexing metamethods ------------------------------------------
  //|
  //|->vmeta_tgets:
  //|  mov TMP1, RC                        // RC = GCstr *
  //|  mov TMP2, LJ_TSTR
  //|  lea RCa, TMP1                        // Store temp. TValue in TMP1/TMP2.
  //|  cmp PC_OP, BC_GGET
  //|  jne >1
  //|  lea RA, [DISPATCH+DISPATCH_GL(tmptv)]  // Store fn->l.env in g->tmptv.
  //|  mov [RA], TAB:RB                       // RB = GCtab *
  //|  mov dword [RA+4], LJ_TTAB
```

```
  //|  mov RB, RA
  //|  jmp >2
  //|
  //|->vmeta_tgetb:
  //|  movzx RC, PC_RC
  //|.if DUALNUM
  //|  mov TMP2, LJ_TISNUM
  //|  mov TMP1, RC
  //|.else
  //|  cvtsi2sd xmm0, RC
  //|  movsd TMPQ, xmm0
  //|.endif
  //|  lea RCa, TMPQ                        // Store temp. TValue in TMPQ.
  //|  jmp >1
  //|
  //|->vmeta_tgetv:
  //|  movzx RC, PC_RC                      // Reload TValue *k from RC.
  //|  lea RC, [BASE+RC*8]
  //|1:
  //|  movzx RB, PC_RB                      // Reload TValue *t from RB.
  dasm_put(Dst, 898, Dt1(->base), LJ_TSTR, BC_GGET, DISPATCH_GL(tmptv), LJ_TTAB, LJ_TISNUM);
#line 874 "vm_x86.dasc"
  //|  lea RB, [BASE+RB*8]
  //|2:
  //|.if X64
  //|  mov L:CARG1d, SAVE_L
  //|  mov L:CARG1d->base, BASE             // Caveat: CARG2d/CARG3d may be BASE.
  //|  mov CARG2d, RB
  //|  mov CARG3, RCa                       // May be 64 bit ptr to stack.
  //|  mov L:RB, L:CARG1d
  //|.else
  //|  mov ARG2, RB
  //|  mov L:RB, SAVE_L
  //|  mov ARG3, RC
  //|  mov ARG1, L:RB
  //|  mov L:RB->base, BASE
  //|.endif
  //|  mov SAVE_PC, PC
  //|  call extern lj_meta_tget             // (lua_State *L, TValue *o, TValue *k)
  //|  // TValue * (finished) or NULL (metamethod) returned in eax (RC).
  //|  mov BASE, L:RB->base
  //|  test RC, RC
  //|  jz >3
  //|->cont_ra:                             // BASE = base, RC = result
  //|  movzx RA, PC_RA
  //|.if X64
  //|  mov RBa, [RC]
  //|  mov [BASE+RA*8], RBa
  //|.else
  //|  mov RB, [RC+4]
  //|  mov RC, [RC]
  //|  mov [BASE+RA*8+4], RB
  //|  mov [BASE+RA*8], RC
  //|.endif
  //|  ins_next
  //|
  //|3:  // Call __index metamethod.
  //|  // BASE = base, L->top = new base, stack = cont/func/t/k
  //|  mov RA, L:RB->top
  //|  mov [RA-12], PC                      // [cont|PC]
  //|  lea PC, [RA+FRAME_CONT]
  //|  sub PC, BASE
  //|  mov LFUNC:RB, [RA-8]                 // Guaranteed to be a function here.
  //|  mov NARGS:RD, 2+1                    // 2 args for func(t, k).
  //|  jmp ->vm_call_dispatch_f
  //|
  //|->vmeta_tgetr:
  //|  mov FCARG1, TAB:RB
  //|  mov RB, BASE                         // Save BASE.
  //|  mov FCARG2, RC                       // Caveat: FCARG2 == BASE
  //|  call extern lj_tab_getinth@8         // (GCtab *t, int32_t key)
  //|  // cTValue * or NULL returned in eax (RC).
  //|  movzx RA, PC_RA
  //|  mov BASE, RB                         // Restore BASE.
  //|  test RC, RC
  //|  jnz ->BC_TGETR_Z
```

```
  //|  mov dword [BASE+RA*8+4], LJ_TNIL
  //|  jmp ->BC_TGETR2_Z
  //|
  //|//-----------------------------------------------------------------------
  //|
  //|->vmeta_tsets:
  //|  mov TMP1, RC                          // RC = GCstr *
  //|  mov TMP2, LJ_TSTR
  //|  lea RCa, TMP1                         // Store temp. TValue in TMP1/TMP2.
  dasm_put(Dst, 1022, Dt1(->base), Dt1(->base), Dt1(->top), FRAME_CONT, 2+1, LJ_TNIL, LJ_TSTR);
#line 937 "vm_x86.dasc"
  //|  cmp PC_OP, BC_GSET
  //|  jne >1
  //|  lea RA, [DISPATCH+DISPATCH_GL(tmptv)]  // Store fn->l.env in g->tmptv.
  //|  mov [RA], TAB:RB                       // RB = GCtab *
  //|  mov dword [RA+4], LJ_TTAB
  //|  mov RB, RA
  //|  jmp >2
  //|
  //|->vmeta_tsetb:
  //|  movzx RC, PC_RC
  //|.if DUALNUM
  //|  mov TMP2, LJ_TISNUM
  //|  mov TMP1, RC
  //|.else
  //|  cvtsi2sd xmm0, RC
  //|  movsd TMPQ, xmm0
  //|.endif
  //|  lea RCa, TMPQ                         // Store temp. TValue in TMPQ.
  //|  jmp >1
  //|
  //|->vmeta_tsetv:
  //|  movzx RC, PC_RC                       // Reload TValue *k from RC.
  //|  lea RC, [BASE+RC*8]
  //|1:
  //|  movzx RB, PC_RB                       // Reload TValue *t from RB.
  //|  lea RB, [BASE+RB*8]
  //|2:
  //|.if X64
  //|  mov L:CARG1d, SAVE_L
  //|  mov L:CARG1d->base, BASE              // Caveat: CARG2d/CARG3d may be BASE.
  //|  mov CARG2d, RB
  //|  mov CARG3, RCa                        // May be 64 bit ptr to stack.
  //|  mov L:RB, L:CARG1d
  //|.else
  //|  mov ARG2, RB
  //|  mov L:RB, SAVE_L
  //|  mov ARG3, RC
  //|  mov ARG1, L:RB
  //|  mov L:RB->base, BASE
  //|.endif
  //|  mov SAVE_PC, PC
  //|  call extern lj_meta_tset               // (lua_State *L, TValue *o, TValue *k)
  //|  // TValue * (finished) or NULL (metamethod) returned in eax (RC).
  //|  mov BASE, L:RB->base
  //|  test RC, RC
  //|  jz >3
  //|  // NOBARRIER: lj_meta_tset ensures the table is not black.
  //|  movzx RA, PC_RA
  //|.if X64
  //|  mov RBa, [BASE+RA*8]
  //|  mov [RC], RBa
  //|.else
  //|  mov RB, [BASE+RA*8+4]
  //|  mov RA, [BASE+RA*8]
  //|  mov [RC+4], RB
  //|  mov [RC], RA
  //|.endif
  //|->cont_nop:                             // BASE = base, (RC = result)
  //|  ins_next
  //|
  //|3:  // Call __newindex metamethod.
  //|  // BASE = base, L->top = new base, stack = cont/func/t/k/(v)
  //|  mov RA, L:RB->top
  dasm_put(Dst, 1171, BC_GSET, DISPATCH_GL(tmptv), LJ_TTAB, LJ_TISNUM, Dt1(->base), Dt1(->base));
#line 1000 "vm_x86.dasc"
```

```
  //|  mov [RA-12], PC                        // [cont|PC]
  //|  movzx RC, PC_RA
  //|  // Copy value to third argument.
  //|.if X64
  //|  mov RBa, [BASE+RC*8]
  //|  mov [RA+16], RBa
  //|.else
  //|  mov RB, [BASE+RC*8+4]
  //|  mov RC, [BASE+RC*8]
  //|  mov [RA+20], RB
  //|  mov [RA+16], RC
  //|.endif
  //|  lea PC, [RA+FRAME_CONT]
  //|  sub PC, BASE
  //|  mov LFUNC:RB, [RA-8]                // Guaranteed to be a function here.
  //|  mov NARGS:RD, 3+1                       // 3 args for func(t, k, v).
  //|  jmp ->vm_call_dispatch_f
  //|
  //|->vmeta_tsetr:
  //|.if X64WIN
  //|  mov L:CARG1d, SAVE_L
  //|  mov CARG3d, RC
  //|  mov L:CARG1d->base, BASE
  //|  xchg CARG2d, TAB:RB                // Caveat: CARG2d == BASE.
  //|.elif X64
  //|  mov L:CARG1d, SAVE_L
  //|  mov CARG2d, TAB:RB
  //|  mov L:CARG1d->base, BASE
  //|  mov RB, BASE                          // Save BASE.
  //|  mov CARG3d, RC                          // Caveat: CARG3d == BASE.
  //|.else
  //|  mov L:RA, SAVE_L
  //|  mov ARG2, TAB:RB
  //|  mov RB, BASE                          // Save BASE.
  //|  mov ARG3, RC
  //|  mov ARG1, L:RA
  //|  mov L:RA->base, BASE
  //|.endif
  //|  mov SAVE_PC, PC
  //|  call extern lj_tab_setinth  // (lua_State *L, GCtab *t, int32_t key)
  //|  // TValue * returned in eax (RC).
  //|  movzx RA, PC_RA
  //|  mov BASE, RB                          // Restore BASE.
  //|  jmp ->BC_TSETR_Z
  //|
  //|//-- Comparison metamethods ---------------------------------------------
  //|
  //|->vmeta_comp:
  //|.if X64
  //|  mov L:RB, SAVE_L
  //|  mov L:RB->base, BASE                // Caveat: CARG2d/CARG3d == BASE.
  //|.if X64WIN
  //|  lea CARG3d, [BASE+RD*8]
  //|  lea CARG2d, [BASE+RA*8]
  //|.else
  //|  lea CARG2d, [BASE+RA*8]
  //|  lea CARG3d, [BASE+RD*8]
  //|.endif
  //|  mov CARG1d, L:RB                        // Caveat: CARG1d/CARG4d == RA.
  //|  movzx CARG4d, PC_OP
  //|.else
  //|  movzx RB, PC_OP
  //|  lea RD, [BASE+RD*8]
  //|  lea RA, [BASE+RA*8]
  //|  mov ARG4, RB
  //|  mov L:RB, SAVE_L
  //|  mov ARG3, RD
  //|  mov ARG2, RA
  //|  mov ARG1, L:RB
  //|  mov L:RB->base, BASE
  //|.endif
  //|  mov SAVE_PC, PC
  //|  call extern lj_meta_comp        // (lua_State *L, TValue *o1, *o2, int op)
  //|  // 0/1 or TValue * (metamethod) returned in eax (RC).
  //|3:
  //|  mov BASE, L:RB->base
```

```
2255   //|  cmp RC, 1
2256   //|  ja ->vmeta_binop
2257   //|4:
2258   //|  lea PC, [PC+4]
2259   //|  jb >6
2260   //|5:
2261   //|  movzx RD, PC_RD
2262   //|  branchPC RD
2263   //|6:
2264   //|  ins_next
2265   dasm_put(Dst, 1316, Dt1(->top), FRAME_CONT, 3+1, Dt1(->base), Dt1(->base), Dt1(->base), -BCBIAS_J*4);
2266 #line 1086 "vm_x86.dasc"
2267   //|
2268   //|->cont_condt:                          // BASE = base, RC = result
2269   //|  add PC, 4
2270   //|  cmp dword [RC+4], LJ_TISTRUECOND       // Branch if result is true.
2271   //|  jb <5
2272   //|  jmp <6
2273   //|
2274   //|->cont_condf:                          // BASE = base, RC = result
2275   //|  cmp dword [RC+4], LJ_TISTRUECOND       // Branch if result is false.
2276   //|  jmp <4
2277   //|
2278   //|->vmeta_equal:
2279   //|  sub PC, 4
2280   //|.if X64WIN
2281   //|  mov CARG3d, RD
2282   //|  mov CARG4d, RB
2283   //|  mov L:RB, SAVE_L
2284   //|  mov L:RB->base, BASE               // Caveat: CARG2d == BASE.
2285   //|  mov CARG2d, RA
2286   //|  mov CARG1d, L:RB                      // Caveat: CARG1d == RA.
2287   //|.elif X64
2288   //|  mov CARG2d, RA
2289   //|  mov CARG4d, RB                       // Caveat: CARG4d == RA.
2290   //|  mov L:RB, SAVE_L
2291   //|  mov L:RB->base, BASE               // Caveat: CARG3d == BASE.
2292   //|  mov CARG3d, RD
2293   //|  mov CARG1d, L:RB
2294   //|.else
2295   //|  mov ARG4, RB
2296   //|  mov L:RB, SAVE_L
2297   //|  mov ARG3, RD
2298   //|  mov ARG2, RA
2299   //|  mov ARG1, L:RB
2300   //|  mov L:RB->base, BASE
2301   //|.endif
2302   //|  mov SAVE_PC, PC
2303   //|  call extern lj_meta_equal        // (lua_State *L, GCobj *o1, *o2, int ne)
2304   //|  // 0/1 or TValue * (metamethod) returned in eax (RC).
2305   //|  jmp <3
2306   //|
2307   //|->vmeta_equal_cd:
2308   //|.if FFI
2309   //|  sub PC, 4
2310   //|  mov L:RB, SAVE_L
2311   //|  mov L:RB->base, BASE
2312   //|  mov FCARG1, L:RB
2313   //|  mov FCARG2, dword [PC-4]
2314   //|  mov SAVE_PC, PC
2315   //|  call extern lj_meta_equal_cd@8        // (lua_State *L, BCIns ins)
2316   //|  // 0/1 or TValue * (metamethod) returned in eax (RC).
2317   //|  jmp <3
2318   //|.endif
2319   //|
2320   //|->vmeta_istype:
2321   //|.if X64
2322   //|  mov L:RB, SAVE_L
2323   //|  mov L:RB->base, BASE               // Caveat: CARG2d/CARG3d may be BASE.
2324   //|  mov CARG2d, RA
2325   dasm_put(Dst, 1455, LJ_TISTRUECOND, LJ_TISTRUECOND, Dt1(->base), Dt1(->base), Dt1(->base));
2326 #line 1144 "vm_x86.dasc"
2327   //|  movzx CARG3d, PC_RD
2328   //|  mov L:CARG1d, L:RB
2329   //|.else
2330   //|  movzx RD, PC_RD
```

```
  //|  mov ARG2, RA
  //|  mov L:RB, SAVE_L
  //|  mov ARG3, RD
  //|  mov ARG1, L:RB
  //|  mov L:RB->base, BASE
  //|.endif
  //|  mov SAVE_PC, PC
  //|  call extern lj_meta_istype  // (lua_State *L, BCReg ra, BCReg tp)
  //|  mov BASE, L:RB->base
  //|  jmp <6
  //|
  //|//-- Arithmetic metamethods ------------------------------------------
  //|
  //|->vmeta_arith_vno:
  //|.if DUALNUM
  //|  movzx RB, PC_RB
  //|.endif
  //|->vmeta_arith_vn:
  //|  lea RC, [KBASE+RC*8]
  //|  jmp >1
  //|
  //|->vmeta_arith_nvo:
  //|.if DUALNUM
  //|  movzx RC, PC_RC
  //|.endif
  //|->vmeta_arith_nv:
  //|  lea RC, [KBASE+RC*8]
  //|  lea RB, [BASE+RB*8]
  //|  xchg RB, RC
  //|  jmp >2
  //|
  //|->vmeta_unm:
  //|  lea RC, [BASE+RD*8]
  //|  mov RB, RC
  //|  jmp >2
  //|
  //|->vmeta_arith_vvo:
  //|.if DUALNUM
  //|  movzx RB, PC_RB
  //|.endif
  //|->vmeta_arith_vv:
  //|  lea RC, [BASE+RC*8]
  //|1:
  //|  lea RB, [BASE+RB*8]
  //|2:
  //|  lea RA, [BASE+RA*8]
  //|.if X64WIN
  //|  mov CARG3d, RB
  //|  mov CARG4d, RC
  //|  movzx RC, PC_OP
  //|  mov ARG5d, RC
  //|  mov L:RB, SAVE_L
  //|  mov L:RB->base, BASE                // Caveat: CARG2d == BASE.
  //|  mov CARG2d, RA
  //|  mov CARG1d, L:RB                        // Caveat: CARG1d == RA.
  //|.elif X64
  //|  movzx CARG5d, PC_OP
  //|  mov CARG2d, RA
  //|  mov CARG4d, RC                        // Caveat: CARG4d == RA.
  //|  mov L:CARG1d, SAVE_L
  //|  mov L:CARG1d->base, BASE                // Caveat: CARG3d == BASE.
  //|  mov CARG3d, RB
  //|  mov L:RB, L:CARG1d
  //|.else
  //|  mov ARG3, RB
  //|  mov L:RB, SAVE_L
  //|  mov ARG4, RC
  //|  movzx RC, PC_OP
  //|  mov ARG2, RA
  //|  mov ARG5, RC
  //|  mov ARG1, L:RB
  //|  mov L:RB->base, BASE
  //|.endif
  //|  mov SAVE_PC, PC
  //|  call extern lj_meta_arith        // (lua_State *L, TValue *ra,*rb,*rc, BCReg op)
  //|  // NULL (finished) or TValue * (metamethod) returned in eax (RC).
```

```
2407   //|  mov BASE, L:RB->base
2408   //|  test RC, RC
2409   dasm_put(Dst, 1581, Dt1(->base), Dt1(->base), Dt1(->base));
2410 #line 1226 "vm_x86.dasc"
2411   //|  jz ->cont_nop
2412   //|
2413   //|  // Call metamethod for binary op.
2414   //|->vmeta_binop:
2415   //|  // BASE = base, RC = new base, stack = cont/func/o1/o2
2416   //|  mov RA, RC
2417   //|  sub RC, BASE
2418   //|  mov [RA-12], PC                        // [cont|PC]
2419   //|  lea PC, [RC+FRAME_CONT]
2420   //|  mov NARGS:RD, 2+1                        // 2 args for func(o1, o2).
2421   //|  jmp ->vm_call_dispatch
2422   //|
2423   //|->vmeta_len:
2424   //|  mov L:RB, SAVE_L
2425   //|  mov L:RB->base, BASE
2426   //|  lea FCARG2, [BASE+RD*8]                // Caveat: FCARG2 == BASE
2427   //|  mov L:FCARG1, L:RB
2428   //|  mov SAVE_PC, PC
2429   //|  call extern lj_meta_len@8                // (lua_State *L, TValue *o)
2430   //|  // NULL (retry) or TValue * (metamethod) returned in eax (RC).
2431   //|  mov BASE, L:RB->base
2432   dasm_put(Dst, 1714, FRAME_CONT, 2+1, Dt1(->base), Dt1(->base));
2433 #line 1247 "vm_x86.dasc"
2434 #if LJ_52
2435   //|  test RC, RC
2436   //|  jne ->vmeta_binop                        // Binop call for compatibility.
2437   //|  movzx RD, PC_RD
2438   //|  mov TAB:FCARG1, [BASE+RD*8]
2439   //|  jmp ->BC_LEN_Z
2440   dasm_put(Dst, 1766);
2441 #line 1253 "vm_x86.dasc"
2442 #else
2443   //|  jmp ->vmeta_binop                        // Binop call for compatibility.
2444   dasm_put(Dst, 1785);
2445 #line 1255 "vm_x86.dasc"
2446 #endif
2447   //|
2448   //|//-- Call metamethod ------------------------------------------------
2449   //|
2450   //|->vmeta_call_ra:
2451   //|  lea RA, [BASE+RA*8+8]
2452   //|->vmeta_call:                          // Resolve and call __call metamethod.
2453   //|  // BASE = old base, RA = new base, RC = nargs+1, PC = return
2454   //|  mov TMP2, RA                          // Save RA, RC for us.
2455   //|  mov TMP1, NARGS:RD
2456   //|  sub RA, 8
2457   //|.if X64
2458   //|  mov L:RB, SAVE_L
2459   //|  mov L:RB->base, BASE                  // Caveat: CARG2d/CARG3d may be BASE.
2460   //|  mov CARG2d, RA
2461   //|  lea CARG3d, [RA+NARGS:RD*8]
2462   //|  mov CARG1d, L:RB                        // Caveat: CARG1d may be RA.
2463   //|.else
2464   //|  lea RC, [RA+NARGS:RD*8]
2465   //|  mov L:RB, SAVE_L
2466   //|  mov ARG2, RA
2467   //|  mov ARG3, RC
2468   //|  mov ARG1, L:RB
2469   //|  mov L:RB->base, BASE                  // This is the callers base!
2470   //|.endif
2471   //|  mov SAVE_PC, PC
2472   //|  call extern lj_meta_call        // (lua_State *L, TValue *func, TValue *top)
2473   //|  mov BASE, L:RB->base
2474   //|  mov RA, TMP2
2475   //|  mov NARGS:RD, TMP1
2476   //|  mov LFUNC:RB, [RA-8]
2477   //|  add NARGS:RD, 1
2478   //|  // This is fragile. L->base must not move, KBASE must always be defined.
2479   //|  cmp KBASE, BASE                        // Continue with CALLT if flag set.
2480   //|  je ->BC_CALLT_Z
2481   //|  mov BASE, RA
2482   //|  ins_call                                // Otherwise call resolved metamethod.
```

```
2483    //|
2484    //|//-- Argument coercion for 'for' statement ----------------------------
2485    //|
2486    //|->vmeta_for:
2487    //|  mov L:RB, SAVE_L
2488    //|  mov L:RB->base, BASE
2489    //|  mov FCARG2, RA                        // Caveat: FCARG2 == BASE
2490    //|  mov L:FCARG1, L:RB                        // Caveat: FCARG1 == RA
2491    //|  mov SAVE_PC, PC
2492    //|  call extern lj_meta_for@8        // (lua_State *L, TValue *base)
2493    //|  mov BASE, L:RB->base
2494    //|  mov RC, [PC-4]
2495    //|  movzx RA, RCH
2496    //|  movzx OP, RCL
2497    //|  shr RC, 16
2498    //|.if X64
2499    //|  jmp aword [DISPATCH+OP*8+GG_DISP2STATIC]        // Retry FORI or JFORI.
2500    //|.else
2501    //|  jmp aword [DISPATCH+OP*4+GG_DISP2STATIC]        // Retry FORI or JFORI.
2502    //|.endif
2503    //|
2504    //|//-----------------------------------------------------------------------
2505    //|//-- Fast functions -----------------------------------------------------
2506    //|//-----------------------------------------------------------------------
2507    //|
2508    //|.macro .ffunc, name
2509    //|->ff_ .. name:
2510    //|.endmacro
2511    //|
2512    //|.macro .ffunc_1, name
2513    //|->ff_ .. name:
2514    //|  cmp NARGS:RD, 1+1;  jb ->fff_fallback
2515    //|.endmacro
2516    //|
2517    //|.macro .ffunc_2, name
2518    //|->ff_ .. name:
2519    //|  cmp NARGS:RD, 2+1;  jb ->fff_fallback
2520    //|.endmacro
2521    //|
2522    //|.macro .ffunc_nsse, name, op
2523    //|  .ffunc_1 name
2524    //|  cmp dword [BASE+4], LJ_TISNUM;  jae ->fff_fallback
2525    //|  op xmm0, qword [BASE]
2526    //|.endmacro
2527    //|
2528    //|.macro .ffunc_nsse, name
2529    //|  .ffunc_nsse name, movsd
2530    //|.endmacro
2531    //|
2532    //|.macro .ffunc_nnsse, name
2533    //|  .ffunc_2 name
2534    //|  cmp dword [BASE+4], LJ_TISNUM;  jae ->fff_fallback
2535    //|  cmp dword [BASE+12], LJ_TISNUM;  jae ->fff_fallback
2536    //|  movsd xmm0, qword [BASE]
2537    //|  movsd xmm1, qword [BASE+8]
2538    //|.endmacro
2539    //|
2540    //|.macro .ffunc_nnr, name
2541    //|  .ffunc_2 name
2542    //|  cmp dword [BASE+4], LJ_TISNUM;  jae ->fff_fallback
2543    //|  cmp dword [BASE+12], LJ_TISNUM;  jae ->fff_fallback
2544    //|  fld qword [BASE+8]
2545    //|  fld qword [BASE]
2546    //|.endmacro
2547    //|
2548    //|// Inlined GC threshold check. Caveat: uses label 1.
2549    //|.macro ffgccheck
2550    //|  mov RB, [DISPATCH+DISPATCH_GL(gc.total)]
2551    //|  cmp RB, [DISPATCH+DISPATCH_GL(gc.threshold)]
2552    //|  jb >1
2553    //|  call ->fff_gcstep
2554    //|1:
2555    //|.endmacro
2556    //|
2557    //|//-- Base library: checks -----------------------------------------------
2558    //|
```

```
2559    //|.ffunc_1 assert
2560    //|  mov RB, [BASE+4]
2561    //|  cmp RB, LJ_TISTRUECOND;  jae ->fff_fallback
2562    //|  mov PC, [BASE-4]
2563    //|  mov MULTRES, RD
2564    //|  mov [BASE-4], RB
2565    //|  mov RB, [BASE]
2566    //|  mov [BASE-8], RB
2567    //|  sub RD, 2
2568    //|  jz >2
2569    //|  mov RA, BASE
2570    dasm_put(Dst, 1790, Dt1(->base), Dt1(->base), Dt7(->pc), Dt1(->base), Dt1(->base), GG_DISP2STATIC, 1+1,
LJ_TISTRUECOND);
2571  #line 1379 "vm_x86.dasc"
2572    //|1:
2573    //|  add RA, 8
2574    //|.if X64
2575    //|  mov RBa, [RA]
2576    //|  mov [RA-8], RBa
2577    //|.else
2578    //|  mov RB, [RA+4]
2579    //|  mov [RA-4], RB
2580    //|  mov RB, [RA]
2581    //|  mov [RA-8], RB
2582    //|.endif
2583    //|  sub RD, 1
2584    //|  jnz <1
2585    //|2:
2586    //|  mov RD, MULTRES
2587    //|  jmp ->fff_res_
2588    //|
2589    //|.ffunc_1 type
2590    //|  mov RB, [BASE+4]
2591    //|.if X64
2592    //|  mov RA, RB
2593    //|  sar RA, 15
2594    //|  cmp RA, -2
2595    //|  je >3
2596    //|.endif
2597    //|  mov RC, ~LJ_TNUMX
2598    //|  not RB
2599    //|  cmp RC, RB
2600    //|  cmova RC, RB
2601    //|2:
2602    //|  mov CFUNC:RB, [BASE-8]
2603    //|  mov STR:RC, [CFUNC:RB+RC*8+((char *)(&((GCfuncC *)0)->upvalue))]
2604    //|  mov PC, [BASE-4]
2605    //|  mov dword [BASE-4], LJ_TSTR
2606    //|  mov [BASE-8], STR:RC
2607    //|  jmp ->fff_res1
2608    //|.if X64
2609    //|3:
2610    //|  mov RC, ~LJ_TLIGHTUD
2611    //|  jmp <2
2612    dasm_put(Dst, 1976, 1+1, ~LJ_TNUMX, ((char *)(&((GCfuncC *)0)->upvalue)), LJ_TSTR, ~LJ_TLIGHTUD);
2613  #line 1419 "vm_x86.dasc"
2614    //|.endif
2615    //|
2616    //|//-- Base library: getters and setters ---------------------------------
2617    //|
2618    //|.ffunc_1 getmetatable
2619    //|  mov RB, [BASE+4]
2620    //|  mov PC, [BASE-4]
2621    //|  cmp RB, LJ_TTAB;  jne >6
2622    //|1:  // Field metatable must be at same offset for GCtab and GCudata!
2623    //|  mov TAB:RB, [BASE]
2624    //|  mov TAB:RB, TAB:RB->metatable
2625    //|2:
2626    //|  test TAB:RB, TAB:RB
2627    //|  mov dword [BASE-4], LJ_TNIL
2628    //|  jz ->fff_res1
2629    //|  mov STR:RC, [DISPATCH+DISPATCH_GL(gcroot)+4*(GCROOT_MMNAME+MM_metatable)]
2630    //|  mov dword [BASE-4], LJ_TTAB        // Store metatable as default result.
2631    //|  mov [BASE-8], TAB:RB
2632    //|  mov RA, TAB:RB->hmask
2633    //|  and RA, STR:RC->hash
```

```
2634    //|  imul RA, #NODE
2635    //|  add NODE:RA, TAB:RB->node
2636    dasm_put(Dst, 2080, 1+1, LJ_TTAB, Dt6(->metatable), LJ_TNIL, DISPATCH_GL(gcroot)+4*
(GCROOT_MMNAME+MM_metatable), LJ_TTAB, Dt6(->hmask), Dt5(->hash), sizeof(Node));
2637  #line 1441 "vm_x86.dasc"
2638    //|3:  // Rearranged logic, because we expect _not_ to find the key.
2639    //|  cmp dword NODE:RA->key.it, LJ_TSTR
2640    //|  jne >4
2641    //|  cmp dword NODE:RA->key.gcr, STR:RC
2642    //|  je >5
2643    //|4:
2644    //|  mov NODE:RA, NODE:RA->next
2645    //|  test NODE:RA, NODE:RA
2646    //|  jnz <3
2647    //|  jmp ->fff_res1                       // Not found, keep default result.
2648    //|5:
2649    //|  mov RB, [RA+4]
2650    //|  cmp RB, LJ_TNIL;  je ->fff_res1        // Ditto for nil value.
2651    //|  mov RC, [RA]
2652    //|  mov [BASE-4], RB                       // Return value of mt.__metatable.
2653    //|  mov [BASE-8], RC
2654    //|  jmp ->fff_res1
2655    //|
2656    //|6:
2657    //|  cmp RB, LJ_TUDATA;  je <1
2658    dasm_put(Dst, 2153, Dt6(->node), DtB(->key.it), LJ_TSTR, DtB(->key.gcr), DtB(->next), LJ_TNIL);
2659  #line 1461 "vm_x86.dasc"
2660    //|.if X64
2661    //|  cmp RB, LJ_TNUMX;  ja >8
2662    //|  cmp RB, LJ_TISNUM;  jbe >7
2663    //|  mov RB, LJ_TLIGHTUD
2664    //|  jmp >8
2665    //|7:
2666    //|.else
2667    //|  cmp RB, LJ_TISNUM;  ja >8
2668    //|.endif
2669    //|  mov RB, LJ_TNUMX
2670    //|8:
2671    //|  not RB
2672    //|  mov TAB:RB, [DISPATCH+RB*4+DISPATCH_GL(gcroot[GCROOT_BASEMT])]
2673    //|  jmp <2
2674    //|
2675    //|.ffunc_2 setmetatable
2676    dasm_put(Dst, 2218, LJ_TUDATA, LJ_TNUMX, LJ_TISNUM, LJ_TLIGHTUD, LJ_TNUMX,
DISPATCH_GL(gcroot[GCROOT_BASEMT]), 2+1);
2677  #line 1477 "vm_x86.dasc"
2678    //|  cmp dword [BASE+4], LJ_TTAB;  jne ->fff_fallback
2679    //|  // Fast path: no mt for table yet and not clearing the mt.
2680    //|  mov TAB:RB, [BASE]
2681    //|  cmp dword TAB:RB->metatable, 0;  jne ->fff_fallback
2682    //|  cmp dword [BASE+12], LJ_TTAB;  jne ->fff_fallback
2683    //|  mov TAB:RC, [BASE+8]
2684    //|  mov TAB:RB->metatable, TAB:RC
2685    //|  mov PC, [BASE-4]
2686    //|  mov dword [BASE-4], LJ_TTAB               // Return original table.
2687    //|  mov [BASE-8], TAB:RB
2688    //|  test byte TAB:RB->marked, LJ_GC_BLACK        // isblack(table)
2689    //|  jz >1
2690    //|  // Possible write barrier. Table is black, but skip iswhite(mt) check.
2691    //|  barrierback TAB:RB, RC
2692    dasm_put(Dst, 2274, LJ_TTAB, Dt6(->metatable), LJ_TTAB, Dt6(->metatable), LJ_TTAB, Dt6(->marked),
LJ_GC_BLACK, Dt6(->marked), (uint8_t)~LJ_GC_BLACK, DISPATCH_GL(gc.grayagain));
2693  #line 1491 "vm_x86.dasc"
2694    //|1:
2695    //|  jmp ->fff_res1
2696    //|
2697    //|.ffunc_2 rawget
2698    //|  cmp dword [BASE+4], LJ_TTAB;  jne ->fff_fallback
2699    //|.if X64WIN
2700    //|  mov RB, BASE                         // Save BASE.
2701    //|  lea CARG3d, [BASE+8]
2702    //|  mov CARG2d, [BASE]                       // Caveat: CARG2d == BASE.
2703    //|  mov CARG1d, SAVE_L
2704    //|.elif X64
2705    //|  mov RB, BASE                         // Save BASE.
2706    //|  mov CARG2d, [BASE]
```

```
  //|  lea CARG3d, [BASE+8]                // Caveat: CARG3d == BASE.
  //|  mov CARG1d, SAVE_L
  //|.else
  //|  mov TAB:RD, [BASE]
  //|  mov L:RB, SAVE_L
  //|  mov ARG2, TAB:RD
  //|  mov ARG1, L:RB
  //|  mov RB, BASE                        // Save BASE.
  //|  add BASE, 8
  //|  mov ARG3, BASE
  //|.endif
  //|  call extern lj_tab_get        // (lua_State *L, GCtab *t, cTValue *key)
  //|  // cTValue * returned in eax (RD).
  //|  mov BASE, RB                        // Restore BASE.
  //|  // Copy table slot.
  //|.if X64
  //|  mov RBa, [RD]
  //|  mov PC, [BASE-4]
  //|  mov [BASE-8], RBa
  //|.else
  //|  mov RB, [RD]
  //|  mov RD, [RD+4]
  //|  mov PC, [BASE-4]
  //|  mov [BASE-8], RB
  //|  mov [BASE-4], RD
  //|.endif
  //|  jmp ->fff_res1
  //|
  //|//-- Base library: conversions ------------------------------------------
  //|
  //|.ffunc tonumber
  //|  // Only handles the number case inline (without a base argument).
  //|  cmp NARGS:RD, 1+1;  jne ->fff_fallback        // Exactly one argument.
  //|  cmp dword [BASE+4], LJ_TISNUM
  //|.if DUALNUM
  //|  jne >1
  //|  mov RB, dword [BASE]; jmp ->fff_resi
   dasm_put(Dst, 2343, DISPATCH_GL(gc.grayagain), Dt6(->gclist), 2+1, LJ_TTAB, 1+1, LJ_TISNUM);
#line 1541 "vm_x86.dasc"
  //|1:
  //|  ja ->fff_fallback
  //|.else
  //|  jae ->fff_fallback
  //|.endif
  //|  movsd xmm0, qword [BASE]; jmp ->fff_resxmm0
  //|
  //|.ffunc_1 tostring
  //|  // Only handles the string or number case inline.
  //|  mov PC, [BASE-4]
  //|  cmp dword [BASE+4], LJ_TSTR;  jne >3
  //|  // A __tostring method in the string base metatable is ignored.
  //|  mov STR:RD, [BASE]
  //|2:
  //|  mov dword [BASE-4], LJ_TSTR
  //|  mov [BASE-8], STR:RD
  //|  jmp ->fff_res1
  //|3:  // Handle numbers inline, unless a number base metatable is present.
  //|  cmp dword [BASE+4], LJ_TISNUM;  ja ->fff_fallback
   dasm_put(Dst, 2429, 1+1, LJ_TSTR, LJ_TSTR, LJ_TISNUM);
#line 1560 "vm_x86.dasc"
  //|  cmp dword [DISPATCH+DISPATCH_GL(gcroot[GCROOT_BASEMT_NUM])], 0
  //|  jne ->fff_fallback
  //|  ffgccheck                              // Caveat: uses label 1.
  //|  mov L:RB, SAVE_L
  //|  mov L:RB->base, BASE                // Add frame since C call can throw.
  //|  mov SAVE_PC, PC                        // Redundant (but a defined value).
  //|.if X64 and not X64WIN
  //|  mov FCARG2, BASE                       // Otherwise: FCARG2 == BASE
  //|.endif
  //|  mov L:FCARG1, L:RB
  //|.if DUALNUM
  //|  call extern lj_strfmt_number@8        // (lua_State *L, cTValue *o)
  //|.else
  //|  call extern lj_strfmt_num@8        // (lua_State *L, lua_Number *np)
  //|.endif
  //|  // GCstr returned in eax (RD).
```

```
2783    //|  mov BASE, L:RB->base
2784    //|  jmp <2
2785    //|
2786    //|//-- Base library: iterators -----------------------------------------
2787    //|
2788    //|.ffunc_1 next
2789    //|  je >2                                // Missing 2nd arg?
2790    dasm_put(Dst, 2498, DISPATCH_GL(gcroot[GCROOT_BASEMT_NUM]), DISPATCH_GL(gc.total),
DISPATCH_GL(gc.threshold), Dt1(->base), Dt1(->base), 1+1);
2791  #line 1583 "vm_x86.dasc"
2792    //|1:
2793    //|  cmp dword [BASE+4], LJ_TTAB;  jne ->fff_fallback
2794    //|  mov L:RB, SAVE_L
2795    //|  mov L:RB->base, BASE                // Add frame since C call can throw.
2796    //|  mov L:RB->top, BASE                 // Dummy frame length is ok.
2797    //|  mov PC, [BASE-4]
2798    //|.if X64WIN
2799    //|  lea CARG3d, [BASE+8]
2800    //|  mov CARG2d, [BASE]                        // Caveat: CARG2d == BASE.
2801    //|  mov CARG1d, L:RB
2802    //|.elif X64
2803    //|  mov CARG2d, [BASE]
2804    //|  lea CARG3d, [BASE+8]                // Caveat: CARG3d == BASE.
2805    //|  mov CARG1d, L:RB
2806    //|.else
2807    //|  mov TAB:RD, [BASE]
2808    //|  mov ARG2, TAB:RD
2809    //|  mov ARG1, L:RB
2810    //|  add BASE, 8
2811    //|  mov ARG3, BASE
2812    //|.endif
2813    //|  mov SAVE_PC, PC                          // Needed for ITERN fallback.
2814    //|  call extern lj_tab_next        // (lua_State *L, GCtab *t, TValue *key)
2815    //|  // Flag returned in eax (RD).
2816    //|  mov BASE, L:RB->base
2817    //|  test RD, RD;  jz >3                 // End of traversal?
2818    //|  // Copy key and value to results.
2819    //|.if X64
2820    //|  mov RBa, [BASE+8]
2821    //|  mov RDa, [BASE+16]
2822    //|  mov [BASE-8], RBa
2823    //|  mov [BASE], RDa
2824    //|.else
2825    //|  mov RB, [BASE+8]
2826    //|  mov RD, [BASE+12]
2827    //|  mov [BASE-8], RB
2828    //|  mov [BASE-4], RD
2829    //|  mov RB, [BASE+16]
2830    //|  mov RD, [BASE+20]
2831    //|  mov [BASE], RB
2832    //|  mov [BASE+4], RD
2833    //|.endif
2834    //|->fff_res2:
2835    //|  mov RD, 1+2
2836    //|  jmp ->fff_res
2837    //|2:  // Set missing 2nd arg to nil.
2838    //|  mov dword [BASE+12], LJ_TNIL
2839    //|  jmp <1
2840    //|3:  // End of traversal: return nil.
2841    //|  mov dword [BASE-4], LJ_TNIL
2842    dasm_put(Dst, 2566, LJ_TTAB, Dt1(->base), Dt1(->top), Dt1(->base), 1+2, LJ_TNIL);
2843  #line 1633 "vm_x86.dasc"
2844    //|  jmp ->fff_res1
2845    //|
2846    //|.ffunc_1 pairs
2847    //|  mov TAB:RB, [BASE]
2848    //|  cmp dword [BASE+4], LJ_TTAB;  jne ->fff_fallback
2849    dasm_put(Dst, 2657, LJ_TNIL, 1+1, LJ_TTAB);
2850  #line 1638 "vm_x86.dasc"
2851  #if LJ_52
2852    //|  cmp dword TAB:RB->metatable, 0; jne ->fff_fallback
2853    dasm_put(Dst, 2688, Dt6(->metatable));
2854  #line 1640 "vm_x86.dasc"
2855  #endif
2856    //|  mov CFUNC:RB, [BASE-8]
2857    //|  mov CFUNC:RD, CFUNC:RB->upvalue[0]
```

```
2858    //|  mov PC, [BASE-4]
2859    //|  mov dword [BASE-4], LJ_TFUNC
2860    //|  mov [BASE-8], CFUNC:RD
2861    //|  mov dword [BASE+12], LJ_TNIL
2862    //|  mov RD, 1+3
2863    //|  jmp ->fff_res
2864    //|
2865    //|.ffunc_2 ipairs_aux
2866    //|  cmp dword [BASE+4], LJ_TTAB;  jne ->fff_fallback
2867    //|  cmp dword [BASE+12], LJ_TISNUM
2868    //|.if DUALNUM
2869    //|  jne ->fff_fallback
2870    //|.else
2871    //|  jae ->fff_fallback
2872    //|.endif
2873    //|  mov PC, [BASE-4]
2874    //|.if DUALNUM
2875    //|  mov RD, dword [BASE+8]
2876    //|  add RD, 1
2877    //|  mov dword [BASE-4], LJ_TISNUM
2878    //|  mov dword [BASE-8], RD
2879    //|.else
2880    //|  movsd xmm0, qword [BASE+8]
2881    //|  sseconst_1 xmm1, RBa
2882    //|  addsd xmm0, xmm1
2883    //|  cvttsd2si RD, xmm0
2884    //|  movsd qword [BASE-8], xmm0
2885    //|.endif
2886    //|  mov TAB:RB, [BASE]
2887    //|  cmp RD, TAB:RB->asize;  jae >2        // Not in array part?
2888    //|  shl RD, 3
2889    dasm_put(Dst, 2697, Dt8(->upvalue[0]), LJ_TFUNC, LJ_TNIL, 1+3, 2+1, LJ_TTAB, LJ_TISNUM, LJ_TISNUM, Dt6(-
>asize));
2890  #line 1674 "vm_x86.dasc"
2891    //|  add RD, TAB:RB->array
2892    //|1:
2893    //|  cmp dword [RD+4], LJ_TNIL;  je ->fff_res0
2894    //|  // Copy array slot.
2895    //|.if X64
2896    //|  mov RBa, [RD]
2897    //|  mov [BASE], RBa
2898    //|.else
2899    //|  mov RB, [RD]
2900    //|  mov RD, [RD+4]
2901    //|  mov [BASE], RB
2902    //|  mov [BASE+4], RD
2903    //|.endif
2904    //|  jmp ->fff_res2
2905    //|2:  // Check for empty hash part first. Otherwise call C function.
2906    //|  cmp dword TAB:RB->hmask, 0; je ->fff_res0
2907    //|  mov FCARG1, TAB:RB
2908    //|  mov RB, BASE                        // Save BASE.
2909    //|  mov FCARG2, RD                        // Caveat: FCARG2 == BASE
2910    //|  call extern lj_tab_getinth@8        // (GCtab *t, int32_t key)
2911    //|  // cTValue * or NULL returned in eax (RD).
2912    //|  mov BASE, RB
2913    //|  test RD, RD
2914    //|  jnz <1
2915    //|->fff_res0:
2916    //|  mov RD, 1+0
2917    //|  jmp ->fff_res
2918    //|
2919    //|.ffunc_1 ipairs
2920    //|  mov TAB:RB, [BASE]
2921    dasm_put(Dst, 2784, Dt6(->array), LJ_TNIL, Dt6(->hmask), 1+0, 1+1);
2922  #line 1704 "vm_x86.dasc"
2923    //|  cmp dword [BASE+4], LJ_TTAB;  jne ->fff_fallback
2924    dasm_put(Dst, 2676, LJ_TTAB);
2925  #line 1705 "vm_x86.dasc"
2926  #if LJ_52
2927    //|  cmp dword TAB:RB->metatable, 0; jne ->fff_fallback
2928    dasm_put(Dst, 2688, Dt6(->metatable));
2929  #line 1707 "vm_x86.dasc"
2930  #endif
2931    //|  mov CFUNC:RB, [BASE-8]
2932    //|  mov CFUNC:RD, CFUNC:RB->upvalue[0]
```

```
2933    //|  mov PC, [BASE-4]
2934    //|  mov dword [BASE-4], LJ_TFUNC
2935    //|  mov [BASE-8], CFUNC:RD
2936    //|.if DUALNUM
2937    //|  mov dword [BASE+12], LJ_TISNUM
2938    //|  mov dword [BASE+8], 0
2939    //|.else
2940    //|  xorps xmm0, xmm0
2941    //|  movsd qword [BASE+8], xmm0
2942    //|.endif
2943    //|  mov RD, 1+3
2944    //|  jmp ->fff_res
2945    //|
2946    //|//-- Base library: catch errors ----------------------------------------
2947    //|
2948    //|.ffunc_1 pcall
2949    //|  lea RA, [BASE+8]
2950    //|  sub NARGS:RD, 1
2951    //|  mov PC, 8+FRAME_PCALL
2952    //|1:
2953    //|  movzx RB, byte [DISPATCH+DISPATCH_GL(hookmask)]
2954    //|  shr RB, HOOK_ACTIVE_SHIFT
2955    //|  and RB, 1
2956    //|  add PC, RB                              // Remember active hook before pcall.
2957    //|  jmp ->vm_call_dispatch
2958    //|
2959    //|.ffunc_2 xpcall
2960    //|  cmp dword [BASE+12], LJ_TFUNC;  jne ->fff_fallback
2961    dasm_put(Dst, 2860, Dt8(->upvalue[0]), LJ_TFUNC, LJ_TISNUM, 1+3, 1+1, 8+FRAME_PCALL,
DISPATCH_GL(hookmask), HOOK_ACTIVE_SHIFT, 2+1);
2962  #line 1738 "vm_x86.dasc"
2963    //|  mov RB, [BASE+4]                       // Swap function and traceback.
2964    //|  mov [BASE+12], RB
2965    //|  mov dword [BASE+4], LJ_TFUNC
2966    //|  mov LFUNC:RB, [BASE]
2967    //|  mov PC, [BASE+8]
2968    //|  mov [BASE+8], LFUNC:RB
2969    //|  mov [BASE], PC
2970    //|  lea RA, [BASE+16]
2971    //|  sub NARGS:RD, 2
2972    //|  mov PC, 16+FRAME_PCALL
2973    //|  jmp <1
2974    //|
2975    //|//-- Coroutine library --------------------------------------------------
2976    //|
2977    //|.macro coroutine_resume_wrap, resume
2978    //|.if resume
2979    //|.ffunc_1 coroutine_resume
2980    //|  mov L:RB, [BASE]
2981    //|.else
2982    //|.ffunc coroutine_wrap_aux
2983    //|  mov CFUNC:RB, [BASE-8]
2984    //|  mov L:RB, CFUNC:RB->upvalue[0].gcr
2985    //|.endif
2986    //|  mov PC, [BASE-4]
2987    //|  mov SAVE_PC, PC
2988    //|.if X64
2989    //|  mov TMP1, L:RB
2990    //|.else
2991    //|  mov ARG1, L:RB
2992    //|.endif
2993    //|.if resume
2994    //|  cmp dword [BASE+4], LJ_TTHREAD;  jne ->fff_fallback
2995    //|.endif
2996    //|  cmp aword L:RB->cframe, 0; jne ->fff_fallback
2997    //|  cmp byte L:RB->status, LUA_YIELD;  ja ->fff_fallback
2998    //|  mov RA, L:RB->top
2999    //|  je >1                                  // Status != LUA_YIELD (i.e. 0)?
3000    //|  cmp RA, L:RB->base                          // Check for presence of initial func.
3001    //|  je ->fff_fallback
3002    //|1:
3003    //|.if resume
3004    //|  lea PC, [RA+NARGS:RD*8-16]                // Check stack space (-1-thread).
3005    //|.else
3006    //|  lea PC, [RA+NARGS:RD*8-8]                // Check stack space (-1).
3007    //|.endif
```

```
3008    //|  cmp PC, L:RB->maxstack; ja ->fff_fallback
3009    //|  mov L:RB->top, PC
3010    //|
3011    //|  mov L:RB, SAVE_L
3012    //|  mov L:RB->base, BASE
3013    //|.if resume
3014    //|  add BASE, 8                        // Keep resumed thread in stack for GC.
3015    //|.endif
3016    //|  mov L:RB->top, BASE
3017    //|.if resume
3018    //|  lea RB, [BASE+NARGS:RD*8-24]        // RB = end of source for stack move.
3019    //|.else
3020    //|  lea RB, [BASE+NARGS:RD*8-16]        // RB = end of source for stack move.
3021    //|.endif
3022    //|  sub RBa, PCa                        // Relative to PC.
3023    //|
3024    //|  cmp PC, RA
3025    //|  je >3
3026    //|2:  // Move args to coroutine.
3027    //|.if X64
3028    //|  mov RCa, [PC+RB]
3029    //|  mov [PC-8], RCa
3030    //|.else
3031    //|  mov RC, [PC+RB+4]
3032    //|  mov [PC-4], RC
3033    //|  mov RC, [PC+RB]
3034    //|  mov [PC-8], RC
3035    //|.endif
3036    //|  sub PC, 8
3037    //|  cmp PC, RA
3038    //|  jne <2
3039    //|3:
3040    //|.if X64
3041    //|  mov CARG2d, RA
3042    //|  mov CARG1d, TMP1
3043    //|.else
3044    //|  mov ARG2, RA
3045    //|  xor RA, RA
3046    //|  mov ARG4, RA
3047    //|  mov ARG3, RA
3048    //|.endif
3049    //|  call ->vm_resume                       // (lua_State *L, TValue *base, 0, 0)
3050    //|
3051    //|  mov L:RB, SAVE_L
3052    //|.if X64
3053    //|  mov L:PC, TMP1
3054    //|.else
3055    //|  mov L:PC, ARG1                       // The callee doesn't modify SAVE_L.
3056    //|.endif
3057    //|  mov BASE, L:RB->base
3058    //|  mov [DISPATCH+DISPATCH_GL(cur_L)], L:RB
3059    //|  set_vmstate INTERP
3060    //|
3061    //|  cmp eax, LUA_YIELD
3062    //|  ja >8
3063    //|4:
3064    //|  mov RA, L:PC->base
3065    //|  mov KBASE, L:PC->top
3066    //|  mov L:PC->top, RA                      // Clear coroutine stack.
3067    //|  mov PC, KBASE
3068    //|  sub PC, RA
3069    //|  je >6                                // No results?
3070    //|  lea RD, [BASE+PC]
3071    //|  shr PC, 3
3072    //|  cmp RD, L:RB->maxstack
3073    //|  ja >9                                // Need to grow stack?
3074    //|
3075    //|  mov RB, BASE
3076    //|  sub RBa, RAa
3077    //|5:  // Move results from coroutine.
3078    //|.if X64
3079    //|  mov RDa, [RA]
3080    //|  mov [RA+RB], RDa
3081    //|.else
3082    //|  mov RD, [RA]
3083    //|  mov [RA+RB], RD
```

```
3084    //|  mov RD, [RA+4]
3085    //|  mov [RA+RB+4], RD
3086    //|.endif
3087    //|  add RA, 8
3088    //|  cmp RA, KBASE
3089    //|  jne <5
3090    //|6:
3091    //|.if resume
3092    //|  lea RD, [PC+2]                      // nresults+1 = 1 + true + results.
3093    //|  mov dword [BASE-4], LJ_TTRUE        // Prepend true to results.
3094    //|.else
3095    //|  lea RD, [PC+1]                      // nresults+1 = 1 + results.
3096    //|.endif
3097    //|7:
3098    //|  mov PC, SAVE_PC
3099    //|  mov MULTRES, RD
3100    //|.if resume
3101    //|  mov RAa, -8
3102    //|.else
3103    //|  xor RA, RA
3104    //|.endif
3105    //|  test PC, FRAME_TYPE
3106    //|  jz ->BC_RET_Z
3107    //|  jmp ->vm_return
3108    //|
3109    //|8:  // Coroutine returned with error (at co->top-1).
3110    //|.if resume
3111    //|  mov dword [BASE-4], LJ_TFALSE        // Prepend false to results.
3112    //|  mov RA, L:PC->top
3113    //|  sub RA, 8
3114    //|  mov L:PC->top, RA                      // Clear error from coroutine stack.
3115    //|  // Copy error message.
3116    //|.if X64
3117    //|  mov RDa, [RA]
3118    //|  mov [BASE], RDa
3119    //|.else
3120    //|  mov RD, [RA]
3121    //|  mov [BASE], RD
3122    //|  mov RD, [RA+4]
3123    //|  mov [BASE+4], RD
3124    //|.endif
3125    //|  mov RD, 1+2                        // nresults+1 = 1 + false + error.
3126    //|  jmp <7
3127    //|.else
3128    //|  mov FCARG2, L:PC
3129    //|  mov FCARG1, L:RB
3130    //|  call extern lj_ffh_coroutine_wrap_err@8  // (lua_State *L, lua_State *co)
3131    //|  // Error function does not return.
3132    //|.endif
3133    //|
3134    //|9:  // Handle stack expansion on return from yield.
3135    //|.if X64
3136    //|  mov L:RA, TMP1
3137    //|.else
3138    //|  mov L:RA, ARG1                       // The callee doesn't modify SAVE_L.
3139    //|.endif
3140    //|  mov L:RA->top, KBASE               // Undo coroutine stack clearing.
3141    //|  mov FCARG2, PC
3142    //|  mov FCARG1, L:RB
3143    //|  call extern lj_state_growstack@8       // (lua_State *L, int n)
3144    //|.if X64
3145    //|  mov L:PC, TMP1
3146    //|.else
3147    //|  mov L:PC, ARG1
3148    //|.endif
3149    //|  mov BASE, L:RB->base
3150    //|  jmp <4                               // Retry the stack move.
3151    //|.endmacro
3152    //|
3153    //|  coroutine_resume_wrap 1               // coroutine.resume
3154    dasm_put(Dst, 2947, LJ_TFUNC, LJ_TFUNC, 16+FRAME_PCALL, 1+1, LJ_TTHREAD, Dt1(->cframe), Dt1(->status),
LUA_YIELD);
3155    dasm_put(Dst, 3038, Dt1(->top), Dt1(->base), Dt1(->maxstack), Dt1(->top), Dt1(->base), Dt1(->top));
3156    dasm_put(Dst, 3127, Dt1(->base), DISPATCH_GL(cur_L), DISPATCH_GL(vmstate), ~LJ_VMST_INTERP, LUA_YIELD,
Dt1(->base), Dt1(->top), Dt1(->top), Dt1(->maxstack));
3157  #line 1929 "vm_x86.dasc"
```

```
3158    //|  coroutine_resume_wrap 0                // coroutine.wrap
3159    dasm_put(Dst, 3218, LJ_TTRUE, FRAME_TYPE, LJ_TFALSE, Dt1(->top), Dt1(->top), 1+2, Dt1(->top), Dt1(->base));
3160    dasm_put(Dst, 3319, Dt8(->upvalue[0].gcr), Dt1(->cframe), Dt1(->status), LUA_YIELD, Dt1(->top), Dt1(->base), Dt1(->maxstack), Dt1(->top), Dt1(->base));
3161    dasm_put(Dst, 3393, Dt1(->top), Dt1(->base), DISPATCH_GL(cur_L), DISPATCH_GL(vmstate), ~LJ_VMST_INTERP, LUA_YIELD, Dt1(->base), Dt1(->top), Dt1(->top));
3162  #line 1930 "vm_x86.dasc"
3163    //|
3164    //|.ffunc coroutine_yield
3165    //|  mov L:RB, SAVE_L
3166    dasm_put(Dst, 3481, Dt1(->maxstack), FRAME_TYPE, Dt1(->top), Dt1(->base));
3167  #line 1933 "vm_x86.dasc"
3168    //|  test aword L:RB->cframe, CFRAME_RESUME
3169    //|  jz ->fff_fallback
3170    //|  mov L:RB->base, BASE
3171    //|  lea RD, [BASE+NARGS:RD*8-8]
3172    //|  mov L:RB->top, RD
3173    //|  xor RD, RD
3174    //|  mov aword L:RB->cframe, RDa
3175    //|  mov al, LUA_YIELD
3176    //|  mov byte L:RB->status, al
3177    //|  jmp ->vm_leave_unw
3178    //|
3179    //|//-- Math library ------------------------------------------------------
3180    //|
3181    //|.if not DUALNUM
3182    //|->fff_resi:  // Dummy.
3183    //|.endif
3184    //|
3185    //|->fff_resn:
3186    //|  mov PC, [BASE-4]
3187    //|  fstp qword [BASE-8]
3188    //|  jmp ->fff_res1
3189    //|
3190    //|  .ffunc_1 math_abs
3191    //|.if DUALNUM
3192    //|  cmp dword [BASE+4], LJ_TISNUM; jne >2
3193    //|  mov RB, dword [BASE]
3194    dasm_put(Dst, 3601, Dt1(->cframe), CFRAME_RESUME, Dt1(->base), Dt1(->top), Dt1(->cframe), LUA_YIELD, Dt1(->status), 1+1, LJ_TISNUM);
3195  #line 1959 "vm_x86.dasc"
3196    //|  cmp RB, 0; jns ->fff_resi
3197    //|  neg RB; js >1
3198    //|->fff_resbit:
3199    //|->fff_resi:
3200    //|  mov PC, [BASE-4]
3201    //|  mov dword [BASE-4], LJ_TISNUM
3202    //|  mov dword [BASE-8], RB
3203    //|  jmp ->fff_res1
3204    //|1:
3205    //|  mov PC, [BASE-4]
3206    //|  mov dword [BASE-4], 0x41e00000  // 2^31.
3207    //|  mov dword [BASE-8], 0
3208    //|  jmp ->fff_res1
3209    //|2:
3210    //|  ja ->fff_fallback
3211    //|.else
3212    //|  cmp dword [BASE+4], LJ_TISNUM; jae ->fff_fallback
3213    //|.endif
3214    //|  movsd xmm0, qword [BASE]
3215    //|  sseconst_abs xmm1, RDa
3216    //|  andps xmm0, xmm1
3217    //|->fff_resxmm0:
3218    //|  mov PC, [BASE-4]
3219    //|  movsd qword [BASE-8], xmm0
3220    //|  // fallthrough
3221    //|
3222    //|->fff_res1:
3223    //|  mov RD, 1+1
3224    //|->fff_res:
3225    //|  mov MULTRES, RD
3226    dasm_put(Dst, 3675, LJ_TISNUM, (unsigned int)(U64x(7fffffff,ffffffff)), (unsigned int)((U64x(7fffffff,ffffffff))>>32), 1+1);
3227  #line 1989 "vm_x86.dasc"
3228    //|->fff_res_:
```

```
3229    //|  test PC, FRAME_TYPE
3230    //|  jnz >7
3231    //|5:
3232    //|  cmp PC_RB, RDL                        // More results expected?
3233    //|  ja >6
3234    //|  // Adjust BASE. KBASE is assumed to be set for the calling frame.
3235    //|  movzx RA, PC_RA
3236    //|  not RAa                                 // Note: ~RA = -(RA+1)
3237    //|  lea BASE, [BASE+RA*8]                 // base = base - (RA+1)*8
3238    //|  ins_next
3239    //|
3240    //|6:  // Fill up results with nil.
3241    //|  mov dword [BASE+RD*8-12], LJ_TNIL
3242    //|  add RD, 1
3243    //|  jmp <5
3244    //|
3245    //|7:  // Non-standard return case.
3246    //|  mov RAa, -8                         // Results start at BASE+RA = BASE-8.
3247    //|  jmp ->vm_return
3248    //|
3249    //|.if X64
3250    //|.define fff_resfp, fff_resxmm0
3251    //|.else
3252    //|.define fff_resfp, fff_resn
3253    //|.endif
3254    //|
3255    //|.macro math_round, func
3256    //|  .ffunc math_ .. func
3257    //|.if DUALNUM
3258    //|  cmp dword [BASE+4], LJ_TISNUM; jne >1
3259    //|  mov RB, dword [BASE]; jmp ->fff_resi
3260    //|1:
3261    //|  ja ->fff_fallback
3262    //|.else
3263    //|  cmp dword [BASE+4], LJ_TISNUM; jae ->fff_fallback
3264    //|.endif
3265    //|  movsd xmm0, qword [BASE]
3266    //|  call ->vm_ .. func .. _sse
3267    //|.if DUALNUM
3268    //|  cvttsd2si RB, xmm0
3269    //|  cmp RB, 0x80000000
3270    //|  jne ->fff_resi
3271    //|  cvtsi2sd xmm1, RB
3272    //|  ucomisd xmm0, xmm1
3273    //|  jp ->fff_resxmm0
3274    //|  je ->fff_resi
3275    //|.endif
3276    //|  jmp ->fff_resxmm0
3277    //|.endmacro
3278    //|
3279    //|  math_round floor
3280    dasm_put(Dst, 3782, FRAME_TYPE, LJ_TNIL, LJ_TISNUM);
3281  #line 2041 "vm_x86.dasc"
3282    //|  math_round ceil
3283    dasm_put(Dst, 3890, LJ_TISNUM);
3284  #line 2042 "vm_x86.dasc"
3285    //|
3286    //|.ffunc_nsse math_sqrt, sqrtsd; jmp ->fff_resxmm0
3287    //|
3288    //|.ffunc math_log
3289    //|  cmp NARGS:RD, 1+1; jne ->fff_fallback        // Exactly one argument.
3290    dasm_put(Dst, 3963, 1+1, LJ_TISNUM);
3291  #line 2047 "vm_x86.dasc"
3292    //|  cmp dword [BASE+4], LJ_TISNUM; jae ->fff_fallback
3293    //|  movsd xmm0, qword [BASE]
3294    //|.if not X64
3295    //|  movsd FPARG1, xmm0
3296    //|.endif
3297    //|  mov RB, BASE
3298    //|  call extern log
3299    //|  mov BASE, RB
3300    //|  jmp ->fff_resfp
3301    //|
3302    //|.macro math_extern, func
3303    //|  .ffunc_nsse math_ .. func
3304    //|.if not X64
```

```
3305   //|  movsd FPARG1, xmm0
3306   //|.endif
3307   //|  mov RB, BASE
3308   //|  call extern func
3309   //|  mov BASE, RB
3310   //|  jmp ->fff_resfp
3311   //|.endmacro
3312   //|
3313   //|.macro math_extern2, func
3314   //|  .ffunc_nnsse math_ .. func
3315   //|.if not X64
3316   //|  movsd FPARG1, xmm0
3317   //|  movsd FPARG3, xmm1
3318   //|.endif
3319   //|  mov RB, BASE
3320   //|  call extern func
3321   //|  mov BASE, RB
3322   //|  jmp ->fff_resfp
3323   //|.endmacro
3324   //|
3325   //|  math_extern log10
3326   //|  math_extern exp
3327   dasm_put(Dst, 4039, 1+1, LJ_TISNUM, 1+1, LJ_TISNUM, 1+1);
3328 #line 2082 "vm_x86.dasc"
3329   //|  math_extern sin
3330   //|  math_extern cos
3331   dasm_put(Dst, 4122, LJ_TISNUM, 1+1, LJ_TISNUM, 1+1, LJ_TISNUM);
3332 #line 2084 "vm_x86.dasc"
3333   //|  math_extern tan
3334   //|  math_extern asin
3335   //|  math_extern acos
3336   dasm_put(Dst, 4206, 1+1, LJ_TISNUM, 1+1, LJ_TISNUM);
3337 #line 2087 "vm_x86.dasc"
3338   //|  math_extern atan
3339   //|  math_extern sinh
3340   dasm_put(Dst, 4301, 1+1, LJ_TISNUM, 1+1, LJ_TISNUM, 1+1);
3341 #line 2089 "vm_x86.dasc"
3342   //|  math_extern cosh
3343   //|  math_extern tanh
3344   dasm_put(Dst, 4384, LJ_TISNUM, 1+1, LJ_TISNUM, 1+1, LJ_TISNUM);
3345 #line 2091 "vm_x86.dasc"
3346   //|  math_extern2 pow
3347   //|  math_extern2 atan2
3348   dasm_put(Dst, 4468, 2+1, LJ_TISNUM, LJ_TISNUM, 2+1, LJ_TISNUM);
3349 #line 2093 "vm_x86.dasc"
3350   //|  math_extern2 fmod
3351   //|
3352   //|.ffunc_nnr math_ldexp;        fscale; fpop1;        jmp ->fff_resn
3353   dasm_put(Dst, 4558, LJ_TISNUM, 2+1, LJ_TISNUM, LJ_TISNUM, 2+1);
3354 #line 2096 "vm_x86.dasc"
3355   //|
3356   //|.ffunc_1 math_frexp
3357   //|  mov RB, [BASE+4]
3358   //|  cmp RB, LJ_TISNUM; jae ->fff_fallback
3359   //|  mov PC, [BASE-4]
3360   //|  mov RC, [BASE]
3361   //|  mov [BASE-4], RB; mov [BASE-8], RC
3362   //|  shl RB, 1; cmp RB, 0xffe00000; jae >3
3363   //|  or RC, RB; jz >3
3364   //|  mov RC, 1022
3365   //|  cmp RB, 0x00200000; jb >4
3366   //|1:
3367   //|  shr RB, 21; sub RB, RC               // Extract and unbias exponent.
3368   dasm_put(Dst, 4654, LJ_TISNUM, LJ_TISNUM, 1+1, LJ_TISNUM);
3369 #line 2109 "vm_x86.dasc"
3370   //|  cvtsi2sd xmm0, RB
3371   //|  mov RB, [BASE-4]
3372   //|  and RB, 0x800fffff                    // Mask off exponent.
3373   //|  or RB, 0x3fe00000                     // Put mantissa in range [0.5,1) or 0.
3374   //|  mov [BASE-4], RB
3375   //|2:
3376   //|  movsd qword [BASE], xmm0
3377   //|  mov RD, 1+2
3378   //|  jmp ->fff_res
3379   //|3:  // Return +-0, +-Inf, NaN unmodified and an exponent of 0.
3380   //|  xorps xmm0, xmm0; jmp <2
```

```
3381    //|4:  // Handle denormals by multiplying with 2^54 and adjusting the bias.
3382    //|  movsd xmm0, qword [BASE]
3383    //|  sseconst_hi xmm1, RBa, 43500000  // 2^54.
3384    //|  mulsd xmm0, xmm1
3385    //|  movsd qword [BASE-8], xmm0
3386    //|  mov RB, [BASE-4]; mov RC, 1076; shl RB, 1; jmp <1
3387    //|
3388    //|.ffunc_nsse math_modf
3389    //|  mov RB, [BASE+4]
3390    //|  mov PC, [BASE-4]
3391    //|  shl RB, 1; cmp RB, 0xffe00000; je >4        // +-Inf?
3392    //|  movaps xmm4, xmm0
3393    dasm_put(Dst, 4761, 1+2, (unsigned int)(U64x(43500000,00000000)), (unsigned int)
((U64x(43500000,00000000))>>32), 1+1, LJ_TISNUM);
3394  #line 2132 "vm_x86.dasc"
3395    //|  call ->vm_trunc_sse
3396    //|  subsd xmm4, xmm0
3397    //|1:
3398    //|  movsd qword [BASE-8], xmm0
3399    //|  movsd qword [BASE], xmm4
3400    //|  mov RC, [BASE-4]; mov RB, [BASE+4]
3401    //|  xor RC, RB; js >3                            // Need to adjust sign?
3402    //|2:
3403    //|  mov RD, 1+2
3404    //|  jmp ->fff_res
3405    //|3:
3406    //|  xor RB, 0x80000000; mov [BASE+4], RB        // Flip sign of fraction.
3407    //|  jmp <2
3408    //|4:
3409    //|  xorps xmm4, xmm4; jmp <1                     // Return +-Inf and +-0.
3410    //|
3411    //|.macro math_minmax, name, cmovop, sseop
3412    //|  .ffunc name
3413    //|  mov RA, 2
3414    //|  cmp dword [BASE+4], LJ_TISNUM
3415    //|.if DUALNUM
3416    //|  jne >4
3417    //|  mov RB, dword [BASE]
3418    //|1:  // Handle integers.
3419    //|  cmp RA, RD; jae ->fff_resi
3420    //|  cmp dword [BASE+RA*8-4], LJ_TISNUM; jne >3
3421    //|  cmp RB, dword [BASE+RA*8-8]
3422    //|  cmovop RB, dword [BASE+RA*8-8]
3423    //|  add RA, 1
3424    //|  jmp <1
3425    //|3:
3426    //|  ja ->fff_fallback
3427    //|  // Convert intermediate result to number and continue below.
3428    //|  cvtsi2sd xmm0, RB
3429    //|  jmp >6
3430    //|4:
3431    //|  ja ->fff_fallback
3432    //|.else
3433    //|  jae ->fff_fallback
3434    //|.endif
3435    //|
3436    //|  movsd xmm0, qword [BASE]
3437    //|5:  // Handle numbers or integers.
3438    //|  cmp RA, RD; jae ->fff_resxmm0
3439    //|  cmp dword [BASE+RA*8-4], LJ_TISNUM
3440    //|.if DUALNUM
3441    //|  jb >6
3442    //|  ja ->fff_fallback
3443    //|  cvtsi2sd xmm1, dword [BASE+RA*8-8]
3444    //|  jmp >7
3445    //|.else
3446    //|  jae ->fff_fallback
3447    //|.endif
3448    //|6:
3449    //|  movsd xmm1, qword [BASE+RA*8-8]
3450    //|7:
3451    //|  sseop xmm0, xmm1
3452    //|  add RA, 1
3453    //|  jmp <5
3454    //|.endmacro
3455    //|
```

```
3456    //|  math_minmax math_min, cmovg, minsd
3457    dasm_put(Dst, 4905, 1+2, LJ_TISNUM);
3458    dasm_put(Dst, 4997, LJ_TISNUM, LJ_TISNUM);
3459  #line 2194 "vm_x86.dasc"
3460    //|  math_minmax math_max, cmovl, maxsd
3461    dasm_put(Dst, 5078, LJ_TISNUM, LJ_TISNUM);
3462  #line 2195 "vm_x86.dasc"
3463    //|
3464    //|//-- String library -----------------------------------------------------
3465    //|
3466    //|.ffunc string_byte                       // Only handle the 1-arg case here.
3467    //|  cmp NARGS:RD, 1+1;  jne ->fff_fallback
3468    dasm_put(Dst, 5176, LJ_TISNUM);
3469  #line 2200 "vm_x86.dasc"
3470    //|  cmp dword [BASE+4], LJ_TSTR;  jne ->fff_fallback
3471    //|  mov STR:RB, [BASE]
3472    //|  mov PC, [BASE-4]
3473    //|  cmp dword STR:RB->len, 1
3474    //|  jb ->fff_res0                        // Return no results for empty string.
3475    //|  movzx RB, byte STR:RB[1]
3476    //|.if DUALNUM
3477    //|  jmp ->fff_resi
3478    //|.else
3479    //|  cvtsi2sd xmm0, RB; jmp ->fff_resxmm0
3480    //|.endif
3481    //|
3482    //|.ffunc string_char                       // Only handle the 1-arg case here.
3483    //|  ffgccheck
3484    //|  cmp NARGS:RD, 1+1;  jne ->fff_fallback        // *Exactly* 1 arg.
3485    dasm_put(Dst, 5262, 1+1, LJ_TSTR, Dt5(->len), Dt5([1]), DISPATCH_GL(gc.total), DISPATCH_GL(gc.threshold));
3486  #line 2215 "vm_x86.dasc"
3487    //|  cmp dword [BASE+4], LJ_TISNUM
3488    //|.if DUALNUM
3489    //|  jne ->fff_fallback
3490    //|  mov RB, dword [BASE]
3491    //|  cmp RB, 255;  ja ->fff_fallback
3492    //|  mov TMP2, RB
3493    //|.else
3494    //|  jae ->fff_fallback
3495    //|  cvttsd2si RB, qword [BASE]
3496    //|  cmp RB, 255;  ja ->fff_fallback
3497    //|  mov TMP2, RB
3498    //|.endif
3499    //|.if X64
3500    //|  mov TMP3, 1
3501    //|.else
3502    //|  mov ARG3, 1
3503    //|.endif
3504    //|  lea RDa, TMP2                        // Points to stack. Little-endian.
3505    //|->fff_newstr:
3506    //|  mov L:RB, SAVE_L
3507    //|  mov L:RB->base, BASE
3508    //|.if X64
3509    //|  mov CARG3d, TMP3                       // Zero-extended to size_t.
3510    //|  mov CARG2, RDa                        // May be 64 bit ptr to stack.
3511    //|  mov CARG1d, L:RB
3512    //|.else
3513    //|  mov ARG2, RD
3514    //|  mov ARG1, L:RB
3515    //|.endif
3516    //|  mov SAVE_PC, PC
3517    //|  call extern lj_str_new               // (lua_State *L, char *str, size_t l)
3518    //|->fff_resstr:
3519    //|  // GCstr * returned in eax (RD).
3520    //|  mov BASE, L:RB->base
3521    //|  mov PC, [BASE-4]
3522    //|  mov dword [BASE-4], LJ_TSTR
3523    //|  mov [BASE-8], STR:RD
3524    //|  jmp ->fff_res1
3525    //|
3526    //|.ffunc string_sub
3527    //|  ffgccheck
3528    dasm_put(Dst, 5321, 1+1, LJ_TISNUM, Dt1(->base), Dt1(->base), LJ_TSTR, DISPATCH_GL(gc.total),
3529  DISPATCH_GL(gc.threshold));
3530  #line 2256 "vm_x86.dasc"
3531    //|  mov TMP2, -1
```

```
3531    //|  cmp NARGS:RD, 1+2;  jb ->fff_fallback
3532    //|  jna >1
3533    //|  cmp dword [BASE+20], LJ_TISNUM
3534    //|.if DUALNUM
3535    //|  jne ->fff_fallback
3536    //|  mov RB, dword [BASE+16]
3537    //|  mov TMP2, RB
3538    //|.else
3539    //|  jae ->fff_fallback
3540    //|  cvttsd2si RB, qword [BASE+16]
3541    //|  mov TMP2, RB
3542    //|.endif
3543    //|1:
3544    //|  cmp dword [BASE+4], LJ_TSTR;  jne ->fff_fallback
3545    //|  cmp dword [BASE+12], LJ_TISNUM
3546    //|.if DUALNUM
3547    //|  jne ->fff_fallback
3548    //|.else
3549    //|  jae ->fff_fallback
3550    //|.endif
3551    //|  mov STR:RB, [BASE]
3552    //|  mov TMP3, STR:RB
3553    //|  mov RB, STR:RB->len
3554    //|.if DUALNUM
3555    //|  mov RA, dword [BASE+8]
3556    //|.else
3557    //|  cvttsd2si RA, qword [BASE+8]
3558    //|.endif
3559    //|  mov RC, TMP2
3560    //|  cmp RB, RC                              // len < end? (unsigned compare)
3561    //|  jb >5
3562    //|2:
3563    //|  test RA, RA                         // start <= 0?
3564    dasm_put(Dst, 5433, 1+2, LJ_TISNUM, LJ_TSTR, LJ_TISNUM, Dt5(->len));
3565  #line 2290 "vm_x86.dasc"
3566    //|  jle >7
3567    //|3:
3568    //|  mov STR:RB, TMP3
3569    //|  sub RC, RA                              // start > end?
3570    //|  jl ->fff_emptystr
3571    //|  lea RB, [STR:RB+RA+#STR-1]
3572    //|  add RC, 1
3573    //|4:
3574    //|.if X64
3575    //|  mov TMP3, RC
3576    //|.else
3577    //|  mov ARG3, RC
3578    //|.endif
3579    //|  mov RD, RB
3580    //|  jmp ->fff_newstr
3581    //|
3582    //|5:  // Negative end or overflow.
3583    //|  jl >6
3584    //|  lea RC, [RC+RB+1]                      // end = end+(len+1)
3585    //|  jmp <2
3586    //|6:  // Overflow.
3587    //|  mov RC, RB                              // end = len
3588    //|  jmp <2
3589    //|
3590    //|7:  // Negative start or underflow.
3591    //|  je >8
3592    //|  add RA, RB                              // start = start+(len+1)
3593    dasm_put(Dst, 5523, sizeof(GCstr)-1);
3594  #line 2317 "vm_x86.dasc"
3595    //|  add RA, 1
3596    //|  jg <3                                // start > 0?
3597    //|8:  // Underflow.
3598    //|  mov RA, 1                               // start = 1
3599    //|  jmp <3
3600    //|
3601    //|->fff_emptystr:  // Range underflow.
3602    //|  xor RC, RC                              // Zero length. Any ptr in RB is ok.
3603    //|  jmp <4
3604    //|
3605    //|.macro ffstring_op, name
3606    //|  .ffunc_1 string_ .. name
```

```
3607    //|  ffgccheck
3608    //|  cmp dword [BASE+4], LJ_TSTR;  jne ->fff_fallback
3609    //|  mov L:RB, SAVE_L
3610    //|   lea SBUF:FCARG1, [DISPATCH+DISPATCH_GL(tmpbuf)]
3611    //|  mov L:RB->base, BASE
3612    //|  mov STR:FCARG2, [BASE]               // Caveat: FCARG2 == BASE
3613    //|   mov RC, SBUF:FCARG1->b
3614    //|   mov SBUF:FCARG1->L, L:RB
3615    //|   mov SBUF:FCARG1->p, RC
3616    //|  mov SAVE_PC, PC
3617    //|  call extern lj_buf_putstr_ .. name .. @8
3618    //|  mov FCARG1, eax
3619    //|  call extern lj_buf_tostr@4
3620    //|  jmp ->fff_resstr
3621    //|.endmacro
3622    //|
3623    //|ffstring_op reverse
3624    dasm_put(Dst, 5590, 1+1, DISPATCH_GL(gc.total), DISPATCH_GL(gc.threshold), LJ_TSTR);
3625  #line 2346 "vm_x86.dasc"
3626    //|ffstring_op lower
3627    dasm_put(Dst, 5652, DISPATCH_GL(tmpbuf), Dt1(->base), DtE(->b), DtE(->L), DtE(->p), 1+1,
DISPATCH_GL(gc.total), DISPATCH_GL(gc.threshold));
3628  #line 2347 "vm_x86.dasc"
3629    //|ffstring_op upper
3630    dasm_put(Dst, 5724, LJ_TSTR, DISPATCH_GL(tmpbuf), Dt1(->base), DtE(->b), DtE(->L), DtE(->p), 1+1,
DISPATCH_GL(gc.total), DISPATCH_GL(gc.threshold));
3631  #line 2348 "vm_x86.dasc"
3632    //|
3633    //|//-- Bit library ------------------------------------------------------
3634    //|
3635    //|.define TOBIT_BIAS, 0x59c00000        // 2^52 + 2^51 (float, not double!).
3636    //|
3637    //|.macro .ffunc_bit, name, kind
3638    //|  .ffunc_1 name
3639    //|.if kind == 2
3640    //|  sseconst_tobit xmm1, RBa
3641    //|.endif
3642    //|  cmp dword [BASE+4], LJ_TISNUM
3643    //|.if DUALNUM
3644    //|  jne >1
3645    //|  mov RB, dword [BASE]
3646    //|.if kind > 0
3647    //|  jmp >2
3648    //|.else
3649    //|  jmp ->fff_resbit
3650    //|.endif
3651    //|1:
3652    //|  ja ->fff_fallback
3653    //|.else
3654    //|  jae ->fff_fallback
3655    //|.endif
3656    //|  movsd xmm0, qword [BASE]
3657    //|.if kind < 2
3658    //|  sseconst_tobit xmm1, RBa
3659    //|.endif
3660    //|  addsd xmm0, xmm1
3661    //|  movd RB, xmm0
3662    //|2:
3663    //|.endmacro
3664    //|
3665    //|.ffunc_bit bit_tobit, 0
3666    dasm_put(Dst, 5801, LJ_TSTR, DISPATCH_GL(tmpbuf), Dt1(->base), DtE(->b), DtE(->L), DtE(->p), 1+1,
LJ_TISNUM);
3667  #line 2382 "vm_x86.dasc"
3668    //|  jmp ->fff_resbit
3669    //|
3670    //|.macro .ffunc_bit_op, name, ins
3671    //|  .ffunc_bit name, 2
3672    //|  mov TMP2, NARGS:RD                      // Save for fallback.
3673    //|  lea RD, [BASE+NARGS:RD*8-16]
3674    //|1:
3675    //|  cmp RD, BASE
3676    //|  jbe ->fff_resbit
3677    //|  cmp dword [RD+4], LJ_TISNUM
3678    //|.if DUALNUM
3679    //|  jne >2
```

```
3680    //|  ins RB, dword [RD]
3681    //|  sub RD, 8
3682    //|  jmp <1
3683    //|2:
3684    //|  ja ->fff_fallback_bit_op
3685    //|.else
3686    //|  jae ->fff_fallback_bit_op
3687    //|.endif
3688    //|  movsd xmm0, qword [RD]
3689    //|  addsd xmm0, xmm1
3690    //|  movd RA, xmm0
3691    //|  ins RB, RA
3692    //|  sub RD, 8
3693    //|  jmp <1
3694    //|.endmacro
3695    //|
3696    //|.ffunc_bit_op bit_band, and
3697    dasm_put(Dst, 5878, (unsigned int)(U64x(43380000,00000000)), (unsigned int)
((U64x(43380000,00000000))>>32), 1+1, (unsigned int)(U64x(43380000,00000000)), (unsigned int)
((U64x(43380000,00000000))>>32), LJ_TISNUM);
3698  #line 2411 "vm_x86.dasc"
3699    //|.ffunc_bit_op bit_bor, or
3700    dasm_put(Dst, 5952, LJ_TISNUM, 1+1, (unsigned int)(U64x(43380000,00000000)), (unsigned int)
((U64x(43380000,00000000))>>32));
3701    dasm_put(Dst, 6047, LJ_TISNUM, LJ_TISNUM);
3702  #line 2412 "vm_x86.dasc"
3703    //|.ffunc_bit_op bit_bxor, xor
3704    dasm_put(Dst, 6131, 1+1, (unsigned int)(U64x(43380000,00000000)), (unsigned int)
((U64x(43380000,00000000))>>32), LJ_TISNUM);
3705  #line 2413 "vm_x86.dasc"
3706    //|
3707    //|.ffunc_bit bit_bswap, 1
3708    dasm_put(Dst, 6228, LJ_TISNUM, 1+1, LJ_TISNUM);
3709  #line 2415 "vm_x86.dasc"
3710    //|  bswap RB
3711    //|  jmp ->fff_resbit
3712    //|
3713    //|.ffunc_bit bit_bnot, 1
3714    dasm_put(Dst, 6303, (unsigned int)(U64x(43380000,00000000)), (unsigned int)
((U64x(43380000,00000000))>>32), 1+1, LJ_TISNUM, (unsigned int)(U64x(43380000,00000000)), (unsigned int)
((U64x(43380000,00000000))>>32));
3715  #line 2419 "vm_x86.dasc"
3716    //|  not RB
3717    //|.if DUALNUM
3718    //|  jmp ->fff_resbit
3719    //|.else
3720    //|->fff_resbit:
3721    //|  cvtsi2sd xmm0, RB
3722    //|  jmp ->fff_resxmm0
3723    //|.endif
3724    //|
3725    //|->fff_fallback_bit_op:
3726    //|  mov NARGS:RD, TMP2                    // Restore for fallback
3727    //|  jmp ->fff_fallback
3728    //|
3729    //|.macro .ffunc_bit_sh, name, ins
3730    //|.if DUALNUM
3731    //|  .ffunc_bit name, 1
3732    //|  // Note: no inline conversion from number for 2nd argument!
3733    //|  cmp dword [BASE+12], LJ_TISNUM; jne ->fff_fallback
3734    //|  mov RA, dword [BASE+8]
3735    //|.else
3736    //|  .ffunc_nnsse name
3737    //|  sseconst_tobit xmm2, RBa
3738    //|  addsd xmm0, xmm2
3739    //|  addsd xmm1, xmm2
3740    //|  movd RB, xmm0
3741    //|  movd RA, xmm1
3742    //|.endif
3743    //|  ins RB, cl                            // Assumes RA is ecx.
3744    //|  jmp ->fff_resbit
3745    //|.endmacro
3746    //|
3747    //|.ffunc_bit_sh bit_lshift, shl
3748    dasm_put(Dst, 6379, 1+1, LJ_TISNUM, (unsigned int)(U64x(43380000,00000000)), (unsigned int)
((U64x(43380000,00000000))>>32));
```

```
3749  #line 2451 "vm_x86.dasc"
3750    //|.ffunc_bit_sh bit_rshift, shr
3751    dasm_put(Dst, 6453, LJ_TISNUM, 1+1, LJ_TISNUM, (unsigned int)(U64x(43380000,00000000)), (unsigned int)
((U64x(43380000,00000000))>>32));
3752  #line 2452 "vm_x86.dasc"
3753    //|.ffunc_bit_sh bit_arshift, sar
3754    dasm_put(Dst, 6528, LJ_TISNUM, 1+1, LJ_TISNUM, (unsigned int)(U64x(43380000,00000000)), (unsigned int)
((U64x(43380000,00000000))>>32));
3755  #line 2453 "vm_x86.dasc"
3756    //|.ffunc_bit_sh bit_rol, rol
3757    dasm_put(Dst, 6604, LJ_TISNUM, 1+1, LJ_TISNUM, (unsigned int)(U64x(43380000,00000000)), (unsigned int)
((U64x(43380000,00000000))>>32));
3758  #line 2454 "vm_x86.dasc"
3759    //|.ffunc_bit_sh bit_ror, ror
3760    dasm_put(Dst, 6680, LJ_TISNUM, 1+1, LJ_TISNUM, (unsigned int)(U64x(43380000,00000000)), (unsigned int)
((U64x(43380000,00000000))>>32));
3761  #line 2455 "vm_x86.dasc"
3762    //|
3763    //|//-----------------------------------------------------------------------
3764    //|
3765    //|->fff_fallback_2:
3766    //|  mov NARGS:RD, 1+2                        // Other args are ignored, anyway.
3767    //|  jmp ->fff_fallback
3768    //|->fff_fallback_1:
3769    //|  mov NARGS:RD, 1+1                        // Other args are ignored, anyway.
3770    //|->fff_fallback:                          // Call fast function fallback handler.
3771    //|  // BASE = new base, RD = nargs+1
3772    //|  mov L:RB, SAVE_L
3773    //|  mov PC, [BASE-4]                         // Fallback may overwrite PC.
3774    //|  mov SAVE_PC, PC                        // Redundant (but a defined value).
3775    //|  mov L:RB->base, BASE
3776    //|  lea RD, [BASE+NARGS:RD*8-8]
3777    //|  lea RA, [RD+8*LUA_MINSTACK]        // Ensure enough space for handler.
3778    //|  mov L:RB->top, RD
3779    //|  mov CFUNC:RD, [BASE-8]
3780    //|  cmp RA, L:RB->maxstack
3781    //|  ja >5                                  // Need to grow stack.
3782    //|.if X64
3783    //|  mov CARG1d, L:RB
3784    //|.else
3785    //|  mov ARG1, L:RB
3786    //|.endif
3787    //|  call aword CFUNC:RD->f                 // (lua_State *L)
3788    //|  mov BASE, L:RB->base
3789    dasm_put(Dst, 6755, LJ_TISNUM, 1+2, 1+1, Dt1(->base), 8*LUA_MINSTACK, Dt1(->top), Dt1(->maxstack), Dt8(-
>f));
3790  #line 2482 "vm_x86.dasc"
3791    //|  // Either throws an error, or recovers and returns -1, 0 or nresults+1.
3792    //|  test RD, RD;  jg ->fff_res               // Returned nresults+1?
3793    //|1:
3794    //|  mov RA, L:RB->top
3795    //|  sub RA, BASE
3796    //|  shr RA, 3
3797    //|  test RD, RD
3798    //|  lea NARGS:RD, [RA+1]
3799    //|  mov LFUNC:RB, [BASE-8]
3800    //|  jne ->vm_call_tail                       // Returned -1?
3801    //|  ins_callt                               // Returned 0: retry fast path.
3802    //|
3803    //|// Reconstruct previous base for vmeta_call during tailcall.
3804    //|->vm_call_tail:
3805    //|  mov RA, BASE
3806    //|  test PC, FRAME_TYPE
3807    //|  jnz >3
3808    //|  movzx RB, PC_RA
3809    //|  not RBa                                  // Note: ~RB = -(RB+1)
3810    //|  lea BASE, [BASE+RB*8]                  // base = base - (RB+1)*8
3811    //|  jmp ->vm_call_dispatch                  // Resolve again for tailcall.
3812    //|3:
3813    //|  mov RB, PC
3814    //|  and RB, -8
3815    //|  sub BASE, RB
3816    //|  jmp ->vm_call_dispatch                  // Resolve again for tailcall.
3817    //|
3818    //|5:  // Grow stack for fallback handler.
3819    //|  mov FCARG2, LUA_MINSTACK
```

```
3820    //|  mov FCARG1, L:RB
3821    //|  call extern lj_state_growstack@8        // (lua_State *L, int n)
3822    //|  mov BASE, L:RB->base
3823    //|  xor RD, RD                                // Simulate a return 0.
3824    dasm_put(Dst, 6848, Dt1(->base), Dt1(->top), Dt7(->pc), FRAME_TYPE, LUA_MINSTACK, Dt1(->base));
3825  #line 2515 "vm_x86.dasc"
3826    //|  jmp <1                                  // Dumb retry (goes through ff first).
3827    //|
3828    //|->fff_gcstep:                           // Call GC step function.
3829    //|  // BASE = new base, RD = nargs+1
3830    //|  pop RBa                                  // Must keep stack at same level.
3831    //|  mov TMPa, RBa                          // Save return address
3832    //|  mov L:RB, SAVE_L
3833    //|  mov SAVE_PC, PC                         // Redundant (but a defined value).
3834    //|  mov L:RB->base, BASE
3835    //|  lea RD, [BASE+NARGS:RD*8-8]
3836    //|  mov FCARG1, L:RB
3837    //|  mov L:RB->top, RD
3838    //|  call extern lj_gc_step@4                // (lua_State *L)
3839    //|  mov BASE, L:RB->base
3840    //|  mov RD, L:RB->top
3841    //|  sub RD, BASE
3842    //|  shr RD, 3
3843    //|  add NARGS:RD, 1
3844    //|  mov RBa, TMPa
3845    //|  push RBa                                 // Restore return address.
3846    //|  ret
3847    //|
3848    //|//-----------------------------------------------------------------------
3849    //|//-- Special dispatch targets -------------------------------------------
3850    //|//-----------------------------------------------------------------------
3851    //|
3852    //|->vm_record:                               // Dispatch target for recording phase.
3853    //|.if JIT
3854    //|  movzx RD, byte [DISPATCH+DISPATCH_GL(hookmask)]
3855    //|  test RDL, HOOK_VMEVENT                 // No recording while in vmevent.
3856    //|  jnz >5
3857    //|  // Decrement the hookcount for consistency, but always do the call.
3858    //|  test RDL, HOOK_ACTIVE
3859    //|  jnz >1
3860    //|  test RDL, LUA_MASKLINE|LUA_MASKCOUNT
3861    //|  jz >1
3862    //|  dec dword [DISPATCH+DISPATCH_GL(hookcount)]
3863    //|  jmp >1
3864    //|.endif
3865    //|
3866    //|->vm_rethook:                           // Dispatch target for return hooks.
3867    //|  movzx RD, byte [DISPATCH+DISPATCH_GL(hookmask)]
3868    dasm_put(Dst, 6961, Dt1(->base), Dt1(->top), Dt1(->base), Dt1(->top), DISPATCH_GL(hookmask), HOOK_VMEVENT,
HOOK_ACTIVE, LUA_MASKLINE|LUA_MASKCOUNT, DISPATCH_GL(hookcount));
3869  #line 2557 "vm_x86.dasc"
3870    //|  test RDL, HOOK_ACTIVE                 // Hook already active?
3871    //|  jnz >5
3872    //|  jmp >1
3873    //|
3874    //|->vm_inshook:                           // Dispatch target for instr/line hooks.
3875    //|  movzx RD, byte [DISPATCH+DISPATCH_GL(hookmask)]
3876    //|  test RDL, HOOK_ACTIVE                 // Hook already active?
3877    //|  jnz >5
3878    //|
3879    //|  test RDL, LUA_MASKLINE|LUA_MASKCOUNT
3880    //|  jz >5
3881    //|  dec dword [DISPATCH+DISPATCH_GL(hookcount)]
3882    //|  jz >1
3883    //|  test RDL, LUA_MASKLINE
3884    //|  jz >5
3885    //|1:
3886    //|  mov L:RB, SAVE_L
3887    dasm_put(Dst, 7059, DISPATCH_GL(hookmask), HOOK_ACTIVE, DISPATCH_GL(hookmask), HOOK_ACTIVE,
LUA_MASKLINE|LUA_MASKCOUNT, DISPATCH_GL(hookcount), LUA_MASKLINE);
3888  #line 2574 "vm_x86.dasc"
3889    //|  mov L:RB->base, BASE
3890    //|  mov FCARG2, PC                          // Caveat: FCARG2 == BASE
3891    //|  mov FCARG1, L:RB
3892    //|  // SAVE_PC must hold the _previous_ PC. The callee updates it with PC.
3893    //|  call extern lj_dispatch_ins@8        // (lua_State *L, const BCIns *pc)
```

```
3894    //|3:
3895    //|  mov BASE, L:RB->base
3896    //|4:
3897    //|  movzx RA, PC_RA
3898    //|5:
3899    //|  movzx OP, PC_OP
3900    //|  movzx RD, PC_RD
3901    //|.if X64
3902    //|  jmp aword [DISPATCH+OP*8+GG_DISP2STATIC]        // Re-dispatch to static ins.
3903    //|.else
3904    //|  jmp aword [DISPATCH+OP*4+GG_DISP2STATIC]        // Re-dispatch to static ins.
3905    //|.endif
3906    //|
3907    //|->cont_hook:                                  // Continue from hook yield.
3908    //|  add PC, 4
3909    //|  mov RA, [RB-24]
3910    //|  mov MULTRES, RA                          // Restore MULTRES for *M ins.
3911    //|  jmp <4
3912    //|
3913    //|->vm_hotloop:                           // Hot loop counter underflow.
3914    //|.if JIT
3915    //|  mov LFUNC:RB, [BASE-8]                  // Same as curr_topL(L).
3916    //|  mov RB, LFUNC:RB->pc
3917    //|  movzx RD, byte [RB+PC2PROTO(framesize)]
3918    //|  lea RD, [BASE+RD*8]
3919    //|  mov L:RB, SAVE_L
3920    //|  mov L:RB->base, BASE
3921    //|  mov L:RB->top, RD
3922    //|  mov FCARG2, PC
3923    //|  lea FCARG1, [DISPATCH+GG_DISP2J]
3924    //|  mov aword [DISPATCH+DISPATCH_J(L)], L:RBa
3925    //|  mov SAVE_PC, PC
3926    //|  call extern lj_trace_hot@8                // (jit_State *J, const BCIns *pc)
3927    //|  jmp <3
3928    //|.endif
3929    //|
3930    //|->vm_callhook:                          // Dispatch target for call hooks.
3931    //|  mov SAVE_PC, PC
3932    //|.if JIT
3933    //|  jmp >1
3934    //|.endif
3935    //|
3936    //|->vm_hotcall:                           // Hot call counter underflow.
3937    //|.if JIT
3938    //|  mov SAVE_PC, PC
3939    dasm_put(Dst, 7111, Dt1(->base), Dt1(->base), GG_DISP2STATIC, Dt7(->pc), PC2PROTO(framesize), Dt1(->base),
Dt1(->top), GG_DISP2J, DISPATCH_J(L));
3940  #line 2624 "vm_x86.dasc"
3941    //|  or PC, 1                                 // Marker for hot call.
3942    //|1:
3943    //|.endif
3944    //|  lea RD, [BASE+NARGS:RD*8-8]
3945    //|  mov L:RB, SAVE_L
3946    //|  mov L:RB->base, BASE
3947    //|  mov L:RB->top, RD
3948    //|  mov FCARG2, PC
3949    //|  mov FCARG1, L:RB
3950    //|  call extern lj_dispatch_call@8        // (lua_State *L, const BCIns *pc)
3951    //|  // ASMFunction returned in eax/rax (RDa).
3952    //|  mov SAVE_PC, 0                          // Invalidate for subsequent line hook.
3953    //|.if JIT
3954    //|  and PC, -2
3955    //|.endif
3956    //|  mov BASE, L:RB->base
3957    //|  mov RAa, RDa
3958    //|  mov RD, L:RB->top
3959    //|  sub RD, BASE
3960    //|  mov RBa, RAa
3961    //|  movzx RA, PC_RA
3962    //|  shr RD, 3
3963    //|  add NARGS:RD, 1
3964    //|  jmp RBa
3965    //|
3966    //|->cont_stitch:                          // Trace stitching.
3967    //|.if JIT
3968    //|  // BASE = base, RC = result, RB = mbase
```

```
3969    //|  mov RA, [RB-24]                        // Save previous trace number.
3970    //|  mov TMP1, RA
3971    //|  mov TMP3, DISPATCH                     // Need one more register.
3972    //|  mov DISPATCH, MULTRES
3973    //|  movzx RA, PC_RA
3974    //|  lea RA, [BASE+RA*8]                 // Call base.
3975    //|  sub DISPATCH, 1
3976    //|  jz >2
3977    //|1:  // Move results down.
3978    //|.if X64
3979    //|  mov RBa, [RC]
3980    //|  mov [RA], RBa
3981    //|.else
3982    //|  mov RB, [RC]
3983    //|  mov [RA], RB
3984    //|  mov RB, [RC+4]
3985    //|  mov [RA+4], RB
3986    //|.endif
3987    //|  add RC, 8
3988    //|  add RA, 8
3989    //|  sub DISPATCH, 1
3990    //|  jnz <1
3991    //|2:
3992    //|  movzx RC, PC_RA
3993    //|  movzx RB, PC_RB
3994    //|  add RC, RB
3995    //|  lea RC, [BASE+RC*8-8]
3996    //|3:
3997    //|  cmp RC, RA
3998    //|  ja >9                                // More results wanted?
3999    //|
4000    //|  mov DISPATCH, TMP3
4001    //|  mov RB, TMP1                        // Get previous trace number.
4002    //|  mov RA, [DISPATCH+DISPATCH_J(trace)]
4003    //|  mov TRACE:RD, [RA+RB*4]
4004    //|  test TRACE:RD, TRACE:RD
4005    //|  jz ->cont_nop
4006    //|  movzx RD, word TRACE:RD->link
4007    //|  cmp RD, RB
4008    //|  je ->cont_nop                        // Blacklisted.
4009    //|  test RD, RD
4010    dasm_put(Dst, 7236, Dt1(->base), Dt1(->top), Dt1(->base), Dt1(->top), DISPATCH_J(trace), DtD(->link));
4011  #line 2693 "vm_x86.dasc"
4012    //|  jne =>BC_JLOOP                       // Jump to stitched trace.
4013    //|
4014    //|  // Stitch a new trace to the previous trace.
4015    //|  mov [DISPATCH+DISPATCH_J(exitno)], RB
4016    //|  mov L:RB, SAVE_L
4017    //|  mov L:RB->base, BASE
4018    //|  mov FCARG2, PC
4019    //|  lea FCARG1, [DISPATCH+GG_DISP2J]
4020    //|  mov aword [DISPATCH+DISPATCH_J(L)], L:RBa
4021    //|  call extern lj_dispatch_stitch@8        // (jit_State *J, const BCIns *pc)
4022    //|  mov BASE, L:RB->base
4023    //|  jmp ->cont_nop
4024    //|
4025    //|9:  // Fill up results with nil.
4026    //|  mov dword [RA+4], LJ_TNIL
4027    //|  add RA, 8
4028    //|  jmp <3
4029    //|.endif
4030    //|
4031    //|->vm_profhook:                         // Dispatch target for profiler hook.
4032    dasm_put(Dst, 7426, BC_JLOOP, DISPATCH_J(exitno), Dt1(->base), GG_DISP2J, DISPATCH_J(L), Dt1(->base),
LJ_TNIL);
4033  #line 2713 "vm_x86.dasc"
4034  #if LJ_HASPROFILE
4035    //|  mov L:RB, SAVE_L
4036    //|  mov L:RB->base, BASE
4037    //|  mov FCARG2, PC                       // Caveat: FCARG2 == BASE
4038    //|  mov FCARG1, L:RB
4039    //|  call extern lj_dispatch_profile@8        // (lua_State *L, const BCIns *pc)
4040    //|  mov BASE, L:RB->base
4041    //|  // HOOK_PROFILE is off again, so re-dispatch to dynamic instruction.
4042    //|  sub PC, 4
4043    //|  jmp ->cont_nop
```

```
4044    dasm_put(Dst, 7479, Dt1(->base), Dt1(->base));
4045  #line 2723 "vm_x86.dasc"
4046  #endif
4047    //|
4048    //|//-----------------------------------------------------------------------
4049    //|//-- Trace exit handler -------------------------------------------------
4050    //|//-----------------------------------------------------------------------
4051    //|
4052    //|// Called from an exit stub with the exit number on the stack.
4053    //|// The 16 bit exit number is stored with two (sign-extended) push imm8.
4054    //|->vm_exit_handler:
4055    //|.if JIT
4056    //|.if X64
4057    //|  push r13; push r12
4058    //|  push r11; push r10; push r9; push r8
4059    //|  push rdi; push rsi; push rbp; lea rbp, [rsp+88]; push rbp
4060    //|  push rbx; push rdx; push rcx; push rax
4061    //|  movzx RC, byte [rbp-8]               // Reconstruct exit number.
4062    //|  mov RCH, byte [rbp-16]
4063    //|  mov [rbp-8], r15; mov [rbp-16], r14
4064    //|.else
4065    //|  push ebp; lea ebp, [esp+12]; push ebp
4066    //|  push ebx; push edx; push ecx; push eax
4067    //|  movzx RC, byte [ebp-4]               // Reconstruct exit number.
4068    //|  mov RCH, byte [ebp-8]
4069    //|  mov [ebp-4], edi; mov [ebp-8], esi
4070    //|.endif
4071    //|  // Caveat: DISPATCH is ebx.
4072    //|  mov DISPATCH, [ebp]
4073    //|  mov RA, [DISPATCH+DISPATCH_GL(vmstate)]        // Get trace number.
4074    //|  set_vmstate EXIT
4075    //|  mov [DISPATCH+DISPATCH_J(exitno)], RC
4076    //|  mov [DISPATCH+DISPATCH_J(parent)], RA
4077    //|.if X64
4078    //|.if X64WIN
4079    //|  sub rsp, 16*8+4*8                      // Room for SSE regs + save area.
4080    //|.else
4081    //|  sub rsp, 16*8                        // Room for SSE regs.
4082    //|.endif
4083    //|  add rbp, -128
4084    //|  movsd qword [rbp-8],   xmm15; movsd qword [rbp-16],  xmm14
4085    //|  movsd qword [rbp-24],  xmm13; movsd qword [rbp-32],  xmm12
4086    //|  movsd qword [rbp-40],  xmm11; movsd qword [rbp-48],  xmm10
4087    //|  movsd qword [rbp-56],  xmm9;  movsd qword [rbp-64],  xmm8
4088    //|  movsd qword [rbp-72],  xmm7;  movsd qword [rbp-80],  xmm6
4089    //|  movsd qword [rbp-88],  xmm5;  movsd qword [rbp-96],  xmm4
4090    //|  movsd qword [rbp-104], xmm3;  movsd qword [rbp-112], xmm2
4091    //|  movsd qword [rbp-120], xmm1;  movsd qword [rbp-128], xmm0
4092    //|.else
4093    //|  sub esp, 8*8+16                        // Room for SSE regs + args.
4094    //|  movsd qword [ebp-40], xmm7; movsd qword [ebp-48], xmm6
4095    //|  movsd qword [ebp-56], xmm5; movsd qword [ebp-64], xmm4
4096    //|  movsd qword [ebp-72], xmm3; movsd qword [ebp-80], xmm2
4097    //|  movsd qword [ebp-88], xmm1; movsd qword [ebp-96], xmm0
4098    //|.endif
4099    //|  // Caveat: RB is ebp.
4100    //|  mov L:RB, [DISPATCH+DISPATCH_GL(cur_L)]
4101    //|  mov BASE, [DISPATCH+DISPATCH_GL(jit_base)]
4102    //|  mov aword [DISPATCH+DISPATCH_J(L)], L:RBa
4103    //|  mov L:RB->base, BASE
4104    //|.if X64WIN
4105    //|  lea CARG2, [rsp+4*8]
4106    //|.elif X64
4107    //|  mov CARG2, rsp
4108    //|.else
4109    //|  lea FCARG2, [esp+16]
4110    //|.endif
4111    //|  lea FCARG1, [DISPATCH+GG_DISP2J]
4112    //|  mov dword [DISPATCH+DISPATCH_GL(jit_base)], 0
4113    //|  call extern lj_trace_exit@8        // (jit_State *J, ExitState *ex)
4114    //|  // MULTRES or negated error code returned in eax (RD).
4115    //|  mov RAa, L:RB->cframe
4116    //|  and RAa, CFRAME_RAWMASK
4117    //|.if X64WIN
4118    //|  // Reposition stack later.
4119    //|.elif X64
```

```
4120    //|  mov rsp, RAa                         // Reposition stack to C frame.
4121    //|.else
4122    //|  mov esp, RAa                         // Reposition stack to C frame.
4123    //|.endif
4124    //|  mov [RAa+CFRAME_OFS_L], L:RB         // Set SAVE_L (on-trace resume/yield).
4125    //|  mov BASE, L:RB->base
4126    //|  mov PC, [RAa+CFRAME_OFS_PC]         // Get SAVE_PC.
4127    //|.if X64
4128    //|  jmp >1
4129    //|.endif
4130    //|.endif
4131    //|->vm_exit_interp:
4132    //|  // RD = MULTRES or negated error code, BASE, PC and DISPATCH set.
4133    //|.if JIT
4134    //|.if X64
4135    //|  // Restore additional callee-save registers only used in compiled code.
4136    //|.if X64WIN
4137    //|  lea RAa, [rsp+9*16+4*8]
4138    //|1:
4139    //|  movdqa xmm15, [RAa-9*16]
4140    //|  movdqa xmm14, [RAa-8*16]
4141    //|  movdqa xmm13, [RAa-7*16]
4142    //|  movdqa xmm12, [RAa-6*16]
4143    //|  movdqa xmm11, [RAa-5*16]
4144    //|  movdqa xmm10, [RAa-4*16]
4145    //|  movdqa xmm9, [RAa-3*16]
4146    //|  movdqa xmm8, [RAa-2*16]
4147    //|  movdqa xmm7, [RAa-1*16]
4148    //|  mov rsp, RAa                         // Reposition stack to C frame.
4149    //|  movdqa xmm6, [RAa]
4150    //|  mov r15, CSAVE_3
4151    //|  mov r14, CSAVE_4
4152    //|.else
4153    //|  add rsp, 16                          // Reposition stack to C frame.
4154    dasm_put(Dst, 7507, DISPATCH_GL(vmstate), DISPATCH_GL(vmstate), ~LJ_VMST_EXIT, DISPATCH_J(exitno),
DISPATCH_J(parent), 16*8, DISPATCH_GL(cur_L), DISPATCH_GL(jit_base), DISPATCH_J(L), Dt1(->base), GG_DISP2J,
DISPATCH_GL(jit_base), Dt1(->cframe), CFRAME_RAWMASK, CFRAME_OFS_L, Dt1(->base), CFRAME_OFS_PC);
4155  #line 2831 "vm_x86.dasc"
4156    //|1:
4157    //|.endif
4158    //|  mov r13, TMPa
4159    //|  mov r12, TMPQ
4160    //|.endif
4161    dasm_put(Dst, 7750);
4162  #line 2836 "vm_x86.dasc"
4163  #ifdef LUA_USE_TRACE_LOGS
4164    //|.if X64
4165    //|  mov FCARG1, SAVE_L
4166    //|  mov L:FCARG1->base, BASE
4167    //|  mov RB, RD     // Save RD
4168    //|  mov TMP1, PC  // Save PC
4169    //|  mov CARG3d, PC   // CARG3d == BASE
4170    //|  mov FCARG2, dword [DISPATCH+DISPATCH_GL(vmstate)]
4171    //|  call extern lj_log_trace_direct_exit@8
4172    //|  mov PC, TMP1
4173    //|  mov RD, RB
4174    //|  mov RB, SAVE_L
4175    //|  mov BASE, L:RB->base
4176    //|.endif
4177    dasm_put(Dst, 7766, Dt1(->base), DISPATCH_GL(vmstate), Dt1(->base));
4178  #line 2850 "vm_x86.dasc"
4179  #endif
4180    //|  test RD, RD; js >9                       // Check for error from exit.
4181    //|  mov L:RB, SAVE_L
4182    //|  mov MULTRES, RD
4183    //|  mov LFUNC:KBASE, [BASE-8]
4184    //|  mov KBASE, LFUNC:KBASE->pc
4185    //|  mov KBASE, [KBASE+PC2PROTO(k)]
4186    //|  mov L:RB->base, BASE
4187    //|  mov dword [DISPATCH+DISPATCH_GL(jit_base)], 0
4188    //|  set_vmstate INTERP
4189    //|  // Modified copy of ins_next which handles function header dispatch, too.
4190    //|  mov RC, [PC]
4191    //|  movzx RA, RCH
4192    //|  movzx OP, RCL
4193    //|  add PC, 4
```

```
4194    //|  shr RC, 16
4195    //|  cmp OP, BC_FUNCF                        // Function header?
4196    //|  jb >3
4197    //|  cmp OP, BC_FUNCC+2                        // Fast function?
4198    //|  jae >4
4199    //|2:
4200    //|  mov RC, MULTRES                        // RC/RD holds nres+1.
4201    //|3:
4202    //|.if X64
4203    //|  jmp aword [DISPATCH+OP*8]
4204    //|.else
4205    //|  jmp aword [DISPATCH+OP*4]
4206    //|.endif
4207    //|
4208    //|4:  // Check frame below fast function.
4209    //|  mov RC, [BASE-4]
4210    //|  test RC, FRAME_TYPE
4211    //|  jnz <2                                 // Trace stitching continuation?
4212    //|  // Otherwise set KBASE for Lua function below fast function.
4213    //|  movzx RC, byte [RC-3]
4214    dasm_put(Dst, 7801, Dt7(->pc), PC2PROTO(k), Dt1(->base), DISPATCH_GL(jit_base), DISPATCH_GL(vmstate),
~LJ_VMST_INTERP, BC_FUNCF, BC_FUNCC+2, FRAME_TYPE);
4215  #line 2885 "vm_x86.dasc"
4216    //|  not RCa
4217    //|  mov LFUNC:KBASE, [BASE+RC*8-8]
4218    //|  mov KBASE, LFUNC:KBASE->pc
4219    //|  mov KBASE, [KBASE+PC2PROTO(k)]
4220    //|  jmp <2
4221    //|
4222    //|9:  // Rethrow error from the right C frame.
4223    //|  neg RD
4224    //|  mov FCARG1, L:RB
4225    //|  mov FCARG2, RD
4226    //|  call extern lj_err_throw@8                // (lua_State *L, int errcode)
4227    //|.endif
4228    //|
4229    //|//-----------------------------------------------------------------------
4230    //|//-- Math helper functions ----------------------------------------------
4231    //|//-----------------------------------------------------------------------
4232    //|
4233    //|// FP value rounding. Called by math.floor/math.ceil fast functions
4234    //|// and from JIT code. arg/ret is xmm0. xmm0-xmm3 and RD (eax) modified.
4235    //|.macro vm_round, name, mode, cond
4236    //|->name:
4237    //|.if not X64 and cond
4238    //|  movsd xmm0, qword [esp+4]
4239    //|  call ->name .. _sse
4240    //|  movsd qword [esp+4], xmm0  // Overwrite callee-owned arg.
4241    //|  fld qword [esp+4]
4242    //|  ret
4243    //|.endif
4244    //|
4245    //|->name .. _sse:
4246    //|  sseconst_abs xmm2, RDa
4247    //|  sseconst_2p52 xmm3, RDa
4248    //|  movaps xmm1, xmm0
4249    //|  andpd xmm1, xmm2                        // |x|
4250    //|  ucomisd xmm3, xmm1                        // No truncation if 2^52 <= |x|.
4251    //|  jbe >1
4252    //|  andnpd xmm2, xmm0                         // Isolate sign bit.
4253    //|.if mode == 2                  // trunc(x)?
4254    //|  movaps xmm0, xmm1
4255    //|  addsd xmm1, xmm3                        // (|x| + 2^52) - 2^52
4256    //|  subsd xmm1, xmm3
4257    //|  sseconst_1 xmm3, RDa
4258    //|  cmpsd xmm0, xmm1, 1                // |x| < result?
4259    //|  andpd xmm0, xmm3
4260    //|  subsd xmm1, xmm0                        // If yes, subtract -1.
4261    //|  orpd xmm1, xmm2                        // Merge sign bit back in.
4262    //|.else
4263    //|  addsd xmm1, xmm3                        // (|x| + 2^52) - 2^52
4264    //|  subsd xmm1, xmm3
4265    //|  orpd xmm1, xmm2                        // Merge sign bit back in.
4266    //|  .if mode == 1                // ceil(x)?
4267    //|    sseconst_m1 xmm2, RDa                // Must subtract -1 to preserve -0.
4268    //|    cmpsd xmm0, xmm1, 6                // x > result?
```

```
4269    //|  .else                        // floor(x)?
4270    //|    sseconst_1 xmm2, RDa
4271    //|    cmpsd xmm0, xmm1, 1               // x < result?
4272    //|  .endif
4273    //|  andpd xmm0, xmm2
4274    //|  subsd xmm1, xmm0                      // If yes, subtract +-1.
4275    //|.endif
4276    //|  movaps xmm0, xmm1
4277    //|1:
4278    //|  ret
4279    //|.endmacro
4280    //|
4281    //|  vm_round vm_floor, 0, 1
4282    //|  vm_round vm_ceil,  1, JIT
4283    dasm_put(Dst, 7901, Dt7(->pc), PC2PROTO(k), (unsigned int)(U64x(7fffffff,ffffffff)), (unsigned int)
((U64x(7fffffff,ffffffff))>>32), (unsigned int)(U64x(43300000,00000000)), (unsigned int)
((U64x(43300000,00000000))>>32), (unsigned int)(U64x(3ff00000,00000000)), (unsigned int)
((U64x(3ff00000,00000000))>>32), (unsigned int)(U64x(7fffffff,ffffffff)), (unsigned int)
((U64x(7fffffff,ffffffff))>>32));
4284  #line 2952 "vm_x86.dasc"
4285    //|  vm_round vm_trunc, 2, JIT
4286    //|
4287    //|// FP modulo x%y. Called by BC_MOD* and vm_arith.
4288    //|->vm_mod:
4289    //|// Args in xmm0/xmm1, return value in xmm0.
4290    //|// Caveat: xmm0-xmm5 and RC (eax) modified!
4291    //|  movaps xmm5, xmm0
4292    //|  divsd xmm0, xmm1
4293    //|  sseconst_abs xmm2, RDa
4294    dasm_put(Dst, 8036, (unsigned int)(U64x(43300000,00000000)), (unsigned int)
((U64x(43300000,00000000))>>32), (unsigned int)(U64x(bff00000,00000000)), (unsigned int)
((U64x(bff00000,00000000))>>32), (unsigned int)(U64x(7fffffff,ffffffff)), (unsigned int)
((U64x(7fffffff,ffffffff))>>32), (unsigned int)(U64x(43300000,00000000)), (unsigned int)
((U64x(43300000,00000000))>>32), (unsigned int)(U64x(3ff00000,00000000)), (unsigned int)
((U64x(3ff00000,00000000))>>32), (unsigned int)(U64x(7fffffff,ffffffff)), (unsigned int)
((U64x(7fffffff,ffffffff))>>32));
4295  #line 2961 "vm_x86.dasc"
4296    //|  sseconst_2p52 xmm3, RDa
4297    //|  movaps xmm4, xmm0
4298    //|  andpd xmm4, xmm2                      // |x/y|
4299    //|  ucomisd xmm3, xmm4                      // No truncation if 2^52 <= |x/y|.
4300    //|  jbe >1
4301    //|  andnpd xmm2, xmm0                       // Isolate sign bit.
4302    //|  addsd xmm4, xmm3                      // (|x/y| + 2^52) - 2^52
4303    //|  subsd xmm4, xmm3
4304    //|  orpd xmm4, xmm2                      // Merge sign bit back in.
4305    //|  sseconst_1 xmm2, RDa
4306    //|  cmpsd xmm0, xmm4, 1                // x/y < result?
4307    //|  andpd xmm0, xmm2
4308    //|  subsd xmm4, xmm0                      // If yes, subtract 1.0.
4309    //|  movaps xmm0, xmm5
4310    //|  mulsd xmm1, xmm4
4311    //|  subsd xmm0, xmm1
4312    //|  ret
4313    //|1:
4314    //|  mulsd xmm1, xmm0
4315    //|  movaps xmm0, xmm5
4316    //|  subsd xmm0, xmm1
4317    //|  ret
4318    //|
4319    //|// Args in xmm0/eax. Ret in xmm0. xmm0-xmm1 and eax modified.
4320    //|->vm_powi_sse:
4321    //|  cmp eax, 1; jle >6                       // i<=1?
4322    //|  // Now 1 < (unsigned)i <= 0x80000000.
4323    //|1:  // Handle leading zeros.
4324    //|  test eax, 1; jnz >2
4325    //|  mulsd xmm0, xmm0
4326    //|  shr eax, 1
4327    //|  jmp <1
4328    //|2:
4329    //|  shr eax, 1; jz >5
4330    //|  movaps xmm1, xmm0
4331    //|3:  // Handle trailing bits.
4332    //|  mulsd xmm0, xmm0
4333    //|  shr eax, 1; jz >4
4334    //|  jnc <3
```

```
4335    dasm_put(Dst, 8216, (unsigned int)(U64x(43300000,00000000)), (unsigned int)
((U64x(43300000,00000000))>>32), (unsigned int)(U64x(3ff00000,00000000)), (unsigned int)
((U64x(3ff00000,00000000))>>32));
4336  #line 3000 "vm_x86.dasc"
4337    //|  mulsd xmm1, xmm0
4338    //|  jmp <3
4339    //|4:
4340    //|  mulsd xmm0, xmm1
4341    //|5:
4342    //|  ret
4343    //|6:
4344    //|  je <5                                  // x^1 ==> x
4345    //|  jb >7                                  // x^0 ==> 1
4346    //|  neg eax
4347    //|  call <1
4348    //|  sseconst_1 xmm1, RDa
4349    //|  divsd xmm1, xmm0
4350    //|  movaps xmm0, xmm1
4351    //|  ret
4352    //|7:
4353    //|  sseconst_1 xmm0, RDa
4354    //|  ret
4355    //|
4356    //|//-----------------------------------------------------------------------
4357    //|//-- Miscellaneous functions -------------------------------------------
4358    //|//-----------------------------------------------------------------------
4359    //|
4360    //|// int lj_vm_cpuid(uint32_t f, uint32_t res[4])
4361    //|->vm_cpuid:
4362    //|.if X64
4363    //|  mov eax, CARG1d
4364    //|  .if X64WIN; push rsi; mov rsi, CARG2; .endif
4365    //|  push rbx
4366    //|  cpuid
4367    //|  mov [rsi], eax
4368    //|  mov [rsi+4], ebx
4369    //|  mov [rsi+8], ecx
4370    //|  mov [rsi+12], edx
4371    //|  pop rbx
4372    //|  .if X64WIN; pop rsi; .endif
4373    //|  ret
4374    //|.else
4375    //|  pushfd
4376    //|  pop edx
4377    //|  mov ecx, edx
4378    //|  xor edx, 0x00200000                // Toggle ID bit in flags.
4379    //|  push edx
4380    //|  popfd
4381    //|  pushfd
4382    //|  pop edx
4383    //|  xor eax, eax                        // Zero means no features supported.
4384    //|  cmp ecx, edx
4385    //|  jz >1                               // No ID toggle means no CPUID support.
4386    //|  mov eax, [esp+4]                      // Argument 1 is function number.
4387    //|  push edi
4388    //|  push ebx
4389    //|  cpuid
4390    //|  mov edi, [esp+16]                      // Argument 2 is result area.
4391    //|  mov [edi], eax
4392    //|  mov [edi+4], ebx
4393    //|  mov [edi+8], ecx
4394    //|  mov [edi+12], edx
4395    //|  pop ebx
4396    //|  pop edi
4397    //|1:
4398    //|  ret
4399    //|.endif
4400    //|
4401    //|//-----------------------------------------------------------------------
4402    //|//-- Assertions ---------------------------------------------------------
4403    //|//-----------------------------------------------------------------------
4404    //|
4405    //|->assert_bad_for_arg_type:
4406    dasm_put(Dst, 8374, (unsigned int)(U64x(3ff00000,00000000)), (unsigned int)
((U64x(3ff00000,00000000))>>32), (unsigned int)(U64x(3ff00000,00000000)), (unsigned int)
((U64x(3ff00000,00000000))>>32));
```

```
4407  #line 3069 "vm_x86.dasc"
4408  #ifdef LUA_USE_ASSERT
4409    //|  int3
4410    dasm_put(Dst, 8467);
4411  #line 3071 "vm_x86.dasc"
4412  #endif
4413    //|  int3
4414    //|
4415    //|//-----------------------------------------------------------------------
4416    //|//-- FFI helper functions ---------------------------------------------
4417    //|//-----------------------------------------------------------------------
4418    //|
4419    //|// Handler for callback functions. Callback slot number in ah/al.
4420    //|->vm_ffi_callback:
4421    //|.if FFI
4422    //|.type CTSTATE, CTState, PC
4423  #define DtF(_V) (int)(ptrdiff_t)&(((CTState *)0)_V)
4424  #line 3082 "vm_x86.dasc"
4425    //|.if not X64
4426    //|  sub esp, 16                        // Leave room for SAVE_ERRF etc.
4427    //|.endif
4428    //|  saveregs_        // ebp/rbp already saved. ebp now holds global_State *.
4429    //|  lea DISPATCH, [ebp+GG_G2DISP]
4430    //|  mov CTSTATE, GL:ebp->ctype_state
4431    //|  movzx eax, ax
4432    //|  mov CTSTATE->cb.slot, eax
4433    //|.if X64
4434    //|  mov CTSTATE->cb.gpr[0], CARG1
4435    //|  mov CTSTATE->cb.gpr[1], CARG2
4436    //|  mov CTSTATE->cb.gpr[2], CARG3
4437    //|  mov CTSTATE->cb.gpr[3], CARG4
4438    //|  movsd qword CTSTATE->cb.fpr[0], xmm0
4439    //|  movsd qword CTSTATE->cb.fpr[1], xmm1
4440    //|  movsd qword CTSTATE->cb.fpr[2], xmm2
4441    //|  movsd qword CTSTATE->cb.fpr[3], xmm3
4442    //|.if X64WIN
4443    //|  lea rax, [rsp+CFRAME_SIZE+4*8]
4444    //|.else
4445    //|  lea rax, [rsp+CFRAME_SIZE]
4446    //|  mov CTSTATE->cb.gpr[4], CARG5
4447    //|  mov CTSTATE->cb.gpr[5], CARG6
4448    //|  movsd qword CTSTATE->cb.fpr[4], xmm4
4449    //|  movsd qword CTSTATE->cb.fpr[5], xmm5
4450    //|  movsd qword CTSTATE->cb.fpr[6], xmm6
4451    //|  movsd qword CTSTATE->cb.fpr[7], xmm7
4452    //|.endif
4453    //|  mov CTSTATE->cb.stack, rax
4454    //|  mov CARG2, rsp
4455    dasm_put(Dst, 8469, GG_G2DISP, Dt2(->ctype_state), DtF(->cb.slot), DtF(->cb.gpr[0]), DtF(->cb.gpr[1]),
DtF(->cb.gpr[2]), DtF(->cb.gpr[3]), DtF(->cb.fpr[0]), DtF(->cb.fpr[1]), DtF(->cb.fpr[2]), DtF(->cb.fpr[3]),
CFRAME_SIZE, DtF(->cb.gpr[4]), DtF(->cb.gpr[5]), DtF(->cb.fpr[4]), DtF(->cb.fpr[5]), DtF(->cb.fpr[6]), DtF(-
>cb.fpr[7]), DtF(->cb.stack));
4456  #line 3112 "vm_x86.dasc"
4457    //|.else
4458    //|  lea eax, [esp+CFRAME_SIZE+16]
4459    //|  mov CTSTATE->cb.gpr[0], FCARG1
4460    //|  mov CTSTATE->cb.gpr[1], FCARG2
4461    //|  mov CTSTATE->cb.stack, eax
4462    //|  mov FCARG1, [esp+CFRAME_SIZE+12]        // Move around misplaced retaddr/ebp.
4463    //|  mov FCARG2, [esp+CFRAME_SIZE+8]
4464    //|  mov SAVE_RET, FCARG1
4465    //|  mov SAVE_R4, FCARG2
4466    //|  mov FCARG2, esp
4467    //|.endif
4468    //|  mov SAVE_PC, CTSTATE                // Any value outside of bytecode is ok.
4469    //|  mov FCARG1, CTSTATE
4470    //|  call extern lj_ccallback_enter@8        // (CTState *cts, void *cf)
4471    //|  // lua_State * returned in eax (RD).
4472    //|  set_vmstate INTERP
4473    //|  mov BASE, L:RD->base
4474    //|  mov RD, L:RD->top
4475    //|  sub RD, BASE
4476    //|  mov LFUNC:RB, [BASE-8]
4477    //|  shr RD, 3
4478    //|  add RD, 1
4479    //|  ins_callt
```

```
4480    //|.endif
4481    //|
4482    //|->cont_ffi_callback:                        // Return from FFI callback.
4483    //|.if FFI
4484    //|  mov L:RA, SAVE_L
4485    //|  mov CTSTATE, [DISPATCH+DISPATCH_GL(ctype_state)]
4486    //|  mov aword CTSTATE->L, L:RAa
4487    //|  mov L:RA->base, BASE
4488    //|  mov L:RA->top, RB
4489    //|  mov FCARG1, CTSTATE
4490    //|  mov FCARG2, RC
4491    //|  call extern lj_ccallback_leave@8        // (CTState *cts, TValue *o)
4492    //|.if X64
4493    //|  mov rax, CTSTATE->cb.gpr[0]
4494    //|  movsd xmm0, qword CTSTATE->cb.fpr[0]
4495    //|  jmp ->vm_leave_unw
4496    //|.else
4497    //|  mov L:RB, SAVE_L
4498    //|  mov eax, CTSTATE->cb.gpr[0]
4499    //|  mov edx, CTSTATE->cb.gpr[1]
4500    //|  cmp dword CTSTATE->cb.gpr[2], 1
4501    //|  jb >7
4502    //|  je >6
4503    //|  fld qword CTSTATE->cb.fpr[0].d
4504    //|  jmp >7
4505    //|6:
4506    //|  fld dword CTSTATE->cb.fpr[0].f
4507    //|7:
4508    //|  mov ecx, L:RB->top
4509    //|  movzx ecx, word [ecx+6]               // Get stack adjustment and copy up.
4510    //|  mov SAVE_L, ecx                       // Must be one slot above SAVE_RET
4511    //|  restoreregs
4512    //|  pop ecx                               // Move return addr from SAVE_RET.
4513    //|  add esp, [esp]                       // Adjust stack.
4514    //|  add esp, 16
4515    //|  push ecx
4516    //|  ret
4517    //|.endif
4518    //|.endif
4519    //|
4520    //|->vm_ffi_call@4:                       // Call C function via FFI.
4521    //|  // Caveat: needs special frame unwinding, see below.
4522    //|.if FFI
4523    //|.if X64
4524    //|  .type CCSTATE, CCallState, rbx
4525  #define Dt10(_V) (int)(ptrdiff_t)&(((CCallState *)0)_V)
4526  #line 3180 "vm_x86.dasc"
4527    //|  push rbp; mov rbp, rsp; push rbx; mov CCSTATE, CARG1
4528    //|.else
4529    //|  .type CCSTATE, CCallState, ebx
4530    //|  push ebp; mov ebp, esp; push ebx; mov CCSTATE, FCARG1
4531    //|.endif
4532    //|
4533    //|  // Readjust stack.
4534    //|.if X64
4535    //|  mov eax, CCSTATE->spadj
4536    //|  sub rsp, rax
4537    //|.else
4538    //|  sub esp, CCSTATE->spadj
4539    //|.if WIN
4540    //|  mov CCSTATE->spadj, esp
4541    //|.endif
4542    //|.endif
4543    //|
4544    //|  // Copy stack slots.
4545    //|  movzx ecx, byte CCSTATE->nsp
4546    //|  sub ecx, 1
4547    //|  js >2
4548    //|1:
4549    //|.if X64
4550    //|  mov rax, [CCSTATE+rcx*8+offsetof(CCallState, stack)]
4551    dasm_put(Dst, 8578, DISPATCH_GL(vmstate), ~LJ_VMST_INTERP, Dt1(->base), Dt1(->top), Dt7(->pc),
DISPATCH_GL(ctype_state), DtF(->L), Dt1(->base), Dt1(->top), DtF(->cb.gpr[0]), DtF(->cb.fpr[0]), Dt10(->spadj),
Dt10(->nsp));
4552  #line 3204 "vm_x86.dasc"
4553    //|  mov [rsp+rcx*8+CCALL_SPS_EXTRA*8], rax
```

```
4554    //|.else
4555    //|  mov eax, [CCSTATE+ecx*4+offsetof(CCallState, stack)]
4556    //|  mov [esp+ecx*4], eax
4557    //|.endif
4558    //|  sub ecx, 1
4559    //|  jns <1
4560    //|2:
4561    //|
4562    //|.if X64
4563    //|  movzx eax, byte CCSTATE->nfpr
4564    //|  mov CARG1, CCSTATE->gpr[0]
4565    //|  mov CARG2, CCSTATE->gpr[1]
4566    //|  mov CARG3, CCSTATE->gpr[2]
4567    //|  mov CARG4, CCSTATE->gpr[3]
4568    //|.if not X64WIN
4569    //|  mov CARG5, CCSTATE->gpr[4]
4570    //|  mov CARG6, CCSTATE->gpr[5]
4571    //|.endif
4572    //|  test eax, eax; jz >5
4573    //|  movaps xmm0, CCSTATE->fpr[0]
4574    //|  movaps xmm1, CCSTATE->fpr[1]
4575    //|  movaps xmm2, CCSTATE->fpr[2]
4576    //|  movaps xmm3, CCSTATE->fpr[3]
4577    //|.if not X64WIN
4578    //|  cmp eax, 4; jbe >5
4579    //|  movaps xmm4, CCSTATE->fpr[4]
4580    dasm_put(Dst, 8709, offsetof(CCallState, stack), CCALL_SPS_EXTRA*8, Dt10(->nfpr), Dt10(->gpr[0]), Dt10(->gpr[1]), Dt10(->gpr[2]), Dt10(->gpr[3]), Dt10(->gpr[4]), Dt10(->gpr[5]), Dt10(->fpr[0]), Dt10(->fpr[1]), Dt10(->fpr[2]), Dt10(->fpr[3]));
4581  #line 3231 "vm_x86.dasc"
4582    //|  movaps xmm5, CCSTATE->fpr[5]
4583    //|  movaps xmm6, CCSTATE->fpr[6]
4584    //|  movaps xmm7, CCSTATE->fpr[7]
4585    //|.endif
4586    //|5:
4587    //|.else
4588    //|  mov FCARG1, CCSTATE->gpr[0]
4589    //|  mov FCARG2, CCSTATE->gpr[1]
4590    //|.endif
4591    //|
4592    //|  call aword CCSTATE->func
4593    //|
4594    //|.if X64
4595    //|  mov CCSTATE->gpr[0], rax
4596    //|  movaps CCSTATE->fpr[0], xmm0
4597    //|.if not X64WIN
4598    //|  mov CCSTATE->gpr[1], rdx
4599    //|  movaps CCSTATE->fpr[1], xmm1
4600    //|.endif
4601    //|.else
4602    //|  mov CCSTATE->gpr[0], eax
4603    //|  mov CCSTATE->gpr[1], edx
4604    //|  cmp byte CCSTATE->resx87, 1
4605    //|  jb >7
4606    //|  je >6
4607    //|  fstp qword CCSTATE->fpr[0].d[0]
4608    //|  jmp >7
4609    //|6:
4610    //|  fstp dword CCSTATE->fpr[0].f[0]
4611    //|7:
4612    //|.if WIN
4613    //|  sub CCSTATE->spadj, esp
4614    //|.endif
4615    //|.endif
4616    //|
4617    //|.if X64
4618    //|  mov rbx, [rbp-8]; leave; ret
4619    //|.else
4620    //|  mov ebx, [ebp-4]; leave; ret
4621    //|.endif
4622    //|.endif
4623    //|// Note: vm_ffi_call must be the last function in this object file!
4624    //|
4625    //|//-----------------------------------------------------------------------
4626    dasm_put(Dst, 8790, Dt10(->fpr[4]), Dt10(->fpr[5]), Dt10(->fpr[6]), Dt10(->fpr[7]), Dt10(->func), Dt10(->gpr[0]), Dt10(->fpr[0]), Dt10(->gpr[1]), Dt10(->fpr[1]));
```

```
4627 #line 3275 "vm_x86.dasc"
4628 }
4629
4630 /* Generate the code for a single instruction. */
4631 static void build_ins(BuildCtx *ctx, BCOp op, int defop)
4632 {
4633   int vk = 0;
4634   //|// Note: aligning all instructions does not pay off.
4635   //|=>defop:
4636   dasm_put(Dst, 8836, defop);
4637 #line 3283 "vm_x86.dasc"
4638
4639   switch (op) {
4640
4641   /* -- Comparison ops ---------------------------------------------------- */
4642
4643   /* Remember: all ops branch for a true comparison, fall through otherwise. */
4644
4645   //|.macro jmp_comp, lt, ge, le, gt, target
4646   //||switch (op) {
4647   //||case BC_ISLT:
4648   //|   lt target
4649   //||break;
4650   //||case BC_ISGE:
4651   //|   ge target
4652   //||break;
4653   //||case BC_ISLE:
4654   //|   le target
4655   //||break;
4656   //||case BC_ISGT:
4657   //|   gt target
4658   //||break;
4659   //||default: break;  /* Shut up GCC. */
4660   //||}
4661   //|.endmacro
4662
4663   case BC_ISLT: case BC_ISGE: case BC_ISLE: case BC_ISGT:
4664     //|  // RA = src1, RD = src2, JMP with RD = target
4665     //|  ins_AD
4666     //|.if DUALNUM
4667     //|  checkint RA, >7
4668     //|  checkint RD, >8
4669     //|  mov RB, dword [BASE+RA*8]
4670     //|  add PC, 4
4671     //|  cmp RB, dword [BASE+RD*8]
4672     //|  jmp_comp jge, jl, jg, jle, >9
4673     dasm_put(Dst, 8838, LJ_TISNUM, LJ_TISNUM);
4674     switch (op) {
4675     case BC_ISLT:
4676     dasm_put(Dst, 8868);
4677     break;
4678     case BC_ISGE:
4679     dasm_put(Dst, 8873);
4680     break;
4681     case BC_ISLE:
4682     dasm_put(Dst, 8878);
4683     break;
4684     case BC_ISGT:
4685     dasm_put(Dst, 8883);
4686     break;
4687     default: break;  /* Shut up GCC. */
4688     }
4689 #line 3318 "vm_x86.dasc"
4690     //|6:
4691     //|  movzx RD, PC_RD
4692     //|  branchPC RD
4693     //|9:
4694     //|  ins_next
4695     //|
4696     //|7:  // RA is not an integer.
4697     //|  ja ->vmeta_comp
4698     //|  // RA is a number.
4699     //|  cmp dword [BASE+RD*8+4], LJ_TISNUM; jb >1; jne ->vmeta_comp
4700     //|  // RA is a number, RD is an integer.
4701     //|  cvtsi2sd xmm0, dword [BASE+RD*8]
4702     //|  jmp >2
```

```
4703      //|
4704      //|8:  // RA is an integer, RD is not an integer.
4705      //|  ja ->vmeta_comp
4706      //|  // RA is an integer, RD is a number.
4707      //|  cvtsi2sd xmm1, dword [BASE+RA*8]
4708      //|  movsd xmm0, qword [BASE+RD*8]
4709      //|  add PC, 4
4710      //|  ucomisd xmm0, xmm1
4711      //|  jmp_comp jbe, ja, jb, jae, <9
4712      dasm_put(Dst, 8888, -BCBIAS_J*4, LJ_TISNUM);
4713      switch (op) {
4714      case BC_ISLT:
4715      dasm_put(Dst, 8978);
4716      break;
4717      case BC_ISGE:
4718      dasm_put(Dst, 8983);
4719      break;
4720      case BC_ISLE:
4721      dasm_put(Dst, 8988);
4722      break;
4723      case BC_ISGT:
4724      dasm_put(Dst, 8993);
4725      break;
4726      default: break;  /* Shut up GCC. */
4727      }
4728  #line 3340 "vm_x86.dasc"
4729      //|  jmp <6
4730      //|.else
4731      //|  checknum RA, ->vmeta_comp
4732      //|  checknum RD, ->vmeta_comp
4733      //|.endif
4734      //|1:
4735      //|  movsd xmm0, qword [BASE+RD*8]
4736      //|2:
4737      //|  add PC, 4
4738      //|  ucomisd xmm0, qword [BASE+RA*8]
4739      //|3:
4740      //|  // Unordered: all of ZF CF PF set, ordered: PF clear.
4741      //|  // To preserve NaN semantics GE/GT branch on unordered, but LT/LE don't.
4742      //|.if DUALNUM
4743      //|  jmp_comp jbe, ja, jb, jae, <9
4744      dasm_put(Dst, 8998);
4745      switch (op) {
4746      case BC_ISLT:
4747      dasm_put(Dst, 8978);
4748      break;
4749      case BC_ISGE:
4750      dasm_put(Dst, 8983);
4751      break;
4752      case BC_ISLE:
4753      dasm_put(Dst, 8988);
4754      break;
4755      case BC_ISGT:
4756      dasm_put(Dst, 8993);
4757      break;
4758      default: break;  /* Shut up GCC. */
4759      }
4760  #line 3355 "vm_x86.dasc"
4761      //|  jmp <6
4762      //|.else
4763      //|  jmp_comp jbe, ja, jb, jae, >1
4764      //|  movzx RD, PC_RD
4765      //|  branchPC RD
4766      //|1:
4767      //|  ins_next
4768      //|.endif
4769      dasm_put(Dst, 9023);
4770  #line 3363 "vm_x86.dasc"
4771      break;
4772
4773    case BC_ISEQV: case BC_ISNEV:
4774      vk = op == BC_ISEQV;
4775      //|  ins_AD        // RA = src1, RD = src2, JMP with RD = target
4776      //|  mov RB, [BASE+RD*8+4]
4777      //|  add PC, 4
4778      //|.if DUALNUM
```

```
4779      //|  cmp RB, LJ_TISNUM; jne >7
4780      //|  checkint RA, >8
4781      //|  mov RB, dword [BASE+RD*8]
4782      //|  cmp RB, dword [BASE+RA*8]
4783      dasm_put(Dst, 9028, LJ_TISNUM, LJ_TISNUM);
4784  #line 3375 "vm_x86.dasc"
4785      if (vk) {
4786        //|  jne >9
4787        dasm_put(Dst, 9060);
4788  #line 3377 "vm_x86.dasc"
4789      } else {
4790        //|  je >9
4791        dasm_put(Dst, 9065);
4792  #line 3379 "vm_x86.dasc"
4793      }
4794      //|  movzx RD, PC_RD
4795      //|  branchPC RD
4796      //|9:
4797      //|  ins_next
4798      //|
4799      //|7:  // RD is not an integer.
4800      //|  ja >5
4801      //|  // RD is a number.
4802      //|  cmp dword [BASE+RA*8+4], LJ_TISNUM; jb >1; jne >5
4803      //|  // RD is a number, RA is an integer.
4804      //|  cvtsi2sd xmm0, dword [BASE+RA*8]
4805      //|  jmp >2
4806      //|
4807      //|8:  // RD is an integer, RA is not an integer.
4808      //|  ja >5
4809      //|  // RD is an integer, RA is a number.
4810      //|  cvtsi2sd xmm0, dword [BASE+RD*8]
4811      //|  ucomisd xmm0, qword [BASE+RA*8]
4812      //|  jmp >4
4813      //|
4814      //|.else
4815      //|  cmp RB, LJ_TISNUM; jae >5
4816      //|  checknum RA, >5
4817      //|.endif
4818      //|1:
4819      //|  movsd xmm0, qword [BASE+RA*8]
4820      //|2:
4821      //|  ucomisd xmm0, qword [BASE+RD*8]
4822      //|4:
4823      dasm_put(Dst, 9070, -BCBIAS_J*4, LJ_TISNUM);
4824  #line 3409 "vm_x86.dasc"
4825    iseqne_fp:
4826      if (vk) {
4827        //|  jp >2                              // Unordered means not equal.
4828        //|  jne >2
4829        dasm_put(Dst, 9171);
4830  #line 3413 "vm_x86.dasc"
4831      } else {
4832        //|  jp >2                              // Unordered means not equal.
4833        //|  je >1
4834        dasm_put(Dst, 9180);
4835  #line 3416 "vm_x86.dasc"
4836      }
4837    iseqne_end:
4838      if (vk) {
4839        //|1:                                  // EQ: Branch to the target.
4840        //|  movzx RD, PC_RD
4841        //|  branchPC RD
4842        //|2:                                  // NE: Fallthrough to next instruction.
4843        //|.if not FFI
4844        //|3:
4845        //|.endif
4846        dasm_put(Dst, 9189, -BCBIAS_J*4);
4847  #line 3426 "vm_x86.dasc"
4848      } else {
4849        //|.if not FFI
4850        //|3:
4851        //|.endif
4852        //|2:                                  // NE: Branch to the target.
4853        //|  movzx RD, PC_RD
4854        //|  branchPC RD
```

```
4855        //|1:                                  // EQ: Fallthrough to next instruction.
4856        dasm_put(Dst, 9204, -BCBIAS_J*4);
4857  #line 3434 "vm_x86.dasc"
4858      }
4859      if (LJ_DUALNUM && (op == BC_ISEQV || op == BC_ISNEV ||
4860                         op == BC_ISEQN || op == BC_ISNEN)) {
4861        //|  jmp <9
4862        dasm_put(Dst, 9219);
4863  #line 3438 "vm_x86.dasc"
4864      } else {
4865        //|  ins_next
4866        dasm_put(Dst, 9224);
4867  #line 3440 "vm_x86.dasc"
4868      }
4869      //|
4870      if (op == BC_ISEQV || op == BC_ISNEV) {
4871        //|5:  // Either or both types are not numbers.
4872        //|.if FFI
4873        //|  cmp RB, LJ_TCDATA; je ->vmeta_equal_cd
4874        //|  checktp RA, LJ_TCDATA; je ->vmeta_equal_cd
4875        //|.endif
4876        //|  checktp RA, RB                        // Compare types.
4877        //|  jne <2                                // Not the same type?
4878        //|  cmp RB, LJ_TISPRI
4879        //|  jae <1                                // Same type and primitive type?
4880        //|
4881        //|  // Same types and not a primitive type. Compare GCobj or pvalue.
4882        //|  mov RA, [BASE+RA*8]
4883        //|  mov RD, [BASE+RD*8]
4884        //|  cmp RA, RD
4885        //|  je <1                                 // Same GCobjs or pvalues?
4886        //|  cmp RB, LJ_TISTABUD
4887        //|  ja <2                                 // Different objects and not table/ud?
4888        //|.if X64
4889        //|  cmp RB, LJ_TUDATA                  // And not 64 bit lightuserdata.
4890        //|  jb <2
4891        //|.endif
4892        //|
4893        //|  // Different tables or userdatas. Need to check __eq metamethod.
4894        //|  // Field metatable must be at same offset for GCtab and GCudata!
4895        //|  mov TAB:RB, TAB:RA->metatable
4896        dasm_put(Dst, 9245, LJ_TCDATA, LJ_TCDATA, LJ_TISPRI, LJ_TISTABUD, LJ_TUDATA);
4897  #line 3468 "vm_x86.dasc"
4898        //|  test TAB:RB, TAB:RB
4899        //|  jz <2                                 // No metatable?
4900        //|  test byte TAB:RB->nomm, 1<<MM_eq
4901        //|  jnz <2                                 // Or 'no __eq' flag set?
4902        dasm_put(Dst, 9310, Dt6(->metatable), Dt6(->nomm), 1<<MM_eq);
4903  #line 3472 "vm_x86.dasc"
4904        if (vk) {
4905          //|  xor RB, RB                        // ne = 0
4906          dasm_put(Dst, 9330);
4907  #line 3474 "vm_x86.dasc"
4908        } else {
4909          //|  mov RB, 1                         // ne = 1
4910          dasm_put(Dst, 9334);
4911  #line 3476 "vm_x86.dasc"
4912        }
4913        //|  jmp ->vmeta_equal                 // Handle __eq metamethod.
4914        dasm_put(Dst, 9340);
4915  #line 3478 "vm_x86.dasc"
4916      } else {
4917        //|.if FFI
4918        //|3:
4919        //|  cmp RB, LJ_TCDATA
4920        dasm_put(Dst, 9345, LJ_TCDATA);
4921  #line 3482 "vm_x86.dasc"
4922        if (LJ_DUALNUM && vk) {
4923          //|  jne <9
4924          dasm_put(Dst, 9352);
4925  #line 3484 "vm_x86.dasc"
4926        } else {
4927          //|  jne <2
4928          dasm_put(Dst, 7896);
4929  #line 3486 "vm_x86.dasc"
4930        }
```

```
4931        //|  jmp ->vmeta_equal_cd
4932        //|.endif
4933        dasm_put(Dst, 9357);
4934  #line 3489 "vm_x86.dasc"
4935      }
4936      break;
4937    case BC_ISEQS: case BC_ISNES:
4938      vk = op == BC_ISEQS;
4939      //|  ins_AND        // RA = src, RD = str const, JMP with RD = target
4940      //|  mov RB, [BASE+RA*8+4]
4941      //|  add PC, 4
4942      //|  cmp RB, LJ_TSTR; jne >3
4943      //|  mov RA, [BASE+RA*8]
4944      //|  cmp RA, [KBASE+RD*4]
4945      dasm_put(Dst, 9362, LJ_TSTR);
4946  #line 3499 "vm_x86.dasc"
4947    iseqne_test:
4948      if (vk) {
4949        //|  jne >2
4950        dasm_put(Dst, 3670);
4951  #line 3502 "vm_x86.dasc"
4952      } else {
4953        //|  je >1
4954        dasm_put(Dst, 9184);
4955  #line 3504 "vm_x86.dasc"
4956      }
4957      goto iseqne_end;
4958    case BC_ISEQN: case BC_ISNEN:
4959      vk = op == BC_ISEQN;
4960      //|  ins_AD        // RA = src, RD = num const, JMP with RD = target
4961      //|  mov RB, [BASE+RA*8+4]
4962      //|  add PC, 4
4963      //|.if DUALNUM
4964      //|  cmp RB, LJ_TISNUM; jne >7
4965      //|  cmp dword [KBASE+RD*8+4], LJ_TISNUM; jne >8
4966      //|  mov RB, dword [KBASE+RD*8]
4967      //|  cmp RB, dword [BASE+RA*8]
4968      dasm_put(Dst, 9389, LJ_TISNUM, LJ_TISNUM);
4969  #line 3516 "vm_x86.dasc"
4970      if (vk) {
4971        //|  jne >9
4972        dasm_put(Dst, 9060);
4973  #line 3518 "vm_x86.dasc"
4974      } else {
4975        //|  je >9
4976        dasm_put(Dst, 9065);
4977  #line 3520 "vm_x86.dasc"
4978      }
4979      //|  movzx RD, PC_RD
4980      //|  branchPC RD
4981      //|9:
4982      //|  ins_next
4983      //|
4984      //|7:  // RA is not an integer.
4985      //|  ja >3
4986      //|  // RA is a number.
4987      //|  cmp dword [KBASE+RD*8+4], LJ_TISNUM; jb >1
4988      //|  // RA is a number, RD is an integer.
4989      //|  cvtsi2sd xmm0, dword [KBASE+RD*8]
4990      //|  jmp >2
4991      //|
4992      //|8:  // RA is an integer, RD is a number.
4993      //|  cvtsi2sd xmm0, dword [BASE+RA*8]
4994      //|  ucomisd xmm0, qword [KBASE+RD*8]
4995      //|  jmp >4
4996      //|.else
4997      //|  cmp RB, LJ_TISNUM; jae >3
4998      //|.endif
4999      //|1:
5000      //|  movsd xmm0, qword [KBASE+RD*8]
5001      //|2:
5002      //|  ucomisd xmm0, qword [BASE+RA*8]
5003      //|4:
5004      dasm_put(Dst, 9423, -BCBIAS_J*4, LJ_TISNUM);
5005  #line 3546 "vm_x86.dasc"
5006      goto iseqne_fp;
```

```
5007    case BC_ISEQP: case BC_ISNEP:
5008      vk = op == BC_ISEQP;
5009      //|  ins_AND        // RA = src, RD = primitive type (~), JMP with RD = target
5010      //|  mov RB, [BASE+RA*8+4]
5011      //|  add PC, 4
5012      //|  cmp RB, RD
5013      dasm_put(Dst, 9520);
5014  #line 3553 "vm_x86.dasc"
5015      if (!LJ_HASFFI) goto iseqne_test;
5016      if (vk) {
5017        //|  jne >3
5018        //|  movzx RD, PC_RD
5019        //|  branchPC RD
5020        //|2:
5021        //|  ins_next
5022        //|3:
5023        //|  cmp RB, LJ_TCDATA; jne <2
5024        //|  jmp ->vmeta_equal_cd
5025        dasm_put(Dst, 9534, -BCBIAS_J*4, LJ_TCDATA);
5026  #line 3563 "vm_x86.dasc"
5027      } else {
5028        //|  je >2
5029        //|  cmp RB, LJ_TCDATA; je ->vmeta_equal_cd
5030        //|  movzx RD, PC_RD
5031        //|  branchPC RD
5032        //|2:
5033        //|  ins_next
5034        dasm_put(Dst, 9585, LJ_TCDATA, -BCBIAS_J*4);
5035  #line 3570 "vm_x86.dasc"
5036      }
5037      break;
5038
5039    /* -- Unary test and copy ops ------------------------------------------- */
5040
5041    case BC_ISTC: case BC_ISFC: case BC_IST: case BC_ISF:
5042      //|  ins_AD        // RA = dst or unused, RD = src, JMP with RD = target
5043      //|  mov RB, [BASE+RD*8+4]
5044      //|  add PC, 4
5045      //|  cmp RB, LJ_TISTRUECOND
5046      dasm_put(Dst, 9630, LJ_TISTRUECOND);
5047  #line 3580 "vm_x86.dasc"
5048      if (op == BC_IST || op == BC_ISTC) {
5049        //|  jae >1
5050        dasm_put(Dst, 9642);
5051  #line 3582 "vm_x86.dasc"
5052      } else {
5053        //|  jb >1
5054        dasm_put(Dst, 5428);
5055  #line 3584 "vm_x86.dasc"
5056      }
5057      if (op == BC_ISTC || op == BC_ISFC) {
5058        //|  mov [BASE+RA*8+4], RB
5059        //|  mov RB, [BASE+RD*8]
5060        //|  mov [BASE+RA*8], RB
5061        dasm_put(Dst, 9647);
5062  #line 3589 "vm_x86.dasc"
5063      }
5064      //|  movzx RD, PC_RD
5065      //|  branchPC RD
5066      //|1:                                    // Fallthrough to the next instruction.
5067      //|  ins_next
5068      dasm_put(Dst, 9658, -BCBIAS_J*4);
5069  #line 3594 "vm_x86.dasc"
5070      break;
5071
5072    case BC_ISTYPE:
5073      //|  ins_AD        // RA = src, RD = -type
5074      //|  add RD, [BASE+RA*8+4]
5075      //|  jne ->vmeta_istype
5076      //|  ins_next
5077      dasm_put(Dst, 9691);
5078  #line 3601 "vm_x86.dasc"
5079      break;
5080    case BC_ISNUM:
5081      //|  ins_AD        // RA = src, RD = -(TISNUM-1)
5082      //|  checknum RA, ->vmeta_istype
```

```
5083      //|  ins_next
5084      dasm_put(Dst, 9720, LJ_TISNUM);
5085  #line 3606 "vm_x86.dasc"
5086      break;
5087
5088    /* -- Unary ops ----------------------------------------------------------- */
5089
5090    case BC_MOV:
5091      //|  ins_AD        // RA = dst, RD = src
5092      //|.if X64
5093      //|  mov RBa, [BASE+RD*8]
5094      //|  mov [BASE+RA*8], RBa
5095      //|.else
5096      //|  mov RB, [BASE+RD*8+4]
5097      //|  mov RD, [BASE+RD*8]
5098      //|  mov [BASE+RA*8+4], RB
5099      //|  mov [BASE+RA*8], RD
5100      //|.endif
5101      //|  ins_next_
5102      dasm_put(Dst, 9751);
5103  #line 3622 "vm_x86.dasc"
5104      break;
5105    case BC_NOT:
5106      //|  ins_AD        // RA = dst, RD = src
5107      //|  xor RB, RB
5108      //|  checktp RD, LJ_TISTRUECOND
5109      //|  adc RB, LJ_TTRUE
5110      //|  mov [BASE+RA*8+4], RB
5111      //|  ins_next
5112      dasm_put(Dst, 9780, LJ_TISTRUECOND, LJ_TTRUE);
5113  #line 3630 "vm_x86.dasc"
5114      break;
5115    case BC_UNM:
5116      //|  ins_AD        // RA = dst, RD = src
5117      //|.if DUALNUM
5118      //|  checkint RD, >5
5119      //|  mov RB, [BASE+RD*8]
5120      //|  neg RB
5121      //|  jo >4
5122      //|  mov dword [BASE+RA*8+4], LJ_TISNUM
5123      //|  mov dword [BASE+RA*8], RB
5124      //|9:
5125      //|  ins_next
5126      //|4:
5127      //|  mov dword [BASE+RA*8+4], 0x41e00000  // 2^31.
5128      //|  mov dword [BASE+RA*8], 0
5129      //|  jmp <9
5130      //|5:
5131      //|  ja ->vmeta_unm
5132      //|.else
5133      //|  checknum RD, ->vmeta_unm
5134      //|.endif
5135      //|  movsd xmm0, qword [BASE+RD*8]
5136      //|  sseconst_sign xmm1, RDa
5137      //|  xorps xmm0, xmm1
5138      //|  movsd qword [BASE+RA*8], xmm0
5139      //|.if DUALNUM
5140      //|  jmp <9
5141      //|.else
5142      //|  ins_next
5143      //|.endif
5144      dasm_put(Dst, 9817, LJ_TISNUM, LJ_TISNUM, (unsigned int)(U64x(80000000,00000000)), (unsigned int)
((U64x(80000000,00000000))>>32));
5145  #line 3660 "vm_x86.dasc"
5146      break;
5147    case BC_LEN:
5148      //|  ins_AD        // RA = dst, RD = src
5149      //|  checkstr RD, >2
5150      //|  mov STR:RD, [BASE+RD*8]
5151      //|.if DUALNUM
5152      //|  mov RD, dword STR:RD->len
5153      //|1:
5154      //|  mov dword [BASE+RA*8+4], LJ_TISNUM
5155      //|  mov dword [BASE+RA*8], RD
5156      //|.else
5157      //|  xorps xmm0, xmm0
```

```
5158      //|  cvtsi2sd xmm0, dword STR:RD->len
5159      //|1:
5160      //|  movsd qword [BASE+RA*8], xmm0
5161      //|.endif
5162      //|  ins_next
5163      //|2:
5164      //|  checktab RD, ->vmeta_len
5165      //|  mov TAB:FCARG1, [BASE+RD*8]
5166      dasm_put(Dst, 9923, LJ_TSTR, Dt5(->len), LJ_TISNUM, LJ_TTAB);
5167  #line 3680 "vm_x86.dasc"
5168  #if LJ_52
5169      //|  mov TAB:RB, TAB:FCARG1->metatable
5170      //|  cmp TAB:RB, 0
5171      //|  jnz >9
5172      //|3:
5173      dasm_put(Dst, 9985, Dt6(->metatable));
5174  #line 3685 "vm_x86.dasc"
5175  #endif
5176      //|->BC_LEN_Z:
5177      //|  mov RB, BASE                        // Save BASE.
5178      //|  call extern lj_tab_len@4               // (GCtab *t)
5179      //|  // Length of table returned in eax (RD).
5180      //|.if DUALNUM
5181      //|  // Nothing to do.
5182      //|.else
5183      //|  cvtsi2sd xmm0, RD
5184      //|.endif
5185      //|  mov BASE, RB                        // Restore BASE.
5186      //|  movzx RA, PC_RA
5187      //|  jmp <1
5188      dasm_put(Dst, 9999);
5189  #line 3698 "vm_x86.dasc"
5190  #if LJ_52
5191      //|9:  // Check for __len.
5192      //|  test byte TAB:RB->nomm, 1<<MM_len
5193      //|  jnz <3
5194      //|  jmp ->vmeta_len                        // 'no __len' flag NOT set: check.
5195      dasm_put(Dst, 10020, Dt6(->nomm), 1<<MM_len);
5196  #line 3703 "vm_x86.dasc"
5197  #endif
5198      break;
5199
5200    /* -- Binary ops ------------------------------------------------------ */
5201
5202      //|.macro ins_arithpre, sseins, ssereg
5203      //|  ins_ABC
5204      //||vk = ((int)op - BC_ADDVN) / (BC_ADDNV-BC_ADDVN);
5205      //||switch (vk) {
5206      //||case 0:
5207      //|   checknum RB, ->vmeta_arith_vn
5208      //|   .if DUALNUM
5209      //|     cmp dword [KBASE+RC*8+4], LJ_TISNUM; jae ->vmeta_arith_vn
5210      //|   .endif
5211      //|   movsd xmm0, qword [BASE+RB*8]
5212      //|   sseins ssereg, qword [KBASE+RC*8]
5213      //||  break;
5214      //||case 1:
5215      //|   checknum RB, ->vmeta_arith_nv
5216      //|   .if DUALNUM
5217      //|     cmp dword [KBASE+RC*8+4], LJ_TISNUM; jae ->vmeta_arith_nv
5218      //|   .endif
5219      //|   movsd xmm0, qword [KBASE+RC*8]
5220      //|   sseins ssereg, qword [BASE+RB*8]
5221      //||  break;
5222      //||default:
5223      //|   checknum RB, ->vmeta_arith_vv
5224      //|   checknum RC, ->vmeta_arith_vv
5225      //|   movsd xmm0, qword [BASE+RB*8]
5226      //|   sseins ssereg, qword [BASE+RC*8]
5227      //||  break;
5228      //||}
5229      //|.endmacro
5230      //|
5231      //|.macro ins_arithdn, intins
5232      //|  ins_ABC
5233      //||vk = ((int)op - BC_ADDVN) / (BC_ADDNV-BC_ADDVN);
```

```
5234      //||switch (vk) {
5235      //||case 0:
5236      //|   checkint RB, ->vmeta_arith_vn
5237      //|   cmp dword [KBASE+RC*8+4], LJ_TISNUM; jne ->vmeta_arith_vn
5238      //|   mov RB, [BASE+RB*8]
5239      //|   intins RB, [KBASE+RC*8]; jo ->vmeta_arith_vno
5240      //||  break;
5241      //||case 1:
5242      //|   checkint RB, ->vmeta_arith_nv
5243      //|   cmp dword [KBASE+RC*8+4], LJ_TISNUM; jne ->vmeta_arith_nv
5244      //|   mov RC, [KBASE+RC*8]
5245      //|   intins RC, [BASE+RB*8]; jo ->vmeta_arith_nvo
5246      //||  break;
5247      //||default:
5248      //|   checkint RB, ->vmeta_arith_vv
5249      //|   checkint RC, ->vmeta_arith_vv
5250      //|   mov RB, [BASE+RB*8]
5251      //|   intins RB, [BASE+RC*8]; jo ->vmeta_arith_vvo
5252      //||  break;
5253      //||}
5254      //|  mov dword [BASE+RA*8+4], LJ_TISNUM
5255      //||if (vk == 1) {
5256      //|   mov dword [BASE+RA*8], RC
5257      //||} else {
5258      //|   mov dword [BASE+RA*8], RB
5259      //||}
5260      //|  ins_next
5261      //|.endmacro
5262      //|
5263      //|.macro ins_arithpost
5264      //|  movsd qword [BASE+RA*8], xmm0
5265      //|.endmacro
5266      //|
5267      //|.macro ins_arith, sseins
5268      //|  ins_arithpre sseins, xmm0
5269      //|  ins_arithpost
5270      //|  ins_next
5271      //|.endmacro
5272      //|
5273      //|.macro ins_arith, intins, sseins
5274      //|.if DUALNUM
5275      //|  ins_arithdn intins
5276      //|.else
5277      //|  ins_arith, sseins
5278      //|.endif
5279      //|.endmacro
5280
5281      //|  // RA = dst, RB = src1 or num const, RC = src2 or num const
5282    case BC_ADDVN: case BC_ADDNV: case BC_ADDVV:
5283      //|  ins_arith add, addsd
5284      dasm_put(Dst, 10036);
5285      vk = ((int)op - BC_ADDVN) / (BC_ADDNV-BC_ADDVN);
5286      switch (vk) {
5287      case 0:
5288      dasm_put(Dst, 10044, LJ_TISNUM, LJ_TISNUM);
5289        break;
5290      case 1:
5291      dasm_put(Dst, 10079, LJ_TISNUM, LJ_TISNUM);
5292        break;
5293      default:
5294      dasm_put(Dst, 10114, LJ_TISNUM, LJ_TISNUM);
5295        break;
5296      }
5297      dasm_put(Dst, 10147, LJ_TISNUM);
5298      if (vk == 1) {
5299      dasm_put(Dst, 10153);
5300      } else {
5301      dasm_put(Dst, 9654);
5302      }
5303      dasm_put(Dst, 9224);
5304  #line 3790 "vm_x86.dasc"
5305      break;
5306    case BC_SUBVN: case BC_SUBNV: case BC_SUBVV:
5307      //|  ins_arith sub, subsd
5308      dasm_put(Dst, 10036);
5309      vk = ((int)op - BC_ADDVN) / (BC_ADDNV-BC_ADDVN);
```

```
    switch (vk) {
    case 0:
    dasm_put(Dst, 10157, LJ_TISNUM, LJ_TISNUM);
      break;
    case 1:
    dasm_put(Dst, 10192, LJ_TISNUM, LJ_TISNUM);
      break;
    default:
    dasm_put(Dst, 10227, LJ_TISNUM, LJ_TISNUM);
      break;
    }
    dasm_put(Dst, 10147, LJ_TISNUM);
    if (vk == 1) {
    dasm_put(Dst, 10153);
    } else {
    dasm_put(Dst, 9654);
    }
    dasm_put(Dst, 9224);
#line 3793 "vm_x86.dasc"
    break;
  case BC_MULVN: case BC_MULNV: case BC_MULVV:
    //|  ins_arith imul, mulsd
    dasm_put(Dst, 10036);
    vk = ((int)op - BC_ADDVN) / (BC_ADDNV-BC_ADDVN);
    switch (vk) {
    case 0:
    dasm_put(Dst, 10260, LJ_TISNUM, LJ_TISNUM);
      break;
    case 1:
    dasm_put(Dst, 10296, LJ_TISNUM, LJ_TISNUM);
      break;
    default:
    dasm_put(Dst, 10332, LJ_TISNUM, LJ_TISNUM);
      break;
    }
    dasm_put(Dst, 10147, LJ_TISNUM);
    if (vk == 1) {
    dasm_put(Dst, 10153);
    } else {
    dasm_put(Dst, 9654);
    }
    dasm_put(Dst, 9224);
#line 3796 "vm_x86.dasc"
    break;
  case BC_DIVVN: case BC_DIVNV: case BC_DIVVV:
    //|  ins_arith divsd
    dasm_put(Dst, 10036);
    vk = ((int)op - BC_ADDVN) / (BC_ADDNV-BC_ADDVN);
    switch (vk) {
    case 0:
    dasm_put(Dst, 10366, LJ_TISNUM, LJ_TISNUM);
      break;
    case 1:
    dasm_put(Dst, 10403, LJ_TISNUM, LJ_TISNUM);
      break;
    default:
    dasm_put(Dst, 10440, LJ_TISNUM, LJ_TISNUM);
      break;
    }
    dasm_put(Dst, 10475);
#line 3799 "vm_x86.dasc"
    break;
  case BC_MODVN:
    //|  ins_arithpre movsd, xmm1
    dasm_put(Dst, 10036);
    vk = ((int)op - BC_ADDVN) / (BC_ADDNV-BC_ADDVN);
    switch (vk) {
    case 0:
    dasm_put(Dst, 10502, LJ_TISNUM, LJ_TISNUM);
      break;
    case 1:
    dasm_put(Dst, 10539, LJ_TISNUM, LJ_TISNUM);
      break;
    default:
    dasm_put(Dst, 10576, LJ_TISNUM, LJ_TISNUM);
      break;
```

```
5386      }
5387  #line 3802 "vm_x86.dasc"
5388      //|->BC_MODVN_Z:
5389      //|  call ->vm_mod
5390      //|  ins_arithpost
5391      //|  ins_next
5392      dasm_put(Dst, 10611);
5393  #line 3806 "vm_x86.dasc"
5394      break;
5395    case BC_MODNV: case BC_MODVV:
5396      //|  ins_arithpre movsd, xmm1
5397      dasm_put(Dst, 10036);
5398      vk = ((int)op - BC_ADDVN) / (BC_ADDNV-BC_ADDVN);
5399      switch (vk) {
5400      case 0:
5401      dasm_put(Dst, 10502, LJ_TISNUM, LJ_TISNUM);
5402        break;
5403      case 1:
5404      dasm_put(Dst, 10539, LJ_TISNUM, LJ_TISNUM);
5405        break;
5406      default:
5407      dasm_put(Dst, 10576, LJ_TISNUM, LJ_TISNUM);
5408        break;
5409      }
5410  #line 3809 "vm_x86.dasc"
5411      //|  jmp ->BC_MODVN_Z                        // Avoid 3 copies. It's slow anyway.
5412      dasm_put(Dst, 10643);
5413  #line 3810 "vm_x86.dasc"
5414      break;
5415    case BC_POW:
5416      //|  ins_arithpre movsd, xmm1
5417      dasm_put(Dst, 10036);
5418      vk = ((int)op - BC_ADDVN) / (BC_ADDNV-BC_ADDVN);
5419      switch (vk) {
5420      case 0:
5421      dasm_put(Dst, 10502, LJ_TISNUM, LJ_TISNUM);
5422        break;
5423      case 1:
5424      dasm_put(Dst, 10539, LJ_TISNUM, LJ_TISNUM);
5425        break;
5426      default:
5427      dasm_put(Dst, 10576, LJ_TISNUM, LJ_TISNUM);
5428        break;
5429      }
5430  #line 3813 "vm_x86.dasc"
5431      //|  mov RB, BASE
5432      //|.if not X64
5433      //|  movsd FPARG1, xmm0
5434      //|  movsd FPARG3, xmm1
5435      //|.endif
5436      //|  call extern pow
5437      //|  movzx RA, PC_RA
5438      //|  mov BASE, RB
5439      //|.if X64
5440      //|  ins_arithpost
5441      //|.else
5442      //|  fstp qword [BASE+RA*8]
5443      //|.endif
5444      //|  ins_next
5445      dasm_put(Dst, 10648);
5446  #line 3827 "vm_x86.dasc"
5447      break;
5448
5449    case BC_CAT:
5450      //|  ins_ABC        // RA = dst, RB = src_start, RC = src_end
5451      //|.if X64
5452      //|  mov L:CARG1d, SAVE_L
5453      //|  mov L:CARG1d->base, BASE
5454      //|  lea CARG2d, [BASE+RC*8]
5455      //|  mov CARG3d, RC
5456      //|  sub CARG3d, RB
5457      //|->BC_CAT_Z:
5458      //|  mov L:RB, L:CARG1d
5459      //|.else
5460      //|  lea RA, [BASE+RC*8]
5461      //|  sub RC, RB
```

```
5462      //|  mov ARG2, RA
5463      //|  mov ARG3, RC
5464      //|->BC_CAT_Z:
5465      //|  mov L:RB, SAVE_L
5466      //|  mov ARG1, L:RB
5467      //|  mov L:RB->base, BASE
5468      //|.endif
5469      //|  mov SAVE_PC, PC
5470      //|  call extern lj_meta_cat                // (lua_State *L, TValue *top, int left)
5471      //|  // NULL (finished) or TValue * (metamethod) returned in eax (RC).
5472      //|  mov BASE, L:RB->base
5473      //|  test RC, RC
5474      //|  jnz ->vmeta_binop
5475      //|  movzx RB, PC_RB                        // Copy result to Stk[RA] from Stk[RB].
5476      //|  movzx RA, PC_RA
5477      //|.if X64
5478      //|  mov RCa, [BASE+RB*8]
5479      //|  mov [BASE+RA*8], RCa
5480      //|.else
5481      //|  mov RC, [BASE+RB*8+4]
5482      //|  mov RB, [BASE+RB*8]
5483      //|  mov [BASE+RA*8+4], RC
5484      //|  mov [BASE+RA*8], RB
5485      //|.endif
5486      //|  ins_next
5487      dasm_put(Dst, 10689, Dt1(->base), Dt1(->base));
5488  #line 3867 "vm_x86.dasc"
5489      break;
5490
5491    /* -- Constant ops ------------------------------------------------------ */
5492
5493    case BC_KSTR:
5494      //|  ins_AND        // RA = dst, RD = str const (~)
5495      //|  mov RD, [KBASE+RD*4]
5496      //|  mov dword [BASE+RA*8+4], LJ_TSTR
5497      //|  mov [BASE+RA*8], RD
5498      //|  ins_next
5499      dasm_put(Dst, 10773, LJ_TSTR);
5500  #line 3877 "vm_x86.dasc"
5501      break;
5502    case BC_KCDATA:
5503      //|.if FFI
5504      //|  ins_AND        // RA = dst, RD = cdata const (~)
5505      //|  mov RD, [KBASE+RD*4]
5506      //|  mov dword [BASE+RA*8+4], LJ_TCDATA
5507      //|  mov [BASE+RA*8], RD
5508      //|  ins_next
5509      //|.endif
5510      dasm_put(Dst, 10773, LJ_TCDATA);
5511  #line 3886 "vm_x86.dasc"
5512      break;
5513    case BC_KSHORT:
5514      //|  ins_AD        // RA = dst, RD = signed int16 literal
5515      //|.if DUALNUM
5516      //|  movsx RD, RDW
5517      //|  mov dword [BASE+RA*8+4], LJ_TISNUM
5518      //|  mov dword [BASE+RA*8], RD
5519      //|.else
5520      //|  movsx RD, RDW                        // Sign-extend literal.
5521      //|  cvtsi2sd xmm0, RD
5522      //|  movsd qword [BASE+RA*8], xmm0
5523      //|.endif
5524      //|  ins_next
5525      dasm_put(Dst, 10810, LJ_TISNUM);
5526  #line 3899 "vm_x86.dasc"
5527      break;
5528    case BC_KNUM:
5529      //|  ins_AD        // RA = dst, RD = num const
5530      //|  movsd xmm0, qword [KBASE+RD*8]
5531      //|  movsd qword [BASE+RA*8], xmm0
5532      //|  ins_next
5533      dasm_put(Dst, 10842);
5534  #line 3905 "vm_x86.dasc"
5535      break;
5536    case BC_KPRI:
5537      //|  ins_AND        // RA = dst, RD = primitive type (~)
```

```
5538      //|  mov [BASE+RA*8+4], RD
5539      //|  ins_next
5540      dasm_put(Dst, 10876);
5541  #line 3910 "vm_x86.dasc"
5542      break;
5543    case BC_KNIL:
5544      //|  ins_AD        // RA = dst_start, RD = dst_end
5545      //|  lea RA, [BASE+RA*8+12]
5546      //|  lea RD, [BASE+RD*8+4]
5547      //|  mov RB, LJ_TNIL
5548      //|  mov [RA-8], RB                        // Sets minimum 2 slots.
5549      //|1:
5550      //|  mov [RA], RB
5551      //|  add RA, 8
5552      //|  cmp RA, RD
5553      //|  jbe <1
5554      //|  ins_next
5555      dasm_put(Dst, 10905, LJ_TNIL);
5556  #line 3923 "vm_x86.dasc"
5557      break;
5558
5559    /* -- Upvalue and function ops ------------------------------------------- */
5560
5561    case BC_UGET:
5562      //|  ins_AD        // RA = dst, RD = upvalue #
5563      //|  mov LFUNC:RB, [BASE-8]
5564      //|  mov UPVAL:RB, [LFUNC:RB+RD*4+offsetof(GCfuncL, uvptr)]
5565      //|  mov RB, UPVAL:RB->v
5566      //|.if X64
5567      //|  mov RDa, [RB]
5568      //|  mov [BASE+RA*8], RDa
5569      //|.else
5570      //|  mov RD, [RB+4]
5571      //|  mov RB, [RB]
5572      //|  mov [BASE+RA*8+4], RD
5573      //|  mov [BASE+RA*8], RB
5574      //|.endif
5575      //|  ins_next
5576      dasm_put(Dst, 10953, offsetof(GCfuncL, uvptr), DtA(->v));
5577  #line 3942 "vm_x86.dasc"
5578      break;
5579    case BC_USETV:
5580  #define TV2MARKOFS \
5581  ((int32_t)offsetof(GCupval, marked)-(int32_t)offsetof(GCupval, tv))
5582      //|  ins_AD        // RA = upvalue #, RD = src
5583      //|  mov LFUNC:RB, [BASE-8]
5584      //|  mov UPVAL:RB, [LFUNC:RB+RA*4+offsetof(GCfuncL, uvptr)]
5585      //|  cmp byte UPVAL:RB->closed, 0
5586      //|  mov RB, UPVAL:RB->v
5587      //|  mov RA, [BASE+RD*8]
5588      //|  mov RD, [BASE+RD*8+4]
5589      //|  mov [RB], RA
5590      //|  mov [RB+4], RD
5591      //|  jz >1
5592      //|  // Check barrier for closed upvalue.
5593      //|  test byte [RB+TV2MARKOFS], LJ_GC_BLACK               // isblack(uv)
5594      //|  jnz >2
5595      //|1:
5596      //|  ins_next
5597      //|
5598      //|2:  // Upvalue is black. Check if new value is collectable and white.
5599      //|  sub RD, LJ_TISGCV
5600      //|  cmp RD, LJ_TNUMX - LJ_TISGCV                        // tvisgcv(v)
5601      //|  jbe <1
5602      //|  test byte GCOBJ:RA->gch.marked, LJ_GC_WHITES        // iswhite(v)
5603      //|  jz <1
5604      //|  // Crossed a write barrier. Move the barrier forward.
5605      //|.if X64 and not X64WIN
5606      //|  mov FCARG2, RB
5607      //|  mov RB, BASE                         // Save BASE.
5608      //|.else
5609      //|  xchg FCARG2, RB                        // Save BASE (FCARG2 == BASE).
5610      //|.endif
5611      //|  lea GL:FCARG1, [DISPATCH+GG_DISP2G]
5612      //|  call extern lj_gc_barrieruv@8        // (global_State *g, TValue *tv)
5613      dasm_put(Dst, 10994, offsetof(GCfuncL, uvptr), DtA(->closed), DtA(->v), TV2MARKOFS, LJ_GC_BLACK,
```

```
LJ_TISGCV, LJ_TNUMX - LJ_TISGCV, Dt4(->gch.marked), LJ_GC_WHITES, GG_DISP2G);
#line 3977 "vm_x86.dasc"
    //|  mov BASE, RB                        // Restore BASE.
    //|  jmp <1
    dasm_put(Dst, 11090);
#line 3979 "vm_x86.dasc"
    break;
#undef TV2MARKOFS
  case BC_USETS:
    //|  ins_AND        // RA = upvalue #, RD = str const (~)
    //|  mov LFUNC:RB, [BASE-8]
    //|  mov UPVAL:RB, [LFUNC:RB+RA*4+offsetof(GCfuncL, uvptr)]
    //|  mov GCOBJ:RA, [KBASE+RD*4]
    //|  mov RD, UPVAL:RB->v
    //|  mov [RD], GCOBJ:RA
    //|  mov dword [RD+4], LJ_TSTR
    //|  test byte UPVAL:RB->marked, LJ_GC_BLACK               // isblack(uv)
    //|  jnz >2
    //|1:
    //|  ins_next
    //|
    //|2:  // Check if string is white and ensure upvalue is closed.
    //|  test byte GCOBJ:RA->gch.marked, LJ_GC_WHITES        // iswhite(str)
    //|  jz <1
    //|  cmp byte UPVAL:RB->closed, 0
    //|  jz <1
    //|  // Crossed a write barrier. Move the barrier forward.
    //|  mov RB, BASE                        // Save BASE (FCARG2 == BASE).
    //|  mov FCARG2, RD
    //|  lea GL:FCARG1, [DISPATCH+GG_DISP2G]
    //|  call extern lj_gc_barrieruv@8        // (global_State *g, TValue *tv)
    //|  mov BASE, RB                        // Restore BASE.
    //|  jmp <1
    dasm_put(Dst, 11102, offsetof(GCfuncL, uvptr), DtA(->v), LJ_TSTR, DtA(->marked), LJ_GC_BLACK, Dt4(-
>gch.marked), LJ_GC_WHITES, DtA(->closed), GG_DISP2G);
#line 4006 "vm_x86.dasc"
    break;
  case BC_USETN:
    //|  ins_AD        // RA = upvalue #, RD = num const
    //|  mov LFUNC:RB, [BASE-8]
    //|  movsd xmm0, qword [KBASE+RD*8]
    //|  mov UPVAL:RB, [LFUNC:RB+RA*4+offsetof(GCfuncL, uvptr)]
    //|  mov RA, UPVAL:RB->v
    //|  movsd qword [RA], xmm0
    //|  ins_next
    dasm_put(Dst, 11198, offsetof(GCfuncL, uvptr), DtA(->v));
#line 4015 "vm_x86.dasc"
    break;
  case BC_USETP:
    //|  ins_AND        // RA = upvalue #, RD = primitive type (~)
    //|  mov LFUNC:RB, [BASE-8]
    //|  mov UPVAL:RB, [LFUNC:RB+RA*4+offsetof(GCfuncL, uvptr)]
    //|  mov RA, UPVAL:RB->v
    //|  mov [RA+4], RD
    //|  ins_next
    dasm_put(Dst, 11243, offsetof(GCfuncL, uvptr), DtA(->v));
#line 4023 "vm_x86.dasc"
    break;
  case BC_UCLO:
    //|  ins_AD        // RA = level, RD = target
    //|  branchPC RD                        // Do this first to free RD.
    //|  mov L:RB, SAVE_L
    //|  cmp dword L:RB->openupval, 0
    //|  je >1
    //|  mov L:RB->base, BASE
    //|  lea FCARG2, [BASE+RA*8]                // Caveat: FCARG2 == BASE
    //|  mov L:FCARG1, L:RB                // Caveat: FCARG1 == RA
    //|  call extern lj_func_closeuv@8        // (lua_State *L, TValue *level)
    //|  mov BASE, L:RB->base
    //|1:
    //|  ins_next
    dasm_put(Dst, 11283, -BCBIAS_J*4, Dt1(->openupval), Dt1(->base), Dt1(->base));
#line 4037 "vm_x86.dasc"
    break;

  case BC_FNEW:
```

```
5688      //|  ins_AND        // RA = dst, RD = proto const (~) (holding function prototype)
5689      //|.if X64
5690      //|  mov L:RB, SAVE_L
5691      //|  mov L:RB->base, BASE                // Caveat: CARG2d/CARG3d may be BASE.
5692      //|  mov CARG3d, [BASE-8]
5693      //|  mov CARG2d, [KBASE+RD*4]                // Fetch GCproto *.
5694      //|  mov CARG1d, L:RB
5695      //|.else
5696      //|  mov LFUNC:RA, [BASE-8]
5697      //|  mov PROTO:RD, [KBASE+RD*4]        // Fetch GCproto *.
5698      //|  mov L:RB, SAVE_L
5699      //|  mov ARG3, LFUNC:RA
5700      //|  mov ARG2, PROTO:RD
5701      //|  mov ARG1, L:RB
5702      //|  mov L:RB->base, BASE
5703      //|.endif
5704      //|  mov SAVE_PC, PC
5705      //|  // (lua_State *L, GCproto *pt, GCfuncL *parent)
5706      //|  call extern lj_func_newL_gc
5707      //|  // GCfuncL * returned in eax (RC).
5708      //|  mov BASE, L:RB->base
5709      //|  movzx RA, PC_RA
5710      //|  mov [BASE+RA*8], LFUNC:RC
5711      //|  mov dword [BASE+RA*8+4], LJ_TFUNC
5712      //|  ins_next
5713      dasm_put(Dst, 11339, Dt1(->base), Dt1(->base), LJ_TFUNC);
5714    #line 4065 "vm_x86.dasc"
5715      break;
5716
5717    /* -- Table ops ---------------------------------------------------- */
5718
5719    case BC_TNEW:
5720      //|  ins_AD        // RA = dst, RD = hbits|asize
5721      //|  mov L:RB, SAVE_L
5722      //|  mov L:RB->base, BASE
5723      //|  mov RA, [DISPATCH+DISPATCH_GL(gc.total)]
5724      //|  cmp RA, [DISPATCH+DISPATCH_GL(gc.threshold)]
5725      //|  mov SAVE_PC, PC
5726      //|  jae >5
5727      //|1:
5728      //|.if X64
5729      //|  mov CARG3d, RD
5730      //|  and RD, 0x7ff
5731      //|  shr CARG3d, 11
5732      //|.else
5733      //|  mov RA, RD
5734      //|  and RD, 0x7ff
5735      //|  shr RA, 11
5736      //|  mov ARG3, RA
5737      //|.endif
5738      //|  cmp RD, 0x7ff
5739      //|  je >3
5740      //|2:
5741      //|.if X64
5742      //|  mov L:CARG1d, L:RB
5743      //|  mov CARG2d, RD
5744      //|.else
5745      //|  mov ARG1, L:RB
5746      //|  mov ARG2, RD
5747      //|.endif
5748      //|  call extern lj_tab_new  // (lua_State *L, int32_t asize, uint32_t hbits)
5749      //|  // Table * returned in eax (RC).
5750      //|  mov BASE, L:RB->base
5751      //|  movzx RA, PC_RA
5752      //|  mov [BASE+RA*8], TAB:RC
5753      //|  mov dword [BASE+RA*8+4], LJ_TTAB
5754      //|  ins_next
5755      //|3:  // Turn 0x7ff into 0x801.
5756      //|  mov RD, 0x801
5757      //|  jmp <2
5758      //|5:
5759      //|  mov L:FCARG1, L:RB
5760      //|  call extern lj_gc_step_fixtop@4        // (lua_State *L)
5761      //|  movzx RD, PC_RD
5762      //|  jmp <1
5763      dasm_put(Dst, 11406, Dt1(->base), DISPATCH_GL(gc.total), DISPATCH_GL(gc.threshold), Dt1(->base),
```

```
LJ_TTAB);
#line 4113 "vm_x86.dasc"
    break;
  case BC_TDUP:
    //|  ins_AND        // RA = dst, RD = table const (~) (holding template table)
    //|  mov L:RB, SAVE_L
    //|  mov RA, [DISPATCH+DISPATCH_GL(gc.total)]
    //|  mov SAVE_PC, PC
    //|  cmp RA, [DISPATCH+DISPATCH_GL(gc.threshold)]
    //|  mov L:RB->base, BASE
    //|  jae >3
    //|2:
    //|  mov TAB:FCARG2, [KBASE+RD*4]        // Caveat: FCARG2 == BASE
    //|  mov L:FCARG1, L:RB                 // Caveat: FCARG1 == RA
    //|  call extern lj_tab_dup@8                // (lua_State *L, Table *kt)
    //|  // Table * returned in eax (RC).
    //|  mov BASE, L:RB->base
    //|  movzx RA, PC_RA
    //|  mov [BASE+RA*8], TAB:RC
    //|  mov dword [BASE+RA*8+4], LJ_TTAB
    //|  ins_next
    //|3:
    //|  mov L:FCARG1, L:RB
    //|  call extern lj_gc_step_fixtop@4        // (lua_State *L)
    //|  movzx RD, PC_RD                        // Need to reload RD.
    //|  not RDa
    //|  jmp <2
    dasm_put(Dst, 11530, DISPATCH_GL(gc.total), DISPATCH_GL(gc.threshold), Dt1(->base), Dt1(->base),
LJ_TTAB);
#line 4138 "vm_x86.dasc"
    break;

  case BC_GGET:
    //|  ins_AND        // RA = dst, RD = str const (~)
    //|  mov LFUNC:RB, [BASE-8]
    //|  mov TAB:RB, LFUNC:RB->env
    //|  mov STR:RC, [KBASE+RD*4]
    //|  jmp ->BC_TGETS_Z
    dasm_put(Dst, 11629, Dt7(->env));
#line 4146 "vm_x86.dasc"
    break;
  case BC_GSET:
    //|  ins_AND        // RA = src, RD = str const (~)
    //|  mov LFUNC:RB, [BASE-8]
    //|  mov TAB:RB, LFUNC:RB->env
    //|  mov STR:RC, [KBASE+RD*4]
    //|  jmp ->BC_TSETS_Z
    dasm_put(Dst, 11649, Dt7(->env));
#line 4153 "vm_x86.dasc"
    break;

  case BC_TGETV:
    //|  ins_ABC        // RA = dst, RB = table, RC = key
    //|  checktab RB, ->vmeta_tgetv
    //|  mov TAB:RB, [BASE+RB*8]
    //|
    //|  // Integer key?
    //|.if DUALNUM
    //|  checkint RC, >5
    //|  mov RC, dword [BASE+RC*8]
    //|.else
    //|  // Convert number to int and back and compare.
    //|  checknum RC, >5
    //|  movsd xmm0, qword [BASE+RC*8]
    //|  cvttsd2si RC, xmm0
    //|  cvtsi2sd xmm1, RC
    //|  ucomisd xmm0, xmm1
    //|  jne ->vmeta_tgetv               // Generic numeric key? Use fallback.
    //|.endif
    //|  cmp RC, TAB:RB->asize        // Takes care of unordered, too.
    //|  jae ->vmeta_tgetv               // Not in array part? Use fallback.
    //|  shl RC, 3
    //|  add RC, TAB:RB->array
    //|  cmp dword [RC+4], LJ_TNIL        // Avoid overwriting RB in fastpath.
    //|  je >2
    //|  // Get array slot.
```

```
5838      //|.if X64
5839      //|  mov RBa, [RC]
5840      //|  mov [BASE+RA*8], RBa
5841      //|.else
5842      //|  mov RB, [RC]
5843      //|  mov RC, [RC+4]
5844      //|  mov [BASE+RA*8], RB
5845      //|  mov [BASE+RA*8+4], RC
5846      //|.endif
5847      //|1:
5848      //|  ins_next
5849      //|
5850      //|2:  // Check for __index if table value is nil.
5851      //|  cmp dword TAB:RB->metatable, 0        // Shouldn't overwrite RA for fastpath.
5852      //|  jz >3
5853      //|  mov TAB:RA, TAB:RB->metatable
5854      //|  test byte TAB:RA->nomm, 1<<MM_index
5855      //|  jz ->vmeta_tgetv                        // 'no __index' flag NOT set: check.
5856      dasm_put(Dst, 11669, LJ_TTAB, LJ_TISNUM, Dt6(->asize), Dt6(->array), LJ_TNIL, Dt6(->metatable), Dt6(-
>metatable), Dt6(->nomm), 1<<MM_index);
5857    #line 4198 "vm_x86.dasc"
5858      //|  movzx RA, PC_RA                        // Restore RA.
5859      //|3:
5860      //|  mov dword [BASE+RA*8+4], LJ_TNIL
5861      //|  jmp <1
5862      //|
5863      //|5:  // String key?
5864      //|  checkstr RC, ->vmeta_tgetv
5865      //|  mov STR:RC, [BASE+RC*8]
5866      //|  jmp ->BC_TGETS_Z
5867      dasm_put(Dst, 11774, LJ_TNIL, LJ_TSTR);
5868    #line 4207 "vm_x86.dasc"
5869      break;
5870    case BC_TGETS:
5871      //|  ins_ABC        // RA = dst, RB = table, RC = str const (~)
5872      //|  not RCa
5873      //|  mov STR:RC, [KBASE+RC*4]
5874      //|  checktab RB, ->vmeta_tgets
5875      //|  mov TAB:RB, [BASE+RB*8]
5876      //|->BC_TGETS_Z:        // RB = GCtab *, RC = GCstr *, refetches PC_RA.
5877      //|  mov RA, TAB:RB->hmask
5878      //|  and RA, STR:RC->hash
5879      //|  imul RA, #NODE
5880      //|  add NODE:RA, TAB:RB->node
5881      //|1:
5882      //|  cmp dword NODE:RA->key.it, LJ_TSTR
5883      //|  jne >4
5884      //|  cmp dword NODE:RA->key.gcr, STR:RC
5885      //|  jne >4
5886      //|  // Ok, key found. Assumes: offsetof(Node, val) == 0
5887      //|  cmp dword [RA+4], LJ_TNIL        // Avoid overwriting RB in fastpath.
5888      //|  je >5                                  // Key found, but nil value?
5889      //|  movzx RC, PC_RA
5890      //|  // Get node value.
5891      //|.if X64
5892      //|  mov RBa, [RA]
5893      //|  mov [BASE+RC*8], RBa
5894      //|.else
5895      //|  mov RB, [RA]
5896      //|  mov RA, [RA+4]
5897      //|  mov [BASE+RC*8], RB
5898      //|  mov [BASE+RC*8+4], RA
5899      //|.endif
5900      //|2:
5901      //|  ins_next
5902      dasm_put(Dst, 11814, LJ_TTAB, Dt6(->hmask), Dt5(->hash), sizeof(Node), Dt6(->node), DtB(->key.it),
LJ_TSTR, DtB(->key.gcr), LJ_TNIL);
5903    #line 4240 "vm_x86.dasc"
5904      //|
5905      //|3:
5906      //|  movzx RC, PC_RA
5907      //|  mov dword [BASE+RC*8+4], LJ_TNIL
5908      //|  jmp <2
5909      //|
5910      //|4:  // Follow hash chain.
5911      //|  mov NODE:RA, NODE:RA->next
```

```
5912      //|  test NODE:RA, NODE:RA
5913      //|  jnz <1
5914      //|  // End of hash chain: key not found, nil result.
5915      //|
5916      //|5:  // Check for __index if table value is nil.
5917      //|  mov TAB:RA, TAB:RB->metatable
5918      //|  test TAB:RA, TAB:RA
5919      //|  jz <3                                  // No metatable: done.
5920      //|  test byte TAB:RA->nomm, 1<<MM_index
5921      //|  jnz <3                                  // 'no __index' flag set: done.
5922      //|  jmp ->vmeta_tgets                 // Caveat: preserve STR:RC.
5923      dasm_put(Dst, 11899, LJ_TNIL, DtB(->next), Dt6(->metatable), Dt6(->nomm), 1<<MM_index);
5924  #line 4259 "vm_x86.dasc"
5925      break;
5926    case BC_TGETB:
5927      //|  ins_ABC        // RA = dst, RB = table, RC = byte literal
5928      //|  checktab RB, ->vmeta_tgetb
5929      //|  mov TAB:RB, [BASE+RB*8]
5930      //|  cmp RC, TAB:RB->asize
5931      //|  jae ->vmeta_tgetb
5932      //|  shl RC, 3
5933      //|  add RC, TAB:RB->array
5934      //|  cmp dword [RC+4], LJ_TNIL        // Avoid overwriting RB in fastpath.
5935      //|  je >2
5936      //|  // Get array slot.
5937      //|.if X64
5938      //|  mov RBa, [RC]
5939      //|  mov [BASE+RA*8], RBa
5940      //|.else
5941      //|  mov RB, [RC]
5942      //|  mov RC, [RC+4]
5943      //|  mov [BASE+RA*8], RB
5944      //|  mov [BASE+RA*8+4], RC
5945      //|.endif
5946      //|1:
5947      //|  ins_next
5948      //|
5949      //|2:  // Check for __index if table value is nil.
5950      //|  cmp dword TAB:RB->metatable, 0        // Shouldn't overwrite RA for fastpath.
5951      //|  jz >3
5952      //|  mov TAB:RA, TAB:RB->metatable
5953      //|  test byte TAB:RA->nomm, 1<<MM_index
5954      //|  jz ->vmeta_tgetb                        // 'no __index' flag NOT set: check.
5955      //|  movzx RA, PC_RA                        // Restore RA.
5956      dasm_put(Dst, 11971, LJ_TTAB, Dt6(->asize), Dt6(->array), LJ_TNIL, Dt6(->metatable), Dt6(->metatable),
Dt6(->nomm), 1<<MM_index);
5957  #line 4290 "vm_x86.dasc"
5958      //|3:
5959      //|  mov dword [BASE+RA*8+4], LJ_TNIL
5960      //|  jmp <1
5961      dasm_put(Dst, 12067, LJ_TNIL);
5962  #line 4293 "vm_x86.dasc"
5963      break;
5964    case BC_TGETR:
5965      //|  ins_ABC        // RA = dst, RB = table, RC = key
5966      //|  mov TAB:RB, [BASE+RB*8]
5967      //|.if DUALNUM
5968      //|  mov RC, dword [BASE+RC*8]
5969      //|.else
5970      //|  cvttsd2si RC, qword [BASE+RC*8]
5971      //|.endif
5972      //|  cmp RC, TAB:RB->asize
5973      //|  jae ->vmeta_tgetr                // Not in array part? Use fallback.
5974      //|  shl RC, 3
5975      //|  add RC, TAB:RB->array
5976      //|  // Get array slot.
5977      //|->BC_TGETR_Z:
5978      //|.if X64
5979      //|  mov RBa, [RC]
5980      //|  mov [BASE+RA*8], RBa
5981      //|.else
5982      //|  mov RB, [RC]
5983      //|  mov RC, [RC+4]
5984      //|  mov [BASE+RA*8], RB
5985      //|  mov [BASE+RA*8+4], RC
5986      //|.endif
```

```
5987      //|->BC_TGETR2_Z:
5988      //|  ins_next
5989      dasm_put(Dst, 12084, Dt6(->asize), Dt6(->array));
5990  #line 4319 "vm_x86.dasc"
5991      break;
5992
5993    case BC_TSETV:
5994      //|  ins_ABC        // RA = src, RB = table, RC = key
5995      //|  checktab RB, ->vmeta_tsetv
5996      //|  mov TAB:RB, [BASE+RB*8]
5997      //|
5998      //|  // Integer key?
5999      //|.if DUALNUM
6000      //|  checkint RC, >5
6001      //|  mov RC, dword [BASE+RC*8]
6002      //|.else
6003      //|  // Convert number to int and back and compare.
6004      //|  checknum RC, >5
6005      //|  movsd xmm0, qword [BASE+RC*8]
6006      //|  cvttsd2si RC, xmm0
6007      //|  cvtsi2sd xmm1, RC
6008      //|  ucomisd xmm0, xmm1
6009      //|  jne ->vmeta_tsetv               // Generic numeric key? Use fallback.
6010      //|.endif
6011      //|  cmp RC, TAB:RB->asize               // Takes care of unordered, too.
6012      //|  jae ->vmeta_tsetv
6013      //|  shl RC, 3
6014      //|  add RC, TAB:RB->array
6015      //|  cmp dword [RC+4], LJ_TNIL
6016      //|  je >3                               // Previous value is nil?
6017      //|1:
6018      //|  test byte TAB:RB->marked, LJ_GC_BLACK        // isblack(table)
6019      //|  jnz >7
6020      //|2:  // Set array slot.
6021      //|.if X64
6022      //|  mov RBa, [BASE+RA*8]
6023      //|  mov [RC], RBa
6024      //|.else
6025      //|  mov RB, [BASE+RA*8+4]
6026      //|  mov RA, [BASE+RA*8]
6027      //|  mov [RC+4], RB
6028      //|  mov [RC], RA
6029      //|.endif
6030      //|  ins_next
6031      //|
6032      //|3:  // Check for __newindex if previous value is nil.
6033      //|  cmp dword TAB:RB->metatable, 0        // Shouldn't overwrite RA for fastpath.
6034      dasm_put(Dst, 12143, LJ_TTAB, LJ_TISNUM, Dt6(->asize), Dt6(->array), LJ_TNIL, Dt6(->marked),
LJ_GC_BLACK);
6035  #line 4362 "vm_x86.dasc"
6036      //|  jz <1
6037      //|  mov TAB:RA, TAB:RB->metatable
6038      //|  test byte TAB:RA->nomm, 1<<MM_newindex
6039      //|  jz ->vmeta_tsetv                     // 'no __newindex' flag NOT set: check.
6040      //|  movzx RA, PC_RA                     // Restore RA.
6041      //|  jmp <1
6042      //|
6043      //|5:  // String key?
6044      //|  checkstr RC, ->vmeta_tsetv
6045      //|  mov STR:RC, [BASE+RC*8]
6046      //|  jmp ->BC_TSETS_Z
6047      //|
6048      //|7:  // Possible table write barrier for the value. Skip valiswhite check.
6049      //|  barrierback TAB:RB, RA
6050      dasm_put(Dst, 12243, Dt6(->metatable), Dt6(->metatable), Dt6(->nomm), 1<<MM_newindex, LJ_TSTR, Dt6(-
>marked), (uint8_t)~LJ_GC_BLACK, DISPATCH_GL(gc.grayagain));
6051  #line 4376 "vm_x86.dasc"
6052      //|  movzx RA, PC_RA                     // Restore RA.
6053      //|  jmp <2
6054      dasm_put(Dst, 12302, DISPATCH_GL(gc.grayagain), Dt6(->gclist));
6055  #line 4378 "vm_x86.dasc"
6056      break;
6057    case BC_TSETS:
6058      //|  ins_ABC        // RA = src, RB = table, RC = str const (~)
6059      //|  not RCa
6060      //|  mov STR:RC, [KBASE+RC*4]
```

```
6061      //|  checktab RB, ->vmeta_tsets
6062      //|  mov TAB:RB, [BASE+RB*8]
6063      //|->BC_TSETS_Z:        // RB = GCtab *, RC = GCstr *, refetches PC_RA.
6064      //|  mov RA, TAB:RB->hmask
6065      //|  and RA, STR:RC->hash
6066      //|  imul RA, #NODE
6067      //|  mov byte TAB:RB->nomm, 0                // Clear metamethod cache.
6068      //|  add NODE:RA, TAB:RB->node
6069      //|1:
6070      //|  cmp dword NODE:RA->key.it, LJ_TSTR
6071      //|  jne >5
6072      //|  cmp dword NODE:RA->key.gcr, STR:RC
6073      //|  jne >5
6074      //|  // Ok, key found. Assumes: offsetof(Node, val) == 0
6075      //|  cmp dword [RA+4], LJ_TNIL
6076      //|  je >4                                   // Previous value is nil?
6077      //|2:
6078      //|  test byte TAB:RB->marked, LJ_GC_BLACK        // isblack(table)
6079      dasm_put(Dst, 12319, LJ_TTAB, Dt6(->hmask), Dt5(->hash), sizeof(Node), Dt6(->nomm), Dt6(->node), DtB(-
>key.it), LJ_TSTR, DtB(->key.gcr), LJ_TNIL);
6080  #line 4401 "vm_x86.dasc"
6081      //|  jnz >7
6082      //|3:  // Set node value.
6083      //|  movzx RC, PC_RA
6084      //|.if X64
6085      //|  mov RBa, [BASE+RC*8]
6086      //|  mov [RA], RBa
6087      //|.else
6088      //|  mov RB, [BASE+RC*8+4]
6089      //|  mov RC, [BASE+RC*8]
6090      //|  mov [RA+4], RB
6091      //|  mov [RA], RC
6092      //|.endif
6093      //|  ins_next
6094      //|
6095      //|4:  // Check for __newindex if previous value is nil.
6096      //|  cmp dword TAB:RB->metatable, 0        // Shouldn't overwrite RA for fastpath.
6097      //|  jz <2
6098      //|  mov TMP1, RA                          // Save RA.
6099      //|  mov TAB:RA, TAB:RB->metatable
6100      //|  test byte TAB:RA->nomm, 1<<MM_newindex
6101      //|  jz ->vmeta_tsets                         // 'no __newindex' flag NOT set: check.
6102      //|  mov RA, TMP1                          // Restore RA.
6103      //|  jmp <2
6104      //|
6105      //|5:  // Follow hash chain.
6106      //|  mov NODE:RA, NODE:RA->next
6107      //|  test NODE:RA, NODE:RA
6108      //|  jnz <1
6109      //|  // End of hash chain: key not found, add a new one.
6110      //|
6111      //|  // But check for __newindex first.
6112      //|  mov TAB:RA, TAB:RB->metatable
6113      dasm_put(Dst, 12396, Dt6(->marked), LJ_GC_BLACK, Dt6(->metatable), Dt6(->metatable), Dt6(->nomm),
1<<MM_newindex, DtB(->next));
6114  #line 4433 "vm_x86.dasc"
6115      //|  test TAB:RA, TAB:RA
6116      //|  jz >6                                   // No metatable: continue.
6117      //|  test byte TAB:RA->nomm, 1<<MM_newindex
6118      //|  jz ->vmeta_tsets                         // 'no __newindex' flag NOT set: check.
6119      //|6:
6120      //|  mov TMP1, STR:RC
6121      //|  mov TMP2, LJ_TSTR
6122      //|  mov TMP3, TAB:RB                         // Save TAB:RB for us.
6123      //|.if X64
6124      //|  mov L:CARG1d, SAVE_L
6125      //|  mov L:CARG1d->base, BASE
6126      //|  lea CARG3, TMP1
6127      //|  mov CARG2d, TAB:RB
6128      //|  mov L:RB, L:CARG1d
6129      //|.else
6130      //|  lea RC, TMP1                          // Store temp. TValue in TMP1/TMP2.
6131      //|  mov ARG2, TAB:RB
6132      //|  mov L:RB, SAVE_L
6133      //|  mov ARG3, RC
6134      //|  mov ARG1, L:RB
```

```
6135      //|  mov L:RB->base, BASE
6136      //|.endif
6137      //|  mov SAVE_PC, PC
6138      //|  call extern lj_tab_newkey        // (lua_State *L, GCtab *t, TValue *k)
6139      //|  // Handles write barrier for the new key. TValue * returned in eax (RC).
6140      //|  mov BASE, L:RB->base
6141      //|  mov TAB:RB, TMP3                       // Need TAB:RB for barrier.
6142      //|  mov RA, eax
6143      //|  jmp <2                                 // Must check write barrier for value.
6144      //|
6145      //|7:  // Possible table write barrier for the value. Skip valiswhite check.
6146      //|  barrierback TAB:RB, RC                 // Destroys STR:RC.
6147      //|  jmp <3
6148      dasm_put(Dst, 12483, Dt6(->metatable), Dt6(->nomm), 1<<MM_newindex, LJ_TSTR, Dt1(->base), Dt1(->base),
Dt6(->marked), (uint8_t)~LJ_GC_BLACK, DISPATCH_GL(gc.grayagain), DISPATCH_GL(gc.grayagain), Dt6(->gclist));
6149  #line 4466 "vm_x86.dasc"
6150      break;
6151    case BC_TSETB:
6152      //|  ins_ABC        // RA = src, RB = table, RC = byte literal
6153      //|  checktab RB, ->vmeta_tsetb
6154      //|  mov TAB:RB, [BASE+RB*8]
6155      //|  cmp RC, TAB:RB->asize
6156      //|  jae ->vmeta_tsetb
6157      //|  shl RC, 3
6158      //|  add RC, TAB:RB->array
6159      //|  cmp dword [RC+4], LJ_TNIL
6160      //|  je >3                                  // Previous value is nil?
6161      //|1:
6162      //|  test byte TAB:RB->marked, LJ_GC_BLACK        // isblack(table)
6163      //|  jnz >7
6164      //|2:         // Set array slot.
6165      //|.if X64
6166      //|  mov RAa, [BASE+RA*8]
6167      //|  mov [RC], RAa
6168      //|.else
6169      //|  mov RB, [BASE+RA*8+4]
6170      //|  mov RA, [BASE+RA*8]
6171      //|  mov [RC+4], RB
6172      //|  mov [RC], RA
6173      //|.endif
6174      //|  ins_next
6175      //|
6176      //|3:  // Check for __newindex if previous value is nil.
6177      //|  cmp dword TAB:RB->metatable, 0        // Shouldn't overwrite RA for fastpath.
6178      //|  jz <1
6179      //|  mov TAB:RA, TAB:RB->metatable
6180      dasm_put(Dst, 12575, LJ_TTAB, Dt6(->asize), Dt6(->array), LJ_TNIL, Dt6(->marked), LJ_GC_BLACK, Dt6(-
>metatable));
6181  #line 4496 "vm_x86.dasc"
6182      //|  test byte TAB:RA->nomm, 1<<MM_newindex
6183      //|  jz ->vmeta_tsetb                        // 'no __newindex' flag NOT set: check.
6184      //|  movzx RA, PC_RA                        // Restore RA.
6185      //|  jmp <1
6186      //|
6187      //|7:  // Possible table write barrier for the value. Skip valiswhite check.
6188      //|  barrierback TAB:RB, RA
6189      //|  movzx RA, PC_RA                        // Restore RA.
6190      //|  jmp <2
6191      dasm_put(Dst, 12670, Dt6(->metatable), Dt6(->nomm), 1<<MM_newindex, Dt6(->marked),
(uint8_t)~LJ_GC_BLACK, DISPATCH_GL(gc.grayagain), DISPATCH_GL(gc.grayagain), Dt6(->gclist));
6192  #line 4505 "vm_x86.dasc"
6193      break;
6194    case BC_TSETR:
6195      //|  ins_ABC        // RA = src, RB = table, RC = key
6196      //|  mov TAB:RB, [BASE+RB*8]
6197      //|.if DUALNUM
6198      //|  mov RC, dword [BASE+RC*8]
6199      //|.else
6200      //|  cvttsd2si RC, qword [BASE+RC*8]
6201      //|.endif
6202      //|  test byte TAB:RB->marked, LJ_GC_BLACK        // isblack(table)
6203      //|  jnz >7
6204      //|2:
6205      //|  cmp RC, TAB:RB->asize
6206      //|  jae ->vmeta_tsetr
6207      //|  shl RC, 3
```

```
6208      //|  add RC, TAB:RB->array
6209      //|  // Set array slot.
6210      //|->BC_TSETR_Z:
6211      //|.if X64
6212      //|  mov RBa, [BASE+RA*8]
6213      //|  mov [RC], RBa
6214      //|.else
6215      //|  mov RB, [BASE+RA*8+4]
6216      //|  mov RA, [BASE+RA*8]
6217      //|  mov [RC+4], RB
6218      //|  mov [RC], RA
6219      //|.endif
6220      //|  ins_next
6221      //|
6222      //|7:  // Possible table write barrier for the value. Skip valiswhite check.
6223      //|  barrierback TAB:RB, RA
6224      //|  movzx RA, PC_RA                        // Restore RA.
6225      //|  jmp <2
6226      dasm_put(Dst, 12718, Dt6(->marked), LJ_GC_BLACK, Dt6(->asize), Dt6(->array), Dt6(->marked),
(uint8_t)~LJ_GC_BLACK, DISPATCH_GL(gc.grayagain), DISPATCH_GL(gc.grayagain), Dt6(->gclist));
6227  #line 4538 "vm_x86.dasc"
6228      break;
6229
6230    case BC_TSETM:
6231      //|  ins_AD        // RA = base (table at base-1), RD = num const (start index)
6232      //|  mov TMP1, KBASE                          // Need one more free register.
6233      //|  mov KBASE, dword [KBASE+RD*8]        // Integer constant is in lo-word.
6234      //|1:
6235      //|  lea RA, [BASE+RA*8]
6236      //|  mov TAB:RB, [RA-8]                // Guaranteed to be a table.
6237      //|  test byte TAB:RB->marked, LJ_GC_BLACK        // isblack(table)
6238      //|  jnz >7
6239      //|2:
6240      //|  mov RD, MULTRES
6241      //|  sub RD, 1
6242      //|  jz >4                                 // Nothing to copy?
6243      //|  add RD, KBASE                         // Compute needed size.
6244      //|  cmp RD, TAB:RB->asize
6245      //|  ja >5                                 // Doesn't fit into array part?
6246      //|  sub RD, KBASE
6247      //|  shl KBASE, 3
6248      //|  add KBASE, TAB:RB->array
6249      //|3:  // Copy result slots to table.
6250      //|.if X64
6251      //|  mov RBa, [RA]
6252      //|  add RA, 8
6253      //|  mov [KBASE], RBa
6254      //|.else
6255      //|  mov RB, [RA]
6256      //|  mov [KBASE], RB
6257      //|  mov RB, [RA+4]
6258      //|  add RA, 8
6259      //|  mov [KBASE+4], RB
6260      //|.endif
6261      //|  add KBASE, 8
6262      //|  sub RD, 1
6263      //|  jnz <3
6264      //|4:
6265      //|  mov KBASE, TMP1
6266      //|  ins_next
6267      //|
6268      //|5:  // Need to resize array part.
6269      //|.if X64
6270      //|  mov L:CARG1d, SAVE_L
6271      //|  mov L:CARG1d->base, BASE               // Caveat: CARG2d/CARG3d may be BASE.
6272      //|  mov CARG2d, TAB:RB
6273      //|  mov CARG3d, RD
6274      //|  mov L:RB, L:CARG1d
6275      //|.else
6276      //|  mov ARG2, TAB:RB
6277      //|  mov L:RB, SAVE_L
6278      //|  mov L:RB->base, BASE
6279      //|  mov ARG3, RD
6280      //|  mov ARG1, L:RB
6281      //|.endif
6282      //|  mov SAVE_PC, PC
```

```
6283      //|  call extern lj_tab_reasize        // (lua_State *L, GCtab *t, int nasize)
6284      //|  mov BASE, L:RB->base
6285      //|  movzx RA, PC_RA                      // Restore RA.
6286      //|  jmp <1                              // Retry.
6287      //|
6288      //|7:  // Possible table write barrier for any value. Skip valiswhite check.
6289      //|  barrierback TAB:RB, RD
6290      dasm_put(Dst, 12812, Dt6(->marked), LJ_GC_BLACK, Dt6(->asize), Dt6(->array), Dt1(->base), Dt1(->base));
6291  #line 4600 "vm_x86.dasc"
6292      //|  jmp <2
6293      dasm_put(Dst, 12962, Dt6(->marked), (uint8_t)~LJ_GC_BLACK, DISPATCH_GL(gc.grayagain),
DISPATCH_GL(gc.grayagain), Dt6(->gclist));
6294  #line 4601 "vm_x86.dasc"
6295      break;
6296
6297    /* -- Calls and vararg handling ----------------------------------------- */
6298
6299    case BC_CALL: case BC_CALLM:
6300      //|  ins_A_C        // RA = base, (RB = nresults+1,) RC = nargs+1 | extra_nargs
6301      dasm_put(Dst, 10040);
6302  #line 4607 "vm_x86.dasc"
6303      if (op == BC_CALLM) {
6304        //|  add NARGS:RD, MULTRES
6305        dasm_put(Dst, 12982);
6306  #line 4609 "vm_x86.dasc"
6307      }
6308      //|  cmp dword [BASE+RA*8+4], LJ_TFUNC
6309      //|  mov LFUNC:RB, [BASE+RA*8]
6310      //|  jne ->vmeta_call_ra
6311      //|  lea BASE, [BASE+RA*8+8]
6312      //|  ins_call
6313      dasm_put(Dst, 12987, LJ_TFUNC, Dt7(->pc));
6314  #line 4615 "vm_x86.dasc"
6315      break;
6316
6317    case BC_CALLMT:
6318      //|  ins_AD        // RA = base, RD = extra_nargs
6319      //|  add NARGS:RD, MULTRES
6320      //|  // Fall through. Assumes BC_CALLT follows and ins_AD is a no-op.
6321      dasm_put(Dst, 12982);
6322  #line 4621 "vm_x86.dasc"
6323      break;
6324    case BC_CALLT:
6325      //|  ins_AD        // RA = base, RD = nargs+1
6326      //|  lea RA, [BASE+RA*8+8]
6327      //|  mov KBASE, BASE                       // Use KBASE for move + vmeta_call hint.
6328      //|  mov LFUNC:RB, [RA-8]
6329      //|  cmp dword [RA-4], LJ_TFUNC
6330      //|  jne ->vmeta_call
6331      //|->BC_CALLT_Z:
6332      //|  mov PC, [BASE-4]
6333      //|  test PC, FRAME_TYPE
6334      //|  jnz >7
6335      //|1:
6336      //|  mov [BASE-8], LFUNC:RB               // Copy function down, reloaded below.
6337      //|  mov MULTRES, NARGS:RD
6338      //|  sub NARGS:RD, 1
6339      //|  jz >3
6340      //|2:  // Move args down.
6341      //|.if X64
6342      //|  mov RBa, [RA]
6343      //|  add RA, 8
6344      //|  mov [KBASE], RBa
6345      //|.else
6346      //|  mov RB, [RA]
6347      //|  mov [KBASE], RB
6348      //|  mov RB, [RA+4]
6349      //|  add RA, 8
6350      //|  mov [KBASE+4], RB
6351      //|.endif
6352      //|  add KBASE, 8
6353      //|  sub NARGS:RD, 1
6354      //|  jnz <2
6355      //|
6356      //|  mov LFUNC:RB, [BASE-8]
6357      //|3:
```

```
6358      //|  mov NARGS:RD, MULTRES
6359      //|  cmp byte LFUNC:RB->ffid, 1        // (> FF_C) Calling a fast function?
6360      //|  ja >5
6361      //|4:
6362      //|  ins_callt
6363      //|
6364      //|5:  // Tailcall to a fast function.
6365      //|  test PC, FRAME_TYPE                // Lua frame below?
6366      dasm_put(Dst, 13030, LJ_TFUNC, FRAME_TYPE, Dt7(->ffid), Dt7(->pc));
6367  #line 4664 "vm_x86.dasc"
6368      //|  jnz <4
6369      //|  movzx RA, PC_RA
6370      //|  not RAa
6371      //|  mov LFUNC:KBASE, [BASE+RA*8-8]        // Need to prepare KBASE.
6372      //|  mov KBASE, LFUNC:KBASE->pc
6373      //|  mov KBASE, [KBASE+PC2PROTO(k)]
6374      //|  jmp <4
6375      //|
6376      //|7:  // Tailcall from a vararg function.
6377      //|  sub PC, FRAME_VARG
6378      //|  test PC, FRAME_TYPEP
6379      //|  jnz >8                             // Vararg frame below?
6380      //|  sub BASE, PC                      // Need to relocate BASE/KBASE down.
6381      //|  mov KBASE, BASE
6382      //|  mov PC, [BASE-4]
6383      //|  jmp <1
6384      //|8:
6385      //|  add PC, FRAME_VARG
6386      //|  jmp <1
6387      dasm_put(Dst, 13148, FRAME_TYPE, Dt7(->pc), PC2PROTO(k), FRAME_VARG, FRAME_TYPEP, FRAME_VARG);
6388  #line 4683 "vm_x86.dasc"
6389      break;
6390
6391    case BC_ITERC:
6392      //|  ins_A        // RA = base, (RB = nresults+1,) RC = nargs+1 (2+1)
6393      //|  lea RA, [BASE+RA*8+8]                // fb = base+1
6394      //|.if X64
6395      //|  mov RBa, [RA-24]                      // Copy state. fb[0] = fb[-3].
6396      //|  mov RCa, [RA-16]                      // Copy control var. fb[1] = fb[-2].
6397      //|  mov [RA], RBa
6398      //|  mov [RA+8], RCa
6399      //|.else
6400      //|  mov RB, [RA-24]                      // Copy state. fb[0] = fb[-3].
6401      //|  mov RC, [RA-20]
6402      //|  mov [RA], RB
6403      //|  mov [RA+4], RC
6404      //|  mov RB, [RA-16]                      // Copy control var. fb[1] = fb[-2].
6405      //|  mov RC, [RA-12]
6406      //|  mov [RA+8], RB
6407      //|  mov [RA+12], RC
6408      //|.endif
6409      //|  mov LFUNC:RB, [RA-32]                // Copy callable. fb[-1] = fb[-4]
6410      //|  mov RC, [RA-28]
6411      //|  mov [RA-8], LFUNC:RB
6412      //|  mov [RA-4], RC
6413      //|  cmp RC, LJ_TFUNC                      // Handle like a regular 2-arg call.
6414      //|  mov NARGS:RD, 2+1
6415      //|  jne ->vmeta_call
6416      //|  mov BASE, RA
6417      //|  ins_call
6418      dasm_put(Dst, 13220, LJ_TFUNC, 2+1, Dt7(->pc));
6419  #line 4712 "vm_x86.dasc"
6420      break;
6421
6422    case BC_ITERN:
6423      //|  ins_A        // RA = base, (RB = nresults+1, RC = nargs+1 (2+1))
6424      //|.if JIT
6425      //|  // NYI: add hotloop, record BC_ITERN.
6426      //|.endif
6427      //|  mov TMP1, KBASE                       // Need two more free registers.
6428      //|  mov TMP2, DISPATCH
6429      //|  mov TAB:RB, [BASE+RA*8-16]
6430      //|  mov RC, [BASE+RA*8-8]                // Get index from control var.
6431      //|  mov DISPATCH, TAB:RB->asize
6432      //|  add PC, 4
6433      //|  mov KBASE, TAB:RB->array
```

```
6434      //|1:  // Traverse array part.
6435      //|  cmp RC, DISPATCH; jae >5                // Index points after array part?
6436      //|  cmp dword [KBASE+RC*8+4], LJ_TNIL; je >4
6437      //|.if DUALNUM
6438      //|  mov dword [BASE+RA*8+4], LJ_TISNUM
6439      //|  mov dword [BASE+RA*8], RC
6440      //|.else
6441      //|  cvtsi2sd xmm0, RC
6442      //|.endif
6443      //|  // Copy array slot to returned value.
6444      //|.if X64
6445      //|  mov RBa, [KBASE+RC*8]
6446      //|  mov [BASE+RA*8+8], RBa
6447      //|.else
6448      //|  mov RB, [KBASE+RC*8+4]
6449      //|  mov [BASE+RA*8+12], RB
6450      //|  mov RB, [KBASE+RC*8]
6451      //|  mov [BASE+RA*8+8], RB
6452      //|.endif
6453      //|  add RC, 1
6454      //|  // Return array index as a numeric key.
6455      //|.if DUALNUM
6456      //|  // See above.
6457      //|.else
6458      //|  movsd qword [BASE+RA*8], xmm0
6459      //|.endif
6460      //|  mov [BASE+RA*8-8], RC                // Update control var.
6461      //|2:
6462      //|  movzx RD, PC_RD                        // Get target from ITERL.
6463      //|  branchPC RD
6464      //|3:
6465      //|  mov DISPATCH, TMP2
6466      //|  mov KBASE, TMP1
6467      //|  ins_next
6468      //|
6469      //|4:  // Skip holes in array part.
6470      //|  add RC, 1
6471      //|  jmp <1
6472      //|
6473      //|5:  // Traverse hash part.
6474      //|  sub RC, DISPATCH
6475      //|6:
6476      //|  cmp RC, TAB:RB->hmask; ja <3        // End of iteration? Branch to ITERL+1.
6477      //|  imul KBASE, RC, #NODE
6478      dasm_put(Dst, 13292, Dt6(->asize), Dt6(->array), LJ_TNIL, LJ_TISNUM, -BCBIAS_J*4, Dt6(->hmask));
6479  #line 4770 "vm_x86.dasc"
6480      //|  add NODE:KBASE, TAB:RB->node
6481      //|  cmp dword NODE:KBASE->val.it, LJ_TNIL; je >7
6482      //|  lea DISPATCH, [RC+DISPATCH+1]
6483      //|  // Copy key and value from hash slot.
6484      //|.if X64
6485      //|  mov RBa, NODE:KBASE->key
6486      //|  mov RCa, NODE:KBASE->val
6487      //|  mov [BASE+RA*8], RBa
6488      //|  mov [BASE+RA*8+8], RCa
6489      //|.else
6490      //|  mov RB, NODE:KBASE->key.gcr
6491      //|  mov RC, NODE:KBASE->key.it
6492      //|  mov [BASE+RA*8], RB
6493      //|  mov [BASE+RA*8+4], RC
6494      //|  mov RB, NODE:KBASE->val.gcr
6495      //|  mov RC, NODE:KBASE->val.it
6496      //|  mov [BASE+RA*8+8], RB
6497      //|  mov [BASE+RA*8+12], RC
6498      //|.endif
6499      //|  mov [BASE+RA*8-8], DISPATCH
6500      //|  jmp <2
6501      //|
6502      //|7:  // Skip holes in hash part.
6503      //|  add RC, 1
6504      //|  jmp <6
6505      dasm_put(Dst, 13436, sizeof(Node), Dt6(->node), DtB(->val.it), LJ_TNIL, DtB(->key), DtB(->val));
6506  #line 4795 "vm_x86.dasc"
6507      break;
6508
6509    case BC_ISNEXT:
```

```
6510      //|  ins_AD        // RA = base, RD = target (points to ITERN)
6511      //|  cmp dword [BASE+RA*8-20], LJ_TFUNC; jne >5
6512      //|  mov CFUNC:RB, [BASE+RA*8-24]
6513      //|  cmp dword [BASE+RA*8-12], LJ_TTAB; jne >5
6514      //|  cmp dword [BASE+RA*8-4], LJ_TNIL; jne >5
6515      //|  cmp byte CFUNC:RB->ffid, FF_next_N; jne >5
6516      //|  branchPC RD
6517      //|  mov dword [BASE+RA*8-8], 0        // Initialize control var.
6518      //|  mov dword [BASE+RA*8-4], 0xfffe7fff
6519      //|1:
6520      //|  ins_next
6521      //|5:  // Despecialize bytecode if any of the checks fail.
6522      //|  mov PC_OP, BC_JMP
6523      //|  branchPC RD
6524      //|  mov byte [PC], BC_ITERC
6525      //|  jmp <1
6526      dasm_put(Dst, 13496, LJ_TFUNC, LJ_TTAB, LJ_TNIL, Dt8(->ffid), FF_next_N, -BCBIAS_J*4, BC_JMP, -
BCBIAS_J*4, BC_ITERC);
6527    #line 4814 "vm_x86.dasc"
6528      break;
6529
6530    case BC_VARG:
6531      //|  ins_ABC        // RA = base, RB = nresults+1, RC = numparams
6532      //|  mov TMP1, KBASE                        // Need one more free register.
6533      //|  lea KBASE, [BASE+RC*8+(8+FRAME_VARG)]
6534      //|  lea RA, [BASE+RA*8]
6535      //|  sub KBASE, [BASE-4]
6536      //|  // Note: KBASE may now be even _above_ BASE if nargs was < numparams.
6537      //|  test RB, RB
6538      //|  jz >5                                 // Copy all varargs?
6539      //|  lea RB, [RA+RB*8-8]
6540      //|  cmp KBASE, BASE                        // No vararg slots?
6541      //|  jnb >2
6542      //|1:  // Copy vararg slots to destination slots.
6543      //|.if X64
6544      //|  mov RCa, [KBASE-8]
6545      //|  add KBASE, 8
6546      //|  mov [RA], RCa
6547      //|.else
6548      //|  mov RC, [KBASE-8]
6549      //|  mov [RA], RC
6550      //|  mov RC, [KBASE-4]
6551      //|  add KBASE, 8
6552      //|  mov [RA+4], RC
6553      //|.endif
6554      //|  add RA, 8
6555      //|  cmp RA, RB                          // All destination slots filled?
6556      //|  jnb >3
6557      //|  cmp KBASE, BASE                        // No more vararg slots?
6558      //|  jb <1
6559      //|2:  // Fill up remainder with nil.
6560      //|  mov dword [RA+4], LJ_TNIL
6561      //|  add RA, 8
6562      //|  cmp RA, RB
6563      //|  jb <2
6564      //|3:
6565      //|  mov KBASE, TMP1
6566      //|  ins_next
6567      //|
6568      //|5:  // Copy all varargs.
6569      //|  mov MULTRES, 1                       // MULTRES = 0+1
6570      //|  mov RC, BASE
6571      //|  sub RC, KBASE
6572      //|  jbe <3                                // No vararg slots?
6573      //|  mov RB, RC
6574      //|  shr RB, 3
6575      //|  add RB, 1
6576      //|  mov MULTRES, RB                        // MULTRES = #varargs+1
6577      //|  mov L:RB, SAVE_L
6578      //|  add RC, RA
6579      //|  cmp RC, L:RB->maxstack
6580      //|  ja >7                                 // Need to grow stack?
6581      //|6:  // Copy all vararg slots.
6582      //|.if X64
6583      //|  mov RCa, [KBASE-8]
6584      dasm_put(Dst, 13609, (8+FRAME_VARG), LJ_TNIL, Dt1(->maxstack));
```

```
#line 4870 "vm_x86.dasc"
    //|  add KBASE, 8
    //|  mov [RA], RCa
    //|.else
    //|  mov RC, [KBASE-8]
    //|  mov [RA], RC
    //|  mov RC, [KBASE-4]
    //|  add KBASE, 8
    //|  mov [RA+4], RC
    //|.endif
    //|  add RA, 8
    //|  cmp KBASE, BASE                        // No more vararg slots?
    //|  jb <6
    //|  jmp <3
    //|
    //|7:  // Grow stack for varargs.
    //|  mov L:RB->base, BASE
    //|  mov L:RB->top, RA
    //|  mov SAVE_PC, PC
    //|  sub KBASE, BASE                        // Need delta, because BASE may change.
    //|  mov FCARG2, MULTRES
    //|  sub FCARG2, 1
    //|  mov FCARG1, L:RB
    //|  call extern lj_state_growstack@8        // (lua_State *L, int n)
    //|  mov BASE, L:RB->base
    //|  mov RA, L:RB->top
    //|  add KBASE, BASE
    //|  jmp <6
    dasm_put(Dst, 13776, Dt1(->base), Dt1(->top), Dt1(->base), Dt1(->top));
#line 4897 "vm_x86.dasc"
    break;

  /* -- Returns ------------------------------------------------------------ */

  case BC_RETM:
    //|  ins_AD        // RA = results, RD = extra_nresults
    //|  add RD, MULTRES                        // MULTRES >=1, so RD >=1.
    //|  // Fall through. Assumes BC_RET follows and ins_AD is a no-op.
    dasm_put(Dst, 12982);
#line 4905 "vm_x86.dasc"
    break;

  case BC_RET: case BC_RET0: case BC_RET1:
    //|  ins_AD        // RA = results, RD = nresults+1
    if (op != BC_RET0) {
      //|  shl RA, 3
      dasm_put(Dst, 13846);
#line 4911 "vm_x86.dasc"
    }
    //|1:
    //|  mov PC, [BASE-4]
    //|  mov MULTRES, RD                        // Save nresults+1.
    //|  test PC, FRAME_TYPE                  // Check frame type marker.
    //|  jnz >7                                // Not returning to a fixarg Lua func?
    dasm_put(Dst, 13850, FRAME_TYPE);
#line 4917 "vm_x86.dasc"
    switch (op) {
    case BC_RET:
      //|->BC_RET_Z:
      //|  mov KBASE, BASE                // Use KBASE for result move.
      //|  sub RD, 1
      //|  jz >3
      //|2:  // Move results down.
      //|.if X64
      //|  mov RBa, [KBASE+RA]
      //|  mov [KBASE-8], RBa
      //|.else
      //|  mov RB, [KBASE+RA]
      //|  mov [KBASE-8], RB
      //|  mov RB, [KBASE+RA+4]
      //|  mov [KBASE-4], RB
      //|.endif
      //|  add KBASE, 8
      //|  sub RD, 1
      //|  jnz <2
      //|3:
```

```
6661        //|  mov RD, MULTRES                // Note: MULTRES may be >255.
6662        //|  movzx RB, PC_RB                // So cannot compare with RDL!
6663        //|5:
6664        //|  cmp RB, RD                        // More results expected?
6665        //|  ja >6
6666        dasm_put(Dst, 13869);
6667 #line 4942 "vm_x86.dasc"
6668        break;
6669      case BC_RET1:
6670        //|.if X64
6671        //|  mov RBa, [BASE+RA]
6672        //|  mov [BASE-8], RBa
6673        //|.else
6674        //|  mov RB, [BASE+RA+4]
6675        //|  mov [BASE-4], RB
6676        //|  mov RB, [BASE+RA]
6677        //|  mov [BASE-8], RB
6678        //|.endif
6679        dasm_put(Dst, 13923);
6680 #line 4953 "vm_x86.dasc"
6681        /* fallthrough */
6682      case BC_RET0:
6683        //|5:
6684        //|  cmp PC_RB, RDL                          // More results expected?
6685        //|  ja >6
6686        dasm_put(Dst, 13933);
6687 #line 4958 "vm_x86.dasc"
6688      default:
6689        break;
6690      }
6691      //|  movzx RA, PC_RA
6692      //|  not RAa                                  // Note: ~RA = -(RA+1)
6693      //|  lea BASE, [BASE+RA*8]                 // base = base - (RA+1)*8
6694      //|  mov LFUNC:KBASE, [BASE-8]
6695      //|  mov KBASE, LFUNC:KBASE->pc
6696      //|  mov KBASE, [KBASE+PC2PROTO(k)]
6697      //|  ins_next
6698      //|
6699      //|6:  // Fill up results with nil.
6700      dasm_put(Dst, 13944, Dt7(->pc), PC2PROTO(k));
6701 #line 4970 "vm_x86.dasc"
6702      if (op == BC_RET) {
6703        //|  mov dword [KBASE-4], LJ_TNIL        // Note: relies on shifted base.
6704        //|  add KBASE, 8
6705        dasm_put(Dst, 13992, LJ_TNIL);
6706 #line 4973 "vm_x86.dasc"
6707      } else {
6708        //|  mov dword [BASE+RD*8-12], LJ_TNIL
6709        dasm_put(Dst, 14003, LJ_TNIL);
6710 #line 4975 "vm_x86.dasc"
6711      }
6712      //|  add RD, 1
6713      //|  jmp <5
6714      //|
6715      //|7:  // Non-standard return case.
6716      //|  lea RB, [PC-FRAME_VARG]
6717      //|  test RB, FRAME_TYPEP
6718      //|  jnz ->vm_return
6719      //|  // Return from vararg function: relocate BASE down and RA up.
6720      //|  sub BASE, RB
6721      dasm_put(Dst, 14010, -FRAME_VARG, FRAME_TYPEP);
6722 #line 4985 "vm_x86.dasc"
6723      if (op != BC_RET0) {
6724        //|  add RA, RB
6725        dasm_put(Dst, 14034);
6726 #line 4987 "vm_x86.dasc"
6727      }
6728      //|  jmp <1
6729      dasm_put(Dst, 10015);
6730 #line 4989 "vm_x86.dasc"
6731      break;
6732
6733    /* -- Loops and branches ------------------------------------------- */
6734
6735    //|.define FOR_IDX,  [RA];    .define FOR_TIDX,  dword [RA+4]
6736    //|.define FOR_STOP, [RA+8];  .define FOR_TSTOP, dword [RA+12]
```

```
6737   //|.define FOR_STEP, [RA+16]; .define FOR_TSTEP, dword [RA+20]
6738   //|.define FOR_EXT,  [RA+24]; .define FOR_TEXT,  dword [RA+28]
6739
6740   case BC_FORL:
6741     //|.if JIT
6742     //|  hotloop RB
6743     //|.endif
6744     //| // Fall through. Assumes BC_IFORL follows and ins_AJ is a no-op.
6745     dasm_put(Dst, 14038, HOTCOUNT_PCMASK, GG_DISP2HOT, HOTCOUNT_LOOP);
6746 #line 5003 "vm_x86.dasc"
6747     break;
6748
6749   case BC_JFORI:
6750   case BC_JFORL:
6751 #if !LJ_HASJIT
6752     break;
6753 #endif
6754   case BC_FORI:
6755   case BC_IFORL:
6756     vk = (op == BC_IFORL || op == BC_JFORL);
6757     //|  ins_AJ        // RA = base, RD = target (after end of loop or start of loop)
6758     //|  lea RA, [BASE+RA*8]
6759     dasm_put(Dst, 14059);
6760 #line 5015 "vm_x86.dasc"
6761     if (LJ_DUALNUM) {
6762       //|  cmp FOR_TIDX, LJ_TISNUM; jne >9
6763       dasm_put(Dst, 14063, LJ_TISNUM);
6764 #line 5017 "vm_x86.dasc"
6765       if (!vk) {
6766         //|  cmp FOR_TSTOP, LJ_TISNUM; jne ->vmeta_for
6767         //|  cmp FOR_TSTEP, LJ_TISNUM; jne ->vmeta_for
6768         //|  mov RB, dword FOR_IDX
6769         //|  cmp dword FOR_STEP, 0; jl >5
6770         dasm_put(Dst, 14073, LJ_TISNUM, LJ_TISNUM);
6771 #line 5022 "vm_x86.dasc"
6772       } else {
6773 #ifdef LUA_USE_ASSERT
6774         //|  cmp FOR_TSTOP, LJ_TISNUM; jne ->assert_bad_for_arg_type
6775         //|  cmp FOR_TSTEP, LJ_TISNUM; jne ->assert_bad_for_arg_type
6776         dasm_put(Dst, 14102, LJ_TISNUM, LJ_TISNUM);
6777 #line 5026 "vm_x86.dasc"
6778 #endif
6779         //|  mov RB, dword FOR_STEP
6780         //|  test RB, RB; js >5
6781         //|  add RB, dword FOR_IDX; jo >1
6782         //|  mov dword FOR_IDX, RB
6783         dasm_put(Dst, 14121);
6784 #line 5031 "vm_x86.dasc"
6785       }
6786       //|  cmp RB, dword FOR_STOP
6787       //|  mov FOR_TEXT, LJ_TISNUM
6788       //|  mov dword FOR_EXT, RB
6789       dasm_put(Dst, 14140, LJ_TISNUM);
6790 #line 5035 "vm_x86.dasc"
6791       if (op == BC_FORI) {
6792         //|  jle >7
6793         //|1:
6794         //|6:
6795         //|  branchPC RD
6796         dasm_put(Dst, 14151, -BCBIAS_J*4);
6797 #line 5040 "vm_x86.dasc"
6798       } else if (op == BC_JFORI) {
6799         //|  branchPC RD
6800         //|  movzx RD, PC_RD
6801         //|  jle =>BC_JLOOP
6802         //|1:
6803         //|6:
6804         dasm_put(Dst, 14165, -BCBIAS_J*4, BC_JLOOP);
6805 #line 5046 "vm_x86.dasc"
6806       } else if (op == BC_IFORL) {
6807         //|  jg >7
6808         //|6:
6809         //|  branchPC RD
6810         //|1:
6811         dasm_put(Dst, 14183, -BCBIAS_J*4);
6812 #line 5051 "vm_x86.dasc"
```

```
6813        } else {
6814          //|  jle =>BC_JLOOP
6815          //|1:
6816          //|6:
6817          dasm_put(Dst, 14175, BC_JLOOP);
6818  #line 5055 "vm_x86.dasc"
6819        }
6820        //|7:
6821        //|  ins_next
6822        //|
6823        //|5:  // Invert check for negative step.
6824        dasm_put(Dst, 14197);
6825  #line 5060 "vm_x86.dasc"
6826        if (vk) {
6827          //|  add RB, dword FOR_IDX; jo <1
6828          //|  mov dword FOR_IDX, RB
6829          dasm_put(Dst, 14222);
6830  #line 5063 "vm_x86.dasc"
6831        }
6832        //|  cmp RB, dword FOR_STOP
6833        //|  mov FOR_TEXT, LJ_TISNUM
6834        //|  mov dword FOR_EXT, RB
6835        dasm_put(Dst, 14140, LJ_TISNUM);
6836  #line 5067 "vm_x86.dasc"
6837        if (op == BC_FORI) {
6838          //|  jge <7
6839          dasm_put(Dst, 14231);
6840  #line 5069 "vm_x86.dasc"
6841        } else if (op == BC_JFORI) {
6842          //|  branchPC RD
6843          //|  movzx RD, PC_RD
6844          //|  jge =>BC_JLOOP
6845          dasm_put(Dst, 14236, -BCBIAS_J*4, BC_JLOOP);
6846  #line 5073 "vm_x86.dasc"
6847        } else if (op == BC_IFORL) {
6848          //|  jl <7
6849          dasm_put(Dst, 14250);
6850  #line 5075 "vm_x86.dasc"
6851        } else {
6852          //|  jge =>BC_JLOOP
6853          dasm_put(Dst, 14246, BC_JLOOP);
6854  #line 5077 "vm_x86.dasc"
6855        }
6856        //|  jmp <6
6857        //|9:  // Fallback to FP variant.
6858        dasm_put(Dst, 14255);
6859  #line 5080 "vm_x86.dasc"
6860      } else if (!vk) {
6861        //|  cmp FOR_TIDX, LJ_TISNUM
6862        dasm_put(Dst, 14262, LJ_TISNUM);
6863  #line 5082 "vm_x86.dasc"
6864      }
6865      if (!vk) {
6866        //|  jae ->vmeta_for
6867        //|  cmp FOR_TSTOP, LJ_TISNUM; jae ->vmeta_for
6868        dasm_put(Dst, 14268, LJ_TISNUM);
6869  #line 5086 "vm_x86.dasc"
6870      } else {
6871  #ifdef LUA_USE_ASSERT
6872        //|  cmp FOR_TSTOP, LJ_TISNUM; jae ->assert_bad_for_arg_type
6873        //|  cmp FOR_TSTEP, LJ_TISNUM; jae ->assert_bad_for_arg_type
6874        dasm_put(Dst, 14282, LJ_TISNUM, LJ_TISNUM);
6875  #line 5090 "vm_x86.dasc"
6876  #endif
6877      }
6878      //|  mov RB, FOR_TSTEP                // Load type/hiword of for step.
6879      dasm_put(Dst, 14301);
6880  #line 5093 "vm_x86.dasc"
6881      if (!vk) {
6882        //|  cmp RB, LJ_TISNUM; jae ->vmeta_for
6883        dasm_put(Dst, 14305, LJ_TISNUM);
6884  #line 5095 "vm_x86.dasc"
6885      }
6886      //|  movsd xmm0, qword FOR_IDX
6887      //|  movsd xmm1, qword FOR_STOP
6888      dasm_put(Dst, 14314);
```

```
#line 5098 "vm_x86.dasc"
    if (vk) {
      //|  addsd xmm0, qword FOR_STEP
      //|  movsd qword FOR_IDX, xmm0
      //|  test RB, RB; js >3
      dasm_put(Dst, 14326);
#line 5102 "vm_x86.dasc"
    } else {
      //|  jl >3
      dasm_put(Dst, 14345);
#line 5104 "vm_x86.dasc"
    }
    //|  ucomisd xmm1, xmm0
    //|1:
    //|  movsd qword FOR_EXT, xmm0
    dasm_put(Dst, 14350);
#line 5108 "vm_x86.dasc"
    if (op == BC_FORI) {
      //|.if DUALNUM
      //|  jnb <7
      //|.else
      //|  jnb >2
      //|  branchPC RD
      //|.endif
      dasm_put(Dst, 14363);
#line 5115 "vm_x86.dasc"
    } else if (op == BC_JFORI) {
      //|  branchPC RD
      //|  movzx RD, PC_RD
      //|  jnb =>BC_JLOOP
      dasm_put(Dst, 14368, -BCBIAS_J*4, BC_JLOOP);
#line 5119 "vm_x86.dasc"
    } else if (op == BC_IFORL) {
      //|.if DUALNUM
      //|  jb <7
      //|.else
      //|  jb >2
      //|  branchPC RD
      //|.endif
      dasm_put(Dst, 14382);
#line 5126 "vm_x86.dasc"
    } else {
      //|  jnb =>BC_JLOOP
      dasm_put(Dst, 14378, BC_JLOOP);
#line 5128 "vm_x86.dasc"
    }
    //|.if DUALNUM
    //|  jmp <6
    //|.else
    //|2:
    //|  ins_next
    //|.endif
    //|
    //|3:  // Invert comparison if step is negative.
    //|  ucomisd xmm0, xmm1
    //|  jmp <1
    dasm_put(Dst, 14387);
#line 5139 "vm_x86.dasc"
    break;

  case BC_ITERL:
    //|.if JIT
    //|  hotloop RB
    //|.endif
    //| // Fall through. Assumes BC_IITERL follows and ins_AJ is a no-op.
    dasm_put(Dst, 14038, HOTCOUNT_PCMASK, GG_DISP2HOT, HOTCOUNT_LOOP);
#line 5146 "vm_x86.dasc"
    break;

  case BC_JITERL:
#if !LJ_HASJIT
    break;
#endif
  case BC_IITERL:
    //|  ins_AJ        // RA = base, RD = target
    //|  lea RA, [BASE+RA*8]
```

```
6965      //|  mov RB, [RA+4]
6966      //|  cmp RB, LJ_TNIL; je >1                // Stop if iterator returned nil.
6967      dasm_put(Dst, 14402, LJ_TNIL);
6968  #line 5157 "vm_x86.dasc"
6969      if (op == BC_JITERL) {
6970        //|  mov [RA-4], RB
6971        //|  mov RB, [RA]
6972        //|  mov [RA-8], RB
6973        //|  jmp =>BC_JLOOP
6974        dasm_put(Dst, 14417, BC_JLOOP);
6975  #line 5162 "vm_x86.dasc"
6976      } else {
6977        //|  branchPC RD                          // Otherwise save control var + branch.
6978        //|  mov RD, [RA]
6979        //|  mov [RA-4], RB
6980        //|  mov [RA-8], RD
6981        dasm_put(Dst, 14431, -BCBIAS_J*4);
6982  #line 5167 "vm_x86.dasc"
6983      }
6984      //|1:
6985      //|  ins_next
6986      dasm_put(Dst, 9668);
6987  #line 5170 "vm_x86.dasc"
6988      break;
6989
6990    case BC_LOOP:
6991      //|  ins_A        // RA = base, RD = target (loop extent)
6992      //|  // Note: RA/RD is only used by trace recorder to determine scope/extent
6993      //|  // This opcode does NOT jump, it's only purpose is to detect a hot loop.
6994      //|.if JIT
6995      //|  hotloop RB
6996      //|.endif
6997      //| // Fall through. Assumes BC_ILOOP follows and ins_A is a no-op.
6998      dasm_put(Dst, 14038, HOTCOUNT_PCMASK, GG_DISP2HOT, HOTCOUNT_LOOP);
6999  #line 5180 "vm_x86.dasc"
7000      break;
7001
7002    case BC_ILOOP:
7003      //|  ins_A        // RA = base, RD = target (loop extent)
7004      //|  ins_next
7005      dasm_put(Dst, 9224);
7006  #line 5185 "vm_x86.dasc"
7007      break;
7008
7009    case BC_JLOOP:
7010      //|.if JIT
7011      //|  ins_AD        // RA = base (ignored), RD = traceno
7012  #ifdef LUA_USE_TRACE_LOGS
7013      //|.if X64
7014      //|  mov L:RB, SAVE_L
7015      //|  mov L:RB->base, BASE  // Save BASE
7016      //|  mov TMP1, RD     // Save RD
7017      //|  mov CARG3d, PC  // CARG3d == BASE
7018      //|  mov FCARG2, RD
7019      //|  mov FCARG1, RB
7020      //|  call extern lj_log_trace_entry@8
7021      //|  mov RD, TMP1
7022      //|  mov BASE, L:RB->base
7023      //|.endif
7024      dasm_put(Dst, 14447, Dt1(->base), Dt1(->base));
7025  #line 5202 "vm_x86.dasc"
7026  #endif
7027      //|  mov RA, [DISPATCH+DISPATCH_J(trace)]
7028      //|  mov TRACE:RD, [RA+RD*4]
7029      //|  mov RDa, TRACE:RD->mcode
7030      //|  mov L:RB, SAVE_L
7031      //|  mov [DISPATCH+DISPATCH_GL(jit_base)], BASE
7032      //|  mov [DISPATCH+DISPATCH_GL(tmpbuf.L)], L:RB
7033      //|  // Save additional callee-save registers only used in compiled code.
7034      //|.if X64WIN
7035      //|  mov TMPQ, r12
7036      //|  mov TMPa, r13
7037      //|  mov CSAVE_4, r14
7038      //|  mov CSAVE_3, r15
7039      //|  mov RAa, rsp
7040      //|  sub rsp, 9*16+4*8
```

```
7041      //|  movdqa [RAa], xmm6
7042      //|  movdqa [RAa-1*16], xmm7
7043      //|  movdqa [RAa-2*16], xmm8
7044      //|  movdqa [RAa-3*16], xmm9
7045      //|  movdqa [RAa-4*16], xmm10
7046      //|  movdqa [RAa-5*16], xmm11
7047      //|  movdqa [RAa-6*16], xmm12
7048      //|  movdqa [RAa-7*16], xmm13
7049      //|  movdqa [RAa-8*16], xmm14
7050      //|  movdqa [RAa-9*16], xmm15
7051      //|.elif X64
7052      //|  mov TMPQ, r12
7053      //|  mov TMPa, r13
7054      //|  sub rsp, 16
7055      //|.endif
7056      //|  jmp RDa
7057      //|.endif
7058      dasm_put(Dst, 14475, DISPATCH_J(trace), DtD(->mcode), DISPATCH_GL(jit_base), DISPATCH_GL(tmpbuf.L));
7059  #line 5234 "vm_x86.dasc"
7060      break;
7061
7062    case BC_JMP:
7063      //|  ins_AJ        // RA = unused, RD = target
7064      //|  branchPC RD
7065      //|  ins_next
7066      dasm_put(Dst, 14516, -BCBIAS_J*4);
7067  #line 5240 "vm_x86.dasc"
7068      break;
7069
7070    /* -- Function headers ---------------------------------------------------- */
7071
7072     /*
7073     ** Reminder: A function may be called with func/args above L->maxstack,
7074     ** i.e. occupying EXTRA_STACK slots. And vmeta_call may add one extra slot,
7075     ** too. This means all FUNC* ops (including fast functions) must check
7076     ** for stack overflow _before_ adding more slots!
7077     */
7078
7079    case BC_FUNCF:
7080      //|.if JIT
7081      //|  hotcall RB
7082      //|.endif
7083      dasm_put(Dst, 14542, HOTCOUNT_PCMASK, GG_DISP2HOT, HOTCOUNT_CALL);
7084  #line 5255 "vm_x86.dasc"
7085    case BC_FUNCV:  /* NYI: compiled vararg functions. */
7086      //| // Fall through. Assumes BC_IFUNCF/BC_IFUNCV follow and ins_AD is a no-op.
7087      break;
7088
7089    case BC_JFUNCF:
7090  #if !LJ_HASJIT
7091      break;
7092  #endif
7093    case BC_IFUNCF:
7094      //|  ins_AD  // BASE = new base, RA = framesize, RD = nargs+1
7095      //|  mov KBASE, [PC-4+PC2PROTO(k)]
7096      //|  mov L:RB, SAVE_L
7097      //|  lea RA, [BASE+RA*8]               // Top of frame.
7098      //|  cmp RA, L:RB->maxstack
7099      //|  ja ->vm_growstack_f
7100      //|  movzx RA, byte [PC-4+PC2PROTO(numparams)]
7101      //|  cmp NARGS:RD, RA                        // Check for missing parameters.
7102      //|  jbe >3
7103      //|2:
7104      dasm_put(Dst, 14563, -4+PC2PROTO(k), Dt1(->maxstack), -4+PC2PROTO(numparams));
7105  #line 5274 "vm_x86.dasc"
7106      if (op == BC_JFUNCF) {
7107        //|  movzx RD, PC_RD
7108        //|  jmp =>BC_JLOOP
7109        dasm_put(Dst, 14594, BC_JLOOP);
7110  #line 5277 "vm_x86.dasc"
7111      } else {
7112        //|  ins_next
7113        dasm_put(Dst, 9224);
7114  #line 5279 "vm_x86.dasc"
7115      }
7116      //|
```

```
7117      //|3:  // Clear missing parameters.
7118      //|  mov dword [BASE+NARGS:RD*8-4], LJ_TNIL
7119      //|  add NARGS:RD, 1
7120      //|  cmp NARGS:RD, RA
7121      //|  jbe <3
7122      //|  jmp <2
7123      dasm_put(Dst, 14603, LJ_TNIL);
7124  #line 5287 "vm_x86.dasc"
7125      break;
7126
7127    case BC_JFUNCV:
7128  #if !LJ_HASJIT
7129      break;
7130  #endif
7131      //| int3  // NYI: compiled vararg functions
7132      dasm_put(Dst, 8467);
7133  #line 5294 "vm_x86.dasc"
7134      break;  /* NYI: compiled vararg functions. */
7135
7136    case BC_IFUNCV:
7137      //|  ins_AD  // BASE = new base, RA = framesize, RD = nargs+1
7138      //|  lea RB, [NARGS:RD*8+FRAME_VARG]
7139      //|  lea RD, [BASE+NARGS:RD*8]
7140      //|  mov LFUNC:KBASE, [BASE-8]
7141      //|  mov [RD-4], RB                        // Store delta + FRAME_VARG.
7142      //|  mov [RD-8], LFUNC:KBASE                // Store copy of LFUNC.
7143      //|  mov L:RB, SAVE_L
7144      //|  lea RA, [RD+RA*8]
7145      //|  cmp RA, L:RB->maxstack
7146      //|  ja ->vm_growstack_v                // Need to grow stack.
7147      //|  mov RA, BASE
7148      //|  mov BASE, RD
7149      //|  movzx RB, byte [PC-4+PC2PROTO(numparams)]
7150      //|  test RB, RB
7151      //|  jz >2
7152      //|1:  // Copy fixarg slots up to new frame.
7153      //|  add RA, 8
7154      //|  cmp RA, BASE
7155      //|  jnb >3                                // Less args than parameters?
7156      //|  mov KBASE, [RA-8]
7157      //|  mov [RD], KBASE
7158      //|  mov KBASE, [RA-4]
7159      //|  mov [RD+4], KBASE
7160      //|  add RD, 8
7161      //|  mov dword [RA-4], LJ_TNIL        // Clear old fixarg slot (help the GC).
7162      //|  sub RB, 1
7163      //|  jnz <1
7164      //|2:
7165      dasm_put(Dst, 14625, FRAME_VARG, Dt1(->maxstack), -4+PC2PROTO(numparams), LJ_TNIL);
7166  #line 5325 "vm_x86.dasc"
7167      if (op == BC_JFUNCV) {
7168        //|  movzx RD, PC_RD
7169        //|  jmp =>BC_JLOOP
7170        dasm_put(Dst, 14594, BC_JLOOP);
7171  #line 5328 "vm_x86.dasc"
7172      } else {
7173        //|  mov KBASE, [PC-4+PC2PROTO(k)]
7174        //|  ins_next
7175        dasm_put(Dst, 14722, -4+PC2PROTO(k));
7176  #line 5331 "vm_x86.dasc"
7177      }
7178      //|
7179      //|3:  // Clear missing parameters.
7180      //|  mov dword [RD+4], LJ_TNIL
7181      //|  add RD, 8
7182      //|  sub RB, 1
7183      //|  jnz <3
7184      //|  jmp <2
7185      dasm_put(Dst, 14747, LJ_TNIL);
7186  #line 5339 "vm_x86.dasc"
7187      break;
7188
7189    case BC_FUNCC:
7190    case BC_FUNCCW:
7191      //|  ins_AD  // BASE = new base, RA = ins RA|RD (unused), RD = nargs+1
7192      //|  mov CFUNC:RB, [BASE-8]
```

```
7193      //|  mov KBASEa, CFUNC:RB->f
7194      //|  mov L:RB, SAVE_L
7195      //|  lea RD, [BASE+NARGS:RD*8-8]
7196      //|  mov L:RB->base, BASE
7197      //|  lea RA, [RD+8*LUA_MINSTACK]
7198      //|  cmp RA, L:RB->maxstack
7199      //|  mov L:RB->top, RD
7200      dasm_put(Dst, 14769, Dt8(->f), Dt1(->base), 8*LUA_MINSTACK, Dt1(->maxstack), Dt1(->top));
7201  #line 5352 "vm_x86.dasc"
7202      if (op == BC_FUNCC) {
7203        //|.if X64
7204        //|  mov CARG1d, L:RB                        // Caveat: CARG1d may be RA.
7205        //|.else
7206        //|  mov ARG1, L:RB
7207        //|.endif
7208        dasm_put(Dst, 14799);
7209  #line 5358 "vm_x86.dasc"
7210      } else {
7211        //|.if X64
7212        //|  mov CARG2, KBASEa
7213        //|  mov CARG1d, L:RB                        // Caveat: CARG1d may be RA.
7214        //|.else
7215        //|  mov ARG2, KBASEa
7216        //|  mov ARG1, L:RB
7217        //|.endif
7218        dasm_put(Dst, 14803);
7219  #line 5366 "vm_x86.dasc"
7220      }
7221      //|  ja ->vm_growstack_c                // Need to grow stack.
7222      //|  set_vmstate C
7223      dasm_put(Dst, 14811, DISPATCH_GL(vmstate), ~LJ_VMST_C);
7224  #line 5369 "vm_x86.dasc"
7225      if (op == BC_FUNCC) {
7226        //|  call KBASEa                          // (lua_State *L)
7227        dasm_put(Dst, 14821);
7228  #line 5371 "vm_x86.dasc"
7229      } else {
7230        //|  // (lua_State *L, lua_CFunction f)
7231        //|  call aword [DISPATCH+DISPATCH_GL(wrapf)]
7232        dasm_put(Dst, 14826, DISPATCH_GL(wrapf));
7233  #line 5374 "vm_x86.dasc"
7234      }
7235      //|  // nresults returned in eax (RD).
7236      //|  mov BASE, L:RB->base
7237      //|  mov [DISPATCH+DISPATCH_GL(cur_L)], L:RB
7238      //|  set_vmstate INTERP
7239      //|  lea RA, [BASE+RD*8]
7240      //|  neg RA
7241      //|  add RA, L:RB->top                // RA = (L->top-(L->base+nresults))*8
7242      //|  mov PC, [BASE-4]                        // Fetch PC of caller.
7243      //|  jmp ->vm_returnc
7244      dasm_put(Dst, 14832, Dt1(->base), DISPATCH_GL(cur_L), DISPATCH_GL(vmstate), ~LJ_VMST_INTERP, Dt1(-
>top));
7245  #line 5384 "vm_x86.dasc"
7246      break;
7247
7248    /* ------------------------------------------------------------------------ */
7249
7250    default:
7251      fprintf(stderr, "Error: undefined opcode BC_%s\n", bc_names[op]);
7252      exit(2);
7253      break;
7254    }
7255  }
7256
7257  static int build_backend(BuildCtx *ctx)
7258  {
7259    int op;
7260    dasm_growpc(Dst, BC__MAX);
7261    build_subroutines(ctx);
7262    //|.code_op
7263    dasm_put(Dst, 14862);
7264  #line 5401 "vm_x86.dasc"
7265    for (op = 0; op < BC__MAX; op++)
7266      build_ins(ctx, (BCOp)op, op);
7267    return BC__MAX;
```

```
}

/* Emit pseudo frame-info for all assembler functions. */
static void emit_asm_debug(BuildCtx *ctx)
{
  int fcofs = (int)((uint8_t *)ctx->glob[GLOB_vm_ffi_call] - ctx->code);
#if LJ_64
#define SZPTR		"8"
#define BSZPTR		"3"
#define REG_SP		"0x7"
#define REG_RA		"0x10"
#else
#define SZPTR		"4"
#define BSZPTR		"2"
#define REG_SP		"0x4"
#define REG_RA		"0x8"
#endif
  switch (ctx->mode) {
  case BUILD_elfasm:
    fprintf(ctx->fp, "\t.section .debug_frame,\"\",@progbits\n");
    fprintf(ctx->fp,
        ".Lframe0:\n"
        "\t.long .LECIE0-.LSCIE0\n"
        ".LSCIE0:\n"
        "\t.long 0xffffffff\n"
        "\t.byte 0x1\n"
        "\t.string \"\"\n"
        "\t.uleb128 0x1\n"
        "\t.sleb128 -" SZPTR "\n"
        "\t.byte " REG_RA "\n"
        "\t.byte 0xc\n\t.uleb128 " REG_SP "\n\t.uleb128 " SZPTR "\n"
        "\t.byte 0x80+" REG_RA "\n\t.uleb128 0x1\n"
        "\t.align " SZPTR "\n"
        ".LECIE0:\n\n");
    fprintf(ctx->fp,
        ".LSFDE0:\n"
        "\t.long .LEFDE0-.LASFDE0\n"
        ".LASFDE0:\n"
        "\t.long .Lframe0\n"
#if LJ_64
        "\t.quad .Lbegin\n"
        "\t.quad %d\n"
        "\t.byte 0xe\n\t.uleb128 %d\n"		/* def_cfa_offset */
        "\t.byte 0x86\n\t.uleb128 0x2\n"	/* offset rbp */
        "\t.byte 0x83\n\t.uleb128 0x3\n"	/* offset rbx */
        "\t.byte 0x8f\n\t.uleb128 0x4\n"	/* offset r15 */
        "\t.byte 0x8e\n\t.uleb128 0x5\n"	/* offset r14 */
#else
        "\t.long .Lbegin\n"
        "\t.long %d\n"
        "\t.byte 0xe\n\t.uleb128 %d\n"		/* def_cfa_offset */
        "\t.byte 0x85\n\t.uleb128 0x2\n"	/* offset ebp */
        "\t.byte 0x87\n\t.uleb128 0x3\n"	/* offset edi */
        "\t.byte 0x86\n\t.uleb128 0x4\n"	/* offset esi */
        "\t.byte 0x83\n\t.uleb128 0x5\n"	/* offset ebx */
#endif
        "\t.align " SZPTR "\n"
        ".LEFDE0:\n\n", fcofs, CFRAME_SIZE);
#if LJ_HASFFI
    fprintf(ctx->fp,
        ".LSFDE1:\n"
        "\t.long .LEFDE1-.LASFDE1\n"
        ".LASFDE1:\n"
        "\t.long .Lframe0\n"
#if LJ_64
        "\t.quad lj_vm_ffi_call\n"
        "\t.quad %d\n"
        "\t.byte 0xe\n\t.uleb128 16\n"		/* def_cfa_offset */
        "\t.byte 0x86\n\t.uleb128 0x2\n"	/* offset rbp */
        "\t.byte 0xd\n\t.uleb128 0x6\n"		/* def_cfa_register rbp */
        "\t.byte 0x83\n\t.uleb128 0x3\n"	/* offset rbx */
#else
        "\t.long lj_vm_ffi_call\n"
        "\t.long %d\n"
        "\t.byte 0xe\n\t.uleb128 8\n"		/* def_cfa_offset */
        "\t.byte 0x85\n\t.uleb128 0x2\n"	/* offset ebp */
```

```
7344          "\t.byte 0xd\n\t.uleb128 0x5\n"                /* def_cfa_register ebp */
7345          "\t.byte 0x83\n\t.uleb128 0x3\n"         /* offset ebx */
7346  #endif
7347          "\t.align " SZPTR "\n"
7348          ".LEFDE1:\n\n", (int)ctx->codesz - fcofs);
7349  #endif
7350  #if (defined(__sun__) && defined(__svr4__))
7351  #if LJ_64
7352      fprintf(ctx->fp, "\t.section .eh_frame,\"a\",@unwind\n");
7353  #else
7354      fprintf(ctx->fp, "\t.section .eh_frame,\"aw\",@progbits\n");
7355  #endif
7356  #else
7357      fprintf(ctx->fp, "\t.section .eh_frame,\"a\",@progbits\n");
7358  #endif
7359      fprintf(ctx->fp,
7360          ".Lframe1:\n"
7361          "\t.long .LECIE1-.LSCIE1\n"
7362          ".LSCIE1:\n"
7363          "\t.long 0\n"
7364          "\t.byte 0x1\n"
7365          "\t.string \"zPR\"\n"
7366          "\t.uleb128 0x1\n"
7367          "\t.sleb128 -" SZPTR "\n"
7368          "\t.byte " REG_RA "\n"
7369          "\t.uleb128 6\n"                         /* augmentation length */
7370          "\t.byte 0x1b\n"                         /* pcrel|sdata4 */
7371          "\t.long lj_err_unwind_dwarf-.\n"
7372          "\t.byte 0x1b\n"                         /* pcrel|sdata4 */
7373          "\t.byte 0xc\n\t.uleb128 " REG_SP "\n\t.uleb128 " SZPTR "\n"
7374          "\t.byte 0x80+" REG_RA "\n\t.uleb128 0x1\n"
7375          "\t.align " SZPTR "\n"
7376          ".LECIE1:\n\n");
7377      fprintf(ctx->fp,
7378          ".LSFDE2:\n"
7379          "\t.long .LEFDE2-.LASFDE2\n"
7380          ".LASFDE2:\n"
7381          "\t.long .LASFDE2-.Lframe1\n"
7382          "\t.long .Lbegin-.\n"
7383          "\t.long %d\n"
7384          "\t.uleb128 0\n"                         /* augmentation length */
7385          "\t.byte 0xe\n\t.uleb128 %d\n"               /* def_cfa_offset */
7386  #if LJ_64
7387          "\t.byte 0x86\n\t.uleb128 0x2\n"         /* offset rbp */
7388          "\t.byte 0x83\n\t.uleb128 0x3\n"         /* offset rbx */
7389          "\t.byte 0x8f\n\t.uleb128 0x4\n"         /* offset r15 */
7390          "\t.byte 0x8e\n\t.uleb128 0x5\n"         /* offset r14 */
7391  #else
7392          "\t.byte 0x85\n\t.uleb128 0x2\n"         /* offset ebp */
7393          "\t.byte 0x87\n\t.uleb128 0x3\n"         /* offset edi */
7394          "\t.byte 0x86\n\t.uleb128 0x4\n"         /* offset esi */
7395          "\t.byte 0x83\n\t.uleb128 0x5\n"         /* offset ebx */
7396  #endif
7397          "\t.align " SZPTR "\n"
7398          ".LEFDE2:\n\n", fcofs, CFRAME_SIZE);
7399  #if LJ_HASFFI
7400      fprintf(ctx->fp,
7401          ".Lframe2:\n"
7402          "\t.long .LECIE2-.LSCIE2\n"
7403          ".LSCIE2:\n"
7404          "\t.long 0\n"
7405          "\t.byte 0x1\n"
7406          "\t.string \"zR\"\n"
7407          "\t.uleb128 0x1\n"
7408          "\t.sleb128 -" SZPTR "\n"
7409          "\t.byte " REG_RA "\n"
7410          "\t.uleb128 1\n"                         /* augmentation length */
7411          "\t.byte 0x1b\n"                         /* pcrel|sdata4 */
7412          "\t.byte 0xc\n\t.uleb128 " REG_SP "\n\t.uleb128 " SZPTR "\n"
7413          "\t.byte 0x80+" REG_RA "\n\t.uleb128 0x1\n"
7414          "\t.align " SZPTR "\n"
7415          ".LECIE2:\n\n");
7416      fprintf(ctx->fp,
7417          ".LSFDE3:\n"
7418          "\t.long .LEFDE3-.LASFDE3\n"
7419          ".LASFDE3:\n"
```

```
7420          "\t.long .LASFDE3-.Lframe2\n"
7421          "\t.long lj_vm_ffi_call-.\n"
7422          "\t.long %d\n"
7423          "\t.uleb128 0\n"                        /* augmentation length */
7424  #if LJ_64
7425          "\t.byte 0xe\n\t.uleb128 16\n"               /* def_cfa_offset */
7426          "\t.byte 0x86\n\t.uleb128 0x2\n"        /* offset rbp */
7427          "\t.byte 0xd\n\t.uleb128 0x6\n"               /* def_cfa_register rbp */
7428          "\t.byte 0x83\n\t.uleb128 0x3\n"        /* offset rbx */
7429  #else
7430          "\t.byte 0xe\n\t.uleb128 8\n"               /* def_cfa_offset */
7431          "\t.byte 0x85\n\t.uleb128 0x2\n"        /* offset ebp */
7432          "\t.byte 0xd\n\t.uleb128 0x5\n"               /* def_cfa_register ebp */
7433          "\t.byte 0x83\n\t.uleb128 0x3\n"        /* offset ebx */
7434  #endif
7435          "\t.align " SZPTR "\n"
7436          ".LEFDE3:\n\n", (int)ctx->codesz - fcofs);
7437  #endif
7438      break;
7439    /* Mental note: never let Apple design an assembler.
7440    ** Or a linker. Or a plastic case. But I digress.
7441    */
7442    case BUILD_machasm: {
7443  #if LJ_HASFFI
7444      int fcsize = 0;
7445  #endif
7446      int i;
7447      fprintf(ctx->fp, "\t.section __TEXT,__eh_frame,coalesced,no_toc+strip_static_syms+live_support\n");
7448      fprintf(ctx->fp,
7449          "EH_frame1:\n"
7450          "\t.set L$set$x,LECIEX-LSCIEX\n"
7451          "\t.long L$set$x\n"
7452          "LSCIEX:\n"
7453          "\t.long 0\n"
7454          "\t.byte 0x1\n"
7455          "\t.ascii \"zPR\\0\"\n"
7456          "\t.byte 0x1\n"
7457          "\t.byte 128-" SZPTR "\n"
7458          "\t.byte " REG_RA "\n"
7459          "\t.byte 6\n"                               /* augmentation length */
7460          "\t.byte 0x9b\n"                        /* indirect|pcrel|sdata4 */
7461  #if LJ_64
7462          "\t.long _lj_err_unwind_dwarf+4@GOTPCREL\n"
7463          "\t.byte 0x1b\n"                         /* pcrel|sdata4 */
7464          "\t.byte 0xc\n\t.byte " REG_SP "\n\t.byte " SZPTR "\n"
7465  #else
7466          "\t.long L_lj_err_unwind_dwarf$non_lazy_ptr-.\n"
7467          "\t.byte 0x1b\n"                         /* pcrel|sdata4 */
7468          "\t.byte 0xc\n\t.byte 0x5\n\t.byte 0x4\n"  /* esp=5 on 32 bit MACH-O. */
7469  #endif
7470          "\t.byte 0x80+" REG_RA "\n\t.byte 0x1\n"
7471          "\t.align " BSZPTR "\n"
7472          "LECIEX:\n\n");
7473      for (i = 0; i < ctx->nsym; i++) {
7474        const char *name = ctx->sym[i].name;
7475        int32_t size = ctx->sym[i+1].ofs - ctx->sym[i].ofs;
7476        if (size == 0) continue;
7477  #if LJ_HASFFI
7478        if (!strcmp(name, "_lj_vm_ffi_call")) { fcsize = size; continue; }
7479  #endif
7480        fprintf(ctx->fp,
7481            "%s.eh:\n"
7482            "LSFDE%d:\n"
7483            "\t.set L$set$%d,LEFDE%d-LASFDE%d\n"
7484            "\t.long L$set$%d\n"
7485            "LASFDE%d:\n"
7486            "\t.long LASFDE%d-EH_frame1\n"
7487            "\t.long %s-.\n"
7488            "\t.long %d\n"
7489            "\t.byte 0\n"                                 /* augmentation length */
7490            "\t.byte 0xe\n\t.byte %d\n"                /* def_cfa_offset */
7491  #if LJ_64
7492            "\t.byte 0x86\n\t.byte 0x2\n"                /* offset rbp */
7493            "\t.byte 0x83\n\t.byte 0x3\n"                /* offset rbx */
7494            "\t.byte 0x8f\n\t.byte 0x4\n"                /* offset r15 */
7495            "\t.byte 0x8e\n\t.byte 0x5\n"                /* offset r14 */
```

```
7496  #else
7497            "\t.byte 0x84\n\t.byte 0x2\n"                /* offset ebp (4 for MACH-O)*/
7498            "\t.byte 0x87\n\t.byte 0x3\n"                /* offset edi */
7499            "\t.byte 0x86\n\t.byte 0x4\n"                /* offset esi */
7500            "\t.byte 0x83\n\t.byte 0x5\n"                /* offset ebx */
7501  #endif
7502            "\t.align " BSZPTR "\n"
7503            "LEFDE%d:\n\n",
7504            name, i, i, i, i, i, i, i, name, size, CFRAME_SIZE, i);
7505      }
7506  #if LJ_HASFFI
7507      if (fcsize) {
7508        fprintf(ctx->fp,
7509            "EH_frame2:\n"
7510            "\t.set L$set$y,LECIEY-LSCIEY\n"
7511            "\t.long L$set$y\n"
7512            "LSCIEY:\n"
7513            "\t.long 0\n"
7514            "\t.byte 0x1\n"
7515            "\t.ascii \"zR\\0\"\n"
7516            "\t.byte 0x1\n"
7517            "\t.byte 128-" SZPTR "\n"
7518            "\t.byte " REG_RA "\n"
7519            "\t.byte 1\n"                                  /* augmentation length */
7520  #if LJ_64
7521            "\t.byte 0x1b\n"                           /* pcrel|sdata4 */
7522            "\t.byte 0xc\n\t.byte " REG_SP "\n\t.byte " SZPTR "\n"
7523  #else
7524            "\t.byte 0x1b\n"                           /* pcrel|sdata4 */
7525            "\t.byte 0xc\n\t.byte 0x5\n\t.byte 0x4\n"  /* esp=5 on 32 bit MACH. */
7526  #endif
7527            "\t.byte 0x80+" REG_RA "\n\t.byte 0x1\n"
7528            "\t.align " BSZPTR "\n"
7529            "LECIEY:\n\n");
7530        fprintf(ctx->fp,
7531            "_lj_vm_ffi_call.eh:\n"
7532            "LSFDEY:\n"
7533            "\t.set L$set$yy,LEFDEY-LASFDEY\n"
7534            "\t.long L$set$yy\n"
7535            "LASFDEY:\n"
7536            "\t.long LASFDEY-EH_frame2\n"
7537            "\t.long _lj_vm_ffi_call-.\n"
7538            "\t.long %d\n"
7539            "\t.byte 0\n"                                  /* augmentation length */
7540  #if LJ_64
7541            "\t.byte 0xe\n\t.byte 16\n"                 /* def_cfa_offset */
7542            "\t.byte 0x86\n\t.byte 0x2\n"                /* offset rbp */
7543            "\t.byte 0xd\n\t.byte 0x6\n"                 /* def_cfa_register rbp */
7544            "\t.byte 0x83\n\t.byte 0x3\n"                /* offset rbx */
7545  #else
7546            "\t.byte 0xe\n\t.byte 8\n"                  /* def_cfa_offset */
7547            "\t.byte 0x84\n\t.byte 0x2\n"                /* offset ebp (4 for MACH-O)*/
7548            "\t.byte 0xd\n\t.byte 0x4\n"                 /* def_cfa_register ebp */
7549            "\t.byte 0x83\n\t.byte 0x3\n"                /* offset ebx */
7550  #endif
7551            "\t.align " BSZPTR "\n"
7552            "LEFDEY:\n\n", fcsize);
7553      }
7554  #endif
7555  #if !LJ_64
7556      fprintf(ctx->fp,
7557        "\t.non_lazy_symbol_pointer\n"
7558        "L_lj_err_unwind_dwarf$non_lazy_ptr:\n"
7559        ".indirect_symbol _lj_err_unwind_dwarf\n"
7560        ".long 0\n\n");
7561      fprintf(ctx->fp, "\t.section
__IMPORT,__jump_table,symbol_stubs,pure_instructions+self_modifying_code,5\n");
7562      {
7563        const char *const *xn;
7564        for (xn = ctx->extnames; *xn; xn++)
7565          if (strncmp(*xn, LABEL_PREFIX, sizeof(LABEL_PREFIX)-1))
7566            fprintf(ctx->fp, "L_%s$stub:\n\t.indirect_symbol _%s\n\t.ascii \"\\364\\364\\364\\364\\364\"\n",
*xn, *xn);
7567      }
7568  #endif
7569      fprintf(ctx->fp, ".subsections_via_symbols\n");
```

```
7570      }
7571      break;
7572    default:  /* Difficult for other modes. */
7573      break;
7574    }
7575  }
7576
```

One Level Up Top Level

src/lj_state.c - luajit-2.0-src

Functions defined

Macros defined

Source code

```
1  /*
2  ** State and stack handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  **
5  ** Portions taken verbatim or adapted from the Lua interpreter.
6  ** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
7  */
8
9  #define lj_state_c
10  #define LUA_CORE
11
12  #include "lj_obj.h"
13  #include "lj_gc.h"
14  #include "lj_err.h"
15  #include "lj_buf.h"
16  #include "lj_str.h"
17  #include "lj_tab.h"
18  #include "lj_func.h"
```

```c
#include "lj_meta.h"
#include "lj_state.h"
#include "lj_frame.h"
#if LJ_HASFFI
#include "lj_ctype.h"
#endif
#include "lj_trace.h"
#include "lj_dispatch.h"
#include "lj_vm.h"
#include "lj_lex.h"
#include "lj_alloc.h"
#include "luajit.h"

/* -- Stack handling ------------------------------------------------------ */

/* Stack sizes. */
#define LJ_STACK_MIN        LUA_MINSTACK        /* Min. stack size. */
#define LJ_STACK_MAX        LUAI_MAXSTACK        /* Max. stack size. */
#define LJ_STACK_START        (2*LJ_STACK_MIN)        /* Starting stack size. */
#define LJ_STACK_MAXEX        (LJ_STACK_MAX + 1 + LJ_STACK_EXTRA)

/* Explanation of LJ_STACK_EXTRA:
**
** Calls to metamethods store their arguments beyond the current top
** without checking for the stack limit. This avoids stack resizes which
** would invalidate passed TValue pointers. The stack check is performed
** later by the function header. This can safely resize the stack or raise
** an error. Thus we need some extra slots beyond the current stack limit.
**
** Most metamethods need 4 slots above top (cont, mobj, arg1, arg2) plus
** one extra slot if mobj is not a function. Only lj_meta_tset needs 5
** slots above top, but then mobj is always a function. So we can get by
** with 5 extra slots.
** LJ_FR2: We need 2 more slots for the frame PC and the continuation PC.
*/

/* Resize stack slots and adjust pointers in state. */
static void resizestack(lua_State *L, MSize n)
{
  TValue *st, *oldst = tvref(L->stack);
  ptrdiff_t delta;
  MSize oldsize = L->stacksize;
  MSize realsize = n + 1 + LJ_STACK_EXTRA;
  GCobj *up;
  lua_assert((MSize)(tvref(L->maxstack)-oldst)==L->stacksize-LJ_STACK_EXTRA-1);
  st = (TValue *)lj_mem_realloc(L, tvref(L->stack),
                                (MSize)(oldsize*sizeof(TValue)),
                                (MSize)(realsize*sizeof(TValue)));
  setmref(L->stack, st);
  delta = (char *)st - (char *)oldst;
  setmref(L->maxstack, st + n);
  while (oldsize < realsize)  /* Clear new slots. */
    setnilV(st + oldsize++);
  L->stacksize = realsize;
  if ((size_t)(mref(G(L)->jit_base, char) - (char *)oldst) < oldsize)
    setmref(G(L)->jit_base, mref(G(L)->jit_base, char) + delta);
  L->base = (TValue *)((char *)L->base + delta);
  L->top = (TValue *)((char *)L->top + delta);
  for (up = gcref(L->openupval); up != NULL; up = gcnext(up))
    setmref(gco2uv(up)->v, (TValue *)((char *)uvval(gco2uv(up)) + delta));
}

/* Relimit stack after error, in case the limit was overdrawn. */
void lj_state_relimitstack(lua_State *L)
{
  if (L->stacksize > LJ_STACK_MAXEX && L->top-tvref(L->stack) < LJ_STACK_MAX-1)
    resizestack(L, LJ_STACK_MAX);
}

/* Try to shrink the stack (called from GC). */
void lj_state_shrinkstack(lua_State *L, MSize used)
{
  if (L->stacksize > LJ_STACK_MAXEX)
    return;  /* Avoid stack shrinking while handling stack overflow. */
  if (4*used < L->stacksize &&
      2*(LJ_STACK_START+LJ_STACK_EXTRA) < L->stacksize &&
```

```
      /* Don't shrink stack of live trace. */
      (tvref(G(L)->jit_base) == NULL || obj2gco(L) != gcref(G(L)->cur_L)))
    resizestack(L, L->stacksize >> 1);
}

/* Try to grow stack. */
void LJ_FASTCALL lj_state_growstack(lua_State *L, MSize need)
{
  MSize n;
  if (L->stacksize > LJ_STACK_MAXEX)  /* Overflow while handling overflow? */
    lj_err_throw(L, LUA_ERRERR);
  n = L->stacksize + need;
  if (n > LJ_STACK_MAX) {
    n += 2*LUA_MINSTACK;
  } else if (n < 2*L->stacksize) {
    n = 2*L->stacksize;
    if (n >= LJ_STACK_MAX)
      n = LJ_STACK_MAX;
  }
  resizestack(L, n);
  if (L->stacksize > LJ_STACK_MAXEX)
    lj_err_msg(L, LJ_ERR_STKOV);
}

void LJ_FASTCALL lj_state_growstack1(lua_State *L)
{
  lj_state_growstack(L, 1);
}

/* Allocate basic stack for new state. */
static void stack_init(lua_State *L1, lua_State *L)
{
  TValue *stend, *st = lj_mem_newvec(L, LJ_STACK_START+LJ_STACK_EXTRA, TValue);
  setmref(L1->stack, st);
  L1->stacksize = LJ_STACK_START + LJ_STACK_EXTRA;
  stend = st + L1->stacksize;
  setmref(L1->maxstack, stend - LJ_STACK_EXTRA - 1);
  setthreadV(L1, st++, L1);  /* Needed for curr_funcisL() on empty stack. */
  if (LJ_FR2) setnilV(st++);
  L1->base = L1->top = st;
  while (st < stend)  /* Clear new slots. */
    setnilV(st++);
}

/* -- State handling ------------------------------------------------------ */

/* Open parts that may cause memory-allocation errors. */
static TValue *cpluaopen(lua_State *L, lua_CFunction dummy, void *ud)
{
  global_State *g = G(L);
  UNUSED(dummy);
  UNUSED(ud);
  stack_init(L, L);
  /* NOBARRIER: State initialization, all objects are white. */
  setgcref(L->env, obj2gco(lj_tab_new(L, 0, LJ_MIN_GLOBAL)));
  settabV(L, registry(L), lj_tab_new(L, 0, LJ_MIN_REGISTRY));
  lj_str_resize(L, LJ_MIN_STRTAB-1);
  lj_meta_init(L);
  lj_lex_init(L);
  fixstring(lj_err_str(L, LJ_ERR_ERRMEM));  /* Preallocate memory error msg. */
  g->gc.threshold = 4*g->gc.total;
  lj_trace_initstate(g);
  return NULL;
}

static void close_state(lua_State *L)
{
  global_State *g = G(L);
  lj_func_closeuv(L, tvref(L->stack));
  lj_gc_freeall(g);
  lua_assert(gcref(g->gc.root) == obj2gco(L));
  lua_assert(g->strnum == 0);
  lj_trace_freestate(g);
#if LJ_HASFFI
  lj_ctype_freestate(g);
#endif
```

```
  lj_mem_freevec(g, g->strhash, g->strmask+1, GCRef);
  lj_buf_free(g, &g->tmpbuf);
  lj_mem_freevec(g, tvref(L->stack), L->stacksize, TValue);
  lua_assert(g->gc.total == sizeof(GG_State));
#ifndef LUAJIT_USE_SYSMALLOC
  if (g->allocf == lj_alloc_f)
    lj_alloc_destroy(g->allocd);
  else
#endif
    g->allocf(g->allocd, G2GG(g), sizeof(GG_State), 0);
}

#if LJ_64 && !(defined(LUAJIT_USE_VALGRIND) && defined(LUAJIT_USE_SYSMALLOC))
lua_State *lj_state_newstate(lua_Alloc f, void *ud)
#else
LUA_API lua_State *lua_newstate(lua_Alloc f, void *ud)
#endif
{
  GG_State *GG = (GG_State *)f(ud, NULL, 0, sizeof(GG_State));
  lua_State *L = &GG->L;
  global_State *g = &GG->g;
  if (GG == NULL || !checkptrGC(GG)) return NULL;
  memset(GG, 0, sizeof(GG_State));
  L->gct = ~LJ_TTHREAD;
  L->marked = LJ_GC_WHITE0 | LJ_GC_FIXED | LJ_GC_SFIXED;  /* Prevent free. */
  L->dummy_ffid = FF_C;
  setmref(L->glref, g);
  g->gc.currentwhite = LJ_GC_WHITE0 | LJ_GC_FIXED;
  g->strempty.marked = LJ_GC_WHITE0;
  g->strempty.gct = ~LJ_TSTR;
  g->allocf = f;
  g->allocd = ud;
  setgcref(g->mainthref, obj2gco(L));
  setgcref(g->uvhead.prev, obj2gco(&g->uvhead));
  setgcref(g->uvhead.next, obj2gco(&g->uvhead));
  g->strmask = ~(MSize)0;
  setnilV(registry(L));
  setnilV(&g->nilnode.val);
  setnilV(&g->nilnode.key);
#if !LJ_GC64
  setmref(g->nilnode.freetop, &g->nilnode);
#endif
  lj_buf_init(NULL, &g->tmpbuf);
  g->gc.state = GCSpause;
  setgcref(g->gc.root, obj2gco(L));
  setmref(g->gc.sweep, &g->gc.root);
  g->gc.total = sizeof(GG_State);
  g->gc.pause = LUAI_GCPAUSE;
  g->gc.stepmul = LUAI_GCMUL;
  lj_dispatch_init((GG_State *)L);
  L->status = LUA_ERRERR+1;  /* Avoid touching the stack upon memory error. */
  if (lj_vm_cpcall(L, NULL, NULL, cpluaopen) != 0) {
    /* Memory allocation error: free partial state. */
    close_state(L);
    return NULL;
  }
  L->status = 0;
  return L;
}

static TValue *cpfinalize(lua_State *L, lua_CFunction dummy, void *ud)
{
  UNUSED(dummy);
  UNUSED(ud);
  lj_gc_finalize_cdata(L);
  lj_gc_finalize_udata(L);
  /* Frame pop omitted. */
  return NULL;
}

LUA_API void lua_close(lua_State *L)
{
  global_State *g = G(L);
  int i;
  L = mainthread(g);  /* Only the main thread can be closed. */
#if LJ_HASPROFILE
```

```
  luaJIT_profile_stop(L);
#endif
  setgcrefnull(g->cur_L);
  lj_func_closeuv(L, tvref(L->stack));
  lj_gc_separateudata(g, 1);  /* Separate udata which have GC metamethods. */
#if LJ_HASJIT
  G2J(g)->flags &= ~JIT_F_ON;
  G2J(g)->state = LJ_TRACE_IDLE;
  lj_dispatch_update(g);
#endif
  for (i = 0;;) {
    hook_enter(g);
    L->status = 0;
    L->base = L->top = tvref(L->stack) + 1 + LJ_FR2;
    L->cframe = NULL;
    if (lj_vm_cpcall(L, NULL, NULL, cpfinalize) == 0) {
      if (++i >= 10) break;
      lj_gc_separateudata(g, 1);  /* Separate udata again. */
      if (gcref(g->gc.mmudata) == NULL)  /* Until nothing is left to do. */
        break;
    }
  }
  close_state(L);
}

lua_State *lj_state_new(lua_State *L)
{
  lua_State *L1 = lj_mem_newobj(L, lua_State);
  L1->gct = ~LJ_TTHREAD;
  L1->dummy_ffid = FF_C;
  L1->status = 0;
  L1->stacksize = 0;
  setmref(L1->stack, NULL);
  L1->cframe = NULL;
  /* NOBARRIER: The lua_State is new (marked white). */
  setgcrefnull(L1->openupval);
  setmrefr(L1->glref, L->glref);
  setgcrefr(L1->env, L->env);
  stack_init(L1, L);  /* init stack */
  lua_assert(iswhite(obj2gco(L1)));
  return L1;
}

void LJ_FASTCALL lj_state_free(global_State *g, lua_State *L)
{
  lua_assert(L != mainthread(g));
  if (obj2gco(L) == gcref(g->cur_L))
    setgcrefnull(g->cur_L);
  lj_func_closeuv(L, tvref(L->stack));
  lua_assert(gcref(L->openupval) == NULL);
  lj_mem_freevec(g, tvref(L->stack), L->stacksize, TValue);
  lj_mem_freet(g, L);
}

```

One Level Up Top Level

src/lj_ctype.c - luajit-2.0-src

Global variables defined

- lj_ctype_typeinfo
- lj_ctype_typenames

Data types defined

- CTRepr
- CTRepr

Functions defined

- ctype_addtype
- ctype_appc
- ctype_appnum
- ctype_prepc
- ctype_prepnum
- ctype_prepqual
- ctype_prepstr
- ctype_preptype
- ctype_repr
- lj_ctype_addname
- lj_ctype_freestate
- lj_ctype_getfieldq
- lj_ctype_getname
- lj_ctype_info
- lj_ctype_init
- lj_ctype_intern
- lj_ctype_meta
- lj_ctype_new
- lj_ctype_rawref
- lj_ctype_repr
- lj_ctype_repr_complex
- lj_ctype_repr_int64
- lj_ctype_size

- lj_ctype_vlsize

Macros defined

- CTKWDEF
- CTKWINFODEF
- CTKWINFODEF
- CTKWNAMEDEF
- CTKWNAMEDEF
- CTREPR_MAX
- CTTDDEF
- CTTDINFODEF
- CTTDINFODEF
- CTTDNAMEDEF
- CTTDNAMEDEF
- CTTYINFODEF
- CTTYINFODEF
- CTTYPEINFO_NUM
- CTTYPETAB_MIN
- CTTYPETAB_MIN
- ct_hashname
- ct_hashtype
- ctype_preplit

Source code

```
/*
** C type management.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#include "lj_obj.h"

#if LJ_HASFFI

 #include "lj_gc.h"
 #include "lj_err.h"
 #include "lj_str.h"
 #include "lj_tab.h"
 #include "lj_strfmt.h"
 #include "lj_ctype.h"
 #include "lj_ccallback.h"

 /* -- C type definitions -------------------------------------------------- */

 /* Predefined typedefs. */
 #define CTTDDEF(_) \
   /* Vararg handling. */ \
   _("va_list",                        P_VOID) \
```

```
  _("__builtin_va_list",        P_VOID) \
  _("__gnuc_va_list",                P_VOID) \
  /* From stddef.h. */ \
  _("ptrdiff_t",                INT_PSZ) \
  _("size_t",                        UINT_PSZ) \
  _("wchar_t",                        WCHAR) \
  /* Subset of stdint.h. */ \
  _("int8_t",                        INT8) \
  _("int16_t",                        INT16) \
  _("int32_t",                        INT32) \
  _("int64_t",                        INT64) \
  _("uint8_t",                        UINT8) \
  _("uint16_t",                        UINT16) \
  _("uint32_t",                        UINT32) \
  _("uint64_t",                        UINT64) \
  _("intptr_t",                        INT_PSZ) \
  _("uintptr_t",                UINT_PSZ) \
  /* End of typedef list. */

/* Keywords (only the ones we actually care for). */
#define CTKWDEF(_) \
  /* Type specifiers. */ \
  _("void",                -1,        CTOK_VOID) \
  _("_Bool",                0,        CTOK_BOOL) \
  _("bool",                1,        CTOK_BOOL) \
  _("char",                1,        CTOK_CHAR) \
  _("int",                4,        CTOK_INT) \
  _("__int8",                1,        CTOK_INT) \
  _("__int16",                2,        CTOK_INT) \
  _("__int32",                4,        CTOK_INT) \
  _("__int64",                8,        CTOK_INT) \
  _("float",                4,        CTOK_FP) \
  _("double",                8,        CTOK_FP) \
  _("long",                0,        CTOK_LONG) \
  _("short",                0,        CTOK_SHORT) \
  _("_Complex",                0,        CTOK_COMPLEX) \
  _("complex",                0,        CTOK_COMPLEX) \
  _("__complex",        0,        CTOK_COMPLEX) \
  _("__complex__",        0,        CTOK_COMPLEX) \
  _("signed",                0,        CTOK_SIGNED) \
  _("__signed",                0,        CTOK_SIGNED) \
  _("__signed__",        0,        CTOK_SIGNED) \
  _("unsigned",                0,        CTOK_UNSIGNED) \
  /* Type qualifiers. */ \
  _("const",                0,        CTOK_CONST) \
  _("__const",                0,        CTOK_CONST) \
  _("__const__",        0,        CTOK_CONST) \
  _("volatile",                0,        CTOK_VOLATILE) \
  _("__volatile",        0,        CTOK_VOLATILE) \
  _("__volatile__",        0,        CTOK_VOLATILE) \
  _("restrict",                0,        CTOK_RESTRICT) \
  _("__restrict",        0,        CTOK_RESTRICT) \
  _("__restrict__",        0,        CTOK_RESTRICT) \
  _("inline",                0,        CTOK_INLINE) \
  _("__inline",                0,        CTOK_INLINE) \
  _("__inline__",        0,        CTOK_INLINE) \
  /* Storage class specifiers. */ \
  _("typedef",                0,        CTOK_TYPEDEF) \
  _("extern",                0,        CTOK_EXTERN) \
  _("static",                0,        CTOK_STATIC) \
  _("auto",                0,        CTOK_AUTO) \
  _("register",                0,        CTOK_REGISTER) \
  /* GCC Attributes. */ \
  _("__extension__",        0,        CTOK_EXTENSION) \
  _("__attribute",        0,        CTOK_ATTRIBUTE) \
  _("__attribute__",        0,        CTOK_ATTRIBUTE) \
  _("asm",                0,        CTOK_ASM) \
  _("__asm",                0,        CTOK_ASM) \
  _("__asm__",                0,        CTOK_ASM) \
  /* MSVC Attributes. */ \
  _("__declspec",        0,        CTOK_DECLSPEC) \
  _("__cdecl",                CTCC_CDECL,        CTOK_CCDECL) \
  _("__thiscall",        CTCC_THISCALL,        CTOK_CCDECL) \
  _("__fastcall",        CTCC_FASTCALL,        CTOK_CCDECL) \
  _("__stdcall",        CTCC_STDCALL,        CTOK_CCDECL) \
  _("__ptr32",                4,        CTOK_PTRSZ) \
```

```
  _("__ptr64",                8,        CTOK_PTRSZ) \
  /* Other type specifiers. */ \
  _("struct",                 0,        CTOK_STRUCT) \
  _("union",                  0,        CTOK_UNION) \
  _("enum",                   0,        CTOK_ENUM) \
  /* Operators. */ \
  _("sizeof",                 0,        CTOK_SIZEOF) \
  _("__alignof",        0,        CTOK_ALIGNOF) \
  _("__alignof__",        0,        CTOK_ALIGNOF) \
  /* End of keyword list. */

/* Type info for predefined types. Size merged in. */
static CTInfo lj_ctype_typeinfo[] = {
#define CTTYINFODEF(id, sz, ct, info)        CTINFO((ct),(((sz)&0x3fu)<<10)+(info)),
#define CTTDINFODEF(name, id)                CTINFO(CT_TYPEDEF, CTID_##id),
#define CTKWINFODEF(name, sz, kw)        CTINFO(CT_KW,(((sz)&0x3fu)<<10)+(kw)),
CTTYDEF(CTTYINFODEF)
CTTDDEF(CTTDINFODEF)
CTKWDEF(CTKWINFODEF)
#undef CTTYINFODEF
#undef CTTDINFODEF
#undef CTKWINFODEF
  0
};

/* Predefined type names collected in a single string. */
static const char * const lj_ctype_typenames =
#define CTTDNAMEDEF(name, id)                name "\0"
#define CTKWNAMEDEF(name, sz, cds)        name "\0"
CTTDDEF(CTTDNAMEDEF)
CTKWDEF(CTKWNAMEDEF)
#undef CTTDNAMEDEF
#undef CTKWNAMEDEF
;

#define CTTYPEINFO_NUM                (sizeof(lj_ctype_typeinfo)/sizeof(CTInfo)-1)
#ifdef LUAJIT_CTYPE_CHECK_ANCHOR
#define CTTYPETAB_MIN                CTTYPEINFO_NUM
#else
#define CTTYPETAB_MIN                128
#endif

/* -- C type interning ---------------------------------------------------- */

#define ct_hashtype(info, size)        (hashrot(info, size) & CTHASH_MASK)
#define ct_hashname(name) \
  (hashrot(u32ptr(name), u32ptr(name) + HASH_BIAS) & CTHASH_MASK)

/* Create new type element. */
CTypeID lj_ctype_new(CTState *cts, CType **ctp)
{
  CTypeID id = cts->top;
  CType *ct;
  lua_assert(cts->L);
  if (LJ_UNLIKELY(id >= cts->sizetab)) {
    if (id >= CTID_MAX) lj_err_msg(cts->L, LJ_ERR_TABOV);
#ifdef LUAJIT_CTYPE_CHECK_ANCHOR
    ct = lj_mem_newvec(cts->L, id+1, CType);
    memcpy(ct, cts->tab, id*sizeof(CType));
    memset(cts->tab, 0, id*sizeof(CType));
    lj_mem_freevec(cts->g, cts->tab, cts->sizetab, CType);
    cts->tab = ct;
    cts->sizetab = id+1;
#else
    lj_mem_growvec(cts->L, cts->tab, cts->sizetab, CTID_MAX, CType);
#endif
  }
  cts->top = id+1;
  *ctp = ct = &cts->tab[id];
  ct->info = 0;
  ct->size = 0;
  ct->sib = 0;
  ct->next = 0;
  setgcrefnull(ct->name);
  return id;
}
```

```
176
177 /* Intern a type element. */
178 CTypeID lj_ctype_intern(CTState *cts, CTInfo info, CTSize size)
179 {
180   uint32_t h = ct_hashtype(info, size);
181   CTypeID id = cts->hash[h];
182   lua_assert(cts->L);
183   while (id) {
184     CType *ct = ctype_get(cts, id);
185     if (ct->info == info && ct->size == size)
186       return id;
187     id = ct->next;
188   }
189   id = cts->top;
190   if (LJ_UNLIKELY(id >= cts->sizetab)) {
191     if (id >= CTID_MAX) lj_err_msg(cts->L, LJ_ERR_TABOV);
192     lj_mem_growvec(cts->L, cts->tab, cts->sizetab, CTID_MAX, CType);
193   }
194   cts->top = id+1;
195   cts->tab[id].info = info;
196   cts->tab[id].size = size;
197   cts->tab[id].sib = 0;
198   cts->tab[id].next = cts->hash[h];
199   setgcrefnull(cts->tab[id].name);
200   cts->hash[h] = (CTypeID1)id;
201   return id;
202 }
203
204 /* Add type element to hash table. */
205 static void ctype_addtype(CTState *cts, CType *ct, CTypeID id)
206 {
207   uint32_t h = ct_hashtype(ct->info, ct->size);
208   ct->next = cts->hash[h];
209   cts->hash[h] = (CTypeID1)id;
210 }
211
212 /* Add named element to hash table. */
213 void lj_ctype_addname(CTState *cts, CType *ct, CTypeID id)
214 {
215   uint32_t h = ct_hashname(gcref(ct->name));
216   ct->next = cts->hash[h];
217   cts->hash[h] = (CTypeID1)id;
218 }
219
220 /* Get a C type by name, matching the type mask. */
221 CTypeID lj_ctype_getname(CTState *cts, CType **ctp, GCstr *name, uint32_t tmask)
222 {
223   CTypeID id = cts->hash[ct_hashname(name)];
224   while (id) {
225     CType *ct = ctype_get(cts, id);
226     if (gcref(ct->name) == obj2gco(name) &&
227         ((tmask >> ctype_type(ct->info)) & 1)) {
228       *ctp = ct;
229       return id;
230     }
231     id = ct->next;
232   }
233   *ctp = &cts->tab[0];  /* Simplify caller logic. ctype_get() would assert. */
234   return 0;
235 }
236
237 /* Get a struct/union/enum/function field by name. */
238 CType *lj_ctype_getfieldq(CTState *cts, CType *ct, GCstr *name, CTSize *ofs,
239                           CTInfo *qual)
240 {
241   while (ct->sib) {
242     ct = ctype_get(cts, ct->sib);
243     if (gcref(ct->name) == obj2gco(name)) {
244       *ofs = ct->size;
245       return ct;
246     }
247     if (ctype_isxattrib(ct->info, CTA_SUBTYPE)) {
248       CType *fct, *cct = ctype_child(cts, ct);
249       CTInfo q = 0;
250       while (ctype_isattrib(cct->info)) {
251         if (ctype_attrib(cct->info) == CTA_QUAL) q |= cct->size;
```

```
        cct = ctype_child(cts, cct);
      }
      fct = lj_ctype_getfieldq(cts, cct, name, ofs, qual);
      if (fct) {
        if (qual) *qual |= q;
        *ofs += ct->size;
        return fct;
      }
    }
  }
  return NULL;  /* Not found. */
}

/* -- C type information -------------------------------------------------- */

/* Follow references and get raw type for a C type ID. */
CType *lj_ctype_rawref(CTState *cts, CTypeID id)
{
  CType *ct = ctype_get(cts, id);
  while (ctype_isattrib(ct->info) || ctype_isref(ct->info))
    ct = ctype_child(cts, ct);
  return ct;
}

/* Get size for a C type ID. Does NOT support VLA/VLS. */
CTSize lj_ctype_size(CTState *cts, CTypeID id)
{
  CType *ct = ctype_raw(cts, id);
  return ctype_hassize(ct->info) ? ct->size : CTSIZE_INVALID;
}

/* Get size for a variable-length C type. Does NOT support other C types. */
CTSize lj_ctype_vlsize(CTState *cts, CType *ct, CTSize nelem)
{
  uint64_t xsz = 0;
  if (ctype_isstruct(ct->info)) {
    CTypeID arrid = 0, fid = ct->sib;
    xsz = ct->size;  /* Add the struct size. */
    while (fid) {
      CType *ctf = ctype_get(cts, fid);
      if (ctype_type(ctf->info) == CT_FIELD)
        arrid = ctype_cid(ctf->info);  /* Remember last field of VLS. */
      fid = ctf->sib;
    }
    ct = ctype_raw(cts, arrid);
  }
  lua_assert(ctype_isvlarray(ct->info));  /* Must be a VLA. */
  ct = ctype_rawchild(cts, ct);  /* Get array element. */
  lua_assert(ctype_hassize(ct->info));
  /* Calculate actual size of VLA and check for overflow. */
  xsz += (uint64_t)ct->size * nelem;
  return xsz < 0x80000000u ? (CTSize)xsz : CTSIZE_INVALID;
}

/* Get type, qualifiers, size and alignment for a C type ID. */
CTInfo lj_ctype_info(CTState *cts, CTypeID id, CTSize *szp)
{
  CTInfo qual = 0;
  CType *ct = ctype_get(cts, id);
  for (;;) {
    CTInfo info = ct->info;
    if (ctype_isenum(info)) {
      /* Follow child. Need to look at its attributes, too. */
    } else if (ctype_isattrib(info)) {
      if (ctype_isxattrib(info, CTA_QUAL))
        qual |= ct->size;
      else if (ctype_isxattrib(info, CTA_ALIGN) && !(qual & CTFP_ALIGNED))
        qual |= CTFP_ALIGNED + CTALIGN(ct->size);
    } else {
      if (!(qual & CTFP_ALIGNED)) qual |= (info & CTF_ALIGN);
      qual |= (info & ~(CTF_ALIGN|CTMASK_CID));
      lua_assert(ctype_hassize(info) || ctype_isfunc(info));
      *szp = ctype_isfunc(info) ? CTSIZE_INVALID : ct->size;
      break;
    }
    ct = ctype_get(cts, ctype_cid(info));
```

```
  }
  return qual;
}

/* Get ctype metamethod. */
cTValue *lj_ctype_meta(CTState *cts, CTypeID id, MMS mm)
{
  CType *ct = ctype_get(cts, id);
  cTValue *tv;
  while (ctype_isattrib(ct->info) || ctype_isref(ct->info)) {
    id = ctype_cid(ct->info);
    ct = ctype_get(cts, id);
  }
  if (ctype_isptr(ct->info) &&
      ctype_isfunc(ctype_get(cts, ctype_cid(ct->info))->info))
    tv = lj_tab_getstr(cts->miscmap, &cts->g->strempty);
  else
    tv = lj_tab_getinth(cts->miscmap, -(int32_t)id);
  if (tv && tvistab(tv) &&
      (tv = lj_tab_getstr(tabV(tv), mmname_str(cts->g, mm))) && !tvisnil(tv))
    return tv;
  return NULL;
}

/* -- C type representation ---------------------------------------------- */

/* Fixed max. length of a C type representation. */
#define CTREPR_MAX              512

typedef struct CTRepr {
  char *pb, *pe;
  CTState *cts;
  lua_State *L;
  int needsp;
  int ok;
  char buf[CTREPR_MAX];
} CTRepr;

/* Prepend string. */
static void ctype_prepstr(CTRepr *ctr, const char *str, MSize len)
{
  char *p = ctr->pb;
  if (ctr->buf + len+1 > p) { ctr->ok = 0; return; }
  if (ctr->needsp) *--p = ' ';
  ctr->needsp = 1;
  p -= len;
  while (len-- > 0) p[len] = str[len];
  ctr->pb = p;
}

#define ctype_preplit(ctr, str)        ctype_prepstr((ctr), "" str, sizeof(str)-1)

/* Prepend char. */
static void ctype_prepc(CTRepr *ctr, int c)
{
  if (ctr->buf >= ctr->pb) { ctr->ok = 0; return; }
  *--ctr->pb = c;
}

/* Prepend number. */
static void ctype_prepnum(CTRepr *ctr, uint32_t n)
{
  char *p = ctr->pb;
  if (ctr->buf + 10+1 > p) { ctr->ok = 0; return; }
  do { *--p = (char)('0' + n % 10); } while (n /= 10);
  ctr->pb = p;
  ctr->needsp = 0;
}

/* Append char. */
static void ctype_appc(CTRepr *ctr, int c)
{
  if (ctr->pe >= ctr->buf + CTREPR_MAX) { ctr->ok = 0; return; }
  *ctr->pe++ = c;
}

```

```
/* Append number. */
static void ctype_appnum(CTRepr *ctr, uint32_t n)
{
  char buf[10];
  char *p = buf+sizeof(buf);
  char *q = ctr->pe;
  if (q > ctr->buf + CTREPR_MAX - 10) { ctr->ok = 0; return; }
  do { *--p = (char)('0' + n % 10); } while (n /= 10);
  do { *q++ = *p++; } while (p < buf+sizeof(buf));
  ctr->pe = q;
}

/* Prepend qualifiers. */
static void ctype_prepqual(CTRepr *ctr, CTInfo info)
{
  if ((info & CTF_VOLATILE)) ctype_preplit(ctr, "volatile");
  if ((info & CTF_CONST)) ctype_preplit(ctr, "const");
}

/* Prepend named type. */
static void ctype_preptype(CTRepr *ctr, CType *ct, CTInfo qual, const char *t)
{
  if (gcref(ct->name)) {
    GCstr *str = gco2str(gcref(ct->name));
    ctype_prepstr(ctr, strdata(str), str->len);
  } else {
    if (ctr->needsp) ctype_prepc(ctr, ' ');
    ctype_prepnum(ctr, ctype_typeid(ctr->cts, ct));
    ctr->needsp = 1;
  }
  ctype_prepstr(ctr, t, (MSize)strlen(t));
  ctype_prepqual(ctr, qual);
}

static void ctype_repr(CTRepr *ctr, CTypeID id)
{
  CType *ct = ctype_get(ctr->cts, id);
  CTInfo qual = 0;
  int ptrto = 0;
  for (;;) {
    CTInfo info = ct->info;
    CTSize size = ct->size;
    switch (ctype_type(info)) {
    case CT_NUM:
      if ((info & CTF_BOOL)) {
        ctype_preplit(ctr, "bool");
      } else if ((info & CTF_FP)) {
        if (size == sizeof(double)) ctype_preplit(ctr, "double");
        else if (size == sizeof(float)) ctype_preplit(ctr, "float");
        else ctype_preplit(ctr, "long double");
      } else if (size == 1) {
        if (!((info ^ CTF_UCHAR) & CTF_UNSIGNED)) ctype_preplit(ctr, "char");
        else if (CTF_UCHAR) ctype_preplit(ctr, "signed char");
        else ctype_preplit(ctr, "unsigned char");
      } else if (size < 8) {
        if (size == 4) ctype_preplit(ctr, "int");
        else ctype_preplit(ctr, "short");
        if ((info & CTF_UNSIGNED)) ctype_preplit(ctr, "unsigned");
      } else {
        ctype_preplit(ctr, "_t");
        ctype_prepnum(ctr, size*8);
        ctype_preplit(ctr, "int");
        if ((info & CTF_UNSIGNED)) ctype_prepc(ctr, 'u');
      }
      ctype_prepqual(ctr, (qual|info));
      return;
    case CT_VOID:
      ctype_preplit(ctr, "void");
      ctype_prepqual(ctr, (qual|info));
      return;
    case CT_STRUCT:
      ctype_preptype(ctr, ct, qual, (info & CTF_UNION) ? "union" : "struct");
      return;
    case CT_ENUM:
      if (id == CTID_CTYPEID) {
        ctype_preplit(ctr, "ctype");
```

```
480         return;
481       }
482       ctype_preptype(ctr, ct, qual, "enum");
483       return;
484     case CT_ATTRIB:
485       if (ctype_attrib(info) == CTA_QUAL) qual |= size;
486       break;
487     case CT_PTR:
488       if ((info & CTF_REF)) {
489         ctype_prepc(ctr, '&');
490       } else {
491         ctype_prepqual(ctr, (qual|info));
492         if (LJ_64 && size == 4) ctype_preplit(ctr, "__ptr32");
493         ctype_prepc(ctr, '*');
494       }
495       qual = 0;
496       ptrto = 1;
497       ctr->needsp = 1;
498       break;
499     case CT_ARRAY:
500       if (ctype_isrefarray(info)) {
501         ctr->needsp = 1;
502         if (ptrto) { ptrto = 0; ctype_prepc(ctr, '('); ctype_appc(ctr, ')'); }
503         ctype_appc(ctr, '[');
504         if (size != CTSIZE_INVALID) {
505           CTSize csize = ctype_child(ctr->cts, ct)->size;
506           ctype_appnum(ctr, csize ? size/csize : 0);
507         } else if ((info & CTF_VLA)) {
508           ctype_appc(ctr, '?');
509         }
510         ctype_appc(ctr, ']');
511       } else if ((info & CTF_COMPLEX)) {
512         if (size == 2*sizeof(float)) ctype_preplit(ctr, "float");
513         ctype_preplit(ctr, "complex");
514         return;
515       } else {
516         ctype_preplit(ctr, ")))");
517         ctype_prepnum(ctr, size);
518         ctype_preplit(ctr, "__attribute__((vector_size(");
519       }
520       break;
521     case CT_FUNC:
522       ctr->needsp = 1;
523       if (ptrto) { ptrto = 0; ctype_prepc(ctr, '('); ctype_appc(ctr, ')'); }
524       ctype_appc(ctr, '(');
525       ctype_appc(ctr, ')');
526       break;
527     default:
528       lua_assert(0);
529       break;
530     }
531     ct = ctype_get(ctr->cts, ctype_cid(info));
532   }
533 }
534
535 /* Return a printable representation of a C type. */
536 GCstr *lj_ctype_repr(lua_State *L, CTypeID id, GCstr *name)
537 {
538   global_State *g = G(L);
539   CTRepr ctr;
540   ctr.pb = ctr.pe = &ctr.buf[CTREPR_MAX/2];
541   ctr.cts = ctype_ctsG(g);
542   ctr.L = L;
543   ctr.ok = 1;
544   ctr.needsp = 0;
545   if (name) ctype_prepstr(&ctr, strdata(name), name->len);
546   ctype_repr(&ctr, id);
547   if (LJ_UNLIKELY(!ctr.ok)) return lj_str_newlit(L, "?");
548   return lj_str_new(L, ctr.pb, ctr.pe - ctr.pb);
549 }
550
551 /* Convert int64_t/uint64_t to string with 'LL' or 'ULL' suffix. */
552 GCstr *lj_ctype_repr_int64(lua_State *L, uint64_t n, int isunsigned)
553 {
554   char buf[1+20+3];
555   char *p = buf+sizeof(buf);
```

```
  int sign = 0;
  *--p = 'L'; *--p = 'L';
  if (isunsigned) {
    *--p = 'U';
  } else if ((int64_t)n < 0) {
    n = (uint64_t)-(int64_t)n;
    sign = 1;
  }
  do { *--p = (char)('0' + n % 10); } while (n /= 10);
  if (sign) *--p = '-';
  return lj_str_new(L, p, (size_t)(buf+sizeof(buf)-p));
}

/* Convert complex to string with 'i' or 'I' suffix. */
GCstr *lj_ctype_repr_complex(lua_State *L, void *sp, CTSize size)
{
  char buf[2*STRFMT_MAXBUF_NUM+2+1], *p = buf;
  TValue re, im;
  if (size == 2*sizeof(double)) {
    re.n = *(double *)sp; im.n = ((double *)sp)[1];
  } else {
    re.n = (double)*(float *)sp; im.n = (double)((float *)sp)[1];
  }
  p = lj_strfmt_wnum(p, &re);
  if (!(im.u32.hi & 0x80000000u) || im.n != im.n) *p++ = '+';
  p = lj_strfmt_wnum(p, &im);
  *p = *(p-1) >= 'a' ? 'I' : 'i';
  p++;
  return lj_str_new(L, buf, p-buf);
}

/* -- C type state ---------------------------------------------------- */

/* Initialize C type table and state. */
CTState *lj_ctype_init(lua_State *L)
{
  CTState *cts = lj_mem_newt(L, sizeof(CTState), CTState);
  CType *ct = lj_mem_newvec(L, CTTYPETAB_MIN, CType);
  const char *name = lj_ctype_typenames;
  CTypeID id;
  memset(cts, 0, sizeof(CTState));
  cts->tab = ct;
  cts->sizetab = CTTYPETAB_MIN;
  cts->top = CTTYPEINFO_NUM;
  cts->L = NULL;
  cts->g = G(L);
  for (id = 0; id < CTTYPEINFO_NUM; id++, ct++) {
    CTInfo info = lj_ctype_typeinfo[id];
    ct->size = (CTSize)((int32_t)(info << 16) >> 26);
    ct->info = info & 0xffff03ffu;
    ct->sib = 0;
    if (ctype_type(info) == CT_KW || ctype_istypedef(info)) {
      size_t len = strlen(name);
      GCstr *str = lj_str_new(L, name, len);
      ctype_setname(ct, str);
      name += len+1;
      lj_ctype_addname(cts, ct, id);
    } else {
      setgcrefnull(ct->name);
      ct->next = 0;
      if (!ctype_isenum(info)) ctype_addtype(cts, ct, id);
    }
  }
  setmref(G(L)->ctype_state, cts);
  return cts;
}

/* Free C type table and state. */
void lj_ctype_freestate(global_State *g)
{
  CTState *cts = ctype_ctsG(g);
  if (cts) {
    lj_ccallback_mcode_free(cts);
    lj_mem_freevec(g, cts->tab, cts->sizetab, CType);
    lj_mem_freevec(g, cts->cb.cbid, cts->cb.sizeid, CTypeID1);
    lj_mem_freet(g, cts);
```

```
632   }
633 }
634
635 #endif
```

One Level Up Top Level

src/lj_ctype.h - luajit-2.0-src

Global variables defined

- CCallback
- FPRCBArg

Data types defined

- CCallback
- CTInfo
- CTSize
- CTState
- CTState
- CType
- CType
- CTypeID
- CTypeID1
- FPRCBArg

Functions defined

- ctype_check
- ctype_child
- ctype_cts
- ctype_get
- ctype_raw
- ctype_rawchild
- ctype_setname

Macros defined

- CCALL_MAX_FPR
- CCALL_MAX_GPR
- CDF_SCL
- CDSDEF
- CDSFLAG
- CDSFLAG

- CKWDEF
- CKWNUM
- CKWNUM
- CTALIGN
- CTALIGN_PTR
- CTALIGN_PTR
- CTATTRIB
- CTBSZ_FIELD
- CTBSZ_MAX
- CTFP_ALIGNED
- CTFP_CCONV
- CTFP_PACKED
- CTF_ALIGN
- CTF_BOOL
- CTF_COMPLEX
- CTF_CONST
- CTF_FP
- CTF_INSERT
- CTF_LONG
- CTF_QUAL
- CTF_REF
- CTF_SSEREGPARM
- CTF_UCHAR
- CTF_UNION
- CTF_UNSIGNED
- CTF_VARARG
- CTF_VECTOR
- CTF_VLA
- CTF_VOLATILE
- CTHASH_MASK
- CTHASH_SIZE
- CTID_INT_PSZ
- CTID_INT_PSZ

- CTID_UINT_PSZ
- CTID_UINT_PSZ
- CTID_WCHAR
- CTID_WCHAR
- CTID_WCHAR
- CTINFO
- CTINFO_REF
- CTMASK_ALIGN
- CTMASK_ATTRIB
- CTMASK_BITBSZ
- CTMASK_BITCSZ
- CTMASK_BITPOS
- CTMASK_CCONV
- CTMASK_CID
- CTMASK_MSIZEP
- CTMASK_NUM
- CTMASK_REGPARM
- CTMASK_VSIZEP
- CTOKDEF
- CTOKNUM
- CTOKNUM
- CTSHIFT_ALIGN
- CTSHIFT_ATTRIB
- CTSHIFT_BITBSZ
- CTSHIFT_BITCSZ
- CTSHIFT_BITPOS
- CTSHIFT_CCONV
- CTSHIFT_MSIZEP
- CTSHIFT_NUM
- CTSHIFT_REGPARM
- CTSHIFT_VSIZEP
- CTSIZE_INVALID
- CTSIZE_PTR

- CTSIZE_PTR
- CTTYDEF
- CTTYDEFP
- CTTYDEFP
- CTTYIDDEF
- CTTYIDDEF
- CT_MEMALIGN
- LJ_CTYPE_RESTORE
- LJ_CTYPE_SAVE
- _LJ_CTYPE_H
- ctype_align
- ctype_attrib
- ctype_bitbsz
- ctype_bitcsz
- ctype_bitpos
- ctype_cconv
- ctype_cid
- ctype_ctsG
- ctype_hassize
- ctype_isarray
- ctype_isattrib
- ctype_isbitfield
- ctype_isbool
- ctype_iscomplex
- ctype_isconstval
- ctype_isenum
- ctype_isextern
- ctype_isfield
- ctype_isfp
- ctype_isfunc
- ctype_isinteger
- ctype_isinteger_or_bool
- ctype_isnum

- ctype_ispointer
- ctype_isptr
- ctype_isref
- ctype_isrefarray
- ctype_isstruct
- ctype_istypedef
- ctype_isvector
- ctype_isvlarray
- ctype_isvltype
- ctype_isvoid
- ctype_isxattrib
- ctype_msizeP
- ctype_type
- ctype_typeid
- ctype_vsizeP
- lj_ctype_getfield

Source code

```
/*
** C type management.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_CTYPE_H
#define _LJ_CTYPE_H

#include "lj_obj.h"
 #include "lj_gc.h"

 #if LJ_HASFFI

 /* -- C type definitions ---------------------------------------------- */

 /* C type numbers. Highest 4 bits of C type info. ORDER CT. */
 enum {
   /* Externally visible types. */
   CT_NUM,               /* Integer or floating-point numbers. */
   CT_STRUCT,               /* Struct or union. */
   CT_PTR,               /* Pointer or reference. */
   CT_ARRAY,               /* Array or complex type. */
   CT_MAYCONVERT = CT_ARRAY,
   CT_VOID,               /* Void type. */
   CT_ENUM,               /* Enumeration. */
   CT_HASSIZE = CT_ENUM,  /* Last type where ct->size holds the actual size. */
   CT_FUNC,               /* Function. */
   CT_TYPEDEF,               /* Typedef. */
   CT_ATTRIB,               /* Miscellaneous attributes. */
   /* Internal element types. */
   CT_FIELD,               /* Struct/union field or function parameter. */
   CT_BITFIELD,               /* Struct/union bitfield. */
   CT_CONSTVAL,               /* Constant value. */
   CT_EXTERN,               /* External reference. */
   CT_KW                  /* Keyword. */
 };
```

```
LJ_STATIC_ASSERT(((int)CT_PTR & (int)CT_ARRAY) == CT_PTR);
LJ_STATIC_ASSERT(((int)CT_STRUCT & (int)CT_ARRAY) == CT_STRUCT);

/*
**  ---------- info ------------
** |type      flags...  A   cid | size   |  sib  | next  | name  |
** +---------------------------+--------+-------+-------+-------+--
** |NUM       BFvcUL..  A       | size   |       | type  |       |
** |STRUCT    ..vcU..V  A       | size   | field | name? | name? |
** |PTR       ..vcR...  A   cid | size   |       | type  |       |
** |ARRAY     VCvc...V  A   cid | size   |       | type  |       |
** |VOID      ..vc....  A       | size   |       | type  |       |
** |ENUM                A   cid | size   | const | name? | name? |
** |FUNC      ....VS.. cc   cid | nargs  | field | name? | name? |
** |TYPEDEF                 cid |        |       | name  | name  |
** |ATTRIB        attrnum   cid | attr   | sib?  | type? |       |
** |FIELD                   cid | offset | field |       | name? |
** |BITFIELD  B.vcU csz bsz pos | offset | field |       | name? |
** |CONSTVAL     c          cid | value  | const | name  | name  |
** |EXTERN                  cid |        | sib?  | name  | name  |
** |KW                      tok | size   |       | name  | name  |
** +---------------------------+--------+-------+-------+-------+--
**        ^^  ^^--- bits used for C type conversion dispatch
*/

/* C type info flags.     TFFArrrr  */
#define CTF_BOOL        0x08000000u        /* Boolean: NUM, BITFIELD. */
#define CTF_FP                0x04000000u        /* Floating-point: NUM. */
#define CTF_CONST        0x02000000u        /* Const qualifier. */
#define CTF_VOLATILE        0x01000000u        /* Volatile qualifier. */
#define CTF_UNSIGNED        0x00800000u        /* Unsigned: NUM, BITFIELD. */
#define CTF_LONG        0x00400000u        /* Long: NUM. */
#define CTF_VLA                0x00100000u        /* Variable-length: ARRAY, STRUCT. */
#define CTF_REF                0x00800000u        /* Reference: PTR. */
#define CTF_VECTOR        0x08000000u        /* Vector: ARRAY. */
#define CTF_COMPLEX        0x04000000u        /* Complex: ARRAY. */
#define CTF_UNION        0x00800000u        /* Union: STRUCT. */
#define CTF_VARARG        0x00800000u        /* Vararg: FUNC. */
#define CTF_SSEREGPARM        0x00400000u        /* SSE register parameters: FUNC. */

#define CTF_QUAL        (CTF_CONST|CTF_VOLATILE)
#define CTF_ALIGN        (CTMASK_ALIGN<<CTSHIFT_ALIGN)
#define CTF_UCHAR        ((char)-1 > 0 ? CTF_UNSIGNED : 0)

/* Flags used in parser.  .F.Ammvf   cp->attr  */
#define CTFP_ALIGNED        0x00000001u        /* cp->attr + ALIGN */
#define CTFP_PACKED        0x00000002u        /* cp->attr */
/*                        ...C...f   cp->fattr */
#define CTFP_CCONV        0x00000001u        /* cp->fattr + CCONV/[SSE]REGPARM */

/* C type info bitfields. */
#define CTMASK_CID        0x0000ffffu        /* Max. 65536 type IDs. */
#define CTMASK_NUM        0xf0000000u        /* Max. 16 type numbers. */
#define CTSHIFT_NUM        28
#define CTMASK_ALIGN        15                /* Max. alignment is 2^15. */
#define CTSHIFT_ALIGN        16
#define CTMASK_ATTRIB        255                /* Max. 256 attributes. */
#define CTSHIFT_ATTRIB        16
#define CTMASK_CCONV        3                /* Max. 4 calling conventions. */
#define CTSHIFT_CCONV        16
#define CTMASK_REGPARM        3                /* Max. 0-3 regparms. */
#define CTSHIFT_REGPARM        18
 /* Bitfields only used in parser. */
 #define CTMASK_VSIZEP        15                /* Max. vector size is 2^15. */
 #define CTSHIFT_VSIZEP        4
 #define CTMASK_MSIZEP        255                /* Max. type size (via mode) is 128. */
 #define CTSHIFT_MSIZEP        8

 /* Info bits for BITFIELD. Max. size of bitfield is 64 bits. */
 #define CTBSZ_MAX        32                /* Max. size of bitfield is 32 bit. */
 #define CTBSZ_FIELD        127                /* Temp. marker for regular field. */
 #define CTMASK_BITPOS        127
 #define CTMASK_BITBSZ        127
 #define CTMASK_BITCSZ        127
 #define CTSHIFT_BITPOS        0
```

```c
#define CTSHIFT_BITBSZ        8
#define CTSHIFT_BITCSZ        16

#define CTF_INSERT(info, field, val) \
  info = (info & ~(CTMASK_##field<<CTSHIFT_##field)) | \
         (((CTSize)(val) & CTMASK_##field) << CTSHIFT_##field)

/* Calling conventions. ORDER CC */
enum { CTCC_CDECL, CTCC_THISCALL, CTCC_FASTCALL, CTCC_STDCALL };

/* Attribute numbers. */
enum {
  CTA_NONE,               /* Ignored attribute. Must be zero. */
  CTA_QUAL,               /* Unmerged qualifiers. */
  CTA_ALIGN,               /* Alignment override. */
  CTA_SUBTYPE,               /* Transparent sub-type. */
  CTA_REDIR,               /* Redirected symbol name. */
  CTA_BAD,               /* To catch bad IDs. */
  CTA__MAX
};

/* Special sizes. */
#define CTSIZE_INVALID        0xffffffffu

typedef uint32_t CTInfo;        /* Type info. */
typedef uint32_t CTSize;        /* Type size. */
typedef uint32_t CTypeID;        /* Type ID. */
typedef uint16_t CTypeID1;        /* Minimum-sized type ID. */

/* C type table element. */
typedef struct CType {
  CTInfo info;               /* Type info. */
  CTSize size;               /* Type size or other info. */
  CTypeID1 sib;               /* Sibling element. */
  CTypeID1 next;        /* Next element in hash chain. */
  GCRef name;               /* Element name (GCstr). */
} CType;

#define CTHASH_SIZE        128        /* Number of hash anchors. */
#define CTHASH_MASK        (CTHASH_SIZE-1)

/* Simplify target-specific configuration. Checked in lj_ccall.h. */
#define CCALL_MAX_GPR               8
#define CCALL_MAX_FPR               8

typedef LJ_ALIGN(8) union FPRCBArg { double d; float f[2]; } FPRCBArg;

/* C callback state. Defined here, to avoid dragging in lj_ccall.h. */

typedef LJ_ALIGN(8) struct CCallback {
  FPRCBArg fpr[CCALL_MAX_FPR];        /* Arguments/results in FPRs. */
  intptr_t gpr[CCALL_MAX_GPR];        /* Arguments/results in GPRs. */
  intptr_t *stack;               /* Pointer to arguments on stack. */
  void *mcode;                    /* Machine code for callback func. pointers. */
  CTypeID1 *cbid;               /* Callback type table. */
  MSize sizeid;                     /* Size of callback type table. */
  MSize topid;                     /* Highest unused callback type table slot. */
  MSize slot;                    /* Current callback slot. */
} CCallback;

/* C type state. */
typedef struct CTState {
  CType *tab;               /* C type table. */
  CTypeID top;               /* Current top of C type table. */
  MSize sizetab;        /* Size of C type table. */
  lua_State *L;               /* Lua state (needed for errors and allocations). */
  global_State *g;        /* Global state. */
  GCtab *finalizer;        /* Map of cdata to finalizer. */
  GCtab *miscmap;        /* Map of -CTypeID to metatable and cb slot to func. */
  CCallback cb;               /* Temporary callback state. */
  CTypeID1 hash[CTHASH_SIZE];  /* Hash anchors for C type table. */
} CTState;

#define CTINFO(ct, flags)        (((CTInfo)(ct) << CTSHIFT_NUM) + (flags))
#define CTALIGN(al)               ((CTSize)(al) << CTSHIFT_ALIGN)
#define CTATTRIB(at)               ((CTInfo)(at) << CTSHIFT_ATTRIB)
```

```
189
190 #define ctype_type(info)        ((info) >> CTSHIFT_NUM)
191 #define ctype_cid(info)               ((CTypeID)((info) & CTMASK_CID))
192 #define ctype_align(info)        (((info) >> CTSHIFT_ALIGN) & CTMASK_ALIGN)
193 #define ctype_attrib(info)        (((info) >> CTSHIFT_ATTRIB) & CTMASK_ATTRIB)
194 #define ctype_bitpos(info)        (((info) >> CTSHIFT_BITPOS) & CTMASK_BITPOS)
195 #define ctype_bitbsz(info)        (((info) >> CTSHIFT_BITBSZ) & CTMASK_BITBSZ)
196 #define ctype_bitcsz(info)        (((info) >> CTSHIFT_BITCSZ) & CTMASK_BITCSZ)
197 #define ctype_vsizeP(info)        (((info) >> CTSHIFT_VSIZEP) & CTMASK_VSIZEP)
198 #define ctype_msizeP(info)        (((info) >> CTSHIFT_MSIZEP) & CTMASK_MSIZEP)
199 #define ctype_cconv(info)        (((info) >> CTSHIFT_CCONV) & CTMASK_CCONV)
200
201 /* Simple type checks. */
202 #define ctype_isnum(info)        (ctype_type((info)) == CT_NUM)
203 #define ctype_isvoid(info)        (ctype_type((info)) == CT_VOID)
204 #define ctype_isptr(info)        (ctype_type((info)) == CT_PTR)
205 #define ctype_isarray(info)        (ctype_type((info)) == CT_ARRAY)
206 #define ctype_isstruct(info)        (ctype_type((info)) == CT_STRUCT)
207 #define ctype_isfunc(info)        (ctype_type((info)) == CT_FUNC)
208 #define ctype_isenum(info)        (ctype_type((info)) == CT_ENUM)
209 #define ctype_istypedef(info)        (ctype_type((info)) == CT_TYPEDEF)
210 #define ctype_isattrib(info)        (ctype_type((info)) == CT_ATTRIB)
211 #define ctype_isfield(info)        (ctype_type((info)) == CT_FIELD)
212 #define ctype_isbitfield(info)        (ctype_type((info)) == CT_BITFIELD)
213 #define ctype_isconstval(info)        (ctype_type((info)) == CT_CONSTVAL)
214 #define ctype_isextern(info)        (ctype_type((info)) == CT_EXTERN)
215 #define ctype_hassize(info)        (ctype_type((info)) <= CT_HASSIZE)
216
217 /* Combined type and flag checks. */
218 #define ctype_isinteger(info) \
219   (((info) & (CTMASK_NUM|CTF_BOOL|CTF_FP)) == CTINFO(CT_NUM, 0))
220 #define ctype_isinteger_or_bool(info) \
221   (((info) & (CTMASK_NUM|CTF_FP)) == CTINFO(CT_NUM, 0))
222 #define ctype_isbool(info) \
223   (((info) & (CTMASK_NUM|CTF_BOOL)) == CTINFO(CT_NUM, CTF_BOOL))
224 #define ctype_isfp(info) \
225   (((info) & (CTMASK_NUM|CTF_FP)) == CTINFO(CT_NUM, CTF_FP))
226
227 #define ctype_ispointer(info) \
228   ((ctype_type(info) >> 1) == (CT_PTR >> 1))  /* Pointer or array. */
229 #define ctype_isref(info) \
230   (((info) & (CTMASK_NUM|CTF_REF)) == CTINFO(CT_PTR, CTF_REF))
231
232 #define ctype_isrefarray(info) \
233   (((info) & (CTMASK_NUM|CTF_VECTOR|CTF_COMPLEX)) == CTINFO(CT_ARRAY, 0))
234 #define ctype_isvector(info) \
235   (((info) & (CTMASK_NUM|CTF_VECTOR)) == CTINFO(CT_ARRAY, CTF_VECTOR))
236 #define ctype_iscomplex(info) \
237   (((info) & (CTMASK_NUM|CTF_COMPLEX)) == CTINFO(CT_ARRAY, CTF_COMPLEX))
238
239 #define ctype_isvltype(info) \
240   (((info) & ((CTMASK_NUM|CTF_VLA) - (2u<<CTSHIFT_NUM))) == \
241    CTINFO(CT_STRUCT, CTF_VLA))  /* VL array or VL struct. */
242 #define ctype_isvlarray(info) \
243   (((info) & (CTMASK_NUM|CTF_VLA)) == CTINFO(CT_ARRAY, CTF_VLA))
244
245 #define ctype_isxattrib(info, at) \
246   (((info) & (CTMASK_NUM|CTATTRIB(CTMASK_ATTRIB))) == \
247    CTINFO(CT_ATTRIB, CTATTRIB(at)))
248
249 /* Target-dependent sizes and alignments. */
250 #if LJ_64
251 #define CTSIZE_PTR        8
252 #define CTALIGN_PTR        CTALIGN(3)
253 #else
254 #define CTSIZE_PTR        4
255 #define CTALIGN_PTR        CTALIGN(2)
256 #endif
257
258 #define CTINFO_REF(ref) \
259   CTINFO(CT_PTR, (CTF_CONST|CTF_REF|CTALIGN_PTR) + (ref))
260
261 #define CT_MEMALIGN        3        /* Alignment guaranteed by memory allocator. */
262
263 /* -- Predefined types ----------------------------------------------------- */
264
```

```
/* Target-dependent types. */
#if LJ_TARGET_PPC
#define CTTYDEFP(_) \
  _(LINT32,                4,        CT_NUM, CTF_LONG|CTALIGN(2))
#else
#define CTTYDEFP(_)
#endif

/* Common types. */
#define CTTYDEF(_) \
  _(NONE,                0,        CT_ATTRIB, CTATTRIB(CTA_BAD)) \
  _(VOID,                -1,        CT_VOID, CTALIGN(0)) \
  _(CVOID,                -1,        CT_VOID, CTF_CONST|CTALIGN(0)) \
  _(BOOL,                1,        CT_NUM, CTF_BOOL|CTF_UNSIGNED|CTALIGN(0)) \
  _(CCHAR,                1,        CT_NUM, CTF_CONST|CTF_UCHAR|CTALIGN(0)) \
  _(INT8,                1,        CT_NUM, CTALIGN(0)) \
  _(UINT8,                1,        CT_NUM, CTF_UNSIGNED|CTALIGN(0)) \
  _(INT16,                2,        CT_NUM, CTALIGN(1)) \
  _(UINT16,                2,        CT_NUM, CTF_UNSIGNED|CTALIGN(1)) \
  _(INT32,                4,        CT_NUM, CTALIGN(2)) \
  _(UINT32,                4,        CT_NUM, CTF_UNSIGNED|CTALIGN(2)) \
  _(INT64,                8,        CT_NUM, CTF_LONG|CTALIGN(3)) \
  _(UINT64,                8,        CT_NUM, CTF_UNSIGNED|CTF_LONG|CTALIGN(3)) \
  _(FLOAT,                4,        CT_NUM, CTF_FP|CTALIGN(2)) \
  _(DOUBLE,                8,        CT_NUM, CTF_FP|CTALIGN(3)) \
  _(COMPLEX_FLOAT,        8,        CT_ARRAY, CTF_COMPLEX|CTALIGN(2)|CTID_FLOAT) \
  _(COMPLEX_DOUBLE,        16,        CT_ARRAY, CTF_COMPLEX|CTALIGN(3)|CTID_DOUBLE) \
  _(P_VOID,        CTSIZE_PTR,        CT_PTR, CTALIGN_PTR|CTID_VOID) \
  _(P_CVOID,        CTSIZE_PTR,        CT_PTR, CTALIGN_PTR|CTID_CVOID) \
  _(P_CCHAR,        CTSIZE_PTR,        CT_PTR, CTALIGN_PTR|CTID_CCHAR) \
  _(A_CCHAR,                -1,        CT_ARRAY, CTF_CONST|CTALIGN(0)|CTID_CCHAR) \
  _(CTYPEID,                4,        CT_ENUM, CTALIGN(2)|CTID_INT32) \
  CTTYDEFP(_) \
  /* End of type list. */

/* Public predefined type IDs. */
enum {
#define CTTYIDDEF(id, sz, ct, info)        CTID_##id,
CTTYDEF(CTTYIDDEF)
#undef CTTYIDDEF
  /* Predefined typedefs and keywords follow. */
  CTID_MAX = 65536
};

/* Target-dependent type IDs. */
#if LJ_64
#define CTID_INT_PSZ        CTID_INT64
#define CTID_UINT_PSZ        CTID_UINT64
#else
#define CTID_INT_PSZ        CTID_INT32
#define CTID_UINT_PSZ        CTID_UINT32
#endif

#if LJ_ABI_WIN
#define CTID_WCHAR        CTID_UINT16
#elif LJ_TARGET_PPC
#define CTID_WCHAR        CTID_LINT32
#else
#define CTID_WCHAR        CTID_INT32
#endif

/* -- C tokens and keywords ---------------------------------------------- */

/* C lexer keywords. */
#define CTOKDEF(_) \
  _(IDENT, "<identifier>") _(STRING, "<string>") \
  _(INTEGER, "<integer>") _(EOF, "<eof>") \
  _(OROR, "||") _(ANDAND, "&&") _(EQ, "==") _(NE, "!=") \
  _(LE, "<=") _(GE, ">=") _(SHL, "<<") _(SHR, ">>") _(DEREF, "->")

/* Simple declaration specifiers. */
#define CDSDEF(_) \
  _(VOID) _(BOOL) _(CHAR) _(INT) _(FP) \
  _(LONG) _(LONGLONG) _(SHORT) _(COMPLEX) _(SIGNED) _(UNSIGNED) \
  _(CONST) _(VOLATILE) _(RESTRICT) _(INLINE) \
  _(TYPEDEF) _(EXTERN) _(STATIC) _(AUTO) _(REGISTER)
```

```
341
342 /* C keywords. */
343 #define CKWDEF(_) \
344   CDSDEF(_) _(EXTENSION) _(ASM) _(ATTRIBUTE) \
345   _(DECLSPEC) _(CCDECL) _(PTRSZ) \
346   _(STRUCT) _(UNION) _(ENUM) \
347   _(SIZEOF) _(ALIGNOF)
348
349 /* C token numbers. */
350 enum {
351   CTOK_OFS = 255,
352 #define CTOKNUM(name, sym)        CTOK_##name,
353 #define CKWNUM(name)                CTOK_##name,
354 CTOKDEF(CTOKNUM)
355 CKWDEF(CKWNUM)
356 #undef CTOKNUM
357 #undef CKWNUM
358   CTOK_FIRSTDECL = CTOK_VOID,
359   CTOK_FIRSTSCL = CTOK_TYPEDEF,
360   CTOK_LASTDECLFLAG = CTOK_REGISTER,
361   CTOK_LASTDECL = CTOK_ENUM
362 };
363
364 /* Declaration specifier flags. */
365 enum {
366 #define CDSFLAG(name)        CDF_##name = (1u << (CTOK_##name - CTOK_FIRSTDECL)),
367 CDSDEF(CDSFLAG)
368 #undef CDSFLAG
369   CDF__END
370 };
371
372 #define CDF_SCL  (CDF_TYPEDEF|CDF_EXTERN|CDF_STATIC|CDF_AUTO|CDF_REGISTER)
373
374 /* -- C type management ------------------------------------------------- */
375
376 #define ctype_ctsG(g)                (mref((g)->ctype_state, CTState))
377
378 /* Get C type state. */
379 static LJ_AINLINE CTState *ctype_cts(lua_State *L)
380 {
381   CTState *cts = ctype_ctsG(G(L));
382   cts->L = L;  /* Save L for errors and allocations. */
383   return cts;
384 }
385
386 /* Save and restore state of C type table. */
387 #define LJ_CTYPE_SAVE(cts)        CTState savects_ = *(cts)
388 #define LJ_CTYPE_RESTORE(cts) \
389   ((cts)->top = savects_.top, \
390    memcpy((cts)->hash, savects_.hash, sizeof(savects_.hash)))
391
392 /* Check C type ID for validity when assertions are enabled. */
393 static LJ_AINLINE CTypeID ctype_check(CTState *cts, CTypeID id)
394 {
395   lua_assert(id > 0 && id < cts->top); UNUSED(cts);
396   return id;
397 }
398
399 /* Get C type for C type ID. */
400 static LJ_AINLINE CType *ctype_get(CTState *cts, CTypeID id)
401 {
402   return &cts->tab[ctype_check(cts, id)];
403 }
404
405 /* Get C type ID for a C type. */
406 #define ctype_typeid(cts, ct)        ((CTypeID)((ct) - (cts)->tab))
407
408 /* Get child C type. */
409 static LJ_AINLINE CType *ctype_child(CTState *cts, CType *ct)
410 {
411   lua_assert(!(ctype_isvoid(ct->info) || ctype_isstruct(ct->info) ||
412                ctype_isbitfield(ct->info)));  /* These don't have children. */
413   return ctype_get(cts, ctype_cid(ct->info));
414 }
415
416 /* Get raw type for a C type ID. */
```

```
417 static LJ_AINLINE CType *ctype_raw(CTState *cts, CTypeID id)
418 {
419   CType *ct = ctype_get(cts, id);
420   while (ctype_isattrib(ct->info)) ct = ctype_child(cts, ct);
421   return ct;
422 }
423
424 /* Get raw type of the child of a C type. */
425 static LJ_AINLINE CType *ctype_rawchild(CTState *cts, CType *ct)
426 {
427   do { ct = ctype_child(cts, ct); } while (ctype_isattrib(ct->info));
428   return ct;
429 }
430
431 /* Set the name of a C type table element. */
432 static LJ_AINLINE void ctype_setname(CType *ct, GCstr *s)
433 {
434   /* NOBARRIER: mark string as fixed -- the C type table is never collected. */
435   fixstring(s);
436   setgcref(ct->name, obj2gco(s));
437 }
438
439 LJ_FUNC CTypeID lj_ctype_new(CTState *cts, CType **ctp);
440 LJ_FUNC CTypeID lj_ctype_intern(CTState *cts, CTInfo info, CTSize size);
441 LJ_FUNC void lj_ctype_addname(CTState *cts, CType *ct, CTypeID id);
442 LJ_FUNC CTypeID lj_ctype_getname(CTState *cts, CType **ctp, GCstr *name,
443                                  uint32_t tmask);
444 LJ_FUNC CType *lj_ctype_getfieldq(CTState *cts, CType *ct, GCstr *name,
445                                   CTSize *ofs, CTInfo *qual);
446 #define lj_ctype_getfield(cts, ct, name, ofs) \
447   lj_ctype_getfieldq((cts), (ct), (name), (ofs), NULL)
448 LJ_FUNC CType *lj_ctype_rawref(CTState *cts, CTypeID id);
449 LJ_FUNC CTSize lj_ctype_size(CTState *cts, CTypeID id);
450 LJ_FUNC CTSize lj_ctype_vlsize(CTState *cts, CType *ct, CTSize nelem);
451 LJ_FUNC CTInfo lj_ctype_info(CTState *cts, CTypeID id, CTSize *szp);
452 LJ_FUNC cTValue *lj_ctype_meta(CTState *cts, CTypeID id, MMS mm);
453 LJ_FUNC GCstr *lj_ctype_repr(lua_State *L, CTypeID id, GCstr *name);
454 LJ_FUNC GCstr *lj_ctype_repr_int64(lua_State *L, uint64_t n, int isunsigned);
455 LJ_FUNC GCstr *lj_ctype_repr_complex(lua_State *L, void *sp, CTSize size);
456 LJ_FUNC CTState *lj_ctype_init(lua_State *L);
457 LJ_FUNC void lj_ctype_freestate(global_State *g);
458
459 #endif
460
461 #endif
```

One Level Up Top Level

src/lj_ccallback.c - luajit-2.0-src

Functions defined

- CALLBACK_OFS2SLOT
- callback_checkfunc
- callback_conv_args
- callback_conv_result
- callback_mcode_init
- callback_mcode_init
- callback_mcode_init
- callback_mcode_init
- callback_mcode_init
- callback_mcode_new
- callback_slot2ptr
- callback_slot_new
- lj_ccallback_enter
- lj_ccallback_leave
- lj_ccallback_mcode_free
- lj_ccallback_new
- lj_ccallback_ptr2slot

Macros defined

- CALLBACK_HANDLE_REGARG
- CALLBACK_HANDLE_REGARG
- CALLBACK_HANDLE_REGARG
- CALLBACK_HANDLE_REGARG
- CALLBACK_HANDLE_REGARG
- CALLBACK_HANDLE_REGARG
- CALLBACK_HANDLE_REGARG
- CALLBACK_HANDLE_REGARG_FP1
- CALLBACK_HANDLE_REGARG_FP1
- CALLBACK_HANDLE_REGARG_FP2
- CALLBACK_HANDLE_REGARG_FP2

- CALLBACK_HANDLE_RET
- CALLBACK_HANDLE_RET
- CALLBACK_MAX_SLOT
- CALLBACK_MAX_SLOT
- CALLBACK_MAX_SLOT
- CALLBACK_MAX_SLOT
- CALLBACK_MCODE_GROUP
- CALLBACK_MCODE_HEAD
- CALLBACK_MCODE_HEAD
- CALLBACK_MCODE_HEAD
- CALLBACK_MCODE_HEAD
- CALLBACK_MCODE_HEAD
- CALLBACK_MCODE_SIZE
- CALLBACK_OFS2SLOT
- CALLBACK_OFS2SLOT
- CALLBACK_OFS2SLOT
- CALLBACK_SLOT2OFS
- CALLBACK_SLOT2OFS
- CALLBACK_SLOT2OFS
- CALLBACK_SLOT2OFS
- MAP_ANONYMOUS
- WIN32_LEAN_AND_MEAN
- callback_mcode_init
- callback_mcode_init

Source code

```
1  /*
2  ** FFI C callback handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #include "lj_obj.h"
7
8  #if LJ_HASFFI
9
10  #include "lj_gc.h"
11  #include "lj_err.h"
12  #include "lj_tab.h"
13  #include "lj_state.h"
14  #include "lj_frame.h"
15  #include "lj_ctype.h"
16  #include "lj_cconv.h"
17  #include "lj_ccall.h"
18  #include "lj_ccallback.h"
```

```c
#include "lj_target.h"
#include "lj_mcode.h"
#include "lj_trace.h"
#include "lj_vm.h"

/* -- Target-specific handling of callback slots ---------------------------- */

#define CALLBACK_MCODE_SIZE	(LJ_PAGESIZE * LJ_NUM_CBPAGE)

#if LJ_OS_NOJIT

/* Callbacks disabled. */
#define CALLBACK_SLOT2OFS(slot)	(0*(slot))
#define CALLBACK_OFS2SLOT(ofs)	(0*(ofs))
#define CALLBACK_MAX_SLOT	0

#elif LJ_TARGET_X86ORX64

#define CALLBACK_MCODE_HEAD	(LJ_64 ? 8 : 0)
#define CALLBACK_MCODE_GROUP	(-2+1+2+5+(LJ_64 ? 6 : 5))

#define CALLBACK_SLOT2OFS(slot) \
  (CALLBACK_MCODE_HEAD + CALLBACK_MCODE_GROUP*((slot)/32) + 4*(slot))

static MSize CALLBACK_OFS2SLOT(MSize ofs)
{
  MSize group;
  ofs -= CALLBACK_MCODE_HEAD;
  group = ofs / (32*4 + CALLBACK_MCODE_GROUP);
  return (ofs % (32*4 + CALLBACK_MCODE_GROUP))/4 + group*32;
}

#define CALLBACK_MAX_SLOT \
  (((CALLBACK_MCODE_SIZE-CALLBACK_MCODE_HEAD)/(CALLBACK_MCODE_GROUP+4*32))*32)

#elif LJ_TARGET_ARM

#define CALLBACK_MCODE_HEAD		32

#elif LJ_TARGET_ARM64

#define CALLBACK_MCODE_HEAD		32

#elif LJ_TARGET_PPC

#define CALLBACK_MCODE_HEAD		24

#elif LJ_TARGET_MIPS

#define CALLBACK_MCODE_HEAD		24

#else

/* Missing support for this architecture. */
#define CALLBACK_SLOT2OFS(slot)	(0*(slot))
#define CALLBACK_OFS2SLOT(ofs)	(0*(ofs))
#define CALLBACK_MAX_SLOT	0

#endif

#ifndef CALLBACK_SLOT2OFS
#define CALLBACK_SLOT2OFS(slot)		(CALLBACK_MCODE_HEAD + 8*(slot))
#define CALLBACK_OFS2SLOT(ofs)		(((ofs)-CALLBACK_MCODE_HEAD)/8)
#define CALLBACK_MAX_SLOT		(CALLBACK_OFS2SLOT(CALLBACK_MCODE_SIZE))
#endif

/* Convert callback slot number to callback function pointer. */
static void *callback_slot2ptr(CTState *cts, MSize slot)
{
  return (uint8_t *)cts->cb.mcode + CALLBACK_SLOT2OFS(slot);
}

/* Convert callback function pointer to slot number. */
MSize lj_ccallback_ptr2slot(CTState *cts, void *p)
{
  uintptr_t ofs = (uintptr_t)((uint8_t *)p -(uint8_t *)cts->cb.mcode);
```

```
 if (ofs < CALLBACK_MCODE_SIZE) {
   MSize slot = CALLBACK_OFS2SLOT((MSize)ofs);
   if (CALLBACK_SLOT2OFS(slot) == (MSize)ofs)
     return slot;
 }
  return ~0u;  /* Not a known callback function pointer. */
}

/* Initialize machine code for callback function pointers. */
#if LJ_OS_NOJIT
/* Disabled callback support. */
#define callback_mcode_init(g, p)        UNUSED(p)
#elif LJ_TARGET_X86ORX64
static void callback_mcode_init(global_State *g, uint8_t *page)
{
  uint8_t *p = page;
  uint8_t *target = (uint8_t *)(void *)lj_vm_ffi_callback;
  MSize slot;
#if LJ_64
  *(void **)p = target; p += 8;
#endif
  for (slot = 0; slot < CALLBACK_MAX_SLOT; slot++) {
    /* mov al, slot; jmp group */
    *p++ = XI_MOVrib | RID_EAX; *p++ = (uint8_t)slot;
    if ((slot & 31) == 31 || slot == CALLBACK_MAX_SLOT-1) {
      /* push ebp/rbp; mov ah, slot>>8; mov ebp, &g. */
      *p++ = XI_PUSH + RID_EBP;
      *p++ = XI_MOVrib | (RID_EAX+4); *p++ = (uint8_t)(slot >> 8);
      *p++ = XI_MOVri | RID_EBP;
      *(int32_t *)p = i32ptr(g); p += 4;
#if LJ_64
      /* jmp [rip-pageofs] where lj_vm_ffi_callback is stored. */
      *p++ = XI_GROUP5; *p++ = XM_OFS0 + (XOg_JMP<<3) + RID_EBP;
      *(int32_t *)p = (int32_t)(page-(p+4)); p += 4;
#else
      /* jmp lj_vm_ffi_callback. */
      *p++ = XI_JMP; *(int32_t *)p = target-(p+4); p += 4;
#endif
    } else {
      *p++ = XI_JMPs; *p++ = (uint8_t)((2+2)*(31-(slot&31)) - 2);
    }
  }
  lua_assert(p - page <= CALLBACK_MCODE_SIZE);
}
#elif LJ_TARGET_ARM
static void callback_mcode_init(global_State *g, uint32_t *page)
{
  uint32_t *p = page;
  void *target = (void *)lj_vm_ffi_callback;
  MSize slot;
  /* This must match with the saveregs macro in buildvm_arm.dasc. */
  *p++ = ARMI_SUB|ARMF_D(RID_R12)|ARMF_N(RID_R12)|ARMF_M(RID_PC);
  *p++ = ARMI_PUSH|ARMF_N(RID_SP)|RSET_RANGE(RID_R4,RID_R11+1)|RID2RSET(RID_LR);
  *p++ = ARMI_SUB|ARMI_K12|ARMF_D(RID_R12)|ARMF_N(RID_R12)|CALLBACK_MCODE_HEAD;
  *p++ = ARMI_STR|ARMI_LS_P|ARMI_LS_W|ARMF_D(RID_R12)|ARMF_N(RID_SP)|(CFRAME_SIZE-4*9);
  *p++ = ARMI_LDR|ARMI_LS_P|ARMI_LS_U|ARMF_D(RID_R12)|ARMF_N(RID_PC);
  *p++ = ARMI_LDR|ARMI_LS_P|ARMI_LS_U|ARMF_D(RID_PC)|ARMF_N(RID_PC);
  *p++ = u32ptr(g);
  *p++ = u32ptr(target);
  for (slot = 0; slot < CALLBACK_MAX_SLOT; slot++) {
    *p++ = ARMI_MOV|ARMF_D(RID_R12)|ARMF_M(RID_PC);
    *p = ARMI_B | ((page-p-2) & 0x00ffffffu);
    p++;
  }
  lua_assert(p - page <= CALLBACK_MCODE_SIZE);
}
#elif LJ_TARGET_ARM64
static void callback_mcode_init(global_State *g, uint32_t *page)
{
  uint32_t *p = page;
  void *target = (void *)lj_vm_ffi_callback;
  MSize slot;
  *p++ = A64I_LDRLx | A64F_D(RID_X11) | A64F_S19(4);
  *p++ = A64I_LDRLx | A64F_D(RID_X10) | A64F_S19(5);
  *p++ = A64I_BR | A64F_N(RID_X11);
  *p++ = A64I_NOP;
```

```
  ((void **)p)[0] = target;
  ((void **)p)[1] = g;
  p += 4;
  for (slot = 0; slot < CALLBACK_MAX_SLOT; slot++) {
    *p++ = A64I_MOVZw | A64F_D(RID_X9) | A64F_U16(slot);
    *p = A64I_B | A64F_S26((page-p) & 0x03ffffffu);
    p++;
  }
  lua_assert(p - page <= CALLBACK_MCODE_SIZE);
}
#elif LJ_TARGET_PPC
static void callback_mcode_init(global_State *g, uint32_t *page)
{
  uint32_t *p = page;
  void *target = (void *)lj_vm_ffi_callback;
  MSize slot;
  *p++ = PPCI_LIS | PPCF_T(RID_TMP) | (u32ptr(target) >> 16);
  *p++ = PPCI_LIS | PPCF_T(RID_R12) | (u32ptr(g) >> 16);
  *p++ = PPCI_ORI | PPCF_A(RID_TMP)|PPCF_T(RID_TMP) | (u32ptr(target) & 0xffff);
  *p++ = PPCI_ORI | PPCF_A(RID_R12)|PPCF_T(RID_R12) | (u32ptr(g) & 0xffff);
  *p++ = PPCI_MTCTR | PPCF_T(RID_TMP);
  *p++ = PPCI_BCTR;
  for (slot = 0; slot < CALLBACK_MAX_SLOT; slot++) {
    *p++ = PPCI_LI | PPCF_T(RID_R11) | slot;
    *p = PPCI_B | (((page-p) & 0x00ffffffu) << 2);
    p++;
  }
  lua_assert(p - page <= CALLBACK_MCODE_SIZE);
}
#elif LJ_TARGET_MIPS
static void callback_mcode_init(global_State *g, uint32_t *page)
{
  uint32_t *p = page;
  void *target = (void *)lj_vm_ffi_callback;
  MSize slot;
  *p++ = MIPSI_SW | MIPSF_T(RID_R1)|MIPSF_S(RID_SP) | 0;
  *p++ = MIPSI_LUI | MIPSF_T(RID_R3) | (u32ptr(target) >> 16);
  *p++ = MIPSI_LUI | MIPSF_T(RID_R2) | (u32ptr(g) >> 16);
  *p++ = MIPSI_ORI | MIPSF_T(RID_R3)|MIPSF_S(RID_R3) |(u32ptr(target)&0xffff);
  *p++ = MIPSI_JR | MIPSF_S(RID_R3);
  *p++ = MIPSI_ORI | MIPSF_T(RID_R2)|MIPSF_S(RID_R2) | (u32ptr(g)&0xffff);
  for (slot = 0; slot < CALLBACK_MAX_SLOT; slot++) {
    *p = MIPSI_B | ((page-p-1) & 0x0000ffffu);
    p++;
    *p++ = MIPSI_LI | MIPSF_T(RID_R1) | slot;
  }
  lua_assert(p - page <= CALLBACK_MCODE_SIZE);
}
#else
/* Missing support for this architecture. */
#define callback_mcode_init(g, p)       UNUSED(p)
#endif

/* -- Machine code management ---------------------------------------- */

#if LJ_TARGET_WINDOWS

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#elif LJ_TARGET_POSIX

#include <sys/mman.h>
#ifndef MAP_ANONYMOUS
#define MAP_ANONYMOUS   MAP_ANON
#endif

#endif

/* Allocate and initialize area for callback function pointers. */
static void callback_mcode_new(CTState *cts)
{
  size_t sz = (size_t)CALLBACK_MCODE_SIZE;
  void *p;
  if (CALLBACK_MAX_SLOT == 0)
    lj_err_caller(cts->L, LJ_ERR_FFI_CBACKOV);
```

```
#if LJ_TARGET_WINDOWS
  p = VirtualAlloc(NULL, sz, MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
  if (!p)
    lj_err_caller(cts->L, LJ_ERR_FFI_CBACKOV);
#elif LJ_TARGET_POSIX
  p = mmap(NULL, sz, (PROT_READ|PROT_WRITE), MAP_PRIVATE|MAP_ANONYMOUS,
           -1, 0);
  if (p == MAP_FAILED)
    lj_err_caller(cts->L, LJ_ERR_FFI_CBACKOV);
#else
  /* Fallback allocator. Fails if memory is not executable by default. */
  p = lj_mem_new(cts->L, sz);
#endif
  cts->cb.mcode = p;
  callback_mcode_init(cts->g, p);
  lj_mcode_sync(p, (char *)p + sz);
#if LJ_TARGET_WINDOWS
  {
    DWORD oprot;
    VirtualProtect(p, sz, PAGE_EXECUTE_READ, &oprot);
  }
#elif LJ_TARGET_POSIX
  mprotect(p, sz, (PROT_READ|PROT_EXEC));
#endif
}

/* Free area for callback function pointers. */
void lj_ccallback_mcode_free(CTState *cts)
{
  size_t sz = (size_t)CALLBACK_MCODE_SIZE;
  void *p = cts->cb.mcode;
  if (p == NULL) return;
#if LJ_TARGET_WINDOWS
  VirtualFree(p, 0, MEM_RELEASE);
  UNUSED(sz);
#elif LJ_TARGET_POSIX
  munmap(p, sz);
#else
  lj_mem_free(cts->g, p, sz);
#endif
}

/* -- C callback entry ---------------------------------------------------- */

/* Target-specific handling of register arguments. Similar to lj_ccall.c. */
#if LJ_TARGET_X86

#define CALLBACK_HANDLE_REGARG \
  if (!isfp) {  /* Only non-FP values may be passed in registers. */ \
    if (n > 1) {  /* Anything > 32 bit is passed on the stack. */ \
      if (!LJ_ABI_WIN) ngpr = maxgpr;  /* Prevent reordering. */ \
    } else if (ngpr + 1 <= maxgpr) { \
      sp = &cts->cb.gpr[ngpr]; \
      ngpr += n; \
      goto done; \
    } \
  }

#elif LJ_TARGET_X64 && LJ_ABI_WIN

/* Windows/x64 argument registers are strictly positional (use ngpr). */
#define CALLBACK_HANDLE_REGARG \
  if (isfp) { \
    if (ngpr < maxgpr) { sp = &cts->cb.fpr[ngpr++]; UNUSED(nfpr); goto done; } \
  } else { \
    if (ngpr < maxgpr) { sp = &cts->cb.gpr[ngpr++]; goto done; } \
  }

#elif LJ_TARGET_X64

#define CALLBACK_HANDLE_REGARG \
  if (isfp) { \
    if (nfpr + n <= CCALL_NARG_FPR) { \
      sp = &cts->cb.fpr[nfpr]; \
      nfpr += n; \
      goto done; \
```

```
    } \
  } else { \
    if (ngpr + n <= maxgpr) { \
      sp = &cts->cb.gpr[ngpr]; \
      ngpr += n; \
      goto done; \
    } \
  }

#elif LJ_TARGET_ARM

#if LJ_ABI_SOFTFP

#define CALLBACK_HANDLE_REGARG_FP1        UNUSED(isfp);
#define CALLBACK_HANDLE_REGARG_FP2

#else

#define CALLBACK_HANDLE_REGARG_FP1 \
  if (isfp) { \
    if (n == 1) { \
      if (fprodd) { \
        sp = &cts->cb.fpr[fprodd-1]; \
        fprodd = 0; \
        goto done; \
      } else if (nfpr + 1 <= CCALL_NARG_FPR) { \
        sp = &cts->cb.fpr[nfpr++]; \
        fprodd = nfpr; \
        goto done; \
      } \
    } else { \
      if (nfpr + 1 <= CCALL_NARG_FPR) { \
        sp = &cts->cb.fpr[nfpr++]; \
        goto done; \
      } \
    } \
    fprodd = 0;  /* No reordering after the first FP value is on stack. */ \
  } else {

#define CALLBACK_HANDLE_REGARG_FP2        }

#endif

#define CALLBACK_HANDLE_REGARG \
  CALLBACK_HANDLE_REGARG_FP1 \
  if (n > 1) ngpr = (ngpr + 1u) & ~1u;  /* Align to regpair. */ \
  if (ngpr + n <= maxgpr) { \
    sp = &cts->cb.gpr[ngpr]; \
    ngpr += n; \
    goto done; \
  } CALLBACK_HANDLE_REGARG_FP2

#elif LJ_TARGET_ARM64

#define CALLBACK_HANDLE_REGARG \
  if (isfp) { \
    if (nfpr + n <= CCALL_NARG_FPR) { \
      sp = &cts->cb.fpr[nfpr]; \
      nfpr += n; \
      goto done; \
    } else { \
      nfpr = CCALL_NARG_FPR;  /* Prevent reordering. */ \
    } \
  } else { \
    if (!LJ_TARGET_IOS && n > 1) \
      ngpr = (ngpr + 1u) & ~1u;  /* Align to regpair. */ \
    if (ngpr + n <= maxgpr) { \
      sp = &cts->cb.gpr[ngpr]; \
      ngpr += n; \
      goto done; \
    } else { \
      ngpr = CCALL_NARG_GPR;  /* Prevent reordering. */ \
    } \
  }

#elif LJ_TARGET_PPC
```

```
399
400 #define CALLBACK_HANDLE_REGARG \
401   if (isfp) { \
402     if (nfpr + 1 <= CCALL_NARG_FPR) { \
403       sp = &cts->cb.fpr[nfpr++]; \
404       cta = ctype_get(cts, CTID_DOUBLE);  /* FPRs always hold doubles. */ \
405       goto done; \
406     } \
407   } else {  /* Try to pass argument in GPRs. */ \
408     if (n > 1) { \
409       lua_assert(ctype_isinteger(cta->info) && n == 2);  /* int64_t. */ \
410       ngpr = (ngpr + 1u) & ~1u;  /* Align int64_t to regpair. */ \
411     } \
412     if (ngpr + n <= maxgpr) { \
413       sp = &cts->cb.gpr[ngpr]; \
414       ngpr += n; \
415       goto done; \
416     } \
417   }
418
419 #define CALLBACK_HANDLE_RET \
420   if (ctype_isfp(ctr->info) && ctr->size == sizeof(float)) \
421     *(double *)dp = *(float *)dp;  /* FPRs always hold doubles. */
422
423 #elif LJ_TARGET_MIPS
424
425 #define CALLBACK_HANDLE_REGARG \
426   if (isfp && nfpr < CCALL_NARG_FPR) {  /* Try to pass argument in FPRs. */ \
427     sp = (void *)((uint8_t *)&cts->cb.fpr[nfpr] + ((LJ_BE && n==1) ? 4 : 0)); \
428     nfpr++; ngpr += n; \
429     goto done; \
430   } else {  /* Try to pass argument in GPRs. */ \
431     nfpr = CCALL_NARG_FPR; \
432     if (n > 1) ngpr = (ngpr + 1u) & ~1u;  /* Align to regpair. */ \
433     if (ngpr + n <= maxgpr) { \
434       sp = &cts->cb.gpr[ngpr]; \
435       ngpr += n; \
436       goto done; \
437     } \
438   }
439
440 #define CALLBACK_HANDLE_RET \
441   if (ctype_isfp(ctr->info) && ctr->size == sizeof(float)) \
442     ((float *)dp)[1] = *(float *)dp;
443
444 #else
445 #error "Missing calling convention definitions for this architecture"
446 #endif
447
448 /* Convert and push callback arguments to Lua stack. */
449 static void callback_conv_args(CTState *cts, lua_State *L)
450 {
451   TValue *o = L->top;
452   intptr_t *stack = cts->cb.stack;
453   MSize slot = cts->cb.slot;
454   CTypeID id = 0, rid, fid;
455   int gcsteps = 0;
456   CType *ct;
457   GCfunc *fn;
458   int fntp;
459   MSize ngpr = 0, nsp = 0, maxgpr = CCALL_NARG_GPR;
460 #if CCALL_NARG_FPR
461   MSize nfpr = 0;
462 #if LJ_TARGET_ARM
463   MSize fprodd = 0;
464 #endif
465 #endif
466
467   if (slot < cts->cb.sizeid && (id = cts->cb.cbid[slot]) != 0) {
468     ct = ctype_get(cts, id);
469     rid = ctype_cid(ct->info);  /* Return type. x86: +(spadj<<16). */
470     fn = funcV(lj_tab_getint(cts->miscmap, (int32_t)slot));
471     fntp = LJ_TFUNC;
472   } else {  /* Must set up frame first, before throwing the error. */
473     ct = NULL;
474     rid = 0;
```

```
    fn = (GCfunc *)L;
    fntp = LJ_TTHREAD;
  }
  /* Continuation returns from callback. */
  if (LJ_FR2) {
    (o++)->u64 = LJ_CONT_FFI_CALLBACK;
    (o++)->u64 = rid;
    o++;
  } else {
    o->u32.lo = LJ_CONT_FFI_CALLBACK;
    o->u32.hi = rid;
    o++;
  }
  setframe_gc(o, obj2gco(fn), fntp);
  setframe_ftsz(o, ((char *)(o+1) - (char *)L->base) + FRAME_CONT);
  L->top = L->base = ++o;
  if (!ct)
    lj_err_caller(cts->L, LJ_ERR_FFI_BADCBACK);
  if (isluafunc(fn))
    setcframe_pc(L->cframe, proto_bc(funcproto(fn))+1);
  lj_state_checkstack(L, LUA_MINSTACK);  /* May throw. */
  o = L->base;  /* Might have been reallocated. */

#if LJ_TARGET_X86
  /* x86 has several different calling conventions. */
  switch (ctype_cconv(ct->info)) {
  case CTCC_FASTCALL: maxgpr = 2; break;
  case CTCC_THISCALL: maxgpr = 1; break;
  default: maxgpr = 0; break;
  }
#endif

  fid = ct->sib;
  while (fid) {
    CType *ctf = ctype_get(cts, fid);
    if (!ctype_isattrib(ctf->info)) {
      CType *cta;
      void *sp;
      CTSize sz;
      int isfp;
      MSize n;
      lua_assert(ctype_isfield(ctf->info));
      cta = ctype_rawchild(cts, ctf);
      isfp = ctype_isfp(cta->info);
      sz = (cta->size + CTSIZE_PTR-1) & ~(CTSIZE_PTR-1);
      n = sz / CTSIZE_PTR;  /* Number of GPRs or stack slots needed. */

      CALLBACK_HANDLE_REGARG  /* Handle register arguments. */

      /* Otherwise pass argument on stack. */
      if (CCALL_ALIGN_STACKARG && LJ_32 && sz == 8)
        nsp = (nsp + 1) & ~1u;  /* Align 64 bit argument on stack. */
      sp = &stack[nsp];
      nsp += n;

    done:
      if (LJ_BE && cta->size < CTSIZE_PTR)
        sp = (void *)((uint8_t *)sp + CTSIZE_PTR-cta->size);
      gcsteps += lj_cconv_tv_ct(cts, cta, 0, o++, sp);
    }
    fid = ctf->sib;
  }
  L->top = o;
#if LJ_TARGET_X86
  /* Store stack adjustment for returns from non-cdecl callbacks. */
  if (ctype_cconv(ct->info) != CTCC_CDECL) {
#if LJ_FR2
    (L->base-3)->u64 |= (nsp << (16+2));
#else
    (L->base-2)->u32.hi |= (nsp << (16+2));
#endif
  }
#endif
  while (gcsteps-- > 0)
    lj_gc_check(L);
}
```

```
551
552 /* Convert Lua object to callback result. */
553 static void callback_conv_result(CTState *cts, lua_State *L, TValue *o)
554 {
555 #if LJ_FR2
556   CType *ctr = ctype_raw(cts, (uint16_t)(L->base-3)->u64);
557 #else
558   CType *ctr = ctype_raw(cts, (uint16_t)(L->base-2)->u32.hi);
559 #endif
560 #if LJ_TARGET_X86
561   cts->cb.gpr[2] = 0;
562 #endif
563   if (!ctype_isvoid(ctr->info)) {
564     uint8_t *dp = (uint8_t *)&cts->cb.gpr[0];
565 #if CCALL_NUM_FPR
566     if (ctype_isfp(ctr->info))
567       dp = (uint8_t *)&cts->cb.fpr[0];
568 #endif
569     lj_cconv_ct_tv(cts, ctr, dp, o, 0);
570 #ifdef CALLBACK_HANDLE_RET
571     CALLBACK_HANDLE_RET
572 #endif
573     /* Extend returned integers to (at least) 32 bits. */
574     if (ctype_isinteger_or_bool(ctr->info) && ctr->size < 4) {
575       if (ctr->info & CTF_UNSIGNED)
576         *(uint32_t *)dp = ctr->size == 1 ? (uint32_t)*(uint8_t *)dp :
577                                            (uint32_t)*(uint16_t *)dp;
578       else
579         *(int32_t *)dp = ctr->size == 1 ? (int32_t)*(int8_t *)dp :
580                                           (int32_t)*(int16_t *)dp;
581     }
582 #if LJ_TARGET_X86
583     if (ctype_isfp(ctr->info))
584       cts->cb.gpr[2] = ctr->size == sizeof(float) ? 1 : 2;
585 #endif
586   }
587 }
588
589 /* Enter callback. */
590 lua_State * LJ_FASTCALL lj_ccallback_enter(CTState *cts, void *cf)
591 {
592   lua_State *L = cts->L;
593   global_State *g = cts->g;
594   lua_assert(L != NULL);
595   if (tvref(g->jit_base)) {
596     setstrV(L, L->top++, lj_err_str(L, LJ_ERR_FFI_BADCBACK));
597     if (g->panic) g->panic(L);
598     exit(EXIT_FAILURE);
599   }
600   lj_trace_abort(g);  /* Never record across callback. */
601   /* Setup C frame. */
602   cframe_prev(cf) = L->cframe;
603   setcframe_L(cf, L);
604   cframe_errfunc(cf) = -1;
605   cframe_nres(cf) = 0;
606   L->cframe = cf;
607   callback_conv_args(cts, L);
608   return L;  /* Now call the function on this stack. */
609 }
610
611 /* Leave callback. */
612 void LJ_FASTCALL lj_ccallback_leave(CTState *cts, TValue *o)
613 {
614   lua_State *L = cts->L;
615   GCfunc *fn;
616   TValue *obase = L->base;
617   L->base = L->top;  /* Keep continuation frame for throwing errors. */
618   if (o >= L->base) {
619     /* PC of RET* is lost. Point to last line for result conv. errors. */
620     fn = curr_func(L);
621     if (isluafunc(fn)) {
622       GCproto *pt = funcproto(fn);
623       setcframe_pc(L->cframe, proto_bc(pt)+pt->sizebc+1);
624     }
625   }
626   callback_conv_result(cts, L, o);
```

```
  /* Finally drop C frame and continuation frame. */
  L->top -= 2+2*LJ_FR2;
  L->base = obase;
  L->cframe = cframe_prev(L->cframe);
  cts->cb.slot = 0;  /* Blacklist C function that called the callback. */
}

/* -- C callback management ------------------------------------------- */

/* Get an unused slot in the callback slot table. */
static MSize callback_slot_new(CTState *cts, CType *ct)
{
  CTypeID id = ctype_typeid(cts, ct);
  CTypeID1 *cbid = cts->cb.cbid;
  MSize top;
  for (top = cts->cb.topid; top < cts->cb.sizeid; top++)
    if (LJ_LIKELY(cbid[top] == 0))
      goto found;
#if CALLBACK_MAX_SLOT
  if (top >= CALLBACK_MAX_SLOT)
#endif
    lj_err_caller(cts->L, LJ_ERR_FFI_CBACKOV);
  if (!cts->cb.mcode)
    callback_mcode_new(cts);
  lj_mem_growvec(cts->L, cbid, cts->cb.sizeid, CALLBACK_MAX_SLOT, CTypeID1);
  cts->cb.cbid = cbid;
  memset(cbid+top, 0, (cts->cb.sizeid-top)*sizeof(CTypeID1));
found:
  cbid[top] = id;
  cts->cb.topid = top+1;
  return top;
}

/* Check for function pointer and supported argument/result types. */
static CType *callback_checkfunc(CTState *cts, CType *ct)
{
  int narg = 0;
  if (!ctype_isptr(ct->info) || (LJ_64 && ct->size != CTSIZE_PTR))
    return NULL;
  ct = ctype_rawchild(cts, ct);
  if (ctype_isfunc(ct->info)) {
    CType *ctr = ctype_rawchild(cts, ct);
    CTypeID fid = ct->sib;
    if (!(ctype_isvoid(ctr->info) || ctype_isenum(ctr->info) ||
          ctype_isptr(ctr->info) || (ctype_isnum(ctr->info) && ctr->size <= 8)))
      return NULL;
    if ((ct->info & CTF_VARARG))
      return NULL;
    while (fid) {
      CType *ctf = ctype_get(cts, fid);
      if (!ctype_isattrib(ctf->info)) {
        CType *cta;
        lua_assert(ctype_isfield(ctf->info));
        cta = ctype_rawchild(cts, ctf);
        if (!(ctype_isenum(cta->info) || ctype_isptr(cta->info) ||
              (ctype_isnum(cta->info) && cta->size <= 8)) ||
            ++narg >= LUA_MINSTACK-3)
          return NULL;
      }
      fid = ctf->sib;
    }
    return ct;
  }
  return NULL;
}

/* Create a new callback and return the callback function pointer. */
void *lj_ccallback_new(CTState *cts, CType *ct, GCfunc *fn)
{
  ct = callback_checkfunc(cts, ct);
  if (ct) {
    MSize slot = callback_slot_new(cts, ct);
    GCtab *t = cts->miscmap;
    setfuncV(cts->L, lj_tab_setint(cts->L, t, (int32_t)slot), fn);
    lj_gc_anybarriert(cts->L, t);
    return callback_slot2ptr(cts, slot);
```

```
703   }
704   return NULL;  /* Bad conversion. */
705 }
706
707 #endif
```

One Level Up Top Level

src/lj_target_arm.h - luajit-2.0-src

Data types defined

- ARMCC
- ARMCC
- ARMIns
- ARMIns
- ARMShift
- ARMShift
- ExitState

Macros defined

- ARMF_CC
- ARMF_D
- ARMF_M
- ARMF_N
- ARMF_RSH
- ARMF_S
- ARMF_SH
- EXITSTATE_CHECKEXIT
- EXITSTATE_PCREG
- EXITSTUBS_PER_GROUP
- EXITSTUB_SPACING
- FPRDEF
- FPRDEF
- GPRDEF
- REGARG_FIRSTFPR
- REGARG_FIRSTFPR
- REGARG_FIRSTGPR
- REGARG_LASTFPR
- REGARG_LASTFPR
- REGARG_LASTGPR
- REGARG_NUMFPR

- REGARG_NUMFPR
- REGARG_NUMGPR
- RIDENUM
- RID_MIN_KREF
- RID_NUM_KREF
- RSET_ALL
- RSET_FPR
- RSET_FPR
- RSET_GPR
- RSET_GPREVEN
- RSET_GPRODD
- RSET_INIT
- RSET_SCRATCH
- RSET_SCRATCH_FPR
- RSET_SCRATCH_FPR
- RSET_SCRATCH_GPR
- RSET_SCRATCH_GPR
- RSET_SCRATCH_GPR_
- SPOFS_TMP
- SPS_FIRST
- SPS_FIXED
- VRIDDEF
- _LJ_TARGET_ARM_H
- sps_align
- sps_scale

Source code

```
/*
** Definitions for ARM CPUs.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_TARGET_ARM_H
#define _LJ_TARGET_ARM_H

/* -- Registers IDs ------------------------------------------------------- */

 #define GPRDEF(_) \
   _(R0) _(R1) _(R2) _(R3) _(R4) _(R5) _(R6) _(R7) \
   _(R8) _(R9) _(R10) _(R11) _(R12) _(SP) _(LR) _(PC)
 #if LJ_SOFTFP
 #define FPRDEF(_)
 #else
```

```
#define FPRDEF(_) \
  _(D0) _(D1) _(D2) _(D3) _(D4) _(D5) _(D6) _(D7) \
  _(D8) _(D9) _(D10) _(D11) _(D12) _(D13) _(D14) _(D15)
#endif
#define VRIDDEF(_)

#define RIDENUM(name)        RID_##name,

enum {
  GPRDEF(RIDENUM)               /* General-purpose registers (GPRs). */
  FPRDEF(RIDENUM)               /* Floating-point registers (FPRs). */
  RID_MAX,
  RID_TMP = RID_LR,

  /* Calling conventions. */
  RID_RET = RID_R0,
  RID_RETLO = RID_R0,
  RID_RETHI = RID_R1,
#if LJ_SOFTFP
  RID_FPRET = RID_R0,
#else
  RID_FPRET = RID_D0,
#endif

  /* These definitions must match with the *.dasc file(s): */
  RID_BASE = RID_R9,               /* Interpreter BASE. */
  RID_LPC = RID_R6,               /* Interpreter PC. */
  RID_DISPATCH = RID_R7,        /* Interpreter DISPATCH table. */
  RID_LREG = RID_R8,               /* Interpreter L. */

  /* Register ranges [min, max) and number of registers. */
  RID_MIN_GPR = RID_R0,
  RID_MAX_GPR = RID_PC+1,
  RID_MIN_FPR = RID_MAX_GPR,
#if LJ_SOFTFP
  RID_MAX_FPR = RID_MIN_FPR,
#else
  RID_MAX_FPR = RID_D15+1,
#endif
  RID_NUM_GPR = RID_MAX_GPR - RID_MIN_GPR,
  RID_NUM_FPR = RID_MAX_FPR - RID_MIN_FPR
};

#define RID_NUM_KREF                RID_NUM_GPR
#define RID_MIN_KREF                RID_R0

/* -- Register sets ---------------------------------------------------- */

/* Make use of all registers, except sp, lr and pc. */
#define RSET_GPR                (RSET_RANGE(RID_MIN_GPR, RID_R12+1))
#define RSET_GPREVEN \
  (RID2RSET(RID_R0)|RID2RSET(RID_R2)|RID2RSET(RID_R4)|RID2RSET(RID_R6)| \
   RID2RSET(RID_R8)|RID2RSET(RID_R10))
#define RSET_GPRODD \
  (RID2RSET(RID_R1)|RID2RSET(RID_R3)|RID2RSET(RID_R5)|RID2RSET(RID_R7)| \
   RID2RSET(RID_R9)|RID2RSET(RID_R11))
#if LJ_SOFTFP
#define RSET_FPR                0
#else
#define RSET_FPR                (RSET_RANGE(RID_MIN_FPR, RID_MAX_FPR))
#endif
#define RSET_ALL                (RSET_GPR|RSET_FPR)
#define RSET_INIT                RSET_ALL

/* ABI-specific register sets. lr is an implicit scratch register. */
#define RSET_SCRATCH_GPR_        (RSET_RANGE(RID_R0, RID_R3+1)|RID2RSET(RID_R12))
#ifdef __APPLE__
#define RSET_SCRATCH_GPR        (RSET_SCRATCH_GPR_|RID2RSET(RID_R9))
#else
#define RSET_SCRATCH_GPR        RSET_SCRATCH_GPR_
#endif
#if LJ_SOFTFP
#define RSET_SCRATCH_FPR        0
#else
#define RSET_SCRATCH_FPR        (RSET_RANGE(RID_D0, RID_D7+1))
#endif
```

```
#define RSET_SCRATCH                (RSET_SCRATCH_GPR|RSET_SCRATCH_FPR)
#define REGARG_FIRSTGPR               RID_R0
#define REGARG_LASTGPR               RID_R3
#define REGARG_NUMGPR                4
#if LJ_ABI_SOFTFP
#define REGARG_FIRSTFPR               0
#define REGARG_LASTFPR               0
#define REGARG_NUMFPR                0
#else
#define REGARG_FIRSTFPR                RID_D0
#define REGARG_LASTFPR               RID_D7
#define REGARG_NUMFPR                8
#endif

/* -- Spill slots --------------------------------------------------------- */

/* Spill slots are 32 bit wide. An even/odd pair is used for FPRs.
**
** SPS_FIXED: Available fixed spill slots in interpreter frame.
** This definition must match with the *.dasc file(s).
**
** SPS_FIRST: First spill slot for general use. Reserve min. two 32 bit slots.
*/
#define SPS_FIXED        2
#define SPS_FIRST        2

#define SPOFS_TMP        0

#define sps_scale(slot)               (4 * (int32_t)(slot))
#define sps_align(slot)               (((slot) - SPS_FIXED + 1) & ~1)

/* -- Exit state ---------------------------------------------------------- */

/* This definition must match with the *.dasc file(s). */
typedef struct {
#if !LJ_SOFTFP
  lua_Number fpr[RID_NUM_FPR];        /* Floating-point registers. */
#endif
  int32_t gpr[RID_NUM_GPR];        /* General-purpose registers. */
  int32_t spill[256];                /* Spill slots. */
} ExitState;

/* PC after instruction that caused an exit. Used to find the trace number. */
#define EXITSTATE_PCREG                RID_PC
/* Highest exit + 1 indicates stack check. */
#define EXITSTATE_CHECKEXIT        1

#define EXITSTUB_SPACING        4
#define EXITSTUBS_PER_GROUP     32

/* -- Instructions -------------------------------------------------------- */

/* Instruction fields. */
#define ARMF_CC(ai, cc)        (((ai) ^ ARMI_CCAL) | ((cc) << 28))
#define ARMF_N(r)        ((r) << 16)
#define ARMF_D(r)        ((r) << 12)
#define ARMF_S(r)        ((r) << 8)
#define ARMF_M(r)        (r)
#define ARMF_SH(sh, n)        (((sh) << 5) | ((n) << 7))
#define ARMF_RSH(sh, r)        (0x10 | ((sh) << 5) | ARMF_S(r))

typedef enum ARMIns {
  ARMI_CCAL = 0xe0000000,
  ARMI_S = 0x000100000,
  ARMI_K12 = 0x02000000,
  ARMI_KNEG = 0x00200000,
  ARMI_LS_W = 0x00200000,
  ARMI_LS_U = 0x00800000,
  ARMI_LS_P = 0x01000000,
  ARMI_LS_R = 0x02000000,
  ARMI_LSX_I = 0x00400000,

  ARMI_AND = 0xe0000000,
  ARMI_EOR = 0xe0200000,
  ARMI_SUB = 0xe0400000,
  ARMI_RSB = 0xe0600000,
```

```
    ARMI_ADD = 0xe0800000,
    ARMI_ADC = 0xe0a00000,
    ARMI_SBC = 0xe0c00000,
    ARMI_RSC = 0xe0e00000,
    ARMI_TST = 0xe1100000,
    ARMI_TEQ = 0xe1300000,
    ARMI_CMP = 0xe1500000,
    ARMI_CMN = 0xe1700000,
    ARMI_ORR = 0xe1800000,
    ARMI_MOV = 0xe1a00000,
    ARMI_BIC = 0xe1c00000,
    ARMI_MVN = 0xe1e00000,

    ARMI_NOP = 0xe1a00000,

    ARMI_MUL = 0xe0000090,
    ARMI_SMULL = 0xe0c00090,

    ARMI_LDR = 0xe4100000,
    ARMI_LDRB = 0xe4500000,
    ARMI_LDRH = 0xe01000b0,
    ARMI_LDRSB = 0xe01000d0,
    ARMI_LDRSH = 0xe01000f0,
    ARMI_LDRD = 0xe00000d0,
    ARMI_STR = 0xe4000000,
    ARMI_STRB = 0xe4400000,
    ARMI_STRH = 0xe00000b0,
    ARMI_STRD = 0xe00000f0,
    ARMI_PUSH = 0xe92d0000,

    ARMI_B = 0xea000000,
    ARMI_BL = 0xeb000000,
    ARMI_BLX = 0xfa000000,
    ARMI_BLXr = 0xe12fff30,

    /* ARMv6 */
    ARMI_REV = 0xe6bf0f30,
    ARMI_SXTB = 0xe6af0070,
    ARMI_SXTH = 0xe6bf0070,
    ARMI_UXTB = 0xe6ef0070,
    ARMI_UXTH = 0xe6ff0070,

    /* ARMv6T2 */
    ARMI_MOVW = 0xe3000000,
    ARMI_MOVT = 0xe3400000,

    /* VFP */
    ARMI_VMOV_D = 0xeeb00b40,
    ARMI_VMOV_S = 0xeeb00a40,
    ARMI_VMOVI_D = 0xeeb00b00,

    ARMI_VMOV_R_S = 0xee100a10,
    ARMI_VMOV_S_R = 0xee000a10,
    ARMI_VMOV_RR_D = 0xec500b10,
    ARMI_VMOV_D_RR = 0xec400b10,

    ARMI_VADD_D = 0xee300b00,
    ARMI_VSUB_D = 0xee300b40,
    ARMI_VMUL_D = 0xee200b00,
    ARMI_VMLA_D = 0xee000b00,
    ARMI_VMLS_D = 0xee000b40,
    ARMI_VNMLS_D = 0xee100b00,
    ARMI_VDIV_D = 0xee800b00,

    ARMI_VABS_D = 0xeeb00bc0,
    ARMI_VNEG_D = 0xeeb10b40,
    ARMI_VSQRT_D = 0xeeb10bc0,

    ARMI_VCMP_D = 0xeeb40b40,
    ARMI_VCMPZ_D = 0xeeb50b40,

    ARMI_VMRS = 0xeef1fa10,

    ARMI_VCVT_S32_F32 = 0xeebd0ac0,
    ARMI_VCVT_S32_F64 = 0xeebd0bc0,
    ARMI_VCVT_U32_F32 = 0xeebc0ac0,
```

```
245   ARMI_VCVT_U32_F64 = 0xeebc0bc0,
246   ARMI_VCVT_F32_S32 = 0xeeb80ac0,
247   ARMI_VCVT_F64_S32 = 0xeeb80bc0,
248   ARMI_VCVT_F32_U32 = 0xeeb80a40,
249   ARMI_VCVT_F64_U32 = 0xeeb80b40,
250   ARMI_VCVT_F32_F64 = 0xeeb70bc0,
251   ARMI_VCVT_F64_F32 = 0xeeb70ac0,
252
253   ARMI_VLDR_S = 0xed100a00,
254   ARMI_VLDR_D = 0xed100b00,
255   ARMI_VSTR_S = 0xed000a00,
256   ARMI_VSTR_D = 0xed000b00,
257 } ARMIns;
258
259 typedef enum ARMShift {
260   ARMSH_LSL, ARMSH_LSR, ARMSH_ASR, ARMSH_ROR
261 } ARMShift;
262
263 /* ARM condition codes. */
264 typedef enum ARMCC {
265   CC_EQ, CC_NE, CC_CS, CC_CC, CC_MI, CC_PL, CC_VS, CC_VC,
266   CC_HI, CC_LS, CC_GE, CC_LT, CC_GT, CC_LE, CC_AL,
267   CC_HS = CC_CS, CC_LO = CC_CC
268 } ARMCC;
269
270 #endif
```

One Level Up Top Level

src/lj_target_x86.h - luajit-2.0-src

Data types defined

- ExitState
- x86Arith
- x86CC
- x86Group
- x86Group3
- x86Group5
- x86ModRM
- x86Mode
- x86Op
- x86Shift

Macros defined

- EXITSTUBS_PER_GROUP
- EXITSTUB_SPACING
- FPRDEF
- FPRDEF
- GPRDEF
- GPRDEF
- REGARG_FIRSTFPR
- REGARG_FIRSTFPR
- REGARG_GPRS
- REGARG_GPRS
- REGARG_GPRS
- REGARG_LASTFPR
- REGARG_LASTFPR
- REGARG_NUMFPR
- REGARG_NUMFPR
- REGARG_NUMFPR
- REGARG_NUMGPR
- REGARG_NUMGPR

- REGARG_NUMGPR
- RIDENUM
- RSET_ACD
- RSET_ALL
- RSET_FPR
- RSET_GPR
- RSET_GPR8
- RSET_GPR8
- RSET_INIT
- RSET_SCRATCH
- RSET_SCRATCH
- RSET_SCRATCH
- SPOFS_TMP
- SPS_FIRST
- SPS_FIRST
- SPS_FIRST
- SPS_FIXED
- SPS_FIXED
- SPS_FIXED
- STACKARG_OFS
- STACKARG_OFS
- STACKARG_OFS
- VRIDDEF
- XG_
- XG_ARITHi
- XG_TOXOi
- XG_TOXOi8
- XO_
- XO_0f
- XO_66
- XO_660f
- XO_ARITH
- XO_ARITHw

- XO_FPU
- XO_f20f
- XO_f30f
- _LJ_TARGET_X86_H
- rset_picktop
- rset_picktop
- sps_align
- sps_scale

Source code

```
/*
** Definitions for x86 and x64 CPUs.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_TARGET_X86_H
#define _LJ_TARGET_X86_H

/* -- Registers IDs ------------------------------------------------------- */

#if LJ_64
#define GPRDEF(_) \
  _(EAX) _(ECX) _(EDX) _(EBX) _(ESP) _(EBP) _(ESI) _(EDI) \
  _(R8D) _(R9D) _(R10D) _(R11D) _(R12D) _(R13D) _(R14D) _(R15D)
#define FPRDEF(_) \
  _(XMM0) _(XMM1) _(XMM2) _(XMM3) _(XMM4) _(XMM5) _(XMM6) _(XMM7) \
  _(XMM8) _(XMM9) _(XMM10) _(XMM11) _(XMM12) _(XMM13) _(XMM14) _(XMM15)
#else
#define GPRDEF(_) \
  _(EAX) _(ECX) _(EDX) _(EBX) _(ESP) _(EBP) _(ESI) _(EDI)
#define FPRDEF(_) \
  _(XMM0) _(XMM1) _(XMM2) _(XMM3) _(XMM4) _(XMM5) _(XMM6) _(XMM7)
#endif
#define VRIDDEF(_) \
  _(MRM)

#define RIDENUM(name)        RID_##name,

enum {
  GPRDEF(RIDENUM)               /* General-purpose registers (GPRs). */
  FPRDEF(RIDENUM)               /* Floating-point registers (FPRs). */
  RID_MAX,
  RID_MRM = RID_MAX,               /* Pseudo-id for ModRM operand. */

  /* Calling conventions. */
  RID_SP = RID_ESP,
  RID_RET = RID_EAX,
#if LJ_64
  RID_FPRET = RID_XMM0,
#else
  RID_RETLO = RID_EAX,
  RID_RETHI = RID_EDX,
#endif

  /* These definitions must match with the *.dasc file(s): */
  RID_BASE = RID_EDX,                /* Interpreter BASE. */
#if LJ_64 && !LJ_ABI_WIN
  RID_LPC = RID_EBX,                /* Interpreter PC. */
  RID_DISPATCH = RID_R14D,        /* Interpreter DISPATCH table. */
#else
  RID_LPC = RID_ESI,                /* Interpreter PC. */
  RID_DISPATCH = RID_EBX,        /* Interpreter DISPATCH table. */
#endif

```

```
  /* Register ranges [min, max) and number of registers. */
  RID_MIN_GPR = RID_EAX,
  RID_MIN_FPR = RID_XMM0,
  RID_MAX_GPR = RID_MIN_FPR,
  RID_MAX_FPR = RID_MAX,
  RID_NUM_GPR = RID_MAX_GPR - RID_MIN_GPR,
  RID_NUM_FPR = RID_MAX_FPR - RID_MIN_FPR,
};

/* -- Register sets ------------------------------------------------------ */

/* Make use of all registers, except the stack pointer. */
#define RSET_GPR        (RSET_RANGE(RID_MIN_GPR, RID_MAX_GPR)-RID2RSET(RID_ESP))
#define RSET_FPR        (RSET_RANGE(RID_MIN_FPR, RID_MAX_FPR))
#define RSET_ALL        (RSET_GPR|RSET_FPR)
#define RSET_INIT        RSET_ALL

#if LJ_64
/* Note: this requires the use of FORCE_REX! */
#define RSET_GPR8        RSET_GPR
#else
#define RSET_GPR8        (RSET_RANGE(RID_EAX, RID_EBX+1))
#endif

/* ABI-specific register sets. */
#define RSET_ACD        (RID2RSET(RID_EAX)|RID2RSET(RID_ECX)|RID2RSET(RID_EDX))
#if LJ_64
#if LJ_ABI_WIN
/* Windows x64 ABI. */
#define RSET_SCRATCH \
  (RSET_ACD|RSET_RANGE(RID_R8D, RID_R11D+1)|RSET_RANGE(RID_XMM0, RID_XMM5+1))
#define REGARG_GPRS \
  (RID_ECX|((RID_EDX|((RID_R8D|(RID_R9D<<5))<<5))<<5))
#define REGARG_NUMGPR        4
#define REGARG_NUMFPR        4
#define REGARG_FIRSTFPR        RID_XMM0
#define REGARG_LASTFPR        RID_XMM3
#define STACKARG_OFS        (4*8)
#else
/* The rest of the civilized x64 world has a common ABI. */
#define RSET_SCRATCH \
  (RSET_ACD|RSET_RANGE(RID_ESI, RID_R11D+1)|RSET_FPR)
#define REGARG_GPRS \
  (RID_EDI|((RID_ESI|((RID_EDX|((RID_ECX|((RID_R8D|(RID_R9D \
   <<5))<<5))<<5))<<5))<<5))
#define REGARG_NUMGPR        6
#define REGARG_NUMFPR        8
#define REGARG_FIRSTFPR        RID_XMM0
#define REGARG_LASTFPR        RID_XMM7
#define STACKARG_OFS        0
#endif
#else
/* Common x86 ABI. */
#define RSET_SCRATCH        (RSET_ACD|RSET_FPR)
#define REGARG_GPRS        (RID_ECX|(RID_EDX<<5))  /* Fastcall only. */
#define REGARG_NUMGPR        2  /* Fastcall only. */
#define REGARG_NUMFPR        0
#define STACKARG_OFS        0
#endif

#if LJ_64
/* Prefer the low 8 regs of each type to reduce REX prefixes. */
#undef rset_picktop
#define rset_picktop(rs)        (lj_fls(lj_bswap(rs)) ^ 0x18)
#endif

/* -- Spill slots -------------------------------------------------------- */

/* Spill slots are 32 bit wide. An even/odd pair is used for FPRs.
**
** SPS_FIXED: Available fixed spill slots in interpreter frame.
** This definition must match with the *.dasc file(s).
**
** SPS_FIRST: First spill slot for general use. Reserve min. two 32 bit slots.
*/
#if LJ_64
```

```
#if LJ_ABI_WIN
#define SPS_FIXED        (4*2)
#define SPS_FIRST        (4*2)        /* Don't use callee register save area. */
#else
#define SPS_FIXED        4
#define SPS_FIRST        2
#endif
#else
#define SPS_FIXED        6
#define SPS_FIRST        2
#endif

#define SPOFS_TMP        0

#define sps_scale(slot)               (4 * (int32_t)(slot))
#define sps_align(slot)               (((slot) - SPS_FIXED + 3) & ~3)

/* -- Exit state ---------------------------------------------------------- */

/* This definition must match with the *.dasc file(s). */
typedef struct {
  lua_Number fpr[RID_NUM_FPR];        /* Floating-point registers. */
  intptr_t gpr[RID_NUM_GPR];        /* General-purpose registers. */
  int32_t spill[256];                /* Spill slots. */
} ExitState;

/* Limited by the range of a short fwd jump (127): (2+2)*(32-1)-2 = 122. */
#define EXITSTUB_SPACING        (2+2)
#define EXITSTUBS_PER_GROUP        32

/* -- x86 ModRM operand encoding ------------------------------------------ */

typedef enum {
  XM_OFS0 = 0x00, XM_OFS8 = 0x40, XM_OFS32 = 0x80, XM_REG = 0xc0,
  XM_SCALE1 = 0x00, XM_SCALE2 = 0x40, XM_SCALE4 = 0x80, XM_SCALE8 = 0xc0,
  XM_MASK = 0xc0
} x86Mode;

/* Structure to hold variable ModRM operand. */
typedef struct {
  int32_t ofs;                /* Offset. */
  uint8_t base;                /* Base register or RID_NONE. */
  uint8_t idx;                /* Index register or RID_NONE. */
  uint8_t scale;        /* Index scale (XM_SCALE1 .. XM_SCALE8). */
} x86ModRM;

/* -- Opcodes ------------------------------------------------------------- */

/* Macros to construct variable-length x86 opcodes. -(len+1) is in LSB. */
#define XO_(o)                ((uint32_t)(0x0000fe + (0x##o<<24)))
#define XO_FPU(a,b)        ((uint32_t)(0x00fd + (0x##a<<16)+(0x##b<<24)))
#define XO_0f(o)        ((uint32_t)(0x0f00fd + (0x##o<<24)))
#define XO_66(o)        ((uint32_t)(0x6600fd + (0x##o<<24)))
#define XO_660f(o)        ((uint32_t)(0x0f66fc + (0x##o<<24)))
#define XO_f20f(o)        ((uint32_t)(0x0ff2fc + (0x##o<<24)))
#define XO_f30f(o)        ((uint32_t)(0x0ff3fc + (0x##o<<24)))

/* This list of x86 opcodes is not intended to be complete. Opcodes are only
** included when needed. Take a look at DynASM or jit.dis_x86 to see the
** whole mess.
*/
typedef enum {
  /* Fixed length opcodes. XI_* prefix. */
  XI_NOP =        0x90,
  XI_XCHGa =        0x90,
  XI_CALL =        0xe8,
  XI_JMP =        0xe9,
  XI_JMPs =        0xeb,
  XI_PUSH =        0x50, /* Really 50+r. */
  XI_JCCs =        0x70, /* Really 7x. */
  XI_JCCn =        0x80, /* Really 0f8x. */
  XI_LEA =        0x8d,
  XI_MOVrib =        0xb0, /* Really b0+r. */
  XI_MOVri =        0xb8, /* Really b8+r. */
  XI_ARITHib =        0x80,
  XI_ARITHi =        0x81,
```

```
  XI_ARITHi8 =        0x83,
  XI_PUSHi8 =        0x6a,
  XI_TESTb =        0x84,
  XI_TEST =        0x85,
  XI_MOVmi =        0xc7,
  XI_GROUP5 =        0xff,

  /* Note: little-endian byte-order! */
  XI_FLDZ =        0xeed9,
  XI_FLD1 =        0xe8d9,
  XI_FLDLG2 =        0xecd9,
  XI_FLDLN2 =        0xedd9,
  XI_FDUP =        0xc0d9,  /* Really fld st0. */
  XI_FPOP =        0xd8dd,  /* Really fstp st0. */
  XI_FPOP1 =        0xd9dd,  /* Really fstp st1. */
  XI_FRNDINT =        0xfcd9,
  XI_FSIN =        0xfed9,
  XI_FCOS =        0xffd9,
  XI_FPTAN =        0xf2d9,
  XI_FPATAN =        0xf3d9,
  XI_FSCALE =        0xfdd9,
  XI_FYL2X =        0xf1d9,

  /* Variable-length opcodes. XO_* prefix. */
  XO_MOV =        XO_(8b),
  XO_MOVto =        XO_(89),
  XO_MOVtow =        XO_66(89),
  XO_MOVtob =        XO_(88),
  XO_MOVmi =        XO_(c7),
  XO_MOVmib =        XO_(c6),
  XO_LEA =        XO_(8d),
  XO_ARITHib =        XO_(80),
  XO_ARITHi =        XO_(81),
  XO_ARITHi8 =        XO_(83),
  XO_ARITHiw8 =        XO_66(83),
  XO_SHIFTi =        XO_(c1),
  XO_SHIFT1 =        XO_(d1),
  XO_SHIFTcl =        XO_(d3),
  XO_IMUL =        XO_0f(af),
  XO_IMULi =        XO_(69),
  XO_IMULi8 =        XO_(6b),
  XO_CMP =        XO_(3b),
  XO_TESTb =        XO_(84),
  XO_TEST =        XO_(85),
  XO_GROUP3b =        XO_(f6),
  XO_GROUP3 =        XO_(f7),
  XO_GROUP5b =        XO_(fe),
  XO_GROUP5 =        XO_(ff),
  XO_MOVZXb =        XO_0f(b6),
  XO_MOVZXw =        XO_0f(b7),
  XO_MOVSXb =        XO_0f(be),
  XO_MOVSXw =        XO_0f(bf),
  XO_MOVSXd =        XO_(63),
  XO_BSWAP =        XO_0f(c8),
  XO_CMOV =        XO_0f(40),

  XO_MOVSD =        XO_f20f(10),
  XO_MOVSDto =        XO_f20f(11),
  XO_MOVSS =        XO_f30f(10),
  XO_MOVSSto =        XO_f30f(11),
  XO_MOVLPD =        XO_660f(12),
  XO_MOVAPS =        XO_0f(28),
  XO_XORPS =        XO_0f(57),
  XO_ANDPS =        XO_0f(54),
  XO_ADDSD =        XO_f20f(58),
  XO_SUBSD =        XO_f20f(5c),
  XO_MULSD =        XO_f20f(59),
  XO_DIVSD =        XO_f20f(5e),
  XO_SQRTSD =        XO_f20f(51),
  XO_MINSD =        XO_f20f(5d),
  XO_MAXSD =        XO_f20f(5f),
  XO_ROUNDSD =        0x0b3a0ffc,  /* Really 66 0f 3a 0b. See asm_fpmath. */
  XO_UCOMISD =        XO_660f(2e),
  XO_CVTSI2SD =        XO_f20f(2a),
  XO_CVTTSD2SI=        XO_f20f(2c),
  XO_CVTSI2SS =        XO_f30f(2a),
```

```
  XO_CVTTSS2SI=        XO_f30f(2c),
  XO_CVTSS2SD =        XO_f30f(5a),
  XO_CVTSD2SS =        XO_f20f(5a),
  XO_ADDSS =         XO_f30f(58),
  XO_MOVD =        XO_660f(6e),
  XO_MOVDto =        XO_660f(7e),

  XO_FLDd =        XO_(d9), XOg_FLDd = 0,
  XO_FLDq =        XO_(dd), XOg_FLDq = 0,
  XO_FILDd =        XO_(db), XOg_FILDd = 0,
  XO_FILDq =        XO_(df), XOg_FILDq = 5,
  XO_FSTPd =        XO_(d9), XOg_FSTPd = 3,
  XO_FSTPq =        XO_(dd), XOg_FSTPq = 3,
  XO_FISTPq =        XO_(df), XOg_FISTPq = 7,
  XO_FISTTPq =        XO_(dd), XOg_FISTTPq = 1,
  XO_FADDq =        XO_(dc), XOg_FADDq = 0,
  XO_FLDCW =        XO_(d9), XOg_FLDCW = 5,
  XO_FNSTCW =        XO_(d9), XOg_FNSTCW = 7
} x86Op;

/* x86 opcode groups. */
typedef uint32_t x86Group;

#define XG_(i8, i, g)        ((x86Group)(((i8) << 16) + ((i) << 8) + (g)))
#define XG_ARITHi(g)        XG_(XI_ARITHi8, XI_ARITHi, g)
#define XG_TOXOi(xg)        ((x86Op)(0x000000fe + (((xg)<<16) & 0xff000000)))
#define XG_TOXOi8(xg)        ((x86Op)(0x000000fe + (((xg)<<8) & 0xff000000)))

#define XO_ARITH(a)        ((x86Op)(0x030000fe + ((a)<<27)))
#define XO_ARITHw(a)        ((x86Op)(0x036600fd + ((a)<<27)))

typedef enum {
  XOg_ADD, XOg_OR, XOg_ADC, XOg_SBB, XOg_AND, XOg_SUB, XOg_XOR, XOg_CMP,
  XOg_X_IMUL
} x86Arith;

typedef enum {
  XOg_ROL, XOg_ROR, XOg_RCL, XOg_RCR, XOg_SHL, XOg_SHR, XOg_SAL, XOg_SAR
} x86Shift;

typedef enum {
  XOg_TEST, XOg_TEST_, XOg_NOT, XOg_NEG, XOg_MUL, XOg_IMUL, XOg_DIV, XOg_IDIV
} x86Group3;

typedef enum {
  XOg_INC, XOg_DEC, XOg_CALL, XOg_CALLfar, XOg_JMP, XOg_JMPfar, XOg_PUSH
} x86Group5;

/* x86 condition codes. */
typedef enum {
  CC_O, CC_NO, CC_B, CC_NB, CC_E, CC_NE, CC_BE, CC_NBE,
  CC_S, CC_NS, CC_P, CC_NP, CC_L, CC_NL, CC_LE, CC_NLE,
  CC_C = CC_B, CC_NAE = CC_C, CC_NC = CC_NB, CC_AE = CC_NB,
  CC_Z = CC_E, CC_NZ = CC_NE, CC_NA = CC_BE, CC_A = CC_NBE,
  CC_PE = CC_P, CC_PO = CC_NP, CC_NGE = CC_L, CC_GE = CC_NL,
  CC_NG = CC_LE, CC_G = CC_NLE
} x86CC;

#endif
```

One Level Up Top Level

src/lj_target.h - luajit-2.0-src

Data types defined

- Reg
- RegCost
- RegSP
- RegSet
- RegSet

Functions defined

- exitstub_addr_

Macros defined

- REGCOST
- REGCOST_PHI_WEIGHT
- REGCOST_REF_T
- REGCOST_T
- REGSP
- REGSP_HINT
- REGSP_INIT
- RID2RSET
- RID_INIT
- RID_MASK
- RID_NONE
- RID_SINK
- RID_SUNK
- RSET_EMPTY
- RSET_RANGE
- SPS_NONE
- _LJ_TARGET_H
- exitstub_addr
- ra_gethint
- ra_hashint
- ra_hasreg

- ra_hasspill
- ra_noreg
- ra_samehint
- ra_sethint
- regcost_ref
- regsp_reg
- regsp_spill
- regsp_used
- rset_clear
- rset_exclude
- rset_pickbot
- rset_pickbot
- rset_picktop
- rset_picktop
- rset_set
- rset_test

Source code

```
/*
** Definitions for target CPU.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_TARGET_H
#define _LJ_TARGET_H

#include "lj_def.h"
 #include "lj_arch.h"

 /* -- Registers and spill slots ------------------------------------------- */

 /* Register type (uint8_t in ir->r). */
 typedef uint32_t Reg;

 /* The hi-bit is NOT set for an allocated register. This means the value
 ** can be directly used without masking. The hi-bit is set for a register
 ** allocation hint or for RID_INIT, RID_SINK or RID_SUNK.
 */
 #define RID_NONE               0x80
 #define RID_MASK               0x7f
 #define RID_INIT               (RID_NONE|RID_MASK)
 #define RID_SINK               (RID_INIT-1)
 #define RID_SUNK               (RID_INIT-2)

 #define ra_noreg(r)               ((r) & RID_NONE)
 #define ra_hasreg(r)               (!((r) & RID_NONE))

 /* The ra_hashint() macro assumes a previous test for ra_noreg(). */
 #define ra_hashint(r)               ((r) < RID_SUNK)
 #define ra_gethint(r)               ((Reg)((r) & RID_MASK))
 #define ra_sethint(rr, r)        rr = (uint8_t)((r)|RID_NONE)
 #define ra_samehint(r1, r2)        (ra_gethint((r1)^(r2)) == 0)

 /* Spill slot 0 means no spill slot has been allocated. */
```

```
#define SPS_NONE		0

#define ra_hasspill(s)		((s) != SPS_NONE)

/* Combined register and spill slot (uint16_t in ir->prev). */
typedef uint32_t RegSP;

#define REGSP(r, s)		((r) + ((s) << 8))
#define REGSP_HINT(r)		((r)|RID_NONE)
#define REGSP_INIT		REGSP(RID_INIT, 0)

#define regsp_reg(rs)		((rs) & 255)
#define regsp_spill(rs)		((rs) >> 8)
#define regsp_used(rs) \
  (((rs) & ~REGSP(RID_MASK, 0)) != REGSP(RID_NONE, 0))

/* -- Register sets ------------------------------------------------------ */

/* Bitset for registers. 32 registers suffice for most architectures.
** Note that one set holds bits for both GPRs and FPRs.
*/
#if LJ_TARGET_PPC || LJ_TARGET_MIPS
typedef uint64_t RegSet;
#else
typedef uint32_t RegSet;
#endif

#define RID2RSET(r)		(((RegSet)1) << (r))
#define RSET_EMPTY		((RegSet)0)
#define RSET_RANGE(lo, hi)	((RID2RSET((hi)-(lo))-1) << (lo))

#define rset_test(rs, r)	((int)((rs) >> (r)) & 1)
#define rset_set(rs, r)		(rs |= RID2RSET(r))
#define rset_clear(rs, r)	(rs &= ~RID2RSET(r))
#define rset_exclude(rs, r)	(rs & ~RID2RSET(r))
#if LJ_TARGET_PPC || LJ_TARGET_MIPS
#define rset_picktop(rs)	((Reg)(__builtin_clzll(rs)^63))
#define rset_pickbot(rs)	((Reg)__builtin_ctzll(rs))
#else
#define rset_picktop(rs)	((Reg)lj_fls(rs))
#define rset_pickbot(rs)	((Reg)lj_ffs(rs))
#endif

/* -- Register allocation cost ------------------------------------------- */

/* The register allocation heuristic keeps track of the cost for allocating
** a specific register:
**
** A free register (obviously) has a cost of 0 and a 1-bit in the free mask.
**
** An already allocated register has the (non-zero) IR reference in the lowest
** bits and the result of a blended cost-model in the higher bits.
**
** The allocator first checks the free mask for a hit. Otherwise an (unrolled)
** linear search for the minimum cost is used. The search doesn't need to
** keep track of the position of the minimum, which makes it very fast.
** The lowest bits of the minimum cost show the desired IR reference whose
** register is the one to evict.
**
** Without the cost-model this degenerates to the standard heuristics for
** (reverse) linear-scan register allocation. Since code generation is done
** in reverse, a live interval extends from the last use to the first def.
** For an SSA IR the IR reference is the first (and only) def and thus
** trivially marks the end of the interval. The LSRA heuristics says to pick
** the register whose live interval has the furthest extent, i.e. the lowest
** IR reference in our case.
**
** A cost-model should take into account other factors, like spill-cost and
** restore- or rematerialization-cost, which depend on the kind of instruction.
** E.g. constants have zero spill costs, variant instructions have higher
** costs than invariants and PHIs should preferably never be spilled.
**
** Here's a first cut at simple, but effective blended cost-model for R-LSRA:
** - Due to careful design of the IR, constants already have lower IR
**   references than invariants and invariants have lower IR references
**   than variants.
```

```
** - The cost in the upper 16 bits is the sum of the IR reference and a
**   weighted score. The score currently only takes into account whether
**   the IRT_ISPHI bit is set in the instruction type.
** - The PHI weight is the minimum distance (in IR instructions) a PHI
**   reference has to be further apart from a non-PHI reference to be spilled.
** - It should be a power of two (for speed) and must be between 2 and 32768.
**   Good values for the PHI weight seem to be between 40 and 150.
** - Further study is required.
*/
#define REGCOST_PHI_WEIGHT        64

/* Cost for allocating a specific register. */
typedef uint32_t RegCost;

/* Note: assumes 16 bit IRRef1. */
#define REGCOST(cost, ref)        ((RegCost)(ref) + ((RegCost)(cost) << 16))
#define regcost_ref(rc)               ((IRRef1)(rc))

#define REGCOST_T(t) \
  ((RegCost)((t)&IRT_ISPHI) * (((RegCost)(REGCOST_PHI_WEIGHT)<<16)/IRT_ISPHI))
#define REGCOST_REF_T(ref, t)        (REGCOST((ref), (ref)) + REGCOST_T((t)))

/* -- Target-specific definitions ---------------------------------------- */

#if LJ_TARGET_X86ORX64
#include "lj_target_x86.h"
#elif LJ_TARGET_ARM
#include "lj_target_arm.h"
#elif LJ_TARGET_ARM64
#include "lj_target_arm64.h"
#elif LJ_TARGET_PPC
#include "lj_target_ppc.h"
#elif LJ_TARGET_MIPS
#include "lj_target_mips.h"
#else
#error "Missing include for target CPU"
#endif

#ifdef EXITSTUBS_PER_GROUP
/* Return the address of an exit stub. */
static LJ_AINLINE char *exitstub_addr_(char **group, uint32_t exitno)
{
  lua_assert(group[exitno / EXITSTUBS_PER_GROUP] != NULL);
  return (char *)group[exitno / EXITSTUBS_PER_GROUP] +
         EXITSTUB_SPACING*(exitno % EXITSTUBS_PER_GROUP);
}
/* Avoid dependence on lj_jit.h if only including lj_target.h. */
#define exitstub_addr(J, exitno) \
  ((MCode *)exitstub_addr_((char **)((J)->exitstubgroup), (exitno)))
#endif

#endif
```

One Level Up Top Level

src/lj_asm.c - luajit-2.0-src

Global variables defined

- field_ofs
- ra_dbg_buf
- ra_dbg_mcp
- ra_dbg_merge
- ra_dbg_p
- ra_regname

Data types defined

- ASMState
- ASMState

Functions defined

- asm_baseslot
- asm_bufhdr
- asm_bufput
- asm_bufstr
- asm_call
- asm_callid
- asm_callx_flags
- asm_collectargs
- asm_conv64
- asm_fpjoin_pow
- asm_fppow
- asm_gcstep
- asm_head_root
- asm_head_side
- asm_ir
- asm_loop
- asm_lref
- asm_mclimit
- asm_newref

- asm_phi
- asm_phi_break
- asm_phi_copyspill
- asm_phi_fixup
- asm_phi_shuffle
- asm_setup_regsp
- asm_snap_alloc
- asm_snap_alloc1
- asm_snap_canremat
- asm_snap_checkrename
- asm_snap_prep
- asm_snew
- asm_stack_adjust
- asm_sunk_store
- asm_tail_link
- asm_tdup
- asm_tnew
- asm_tostr
- checkmclim
- ir_khash
- lj_asm_trace
- ra_alloc1
- ra_allock
- ra_allockreg
- ra_allocref
- ra_dest
- ra_destpair
- ra_destreg
- ra_dflush
- ra_dprintf
- ra_dstart
- ra_evict
- ra_evictk

- ra_evictset
- ra_left
- ra_leftov
- ra_pick
- ra_releasetmp
- ra_rematk
- ra_rename
- ra_restore
- ra_save
- ra_scratch
- ra_setkref
- ra_setup
- ra_spill

Macros defined

- ASMREF_L
- ASMREF_TMP1
- ASMREF_TMP2
- FLOFS
- FLOFS
- FUSE_DISABLED
- IR
- IR
- LUA_CORE
- MCLIM_REDZONE
- MINCOST
- RA_DBGX
- RA_DBGX
- RA_DBG_FLUSH
- RA_DBG_FLUSH
- RA_DBG_REF
- RA_DBG_REF
- RA_DBG_START
- RA_DBG_START

- RIDNAME
- RIDNAME
- canfuse
- emit_sload
- emit_spstore
- iscrossref
- lj_asm_c
- mayfuse
- neverfuse
- opisfusableload
- ra_allockreg
- ra_free
- ra_iskref
- ra_iskref
- ra_krefk
- ra_krefk
- ra_krefreg
- ra_krefreg
- ra_modified
- ra_noweak
- ra_used
- ra_weak

Source code

```
/*
** IR assembler (SSA IR -> machine code).
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_asm_c
#define LUA_CORE

#include "lj_obj.h"

 #if LJ_HASJIT

 #include "lj_gc.h"
 #include "lj_str.h"
 #include "lj_tab.h"
 #include "lj_frame.h"
 #if LJ_HASFFI
 #include "lj_ctype.h"
 #endif
 #include "lj_ir.h"
 #include "lj_jit.h"
 #include "lj_ircall.h"
```

```
#include "lj_iropt.h"
#include "lj_mcode.h"
#include "lj_iropt.h"
#include "lj_trace.h"
#include "lj_snap.h"
#include "lj_asm.h"
#include "lj_dispatch.h"
#include "lj_vm.h"
#include "lj_target.h"

#ifdef LUA_USE_ASSERT
#include <stdio.h>
#endif

/* -- Assembler state and common macros ----------------------------------- */

/* Assembler state. */
typedef struct ASMState {
  RegCost cost[RID_MAX];  /* Reference and blended allocation cost for regs. */

  MCode *mcp;               /* Current MCode pointer (grows down). */
  MCode *mclim;               /* Lower limit for MCode memory + red zone. */
#ifdef LUA_USE_ASSERT
  MCode *mcp_prev;        /* Red zone overflow check. */
#endif

  IRIns *ir;               /* Copy of pointer to IR instructions/constants. */
  jit_State *J;               /* JIT compiler state. */

#if LJ_TARGET_X86ORX64
  x86ModRM mrm;               /* Fused x86 address operand. */
#endif

  RegSet freeset;        /* Set of free registers. */
  RegSet modset;        /* Set of registers modified inside the loop. */
  RegSet weakset;        /* Set of weakly referenced registers. */
  RegSet phiset;        /* Set of PHI registers. */

  uint32_t flags;        /* Copy of JIT compiler flags. */
  int loopinv;               /* Loop branch inversion (0:no, 1:yes, 2:yes+CC_P). */

  int32_t evenspill;        /* Next even spill slot. */
  int32_t oddspill;        /* Next odd spill slot (or 0). */

  IRRef curins;               /* Reference of current instruction. */
  IRRef stopins;        /* Stop assembly before hitting this instruction. */
  IRRef orignins;        /* Original T->nins. */

  IRRef snapref;        /* Current snapshot is active after this reference. */
  IRRef snaprename;        /* Rename highwater mark for snapshot check. */
  SnapNo snapno;        /* Current snapshot number. */
  SnapNo loopsnapno;        /* Loop snapshot number. */

  IRRef fuseref;        /* Fusion limit (loopref, 0 or FUSE_DISABLED). */
  IRRef sectref;        /* Section base reference (loopref or 0). */
  IRRef loopref;        /* Reference of LOOP instruction (or 0). */

  BCReg topslot;        /* Number of slots for stack check (unless 0). */
  int32_t gcsteps;        /* Accumulated number of GC steps (per section). */

  GCtrace *T;               /* Trace to assemble. */
  GCtrace *parent;        /* Parent trace (or NULL). */

  MCode *mcbot;               /* Bottom of reserved MCode. */
  MCode *mctop;               /* Top of generated MCode. */
  MCode *mcloop;        /* Pointer to loop MCode (or NULL). */
  MCode *invmcp;        /* Points to invertible loop branch (or NULL). */
  MCode *flagmcp;        /* Pending opportunity to merge flag setting ins. */
  MCode *realign;        /* Realign loop if not NULL. */

#ifdef RID_NUM_KREF
  int32_t krefk[RID_NUM_KREF];
#endif
  IRRef1 phireg[RID_MAX];  /* PHI register references. */
  uint16_t parentmap[LJ_MAX_JSLOTS];  /* Parent instruction to RegSP map. */
} ASMState;
```

```
99
100 #define IR(ref)                         (&as->ir[(ref)])
101
102 #define ASMREF_TMP1                REF_TRUE        /* Temp. register. */
103 #define ASMREF_TMP2                REF_FALSE        /* Temp. register. */
104 #define ASMREF_L                REF_NIL                /* Stores register for L. */
105
106 /* Check for variant to invariant references. */
107 #define iscrossref(as, ref)        ((ref) < as->sectref)
108
109 /* Inhibit memory op fusion from variant to invariant references. */
110 #define FUSE_DISABLED                (~(IRRef)0)
111 #define mayfuse(as, ref)        ((ref) > as->fuseref)
112 #define neverfuse(as)                (as->fuseref == FUSE_DISABLED)
113 #define canfuse(as, ir)                (!neverfuse(as) && !irt_isphi((ir)->t))
114 #define opisfusableload(o) \
115   ((o) == IR_ALOAD || (o) == IR_HLOAD || (o) == IR_ULOAD || \
116    (o) == IR_FLOAD || (o) == IR_XLOAD || (o) == IR_SLOAD || (o) == IR_VLOAD)
117
118 /* Sparse limit checks using a red zone before the actual limit. */
119 #define MCLIM_REDZONE        64
120
121 static LJ_NORET LJ_NOINLINE void asm_mclimit(ASMState *as)
122 {
123   lj_mcode_limiterr(as->J, (size_t)(as->mctop - as->mcp + 4*MCLIM_REDZONE));
124 }
125
126 static LJ_AINLINE void checkmclim(ASMState *as)
127 {
128 #ifdef LUA_USE_ASSERT
129   if (as->mcp + MCLIM_REDZONE < as->mcp_prev) {
130     IRIns *ir = IR(as->curins+1);
131     fprintf(stderr, "RED ZONE OVERFLOW: %p IR %04d  %02d %04d %04d\n", as->mcp,
132             as->curins+1-REF_BIAS, ir->o, ir->op1-REF_BIAS, ir->op2-REF_BIAS);
133     lua_assert(0);
134   }
135 #endif
136   if (LJ_UNLIKELY(as->mcp < as->mclim)) asm_mclimit(as);
137 #ifdef LUA_USE_ASSERT
138   as->mcp_prev = as->mcp;
139 #endif
140 }
141
142 #ifdef RID_NUM_KREF
143 #define ra_iskref(ref)                ((ref) < RID_NUM_KREF)
144 #define ra_krefreg(ref)                ((Reg)(RID_MIN_KREF + (Reg)(ref)))
145 #define ra_krefk(as, ref)        (as->krefk[(ref)])
146
147 static LJ_AINLINE void ra_setkref(ASMState *as, Reg r, int32_t k)
148 {
149   IRRef ref = (IRRef)(r - RID_MIN_KREF);
150   as->krefk[ref] = k;
151   as->cost[r] = REGCOST(ref, ref);
152 }
153
154 #else
155 #define ra_iskref(ref)                0
156 #define ra_krefreg(ref)                RID_MIN_GPR
157 #define ra_krefk(as, ref)        0
158 #endif
159
160 /* Arch-specific field offsets. */
161 static const uint8_t field_ofs[IRFL__MAX+1] = {
162 #define FLOFS(name, ofs)        (uint8_t)(ofs),
163 IRFLDEF(FLOFS)
164 #undef FLOFS
165   0
166 };
167
168 /* -- Target-specific instruction emitter ---------------------------------- */
169
170 #if LJ_TARGET_X86ORX64
171 #include "lj_emit_x86.h"
172 #elif LJ_TARGET_ARM
173 #include "lj_emit_arm.h"
174 #elif LJ_TARGET_PPC
```

```
#include "lj_emit_ppc.h"
#elif LJ_TARGET_MIPS
#include "lj_emit_mips.h"
#else
#error "Missing instruction emitter for target CPU"
#endif

/* Generic load/store of register from/to stack slot. */
#define emit_spload(as, ir, r, ofs) \
  emit_loadofs(as, ir, (r), RID_SP, (ofs))
#define emit_spstore(as, ir, r, ofs) \
  emit_storeofs(as, ir, (r), RID_SP, (ofs))

/* -- Register allocator debugging ---------------------------------------- */

/* #define LUAJIT_DEBUG_RA */

#ifdef LUAJIT_DEBUG_RA

#include <stdio.h>
#include <stdarg.h>

#define RIDNAME(name)        #name,
static const char *const ra_regname[] = {
  GPRDEF(RIDNAME)
  FPRDEF(RIDNAME)
  VRIDDEF(RIDNAME)
  NULL
};
#undef RIDNAME

static char ra_dbg_buf[65536];
static char *ra_dbg_p;
static char *ra_dbg_merge;
static MCode *ra_dbg_mcp;

static void ra_dstart(void)
{
  ra_dbg_p = ra_dbg_buf;
  ra_dbg_merge = NULL;
  ra_dbg_mcp = NULL;
}

static void ra_dflush(void)
{
  fwrite(ra_dbg_buf, 1, (size_t)(ra_dbg_p-ra_dbg_buf), stdout);
  ra_dstart();
}

static void ra_dprintf(ASMState *as, const char *fmt, ...)
{
  char *p;
  va_list argp;
  va_start(argp, fmt);
  p = ra_dbg_mcp == as->mcp ? ra_dbg_merge : ra_dbg_p;
  ra_dbg_mcp = NULL;
  p += sprintf(p, "%08x  \e[36m%04d ", (uintptr_t)as->mcp, as->curins-REF_BIAS);
  for (;;) {
    const char *e = strchr(fmt, '$');
    if (e == NULL) break;
    memcpy(p, fmt, (size_t)(e-fmt));
    p += e-fmt;
    if (e[1] == 'r') {
      Reg r = va_arg(argp, Reg) & RID_MASK;
      if (r <= RID_MAX) {
        const char *q;
        for (q = ra_regname[r]; *q; q++)
          *p++ = *q >= 'A' && *q <= 'Z' ? *q + 0x20 : *q;
      } else {
        *p++ = '?';
        lua_assert(0);
      }
    } else if (e[1] == 'f' || e[1] == 'i') {
      IRRef ref;
      if (e[1] == 'f')
        ref = va_arg(argp, IRRef);
```

```
      else
        ref = va_arg(argp, IRIns *) - as->ir;
      if (ref >= REF_BIAS)
        p += sprintf(p, "%04d", ref - REF_BIAS);
      else
        p += sprintf(p, "K%03d", REF_BIAS - ref);
    } else if (e[1] == 's') {
      uint32_t slot = va_arg(argp, uint32_t);
      p += sprintf(p, "[sp+0x%x]", sps_scale(slot));
    } else if (e[1] == 'x') {
      p += sprintf(p, "%08x", va_arg(argp, int32_t));
    } else {
      lua_assert(0);
    }
    fmt = e+2;
  }
  va_end(argp);
  while (*fmt)
    *p++ = *fmt++;
  *p++ = '\e'; *p++ = '['; *p++ = 'm'; *p++ = '\n';
  if (p > ra_dbg_buf+sizeof(ra_dbg_buf)-256) {
    fwrite(ra_dbg_buf, 1, (size_t)(p-ra_dbg_buf), stdout);
    p = ra_dbg_buf;
  }
  ra_dbg_p = p;
}

#define RA_DBG_START()        ra_dstart()
#define RA_DBG_FLUSH()        ra_dflush()
#define RA_DBG_REF() \
  do { char *_p = ra_dbg_p; ra_dprintf(as, ""); \
       ra_dbg_merge = _p; ra_dbg_mcp = as->mcp; } while (0)
#define RA_DBGX(x)        ra_dprintf x

#else
#define RA_DBG_START()        ((void)0)
#define RA_DBG_FLUSH()        ((void)0)
#define RA_DBG_REF()        ((void)0)
#define RA_DBGX(x)        ((void)0)
#endif

/* -- Register allocator ---------------------------------------------- */

#define ra_free(as, r)                rset_set(as->freeset, (r))
#define ra_modified(as, r)        rset_set(as->modset, (r))
#define ra_weak(as, r)                rset_set(as->weakset, (r))
#define ra_noweak(as, r)        rset_clear(as->weakset, (r))

#define ra_used(ir)                (ra_hasreg((ir)->r) || ra_hasspill((ir)->s))

/* Setup register allocator. */
static void ra_setup(ASMState *as)
{
  Reg r;
  /* Initially all regs (except the stack pointer) are free for use. */
  as->freeset = RSET_INIT;
  as->modset = RSET_EMPTY;
  as->weakset = RSET_EMPTY;
  as->phiset = RSET_EMPTY;
  memset(as->phireg, 0, sizeof(as->phireg));
  for (r = RID_MIN_GPR; r < RID_MAX; r++)
    as->cost[r] = REGCOST(~0u, 0u);
}

/* Rematerialize constants. */
static Reg ra_rematk(ASMState *as, IRRef ref)
{
  IRIns *ir;
  Reg r;
  if (ra_iskref(ref)) {
    r = ra_krefreg(ref);
    lua_assert(!rset_test(as->freeset, r));
    ra_free(as, r);
    ra_modified(as, r);
    emit_loadi(as, r, ra_krefk(as, ref));
    return r;
```

```
  }
  ir = IR(ref);
  r = ir->r;
  lua_assert(ra_hasreg(r) && !ra_hasspill(ir->s));
  ra_free(as, r);
  ra_modified(as, r);
  ir->r = RID_INIT;  /* Do not keep any hint. */
  RA_DBGX((as, "remat     $i $r", ir, r));
#if !LJ_SOFTFP
  if (ir->o == IR_KNUM) {
    emit_loadn(as, r, ir_knum(ir));
  } else
#endif
  if (emit_canremat(REF_BASE) && ir->o == IR_BASE) {
    ra_sethint(ir->r, RID_BASE);  /* Restore BASE register hint. */
    emit_getgl(as, r, jit_base);
  } else if (emit_canremat(ASMREF_L) && ir->o == IR_KPRI) {
    lua_assert(irt_isnil(ir->t));  /* REF_NIL stores ASMREF_L register. */
    emit_getgl(as, r, cur_L);
#if LJ_64
  } else if (ir->o == IR_KINT64) {
    emit_loadu64(as, r, ir_kint64(ir)->u64);
#endif
  } else {
    lua_assert(ir->o == IR_KINT || ir->o == IR_KGC ||
               ir->o == IR_KPTR || ir->o == IR_KKPTR || ir->o == IR_KNULL);
    emit_loadi(as, r, ir->i);
  }
  return r;
}

/* Force a spill. Allocate a new spill slot if needed. */
static int32_t ra_spill(ASMState *as, IRIns *ir)
{
  int32_t slot = ir->s;
  lua_assert(ir >= as->ir + REF_TRUE);
  if (!ra_hasspill(slot)) {
    if (irt_is64(ir->t)) {
      slot = as->evenspill;
      as->evenspill += 2;
    } else if (as->oddspill) {
      slot = as->oddspill;
      as->oddspill = 0;
    } else {
      slot = as->evenspill;
      as->oddspill = slot+1;
      as->evenspill += 2;
    }
    if (as->evenspill > 256)
      lj_trace_err(as->J, LJ_TRERR_SPILLOV);
    ir->s = (uint8_t)slot;
  }
  return sps_scale(slot);
}

/* Release the temporarily allocated register in ASMREF_TMP1/ASMREF_TMP2. */
static Reg ra_releasetmp(ASMState *as, IRRef ref)
{
  IRIns *ir = IR(ref);
  Reg r = ir->r;
  lua_assert(ra_hasreg(r) && !ra_hasspill(ir->s));
  ra_free(as, r);
  ra_modified(as, r);
  ir->r = RID_INIT;
  return r;
}

/* Restore a register (marked as free). Rematerialize or force a spill. */
static Reg ra_restore(ASMState *as, IRRef ref)
{
  if (emit_canremat(ref)) {
    return ra_rematk(as, ref);
  } else {
    IRIns *ir = IR(ref);
    int32_t ofs = ra_spill(as, ir);  /* Force a spill slot. */
    Reg r = ir->r;
```

```
    lua_assert(ra_hasreg(r));
    ra_sethint(ir->r, r);  /* Keep hint. */
    ra_free(as, r);
    if (!rset_test(as->weakset, r)) {  /* Only restore non-weak references. */
      ra_modified(as, r);
      RA_DBGX((as, "restore   $i $r", ir, r));
      emit_spload(as, ir, r, ofs);
    }
    return r;
  }
}

/* Save a register to a spill slot. */
static void ra_save(ASMState *as, IRIns *ir, Reg r)
{
  RA_DBGX((as, "save      $i $r", ir, r));
  emit_spstore(as, ir, r, sps_scale(ir->s));
}

#define MINCOST(name) \
  if (rset_test(RSET_ALL, RID_##name) && \
      LJ_LIKELY(allow&RID2RSET(RID_##name)) && as->cost[RID_##name] < cost) \
    cost = as->cost[RID_##name];

/* Evict the register with the lowest cost, forcing a restore. */
static Reg ra_evict(ASMState *as, RegSet allow)
{
  IRRef ref;
  RegCost cost = ~(RegCost)0;
  lua_assert(allow != RSET_EMPTY);
  if (RID_NUM_FPR == 0 || allow < RID2RSET(RID_MAX_GPR)) {
    GPRDEF(MINCOST)
  } else {
    FPRDEF(MINCOST)
  }
  ref = regcost_ref(cost);
  lua_assert(ra_iskref(ref) || (ref >= as->T->nk && ref < as->T->nins));
  /* Preferably pick any weak ref instead of a non-weak, non-const ref. */
  if (!irref_isk(ref) && (as->weakset & allow)) {
    IRIns *ir = IR(ref);
    if (!rset_test(as->weakset, ir->r))
      ref = regcost_ref(as->cost[rset_pickbot((as->weakset & allow))]);
  }
  return ra_restore(as, ref);
}

/* Pick any register (marked as free). Evict on-demand. */
static Reg ra_pick(ASMState *as, RegSet allow)
{
  RegSet pick = as->freeset & allow;
  if (!pick)
    return ra_evict(as, allow);
  else
    return rset_picktop(pick);
}

/* Get a scratch register (marked as free). */
static Reg ra_scratch(ASMState *as, RegSet allow)
{
  Reg r = ra_pick(as, allow);
  ra_modified(as, r);
  RA_DBGX((as, "scratch        $r", r));
  return r;
}

/* Evict all registers from a set (if not free). */
static void ra_evictset(ASMState *as, RegSet drop)
{
  RegSet work;
  as->modset |= drop;
#if !LJ_SOFTFP
  work = (drop & ~as->freeset) & RSET_FPR;
  while (work) {
    Reg r = rset_pickbot(work);
    ra_restore(as, regcost_ref(as->cost[r]));
    rset_clear(work, r);
```

```
    checkmclim(as);
  }
#endif
  work = (drop & ~as->freeset);
  while (work) {
    Reg r = rset_pickbot(work);
    ra_restore(as, regcost_ref(as->cost[r]));
    rset_clear(work, r);
    checkmclim(as);
  }
}

/* Evict (rematerialize) all registers allocated to constants. */
static void ra_evictk(ASMState *as)
{
  RegSet work;
#if !LJ_SOFTFP
  work = ~as->freeset & RSET_FPR;
  while (work) {
    Reg r = rset_pickbot(work);
    IRRef ref = regcost_ref(as->cost[r]);
    if (emit_canremat(ref) && irref_isk(ref)) {
      ra_rematk(as, ref);
      checkmclim(as);
    }
    rset_clear(work, r);
  }
#endif
  work = ~as->freeset & RSET_GPR;
  while (work) {
    Reg r = rset_pickbot(work);
    IRRef ref = regcost_ref(as->cost[r]);
    if (emit_canremat(ref) && irref_isk(ref)) {
      ra_rematk(as, ref);
      checkmclim(as);
    }
    rset_clear(work, r);
  }
}

#ifdef RID_NUM_KREF
/* Allocate a register for a constant. */
static Reg ra_allock(ASMState *as, int32_t k, RegSet allow)
{
  /* First try to find a register which already holds the same constant. */
  RegSet pick, work = ~as->freeset & RSET_GPR;
  Reg r;
  while (work) {
    IRRef ref;
    r = rset_pickbot(work);
    ref = regcost_ref(as->cost[r]);
    if (ref < ASMREF_L &&
        k == (ra_iskref(ref) ? ra_krefk(as, ref) : IR(ref)->i))
      return r;
    rset_clear(work, r);
  }
  pick = as->freeset & allow;
  if (pick) {
    /* Constants should preferably get unmodified registers. */
    if ((pick & ~as->modset))
      pick &= ~as->modset;
    r = rset_pickbot(pick);  /* Reduce conflicts with inverse allocation. */
  } else {
    r = ra_evict(as, allow);
  }
  RA_DBGX((as, "allock    $x $r", k, r));
  ra_setkref(as, r, k);
  rset_clear(as->freeset, r);
  ra_noweak(as, r);
  return r;
}

/* Allocate a specific register for a constant. */
static void ra_allockreg(ASMState *as, int32_t k, Reg r)
{
  Reg kr = ra_allock(as, k, RID2RSET(r));
```

```
  if (kr != r) {
    IRIns irdummy;
    irdummy.t.irt = IRT_INT;
    ra_scratch(as, RID2RSET(r));
    emit_movrr(as, &irdummy, r, kr);
  }
}
#else
#define ra_allockreg(as, k, r)                emit_loadi(as, (r), (k))
#endif

/* Allocate a register for ref from the allowed set of registers.
** Note: this function assumes the ref does NOT have a register yet!
** Picks an optimal register, sets the cost and marks the register as non-free.
*/
static Reg ra_allocref(ASMState *as, IRRef ref, RegSet allow)
{
  IRIns *ir = IR(ref);
  RegSet pick = as->freeset & allow;
  Reg r;
  lua_assert(ra_noreg(ir->r));
  if (pick) {
    /* First check register hint from propagation or PHI. */
    if (ra_hashint(ir->r)) {
      r = ra_gethint(ir->r);
      if (rset_test(pick, r))  /* Use hint register if possible. */
        goto found;
      /* Rematerialization is cheaper than missing a hint. */
      if (rset_test(allow, r) && emit_canremat(regcost_ref(as->cost[r]))) {
        ra_rematk(as, regcost_ref(as->cost[r]));
        goto found;
      }
      RA_DBGX((as, "hintmiss  $f $r", ref, r));
    }
    /* Invariants should preferably get unmodified registers. */
    if (ref < as->loopref && !irt_isphi(ir->t)) {
      if ((pick & ~as->modset))
        pick &= ~as->modset;
      r = rset_pickbot(pick);  /* Reduce conflicts with inverse allocation. */
    } else {
      /* We've got plenty of regs, so get callee-save regs if possible. */
      if (RID_NUM_GPR > 8 && (pick & ~RSET_SCRATCH))
        pick &= ~RSET_SCRATCH;
      r = rset_picktop(pick);
    }
  } else {
    r = ra_evict(as, allow);
  }
found:
  RA_DBGX((as, "alloc     $f $r", ref, r));
  ir->r = (uint8_t)r;
  rset_clear(as->freeset, r);
  ra_noweak(as, r);
  as->cost[r] = REGCOST_REF_T(ref, irt_t(ir->t));
  return r;
}

/* Allocate a register on-demand. */
static Reg ra_alloc1(ASMState *as, IRRef ref, RegSet allow)
{
  Reg r = IR(ref)->r;
  /* Note: allow is ignored if the register is already allocated. */
  if (ra_noreg(r)) r = ra_allocref(as, ref, allow);
  ra_noweak(as, r);
  return r;
}

/* Rename register allocation and emit move. */
static void ra_rename(ASMState *as, Reg down, Reg up)
{
  IRRef ren, ref = regcost_ref(as->cost[up] = as->cost[down]);
  IRIns *ir = IR(ref);
  ir->r = (uint8_t)up;
  as->cost[down] = 0;
  lua_assert((down < RID_MAX_GPR) == (up < RID_MAX_GPR));
  lua_assert(!rset_test(as->freeset, down) && rset_test(as->freeset, up));
```

```
  ra_free(as, down);  /* 'down' is free ... */
  ra_modified(as, down);
  rset_clear(as->freeset, up);  /* ... and 'up' is now allocated. */
  ra_noweak(as, up);
  RA_DBGX((as, "rename    $f $r $r", regcost_ref(as->cost[up]), down, up));
  emit_movrr(as, ir, down, up);  /* Backwards codegen needs inverse move. */
  if (!ra_hasspill(IR(ref)->s)) {  /* Add the rename to the IR. */
    lj_ir_set(as->J, IRT(IR_RENAME, IRT_NIL), ref, as->snapno);
    ren = tref_ref(lj_ir_emit(as->J));
    as->ir = as->T->ir;  /* The IR may have been reallocated. */
    IR(ren)->r = (uint8_t)down;
    IR(ren)->s = SPS_NONE;
  }
}

/* Pick a destination register (marked as free).
** Caveat: allow is ignored if there's already a destination register.
** Use ra_destreg() to get a specific register.
*/
static Reg ra_dest(ASMState *as, IRIns *ir, RegSet allow)
{
  Reg dest = ir->r;
  if (ra_hasreg(dest)) {
    ra_free(as, dest);
    ra_modified(as, dest);
  } else {
    if (ra_hashint(dest) && rset_test((as->freeset&allow), ra_gethint(dest))) {
      dest = ra_gethint(dest);
      ra_modified(as, dest);
      RA_DBGX((as, "dest           $r", dest));
    } else {
      dest = ra_scratch(as, allow);
    }
    ir->r = dest;
  }
  if (LJ_UNLIKELY(ra_hasspill(ir->s))) ra_save(as, ir, dest);
  return dest;
}

/* Force a specific destination register (marked as free). */
static void ra_destreg(ASMState *as, IRIns *ir, Reg r)
{
  Reg dest = ra_dest(as, ir, RID2RSET(r));
  if (dest != r) {
    lua_assert(rset_test(as->freeset, r));
    ra_modified(as, r);
    emit_movrr(as, ir, dest, r);
  }
}

#if LJ_TARGET_X86ORX64
/* Propagate dest register to left reference. Emit moves as needed.
** This is a required fixup step for all 2-operand machine instructions.
*/
static void ra_left(ASMState *as, Reg dest, IRRef lref)
{
  IRIns *ir = IR(lref);
  Reg left = ir->r;
  if (ra_noreg(left)) {
    if (irref_isk(lref)) {
      if (ir->o == IR_KNUM) {
        cTValue *tv = ir_knum(ir);
        /* FP remat needs a load except for +0. Still better than eviction. */
        if (tvispzero(tv) || !(as->freeset & RSET_FPR)) {
          emit_loadn(as, dest, tv);
          return;
        }
#if LJ_64
      } else if (ir->o == IR_KINT64) {
        emit_loadu64(as, dest, ir_kint64(ir)->u64);
        return;
#endif
      } else if (ir->o != IR_KPRI) {
        lua_assert(ir->o == IR_KINT || ir->o == IR_KGC ||
                   ir->o == IR_KPTR || ir->o == IR_KKPTR || ir->o == IR_KNULL);
        emit_loadi(as, dest, ir->i);
```

```
        return;
      }
    }
    if (!ra_hashint(left) && !iscrossref(as, lref))
      ra_sethint(ir->r, dest);  /* Propagate register hint. */
    left = ra_allocref(as, lref, dest < RID_MAX_GPR ? RSET_GPR : RSET_FPR);
  }
  ra_noweak(as, left);
  /* Move needed for true 3-operand instruction: y=a+b ==> y=a; y+=b. */
  if (dest != left) {
    /* Use register renaming if dest is the PHI reg. */
    if (irt_isphi(ir->t) && as->phireg[dest] == lref) {
      ra_modified(as, left);
      ra_rename(as, left, dest);
    } else {
      emit_movrr(as, ir, dest, left);
    }
  }
}
#else
/* Similar to ra_left, except we override any hints. */
static void ra_leftov(ASMState *as, Reg dest, IRRef lref)
{
  IRIns *ir = IR(lref);
  Reg left = ir->r;
  if (ra_noreg(left)) {
    ra_sethint(ir->r, dest);  /* Propagate register hint. */
    left = ra_allocref(as, lref,
                       (LJ_SOFTFP || dest < RID_MAX_GPR) ? RSET_GPR : RSET_FPR);
  }
  ra_noweak(as, left);
  if (dest != left) {
    /* Use register renaming if dest is the PHI reg. */
    if (irt_isphi(ir->t) && as->phireg[dest] == lref) {
      ra_modified(as, left);
      ra_rename(as, left, dest);
    } else {
      emit_movrr(as, ir, dest, left);
    }
  }
}
#endif

#if !LJ_64
/* Force a RID_RETLO/RID_RETHI destination register pair (marked as free). */
static void ra_destpair(ASMState *as, IRIns *ir)
{
  Reg destlo = ir->r, desthi = (ir+1)->r;
  /* First spill unrelated refs blocking the destination registers. */
  if (!rset_test(as->freeset, RID_RETLO) &&
      destlo != RID_RETLO && desthi != RID_RETLO)
    ra_restore(as, regcost_ref(as->cost[RID_RETLO]));
  if (!rset_test(as->freeset, RID_RETHI) &&
      destlo != RID_RETHI && desthi != RID_RETHI)
    ra_restore(as, regcost_ref(as->cost[RID_RETHI]));
  /* Next free the destination registers (if any). */
  if (ra_hasreg(destlo)) {
    ra_free(as, destlo);
    ra_modified(as, destlo);
  } else {
    destlo = RID_RETLO;
  }
  if (ra_hasreg(desthi)) {
    ra_free(as, desthi);
    ra_modified(as, desthi);
  } else {
    desthi = RID_RETHI;
  }
  /* Check for conflicts and shuffle the registers as needed. */
  if (destlo == RID_RETHI) {
    if (desthi == RID_RETLO) {
#if LJ_TARGET_X86
      *--as->mcp = XI_XCHGa + RID_RETHI;
#else
      emit_movrr(as, ir, RID_RETHI, RID_TMP);
      emit_movrr(as, ir, RID_RETLO, RID_RETHI);
```

```
      emit_movrr(as, ir, RID_TMP, RID_RETLO);
#endif
    } else {
      emit_movrr(as, ir, RID_RETHI, RID_RETLO);
      if (desthi != RID_RETHI) emit_movrr(as, ir, desthi, RID_RETHI);
    }
  } else if (desthi == RID_RETLO) {
    emit_movrr(as, ir, RID_RETLO, RID_RETHI);
    if (destlo != RID_RETLO) emit_movrr(as, ir, destlo, RID_RETLO);
  } else {
    if (desthi != RID_RETHI) emit_movrr(as, ir, desthi, RID_RETHI);
    if (destlo != RID_RETLO) emit_movrr(as, ir, destlo, RID_RETLO);
  }
  /* Restore spill slots (if any). */
  if (ra_hasspill((ir+1)->s)) ra_save(as, ir+1, RID_RETHI);
  if (ra_hasspill(ir->s)) ra_save(as, ir, RID_RETLO);
}
#endif

/* -- Snapshot handling --------- ----------------------------------- */

/* Can we rematerialize a KNUM instead of forcing a spill? */
static int asm_snap_canremat(ASMState *as)
{
  Reg r;
  for (r = RID_MIN_FPR; r < RID_MAX_FPR; r++)
    if (irref_isk(regcost_ref(as->cost[r])))
      return 1;
  return 0;
}

/* Check whether a sunk store corresponds to an allocation. */
static int asm_sunk_store(ASMState *as, IRIns *ira, IRIns *irs)
{
  if (irs->s == 255) {
    if (irs->o == IR_ASTORE || irs->o == IR_HSTORE ||
        irs->o == IR_FSTORE || irs->o == IR_XSTORE) {
      IRIns *irk = IR(irs->op1);
      if (irk->o == IR_AREF || irk->o == IR_HREFK)
        irk = IR(irk->op1);
      return (IR(irk->op1) == ira);
    }
    return 0;
  } else {
    return (ira + irs->s == irs);  /* Quick check. */
  }
}

/* Allocate register or spill slot for a ref that escapes to a snapshot. */
static void asm_snap_alloc1(ASMState *as, IRRef ref)
{
  IRIns *ir = IR(ref);
  if (!irref_isk(ref) && (!(ra_used(ir) || ir->r == RID_SUNK))) {
    if (ir->r == RID_SINK) {
      ir->r = RID_SUNK;
#if LJ_HASFFI
      if (ir->o == IR_CNEWI) {  /* Allocate CNEWI value. */
        asm_snap_alloc1(as, ir->op2);
        if (LJ_32 && (ir+1)->o == IR_HIOP)
          asm_snap_alloc1(as, (ir+1)->op2);
      } else
#endif
      {  /* Allocate stored values for TNEW, TDUP and CNEW. */
        IRIns *irs;
        lua_assert(ir->o == IR_TNEW || ir->o == IR_TDUP || ir->o == IR_CNEW);
        for (irs = IR(as->snapref-1); irs > ir; irs--)
          if (irs->r == RID_SINK && asm_sunk_store(as, ir, irs)) {
            lua_assert(irs->o == IR_ASTORE || irs->o == IR_HSTORE ||
                       irs->o == IR_FSTORE || irs->o == IR_XSTORE);
            asm_snap_alloc1(as, irs->op2);
            if (LJ_32 && (irs+1)->o == IR_HIOP)
              asm_snap_alloc1(as, (irs+1)->op2);
          }
      }
    } else {
      RegSet allow;
```

```
      if (ir->o == IR_CONV && ir->op2 == IRCONV_NUM_INT) {
        IRIns *irc;
        for (irc = IR(as->curins); irc > ir; irc--)
          if ((irc->op1 == ref || irc->op2 == ref) &&
              !(irc->r == RID_SINK || irc->r == RID_SUNK))
            goto nosink;  /* Don't sink conversion if result is used. */
        asm_snap_alloc1(as, ir->op1);
        return;
      }
    nosink:
      allow = (!LJ_SOFTFP && irt_isfp(ir->t)) ? RSET_FPR : RSET_GPR;
      if ((as->freeset & allow) ||
               (allow == RSET_FPR && asm_snap_canremat(as))) {
        /* Get a weak register if we have a free one or can rematerialize. */
        Reg r = ra_allocref(as, ref, allow);  /* Allocate a register. */
        if (!irt_isphi(ir->t))
          ra_weak(as, r);  /* But mark it as weakly referenced. */
        checkmclim(as);
        RA_DBGX((as, "snapreg   $f $r", ref, ir->r));
      } else {
        ra_spill(as, ir);  /* Otherwise force a spill slot. */
        RA_DBGX((as, "snapspill $f $s", ref, ir->s));
      }
    }
  }
}

/* Allocate refs escaping to a snapshot. */
static void asm_snap_alloc(ASMState *as)
{
  SnapShot *snap = &as->T->snap[as->snapno];
  SnapEntry *map = &as->T->snapmap[snap->mapofs];
  MSize n, nent = snap->nent;
  for (n = 0; n < nent; n++) {
    SnapEntry sn = map[n];
    IRRef ref = snap_ref(sn);
    if (!irref_isk(ref)) {
      asm_snap_alloc1(as, ref);
      if (LJ_SOFTFP && (sn & SNAP_SOFTFPNUM)) {
        lua_assert(irt_type(IR(ref+1)->t) == IRT_SOFTFP);
        asm_snap_alloc1(as, ref+1);
      }
    }
  }
}

/* All guards for a snapshot use the same exitno. This is currently the
** same as the snapshot number. Since the exact origin of the exit cannot
** be determined, all guards for the same snapshot must exit with the same
** RegSP mapping.
** A renamed ref which has been used in a prior guard for the same snapshot
** would cause an inconsistency. The easy way out is to force a spill slot.
*/
static int asm_snap_checkrename(ASMState *as, IRRef ren)
{
  SnapShot *snap = &as->T->snap[as->snapno];
  SnapEntry *map = &as->T->snapmap[snap->mapofs];
  MSize n, nent = snap->nent;
  for (n = 0; n < nent; n++) {
    SnapEntry sn = map[n];
    IRRef ref = snap_ref(sn);
    if (ref == ren || (LJ_SOFTFP && (sn & SNAP_SOFTFPNUM) && ++ref == ren)) {
      IRIns *ir = IR(ref);
      ra_spill(as, ir);  /* Register renamed, so force a spill slot. */
      RA_DBGX((as, "snaprensp $f $s", ref, ir->s));
      return 1;  /* Found. */
    }
  }
  return 0;  /* Not found. */
}

/* Prepare snapshot for next guard instruction. */
static void asm_snap_prep(ASMState *as)
{
  if (as->curins < as->snapref) {
    do {
```

```
      if (as->snapno == 0) return;  /* Called by sunk stores before snap #0. */
      as->snapno--;
      as->snapref = as->T->snap[as->snapno].ref;
    } while (as->curins < as->snapref);
    asm_snap_alloc(as);
    as->snaprename = as->T->nins;
  } else {
    /* Process any renames above the highwater mark. */
    for (; as->snaprename < as->T->nins; as->snaprename++) {
      IRIns *ir = IR(as->snaprename);
      if (asm_snap_checkrename(as, ir->op1))
        ir->op2 = REF_BIAS-1;  /* Kill rename. */
    }
  }
}

/* -- Miscellaneous helpers ----------------------------------------------- */

/* Calculate stack adjustment. */
static int32_t asm_stack_adjust(ASMState *as)
{
  if (as->evenspill <= SPS_FIXED)
    return 0;
  return sps_scale(sps_align(as->evenspill));
}

/* Must match with hash*() in lj_tab.c. */
static uint32_t ir_khash(IRIns *ir)
{
  uint32_t lo, hi;
  if (irt_isstr(ir->t)) {
    return ir_kstr(ir)->hash;
  } else if (irt_isnum(ir->t)) {
    lo = ir_knum(ir)->u32.lo;
    hi = ir_knum(ir)->u32.hi << 1;
  } else if (irt_ispri(ir->t)) {
    lua_assert(!irt_isnil(ir->t));
    return irt_type(ir->t)-IRT_FALSE;
  } else {
    lua_assert(irt_isgcv(ir->t));
    lo = u32ptr(ir_kgc(ir));
    hi = lo + HASH_BIAS;
  }
  return hashrot(lo, hi);
}

/* -- Allocations --------------------------------------------------------- */

static void asm_gencall(ASMState *as, const CCallInfo *ci, IRRef *args);
static void asm_setupresult(ASMState *as, IRIns *ir, const CCallInfo *ci);

static void asm_snew(ASMState *as, IRIns *ir)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_str_new];
  IRRef args[3];
  args[0] = ASMREF_L;  /* lua_State *L     */
  args[1] = ir->op1;   /* const char *str */
  args[2] = ir->op2;   /* size_t len      */
  as->gcsteps++;
  asm_setupresult(as, ir, ci);  /* GCstr * */
  asm_gencall(as, ci, args);
}

static void asm_tnew(ASMState *as, IRIns *ir)
{
   const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_tab_new1];
   IRRef args[2];
   args[0] = ASMREF_L;     /* lua_State *L     */
   args[1] = ASMREF_TMP1;  /* uint32_t ahsize */
   as->gcsteps++;
   asm_setupresult(as, ir, ci);  /* GCtab * */
   asm_gencall(as, ci, args);
   ra_allockreg(as, ir->op1 | (ir->op2 << 24), ra_releasetmp(as, ASMREF_TMP1));
 }

 static void asm_tdup(ASMState *as, IRIns *ir)
```

```
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_tab_dup];
  IRRef args[2];
  args[0] = ASMREF_L;  /* lua_State *L    */
  args[1] = ir->op1;   /* const GCtab *kt */
  as->gcsteps++;
  asm_setupresult(as, ir, ci);  /* GCtab * */
  asm_gencall(as, ci, args);
}

static void asm_gc_check(ASMState *as);

/* Explicit GC step. */
static void asm_gcstep(ASMState *as, IRIns *ir)
{
  IRIns *ira;
  for (ira = IR(as->stopins+1); ira < ir; ira++)
    if ((ira->o == IR_TNEW || ira->o == IR_TDUP ||
         (LJ_HASFFI && (ira->o == IR_CNEW || ira->o == IR_CNEWI))) &&
        ra_used(ira))
      as->gcsteps++;
  if (as->gcsteps)
    asm_gc_check(as);
  as->gcsteps = 0x80000000;  /* Prevent implicit GC check further up. */
}

/* -- Buffer operations -------------------------------------------------- */

static void asm_tvptr(ASMState *as, Reg dest, IRRef ref);

static void asm_bufhdr(ASMState *as, IRIns *ir)
{
  Reg sb = ra_dest(as, ir, RSET_GPR);
  if ((ir->op2 & IRBUFHDR_APPEND)) {
    /* Rematerialize const buffer pointer instead of likely spill. */
    IRIns *irp = IR(ir->op1);
    if (!(ra_hasreg(irp->r) || irp == ir-1 ||
          (irp == ir-2 && !ra_used(ir-1)))) {
      while (!(irp->o == IR_BUFHDR && !(irp->op2 & IRBUFHDR_APPEND)))
        irp = IR(irp->op1);
      if (irref_isk(irp->op1)) {
        ra_weak(as, ra_allocref(as, ir->op1, RSET_GPR));
        ir = irp;
      }
    }
  } else {
    Reg tmp = ra_scratch(as, rset_exclude(RSET_GPR, sb));
    /* Passing ir isn't strictly correct, but it's an IRT_P32, too. */
    emit_storeofs(as, ir, tmp, sb, offsetof(SBuf, p));
    emit_loadofs(as, ir, tmp, sb, offsetof(SBuf, b));
  }
#if LJ_TARGET_X86ORX64
  ra_left(as, sb, ir->op1);
#else
  ra_leftov(as, sb, ir->op1);
#endif
}

static void asm_bufput(ASMState *as, IRIns *ir)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_buf_putstr];
  IRRef args[3];
  IRIns *irs;
  int kchar = -1;
  args[0] = ir->op1;  /* SBuf * */
  args[1] = ir->op2;  /* GCstr * */
  irs = IR(ir->op2);
  lua_assert(irt_isstr(irs->t));
  if (irs->o == IR_KGC) {
    GCstr *s = ir_kstr(irs);
    if (s->len == 1) {  /* Optimize put of single-char string constant. */
      kchar = strdata(s)[0];
      args[1] = ASMREF_TMP1;  /* int, truncated to char */
      ci = &lj_ir_callinfo[IRCALL_lj_buf_putchar];
    }
  } else if (mayfuse(as, ir->op2) && ra_noreg(irs->r)) {
```

```
1087      if (irs->o == IR_TOSTR) {  /* Fuse number to string conversions. */
1088        if (irs->op2 == IRTOSTR_NUM) {
1089          args[1] = ASMREF_TMP1;  /* TValue * */
1090          ci = &lj_ir_callinfo[IRCALL_lj_strfmt_putnum];
1091        } else {
1092          lua_assert(irt_isinteger(IR(irs->op1)->t));
1093          args[1] = irs->op1;  /* int */
1094          if (irs->op2 == IRTOSTR_INT)
1095            ci = &lj_ir_callinfo[IRCALL_lj_strfmt_putint];
1096          else
1097            ci = &lj_ir_callinfo[IRCALL_lj_buf_putchar];
1098        }
1099      } else if (irs->o == IR_SNEW) {  /* Fuse string allocation. */
1100        args[1] = irs->op1;  /* const void * */
1101        args[2] = irs->op2;  /* MSize */
1102        ci = &lj_ir_callinfo[IRCALL_lj_buf_putmem];
1103      }
1104    }
1105    asm_setupresult(as, ir, ci);  /* SBuf * */
1106    asm_gencall(as, ci, args);
1107    if (args[1] == ASMREF_TMP1) {
1108      Reg tmp = ra_releasetmp(as, ASMREF_TMP1);
1109      if (kchar == -1)
1110        asm_tvptr(as, tmp, irs->op1);
1111      else
1112        ra_allockreg(as, kchar, tmp);
1113    }
1114  }
1115
1116  static void asm_bufstr(ASMState *as, IRIns *ir)
1117  {
1118    const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_buf_tostr];
1119    IRRef args[1];
1120    args[0] = ir->op1;  /* SBuf *sb */
1121    as->gcsteps++;
1122    asm_setupresult(as, ir, ci);  /* GCstr * */
1123    asm_gencall(as, ci, args);
1124  }
1125
1126  /* -- Type conversions ---------------------------------------------------- */
1127
1128  static void asm_tostr(ASMState *as, IRIns *ir)
1129  {
1130    const CCallInfo *ci;
1131    IRRef args[2];
1132    args[0] = ASMREF_L;
1133    as->gcsteps++;
1134    if (ir->op2 == IRTOSTR_NUM) {
1135      args[1] = ASMREF_TMP1;  /* cTValue * */
1136      ci = &lj_ir_callinfo[IRCALL_lj_strfmt_num];
1137    } else {
1138      args[1] = ir->op1;  /* int32_t k */
1139      if (ir->op2 == IRTOSTR_INT)
1140        ci = &lj_ir_callinfo[IRCALL_lj_strfmt_int];
1141      else
1142        ci = &lj_ir_callinfo[IRCALL_lj_strfmt_char];
1143    }
1144    asm_setupresult(as, ir, ci);  /* GCstr * */
1145    asm_gencall(as, ci, args);
1146    if (ir->op2 == IRTOSTR_NUM)
1147      asm_tvptr(as, ra_releasetmp(as, ASMREF_TMP1), ir->op1);
1148  }
1149
1150  #if LJ_32 && LJ_HASFFI && !LJ_SOFTFP && !LJ_TARGET_X86
1151  static void asm_conv64(ASMState *as, IRIns *ir)
1152  {
1153    IRType st = (IRType)((ir-1)->op2 & IRCONV_SRCMASK);
1154    IRType dt = (((ir-1)->op2 & IRCONV_DSTMASK) >> IRCONV_DSH);
1155    IRCallID id;
1156    IRRef args[2];
1157    lua_assert((ir-1)->o == IR_CONV && ir->o == IR_HIOP);
1158    args[LJ_BE] = (ir-1)->op1;
1159    args[LJ_LE] = ir->op1;
1160    if (st == IRT_NUM || st == IRT_FLOAT) {
1161      id = IRCALL_fp64_d2l + ((st == IRT_FLOAT) ? 2 : 0) + (dt - IRT_I64);
1162      ir--;
```

```
  } else {
    id = IRCALL_fp64_l2d + ((dt == IRT_FLOAT) ? 2 : 0) + (st - IRT_I64);
  }
  {
#if LJ_TARGET_ARM && !LJ_ABI_SOFTFP
    CCallInfo cim = lj_ir_callinfo[id], *ci = &cim;
    cim.flags |= CCI_VARARG;  /* These calls don't use the hard-float ABI! */
#else
    const CCallInfo *ci = &lj_ir_callinfo[id];
#endif
    asm_setupresult(as, ir, ci);
    asm_gencall(as, ci, args);
  }
}
#endif

/* -- Memory references --------------------------------------------------- */

static void asm_newref(ASMState *as, IRIns *ir)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_tab_newkey];
  IRRef args[3];
  if (ir->r == RID_SINK)
    return;
  args[0] = ASMREF_L;     /* lua_State *L */
  args[1] = ir->op1;      /* GCtab *t     */
  args[2] = ASMREF_TMP1;  /* cTValue *key */
  asm_setupresult(as, ir, ci);  /* TValue * */
  asm_gencall(as, ci, args);
  asm_tvptr(as, ra_releasetmp(as, ASMREF_TMP1), ir->op2);
}

static void asm_lref(ASMState *as, IRIns *ir)
{
  Reg r = ra_dest(as, ir, RSET_GPR);
#if LJ_TARGET_X86ORX64
  ra_left(as, r, ASMREF_L);
#else
  ra_leftov(as, r, ASMREF_L);
#endif
}

/* -- Calls --------------------------------------------------------------- */

/* Collect arguments from CALL* and CARG instructions. */
static void asm_collectargs(ASMState *as, IRIns *ir,
                            const CCallInfo *ci, IRRef *args)
{
  uint32_t n = CCI_XNARGS(ci);
  lua_assert(n <= CCI_NARGS_MAX*2);  /* Account for split args. */
  if ((ci->flags & CCI_L)) { *args++ = ASMREF_L; n--; }
  while (n-- > 1) {
    ir = IR(ir->op1);
    lua_assert(ir->o == IR_CARG);
    args[n] = ir->op2 == REF_NIL ? 0 : ir->op2;
  }
  args[0] = ir->op1 == REF_NIL ? 0 : ir->op1;
  lua_assert(IR(ir->op1)->o != IR_CARG);
}

/* Reconstruct CCallInfo flags for CALLX*. */
static uint32_t asm_callx_flags(ASMState *as, IRIns *ir)
{
  uint32_t nargs = 0;
  if (ir->op1 != REF_NIL) {  /* Count number of arguments first. */
    IRIns *ira = IR(ir->op1);
    nargs++;
    while (ira->o == IR_CARG) { nargs++; ira = IR(ira->op1); }
  }
#if LJ_HASFFI
  if (IR(ir->op2)->o == IR_CARG) {  /* Copy calling convention info. */
    CTypeID id = (CTypeID)IR(IR(ir->op2)->op2)->i;
    CType *ct = ctype_get(ctype_ctsG(J2G(as->J)), id);
    nargs |= ((ct->info & CTF_VARARG) ? CCI_VARARG : 0);
#if LJ_TARGET_X86
    nargs |= (ctype_cconv(ct->info) << CCI_CC_SHIFT);
```

```
#endif
  }
#endif
  return (nargs | (ir->t.irt << CCI_OTSHIFT));
}

static void asm_callid(ASMState *as, IRIns *ir, IRCallID id)
{
  const CCallInfo *ci = &lj_ir_callinfo[id];
  IRRef args[2];
  args[0] = ir->op1;
  args[1] = ir->op2;
  asm_setupresult(as, ir, ci);
  asm_gencall(as, ci, args);
}

static void asm_call(ASMState *as, IRIns *ir)
{
  IRRef args[CCI_NARGS_MAX];
  const CCallInfo *ci = &lj_ir_callinfo[ir->op2];
  asm_collectargs(as, ir, ci, args);
  asm_setupresult(as, ir, ci);
  asm_gencall(as, ci, args);
}

#if !LJ_SOFTFP
static void asm_fppow(ASMState *as, IRIns *ir, IRRef lref, IRRef rref)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_pow];
  IRRef args[2];
  args[0] = lref;
  args[1] = rref;
  asm_setupresult(as, ir, ci);
  asm_gencall(as, ci, args);
}

static int asm_fpjoin_pow(ASMState *as, IRIns *ir)
{
  IRIns *irp = IR(ir->op1);
  if (irp == ir-1 && irp->o == IR_MUL && !ra_used(irp)) {
    IRIns *irpp = IR(irp->op1);
    if (irpp == ir-2 && irpp->o == IR_FPMATH &&
        irpp->op2 == IRFPM_LOG2 && !ra_used(irpp)) {
      asm_fppow(as, ir, irpp->op1, irp->op2);
      return 1;
    }
  }
  return 0;
}
#endif

/* -- PHI and loop handling ------------------------------------------------ */

/* Break a PHI cycle by renaming to a free register (evict if needed). */
static void asm_phi_break(ASMState *as, RegSet blocked, RegSet blockedby,
                          RegSet allow)
{
  RegSet candidates = blocked & allow;
  if (candidates) {  /* If this register file has candidates. */
    /* Note: the set for ra_pick cannot be empty, since each register file
    ** has some registers never allocated to PHIs.
    */
    Reg down, up = ra_pick(as, ~blocked & allow);  /* Get a free register. */
    if (candidates & ~blockedby)  /* Optimize shifts, else it's a cycle. */
      candidates = candidates & ~blockedby;
    down = rset_picktop(candidates);  /* Pick candidate PHI register. */
    ra_rename(as, down, up);  /* And rename it to the free register. */
  }
}

/* PHI register shuffling.
**
** The allocator tries hard to preserve PHI register assignments across
** the loop body. Most of the time this loop does nothing, since there
** are no register mismatches.
**
```

```
1315  ** If a register mismatch is detected and ...
1316  ** - the register is currently free: rename it.
1317  ** - the register is blocked by an invariant: restore/remat and rename it.
1318  ** - Otherwise the register is used by another PHI, so mark it as blocked.
1319  **
1320  ** The renames are order-sensitive, so just retry the loop if a register
1321  ** is marked as blocked, but has been freed in the meantime. A cycle is
1322  ** detected if all of the blocked registers are allocated. To break the
1323  ** cycle rename one of them to a free register and retry.
1324  **
1325  ** Note that PHI spill slots are kept in sync and don't need to be shuffled.
1326  */
1327 static void asm_phi_shuffle(ASMState *as)
1328 {
1329   RegSet work;
1330
1331   /* Find and resolve PHI register mismatches. */
1332   for (;;) {
1333     RegSet blocked = RSET_EMPTY;
1334     RegSet blockedby = RSET_EMPTY;
1335     RegSet phiset = as->phiset;
1336     while (phiset) {  /* Check all left PHI operand registers. */
1337       Reg r = rset_pickbot(phiset);
1338       IRIns *irl = IR(as->phireg[r]);
1339       Reg left = irl->r;
1340       if (r != left) {  /* Mismatch? */
1341         if (!rset_test(as->freeset, r)) {  /* PHI register blocked? */
1342           IRRef ref = regcost_ref(as->cost[r]);
1343           /* Blocked by other PHI (w/reg)? */
1344           if (!ra_iskref(ref) && irt_ismarked(IR(ref)->t)) {
1345             rset_set(blocked, r);
1346             if (ra_hasreg(left))
1347               rset_set(blockedby, left);
1348             left = RID_NONE;
1349           } else {  /* Otherwise grab register from invariant. */
1350             ra_restore(as, ref);
1351             checkmclim(as);
1352           }
1353         }
1354         if (ra_hasreg(left)) {
1355           ra_rename(as, left, r);
1356           checkmclim(as);
1357         }
1358       }
1359       rset_clear(phiset, r);
1360     }
1361     if (!blocked) break;  /* Finished. */
1362     if (!(as->freeset & blocked)) {  /* Break cycles if none are free. */
1363       asm_phi_break(as, blocked, blockedby, RSET_GPR);
1364       if (!LJ_SOFTFP) asm_phi_break(as, blocked, blockedby, RSET_FPR);
1365       checkmclim(as);
1366     }  /* Else retry some more renames. */
1367   }
1368
1369   /* Restore/remat invariants whose registers are modified inside the loop. */
1370 #if !LJ_SOFTFP
1371   work = as->modset & ~(as->freeset | as->phiset) & RSET_FPR;
1372   while (work) {
1373     Reg r = rset_pickbot(work);
1374     ra_restore(as, regcost_ref(as->cost[r]));
1375     rset_clear(work, r);
1376     checkmclim(as);
1377   }
1378 #endif
1379   work = as->modset & ~(as->freeset | as->phiset);
1380   while (work) {
1381     Reg r = rset_pickbot(work);
1382     ra_restore(as, regcost_ref(as->cost[r]));
1383     rset_clear(work, r);
1384     checkmclim(as);
1385   }
1386
1387   /* Allocate and save all unsaved PHI regs and clear marks. */
1388   work = as->phiset;
1389   while (work) {
1390     Reg r = rset_picktop(work);
```

```
    IRRef lref = as->phireg[r];
    IRIns *ir = IR(lref);
    if (ra_hasspill(ir->s)) {  /* Left PHI gained a spill slot? */
      irt_clearmark(ir->t);  /* Handled here, so clear marker now. */
      ra_alloc1(as, lref, RID2RSET(r));
      ra_save(as, ir, r);  /* Save to spill slot inside the loop. */
      checkmclim(as);
    }
    rset_clear(work, r);
  }
}

/* Copy unsynced left/right PHI spill slots. Rarely needed. */
static void asm_phi_copyspill(ASMState *as)
{
  int need = 0;
  IRIns *ir;
  for (ir = IR(as->orignins-1); ir->o == IR_PHI; ir--)
    if (ra_hasspill(ir->s) && ra_hasspill(IR(ir->op1)->s))
      need |= irt_isfp(ir->t) ? 2 : 1;  /* Unsynced spill slot? */
  if ((need & 1)) {  /* Copy integer spill slots. */
#if !LJ_TARGET_X86ORX64
    Reg r = RID_TMP;
#else
    Reg r = RID_RET;
    if ((as->freeset & RSET_GPR))
      r = rset_pickbot((as->freeset & RSET_GPR));
    else
      emit_spload(as, IR(regcost_ref(as->cost[r])), r, SPOFS_TMP);
#endif
    for (ir = IR(as->orignins-1); ir->o == IR_PHI; ir--) {
      if (ra_hasspill(ir->s)) {
        IRIns *irl = IR(ir->op1);
        if (ra_hasspill(irl->s) && !irt_isfp(ir->t)) {
          emit_spstore(as, irl, r, sps_scale(irl->s));
          emit_spload(as, ir, r, sps_scale(ir->s));
          checkmclim(as);
        }
      }
    }
#if LJ_TARGET_X86ORX64
    if (!rset_test(as->freeset, r))
      emit_spstore(as, IR(regcost_ref(as->cost[r])), r, SPOFS_TMP);
#endif
  }
#if !LJ_SOFTFP
  if ((need & 2)) {  /* Copy FP spill slots. */
#if LJ_TARGET_X86
    Reg r = RID_XMM0;
#else
    Reg r = RID_FPRET;
#endif
    if ((as->freeset & RSET_FPR))
      r = rset_pickbot((as->freeset & RSET_FPR));
    if (!rset_test(as->freeset, r))
      emit_spload(as, IR(regcost_ref(as->cost[r])), r, SPOFS_TMP);
    for (ir = IR(as->orignins-1); ir->o == IR_PHI; ir--) {
      if (ra_hasspill(ir->s)) {
        IRIns *irl = IR(ir->op1);
        if (ra_hasspill(irl->s) && irt_isfp(ir->t)) {
          emit_spstore(as, irl, r, sps_scale(irl->s));
          emit_spload(as, ir, r, sps_scale(ir->s));
          checkmclim(as);
        }
      }
    }
    if (!rset_test(as->freeset, r))
      emit_spstore(as, IR(regcost_ref(as->cost[r])), r, SPOFS_TMP);
  }
#endif
}

/* Emit renames for left PHIs which are only spilled outside the loop. */
static void asm_phi_fixup(ASMState *as)
{
  RegSet work = as->phiset;
```

```
  while (work) {
    Reg r = rset_picktop(work);
    IRRef lref = as->phireg[r];
    IRIns *ir = IR(lref);
    if (irt_ismarked(ir->t)) {
      irt_clearmark(ir->t);
      /* Left PHI gained a spill slot before the loop? */
      if (ra_hasspill(ir->s)) {
        IRRef ren;
        lj_ir_set(as->J, IRT(IR_RENAME, IRT_NIL), lref, as->loopsnapno);
        ren = tref_ref(lj_ir_emit(as->J));
        as->ir = as->T->ir;  /* The IR may have been reallocated. */
        IR(ren)->r = (uint8_t)r;
        IR(ren)->s = SPS_NONE;
      }
    }
    rset_clear(work, r);
  }
}

/* Setup right PHI reference. */
static void asm_phi(ASMState *as, IRIns *ir)
{
  RegSet allow = ((!LJ_SOFTFP && irt_isfp(ir->t)) ? RSET_FPR : RSET_GPR) &
                 ~as->phiset;
  RegSet afree = (as->freeset & allow);
  IRIns *irl = IR(ir->op1);
  IRIns *irr = IR(ir->op2);
  if (ir->r == RID_SINK)  /* Sink PHI. */
    return;
  /* Spill slot shuffling is not implemented yet (but rarely needed). */
  if (ra_hasspill(irl->s) || ra_hasspill(irr->s))
    lj_trace_err(as->J, LJ_TRERR_NYIPHI);
  /* Leave at least one register free for non-PHIs (and PHI cycle breaking). */
  if ((afree & (afree-1))) {  /* Two or more free registers? */
    Reg r;
    if (ra_noreg(irr->r)) {  /* Get a register for the right PHI. */
      r = ra_allocref(as, ir->op2, allow);
    } else {  /* Duplicate right PHI, need a copy (rare). */
      r = ra_scratch(as, allow);
      emit_movrr(as, irr, r, irr->r);
    }
    ir->r = (uint8_t)r;
    rset_set(as->phiset, r);
    as->phireg[r] = (IRRef1)ir->op1;
    irt_setmark(irl->t);  /* Marks left PHIs _with_ register. */
    if (ra_noreg(irl->r))
      ra_sethint(irl->r, r); /* Set register hint for left PHI. */
  } else {  /* Otherwise allocate a spill slot. */
    /* This is overly restrictive, but it triggers only on synthetic code. */
    if (ra_hasreg(irl->r) || ra_hasreg(irr->r))
      lj_trace_err(as->J, LJ_TRERR_NYIPHI);
    ra_spill(as, ir);
    irr->s = ir->s;  /* Set right PHI spill slot. Sync left slot later. */
  }
}

static void asm_loop_fixup(ASMState *as);

/* Middle part of a loop. */
static void asm_loop(ASMState *as)
{
  MCode *mcspill;
  /* LOOP is a guard, so the snapno is up to date. */
  as->loopsnapno = as->snapno;
  if (as->gcsteps)
    asm_gc_check(as);
  /* LOOP marks the transition from the variant to the invariant part. */
  as->flagmcp = as->invmcp = NULL;
  as->sectref = 0;
  if (!neverfuse(as)) as->fuseref = 0;
  asm_phi_shuffle(as);
  mcspill = as->mcp;
  asm_phi_copyspill(as);
  asm_loop_fixup(as);
  as->mcloop = as->mcp;
```

```
1543    RA_DBGX((as, "===== LOOP ====="));
1544    if (!as->realign) RA_DBG_FLUSH();
1545    if (as->mcp != mcspill)
1546      emit_jmp(as, mcspill);
1547  }
1548
1549  /* -- Target-specific assembler ----------------------------------------- */
1550
1551  #if LJ_TARGET_X86ORX64
1552  #include "lj_asm_x86.h"
1553  #elif LJ_TARGET_ARM
1554  #include "lj_asm_arm.h"
1555  #elif LJ_TARGET_PPC
1556  #include "lj_asm_ppc.h"
1557  #elif LJ_TARGET_MIPS
1558  #include "lj_asm_mips.h"
1559  #else
1560  #error "Missing assembler for target CPU"
1561  #endif
1562
1563  /* -- Instruction dispatch ---------------------------------------------- */
1564
1565  /* Assemble a single instruction. */
1566  static void asm_ir(ASMState *as, IRIns *ir)
1567  {
1568    switch ((IROp)ir->o) {
1569    /* Miscellaneous ops. */
1570    case IR_LOOP: asm_loop(as); break;
1571    case IR_NOP: case IR_XBAR: lua_assert(!ra_used(ir)); break;
1572    case IR_USE:
1573      ra_alloc1(as, ir->op1, irt_isfp(ir->t) ? RSET_FPR : RSET_GPR); break;
1574    case IR_PHI: asm_phi(as, ir); break;
1575    case IR_HIOP: asm_hiop(as, ir); break;
1576    case IR_GCSTEP: asm_gcstep(as, ir); break;
1577    case IR_PROF: asm_prof(as, ir); break;
1578
1579    /* Guarded assertions. */
1580    case IR_LT: case IR_GE: case IR_LE: case IR_GT:
1581    case IR_ULT: case IR_UGE: case IR_ULE: case IR_UGT:
1582    case IR_ABC:
1583      asm_comp(as, ir);
1584      break;
1585    case IR_EQ: case IR_NE:
1586      if ((ir-1)->o == IR_HREF && ir->op1 == as->curins-1) {
1587        as->curins--;
1588        asm_href(as, ir-1, (IROp)ir->o);
1589      } else {
1590        asm_equal(as, ir);
1591      }
1592      break;
1593
1594    case IR_RETF: asm_retf(as, ir); break;
1595
1596    /* Bit ops. */
1597    case IR_BNOT: asm_bnot(as, ir); break;
1598    case IR_BSWAP: asm_bswap(as, ir); break;
1599    case IR_BAND: asm_band(as, ir); break;
1600    case IR_BOR: asm_bor(as, ir); break;
1601    case IR_BXOR: asm_bxor(as, ir); break;
1602    case IR_BSHL: asm_bshl(as, ir); break;
1603    case IR_BSHR: asm_bshr(as, ir); break;
1604    case IR_BSAR: asm_bsar(as, ir); break;
1605    case IR_BROL: asm_brol(as, ir); break;
1606    case IR_BROR: asm_bror(as, ir); break;
1607
1608    /* Arithmetic ops. */
1609    case IR_ADD: asm_add(as, ir); break;
1610    case IR_SUB: asm_sub(as, ir); break;
1611    case IR_MUL: asm_mul(as, ir); break;
1612    case IR_DIV: asm_div(as, ir); break;
1613    case IR_MOD: asm_mod(as, ir); break;
1614    case IR_POW: asm_pow(as, ir); break;
1615    case IR_NEG: asm_neg(as, ir); break;
1616    case IR_ABS: asm_abs(as, ir); break;
1617    case IR_ATAN2: asm_atan2(as, ir); break;
1618    case IR_LDEXP: asm_ldexp(as, ir); break;
```

```
1619    case IR_MIN: asm_min(as, ir); break;
1620    case IR_MAX: asm_max(as, ir); break;
1621    case IR_FPMATH: asm_fpmath(as, ir); break;
1622
1623    /* Overflow-checking arithmetic ops. */
1624    case IR_ADDOV: asm_addov(as, ir); break;
1625    case IR_SUBOV: asm_subov(as, ir); break;
1626    case IR_MULOV: asm_mulov(as, ir); break;
1627
1628    /* Memory references. */
1629    case IR_AREF: asm_aref(as, ir); break;
1630    case IR_HREF: asm_href(as, ir, 0); break;
1631    case IR_HREFK: asm_hrefk(as, ir); break;
1632    case IR_NEWREF: asm_newref(as, ir); break;
1633    case IR_UREFO: case IR_UREFC: asm_uref(as, ir); break;
1634    case IR_FREF: asm_fref(as, ir); break;
1635    case IR_STRREF: asm_strref(as, ir); break;
1636    case IR_LREF: asm_lref(as, ir); break;
1637
1638    /* Loads and stores. */
1639    case IR_ALOAD: case IR_HLOAD: case IR_ULOAD: case IR_VLOAD:
1640      asm_ahuvload(as, ir);
1641      break;
1642    case IR_FLOAD: asm_fload(as, ir); break;
1643    case IR_XLOAD: asm_xload(as, ir); break;
1644    case IR_SLOAD: asm_sload(as, ir); break;
1645
1646    case IR_ASTORE: case IR_HSTORE: case IR_USTORE: asm_ahustore(as, ir); break;
1647    case IR_FSTORE: asm_fstore(as, ir); break;
1648    case IR_XSTORE: asm_xstore(as, ir); break;
1649
1650    /* Allocations. */
1651    case IR_SNEW: case IR_XSNEW: asm_snew(as, ir); break;
1652    case IR_TNEW: asm_tnew(as, ir); break;
1653    case IR_TDUP: asm_tdup(as, ir); break;
1654    case IR_CNEW: case IR_CNEWI: asm_cnew(as, ir); break;
1655
1656    /* Buffer operations. */
1657    case IR_BUFHDR: asm_bufhdr(as, ir); break;
1658    case IR_BUFPUT: asm_bufput(as, ir); break;
1659    case IR_BUFSTR: asm_bufstr(as, ir); break;
1660
1661    /* Write barriers. */
1662    case IR_TBAR: asm_tbar(as, ir); break;
1663    case IR_OBAR: asm_obar(as, ir); break;
1664
1665    /* Type conversions. */
1666    case IR_TOBIT: asm_tobit(as, ir); break;
1667    case IR_CONV: asm_conv(as, ir); break;
1668    case IR_TOSTR: asm_tostr(as, ir); break;
1669    case IR_STRTO: asm_strto(as, ir); break;
1670
1671    /* Calls. */
1672    case IR_CALLA:
1673      as->gcsteps++;
1674      /* fallthrough */
1675    case IR_CALLN: case IR_CALLL: case IR_CALLS: asm_call(as, ir); break;
1676    case IR_CALLXS: asm_callx(as, ir); break;
1677    case IR_CARG: break;
1678
1679    default:
1680      setintV(&as->J->errinfo, ir->o);
1681      lj_trace_err_info(as->J, LJ_TRERR_NYIIR);
1682      break;
1683    }
1684  }
1685
1686  /* -- Head of trace ------------------------------------------------------ */
1687
1688  /* Head of a root trace. */
1689  static void asm_head_root(ASMState *as)
1690  {
1691    int32_t spadj;
1692    asm_head_root_base(as);
1693    emit_setvmstate(as, (int32_t)as->T->traceno);
1694    spadj = asm_stack_adjust(as);
```

```
  as->T->spadjust = (uint16_t)spadj;
  emit_spsub(as, spadj);
  /* Root traces assume a checked stack for the starting proto. */
  as->T->topslot = gcref(as->T->startpt)->pt.framesize;
}

/* Head of a side trace.
**
** The current simplistic algorithm requires that all slots inherited
** from the parent are live in a register between pass 2 and pass 3. This
** avoids the complexity of stack slot shuffling. But of course this may
** overflow the register set in some cases and cause the dreaded error:
** "NYI: register coalescing too complex". A refined algorithm is needed.
*/
static void asm_head_side(ASMState *as)
{
  IRRef1 sloadins[RID_MAX];
  RegSet allow = RSET_ALL;  /* Inverse of all coalesced registers. */
  RegSet live = RSET_EMPTY;  /* Live parent registers. */
  IRIns *irp = &as->parent->ir[REF_BASE];  /* Parent base. */
  int32_t spadj, spdelta;
  int pass2 = 0;
  int pass3 = 0;
  IRRef i;

  if (as->snapno && as->topslot > as->parent->topslot) {
    /* Force snap #0 alloc to prevent register overwrite in stack check. */
    as->snapno = 0;
    asm_snap_alloc(as);
  }
  allow = asm_head_side_base(as, irp, allow);

  /* Scan all parent SLOADs and collect register dependencies. */
  for (i = as->stopins; i > REF_BASE; i--) {
    IRIns *ir = IR(i);
    RegSP rs;
    lua_assert((ir->o == IR_SLOAD && (ir->op2 & IRSLOAD_PARENT)) ||
               (LJ_SOFTFP && ir->o == IR_HIOP) || ir->o == IR_PVAL);
    rs = as->parentmap[i - REF_FIRST];
    if (ra_hasreg(ir->r)) {
      rset_clear(allow, ir->r);
      if (ra_hasspill(ir->s)) {
        ra_save(as, ir, ir->r);
        checkmclim(as);
      }
    } else if (ra_hasspill(ir->s)) {
      irt_setmark(ir->t);
      pass2 = 1;
    }
    if (ir->r == rs) {  /* Coalesce matching registers right now. */
      ra_free(as, ir->r);
    } else if (ra_hasspill(regsp_spill(rs))) {
      if (ra_hasreg(ir->r))
        pass3 = 1;
    } else if (ra_used(ir)) {
      sloadins[rs] = (IRRef1)i;
      rset_set(live, rs);  /* Block live parent register. */
    }
  }

  /* Calculate stack frame adjustment. */
  spadj = asm_stack_adjust(as);
  spdelta = spadj - (int32_t)as->parent->spadjust;
  if (spdelta < 0) {  /* Don't shrink the stack frame. */
    spadj = (int32_t)as->parent->spadjust;
    spdelta = 0;
  }
  as->T->spadjust = (uint16_t)spadj;

  /* Reload spilled target registers. */
  if (pass2) {
    for (i = as->stopins; i > REF_BASE; i--) {
      IRIns *ir = IR(i);
      if (irt_ismarked(ir->t)) {
        RegSet mask;
        Reg r;
```

```
        RegSP rs;
        irt_clearmark(ir->t);
        rs = as->parentmap[i - REF_FIRST];
        if (!ra_hasspill(regsp_spill(rs)))
          ra_sethint(ir->r, rs);  /* Hint may be gone, set it again. */
        else if (sps_scale(regsp_spill(rs))+spdelta == sps_scale(ir->s))
          continue;  /* Same spill slot, do nothing. */
        mask = ((!LJ_SOFTFP && irt_isfp(ir->t)) ? RSET_FPR : RSET_GPR) & allow;
        if (mask == RSET_EMPTY)
          lj_trace_err(as->J, LJ_TRERR_NYICOAL);
        r = ra_allocref(as, i, mask);
        ra_save(as, ir, r);
        rset_clear(allow, r);
        if (r == rs) {  /* Coalesce matching registers right now. */
          ra_free(as, r);
          rset_clear(live, r);
        } else if (ra_hasspill(regsp_spill(rs))) {
          pass3 = 1;
        }
        checkmclim(as);
      }
    }
  }

  /* Store trace number and adjust stack frame relative to the parent. */
  emit_setvmstate(as, (int32_t)as->T->traceno);
  emit_spsub(as, spdelta);

#if !LJ_TARGET_X86ORX64
  /* Restore BASE register from parent spill slot. */
  if (ra_hasspill(irp->s))
    emit_spload(as, IR(REF_BASE), IR(REF_BASE)->r, sps_scale(irp->s));
#endif

  /* Restore target registers from parent spill slots. */
  if (pass3) {
    RegSet work = ~as->freeset & RSET_ALL;
    while (work) {
      Reg r = rset_pickbot(work);
      IRRef ref = regcost_ref(as->cost[r]);
      RegSP rs = as->parentmap[ref - REF_FIRST];
      rset_clear(work, r);
      if (ra_hasspill(regsp_spill(rs))) {
        int32_t ofs = sps_scale(regsp_spill(rs));
        ra_free(as, r);
        emit_spload(as, IR(ref), r, ofs);
        checkmclim(as);
      }
    }
  }

  /* Shuffle registers to match up target regs with parent regs. */
  for (;;) {
    RegSet work;

    /* Repeatedly coalesce free live registers by moving to their target. */
    while ((work = as->freeset & live) != RSET_EMPTY) {
      Reg rp = rset_pickbot(work);
      IRIns *ir = IR(sloadins[rp]);
      rset_clear(live, rp);
      rset_clear(allow, rp);
      ra_free(as, ir->r);
      emit_movrr(as, ir, ir->r, rp);
      checkmclim(as);
    }

    /* We're done if no live registers remain. */
    if (live == RSET_EMPTY)
      break;

    /* Break cycles by renaming one target to a temp. register. */
    if (live & RSET_GPR) {
      RegSet tmpset = as->freeset & ~live & allow & RSET_GPR;
      if (tmpset == RSET_EMPTY)
        lj_trace_err(as->J, LJ_TRERR_NYICOAL);
      ra_rename(as, rset_pickbot(live & RSET_GPR), rset_pickbot(tmpset));
```

```
    }
    if (!LJ_SOFTFP && (live & RSET_FPR)) {
      RegSet tmpset = as->freeset & ~live & allow & RSET_FPR;
      if (tmpset == RSET_EMPTY)
        lj_trace_err(as->J, LJ_TRERR_NYICOAL);
      ra_rename(as, rset_pickbot(live & RSET_FPR), rset_pickbot(tmpset));
    }
    checkmclim(as);
    /* Continue with coalescing to fix up the broken cycle(s). */
  }

  /* Inherit top stack slot already checked by parent trace. */
  as->T->topslot = as->parent->topslot;
  if (as->topslot > as->T->topslot) {  /* Need to check for higher slot? */
#ifdef EXITSTATE_CHECKEXIT
    /* Highest exit + 1 indicates stack check. */
    ExitNo exitno = as->T->nsnap;
#else
    /* Reuse the parent exit in the context of the parent trace. */
    ExitNo exitno = as->J->exitno;
#endif
    as->T->topslot = (uint8_t)as->topslot;  /* Remember for child traces. */
    asm_stack_check(as, as->topslot, irp, allow & RSET_GPR, exitno);
  }
}

/* -- Tail of trace ------------------------------------------------------ */

/* Get base slot for a snapshot. */
static BCReg asm_baseslot(ASMState *as, SnapShot *snap, int *gotframe)
{
  SnapEntry *map = &as->T->snapmap[snap->mapofs];
  MSize n;
  for (n = snap->nent; n > 0; n--) {
    SnapEntry sn = map[n-1];
    if ((sn & SNAP_FRAME)) {
      *gotframe = 1;
      return snap_slot(sn);
    }
  }
  return 0;
}

/* Link to another trace. */
static void asm_tail_link(ASMState *as)
{
  SnapNo snapno = as->T->nsnap-1;  /* Last snapshot. */
  SnapShot *snap = &as->T->snap[snapno];
  int gotframe = 0;
  BCReg baseslot = asm_baseslot(as, snap, &gotframe);

  as->topslot = snap->topslot;
  checkmclim(as);
  ra_allocref(as, REF_BASE, RID2RSET(RID_BASE));

  if (as->T->link == 0) {
    /* Setup fixed registers for exit to interpreter. */
    const BCIns *pc = snap_pc(as->T->snapmap[snap->mapofs + snap->nent]);
    int32_t mres;
    if (bc_op(*pc) == BC_JLOOP) {  /* NYI: find a better way to do this. */
      BCIns *retpc = &traceref(as->J, bc_d(*pc))->startins;
      if (bc_isret(bc_op(*retpc)))
        pc = retpc;
    }
    ra_allockreg(as, i32ptr(J2GG(as->J)->dispatch), RID_DISPATCH);
    ra_allockreg(as, i32ptr(pc), RID_LPC);
    mres = (int32_t)(snap->nslots - baseslot);
    switch (bc_op(*pc)) {
    case BC_CALLM: case BC_CALLMT:
      mres -= (int32_t)(1 + LJ_FR2 + bc_a(*pc) + bc_c(*pc)); break;
    case BC_RETM: mres -= (int32_t)(bc_a(*pc) + bc_d(*pc)); break;
    case BC_TSETM: mres -= (int32_t)bc_a(*pc); break;
    default: if (bc_op(*pc) < BC_FUNCF) mres = 0; break;
    }
    ra_allockreg(as, mres, RID_RET);  /* Return MULTRES or 0. */
  } else if (baseslot) {
```

```
    /* Save modified BASE for linking to trace with higher start frame. */
    emit_setgl(as, RID_BASE, jit_base);
  }
  emit_addptr(as, RID_BASE, 8*(int32_t)baseslot);

  /* Sync the interpreter state with the on-trace state. */
  asm_stack_restore(as, snap);

  /* Root traces that add frames need to check the stack at the end. */
  if (!as->parent && gotframe)
    asm_stack_check(as, as->topslot, NULL, as->freeset & RSET_GPR, snapno);
}

/* -- Trace setup --------------------------------------------------------- */

/* Clear reg/sp for all instructions and add register hints. */
static void asm_setup_regsp(ASMState *as)
{
  GCtrace *T = as->T;
  int sink = T->sinktags;
  IRRef nins = T->nins;
  IRIns *ir, *lastir;
  int inloop;
#if LJ_TARGET_ARM
  uint32_t rload = 0xa6402a64;
#endif

  ra_setup(as);

  /* Clear reg/sp for constants. */
  for (ir = IR(T->nk), lastir = IR(REF_BASE); ir < lastir; ir++)
    ir->prev = REGSP_INIT;

  /* REF_BASE is used for implicit references to the BASE register. */
  lastir->prev = REGSP_HINT(RID_BASE);

  ir = IR(nins-1);
  if (ir->o == IR_RENAME) {
    do { ir--; nins--; } while (ir->o == IR_RENAME);
    T->nins = nins;  /* Remove any renames left over from ASM restart. */
  }
  as->snaprename = nins;
  as->snapref = nins;
  as->snapno = T->nsnap;

  as->stopins = REF_BASE;
  as->orignins = nins;
  as->curins = nins;

  /* Setup register hints for parent link instructions. */
  ir = IR(REF_FIRST);
  if (as->parent) {
    uint16_t *p;
    lastir = lj_snap_regspmap(as->parent, as->J->exitno, ir);
    if (lastir - ir > LJ_MAX_JSLOTS)
      lj_trace_err(as->J, LJ_TRERR_NYICOAL);
    as->stopins = (IRRef)((lastir-1) - as->ir);
    for (p = as->parentmap; ir < lastir; ir++) {
      RegSP rs = ir->prev;
      *p++ = (uint16_t)rs;  /* Copy original parent RegSP to parentmap. */
      if (!ra_hasspill(regsp_spill(rs)))
        ir->prev = (uint16_t)REGSP_HINT(regsp_reg(rs));
      else
        ir->prev = REGSP_INIT;
    }
  }

  inloop = 0;
  as->evenspill = SPS_FIRST;
  for (lastir = IR(nins); ir < lastir; ir++) {
    if (sink) {
      if (ir->r == RID_SINK)
        continue;
      if (ir->r == RID_SUNK) {  /* Revert after ASM restart. */
        ir->r = RID_SINK;
        continue;
```

```
1999        }
2000      }
2001      switch (ir->o) {
2002      case IR_LOOP:
2003        inloop = 1;
2004        break;
2005  #if LJ_TARGET_ARM
2006      case IR_SLOAD:
2007        if (!((ir->op2 & IRSLOAD_TYPECHECK) || (ir+1)->o == IR_HIOP))
2008          break;
2009        /* fallthrough */
2010      case IR_ALOAD: case IR_HLOAD: case IR_ULOAD: case IR_VLOAD:
2011        if (!LJ_SOFTFP && irt_isnum(ir->t)) break;
2012        ir->prev = (uint16_t)REGSP_HINT((rload & 15));
2013        rload = lj_ror(rload, 4);
2014        continue;
2015  #endif
2016      case IR_CALLXS: {
2017        CCallInfo ci;
2018        ci.flags = asm_callx_flags(as, ir);
2019        ir->prev = asm_setup_call_slots(as, ir, &ci);
2020        if (inloop)
2021          as->modset |= RSET_SCRATCH;
2022        continue;
2023        }
2024      case IR_CALLN: case IR_CALLA: case IR_CALLL: case IR_CALLS: {
2025        const CCallInfo *ci = &lj_ir_callinfo[ir->op2];
2026        ir->prev = asm_setup_call_slots(as, ir, ci);
2027        if (inloop)
2028          as->modset |= (ci->flags & CCI_NOFPRCLOBBER) ?
2029                        (RSET_SCRATCH & ~RSET_FPR) : RSET_SCRATCH;
2030        continue;
2031        }
2032  #if LJ_SOFTFP || (LJ_32 && LJ_HASFFI)
2033      case IR_HIOP:
2034        switch ((ir-1)->o) {
2035  #if LJ_SOFTFP && LJ_TARGET_ARM
2036        case IR_SLOAD: case IR_ALOAD: case IR_HLOAD: case IR_ULOAD: case IR_VLOAD:
2037          if (ra_hashint((ir-1)->r)) {
2038            ir->prev = (ir-1)->prev + 1;
2039            continue;
2040          }
2041          break;
2042  #endif
2043  #if !LJ_SOFTFP && LJ_NEED_FP64
2044        case IR_CONV:
2045          if (irt_isfp((ir-1)->t)) {
2046            ir->prev = REGSP_HINT(RID_FPRET);
2047            continue;
2048          }
2049          /* fallthrough */
2050  #endif
2051        case IR_CALLN: case IR_CALLXS:
2052  #if LJ_SOFTFP
2053        case IR_MIN: case IR_MAX:
2054  #endif
2055          (ir-1)->prev = REGSP_HINT(RID_RETLO);
2056          ir->prev = REGSP_HINT(RID_RETHI);
2057          continue;
2058        default:
2059          break;
2060        }
2061        break;
2062  #endif
2063  #if LJ_SOFTFP
2064      case IR_MIN: case IR_MAX:
2065        if ((ir+1)->o != IR_HIOP) break;
2066        /* fallthrough */
2067  #endif
2068      /* C calls evict all scratch regs and return results in RID_RET. */
2069      case IR_SNEW: case IR_XSNEW: case IR_NEWREF: case IR_BUFPUT:
2070        if (REGARG_NUMGPR < 3 && as->evenspill < 3)
2071          as->evenspill = 3;  /* lj_str_new and lj_tab_newkey need 3 args. */
2072  #if LJ_TARGET_X86 && LJ_HASFFI
2073        if (0) {
2074      case IR_CNEW:
```

```
2075          if (ir->op2 != REF_NIL && as->evenspill < 4)
2076            as->evenspill = 4;  /* lj_cdata_newv needs 4 args. */
2077        }
2078  #else
2079      case IR_CNEW:
2080  #endif
2081      case IR_TNEW: case IR_TDUP: case IR_CNEWI: case IR_TOSTR:
2082      case IR_BUFSTR:
2083        ir->prev = REGSP_HINT(RID_RET);
2084        if (inloop)
2085          as->modset = RSET_SCRATCH;
2086        continue;
2087      case IR_STRTO: case IR_OBAR:
2088        if (inloop)
2089          as->modset = RSET_SCRATCH;
2090        break;
2091  #if !LJ_SOFTFP
2092      case IR_ATAN2:
2093  #if LJ_TARGET_X86
2094        if (as->evenspill < 4)  /* Leave room to call atan2(). */
2095          as->evenspill = 4;
2096  #endif
2097  #if !LJ_TARGET_X86ORX64
2098      case IR_LDEXP:
2099  #endif
2100  #endif
2101      case IR_POW:
2102        if (!LJ_SOFTFP && irt_isnum(ir->t)) {
2103          if (inloop)
2104            as->modset |= RSET_SCRATCH;
2105  #if LJ_TARGET_X86
2106          break;
2107  #else
2108          ir->prev = REGSP_HINT(RID_FPRET);
2109          continue;
2110  #endif
2111        }
2112        /* fallthrough for integer POW */
2113      case IR_DIV: case IR_MOD:
2114        if (!irt_isnum(ir->t)) {
2115          ir->prev = REGSP_HINT(RID_RET);
2116          if (inloop)
2117            as->modset |= (RSET_SCRATCH & RSET_GPR);
2118          continue;
2119        }
2120        break;
2121      case IR_FPMATH:
2122  #if LJ_TARGET_X86ORX64
2123        if (ir->op2 <= IRFPM_TRUNC) {
2124          if (!(as->flags & JIT_F_SSE4_1)) {
2125            ir->prev = REGSP_HINT(RID_XMM0);
2126            if (inloop)
2127              as->modset |= RSET_RANGE(RID_XMM0, RID_XMM3+1)|RID2RSET(RID_EAX);
2128            continue;
2129          }
2130          break;
2131        } else if (ir->op2 == IRFPM_EXP2 && !LJ_64) {
2132          if (as->evenspill < 4)  /* Leave room to call pow(). */
2133            as->evenspill = 4;
2134        }
2135  #endif
2136        if (inloop)
2137          as->modset |= RSET_SCRATCH;
2138  #if LJ_TARGET_X86
2139        break;
2140  #else
2141        ir->prev = REGSP_HINT(RID_FPRET);
2142        continue;
2143  #endif
2144  #if LJ_TARGET_X86ORX64
2145      /* Non-constant shift counts need to be in RID_ECX on x86/x64. */
2146      case IR_BSHL: case IR_BSHR: case IR_BSAR: case IR_BROL: case IR_BROR:
2147        if (!irref_isk(ir->op2) && !ra_hashint(IR(ir->op2)->r)) {
2148          IR(ir->op2)->r = REGSP_HINT(RID_ECX);
2149          if (inloop)
2150            rset_set(as->modset, RID_ECX);
```

```
      }
      break;
#endif
    /* Do not propagate hints across type conversions or loads. */
    case IR_TOBIT:
    case IR_XLOAD:
#if !LJ_TARGET_ARM
    case IR_ALOAD: case IR_HLOAD: case IR_ULOAD: case IR_VLOAD:
#endif
      break;
    case IR_CONV:
      if (irt_isfp(ir->t) || (ir->op2 & IRCONV_SRCMASK) == IRT_NUM ||
          (ir->op2 & IRCONV_SRCMASK) == IRT_FLOAT)
        break;
      /* fallthrough */
    default:
      /* Propagate hints across likely 'op reg, imm' or 'op reg'. */
      if (irref_isk(ir->op2) && !irref_isk(ir->op1) &&
          ra_hashint(regsp_reg(IR(ir->op1)->prev))) {
        ir->prev = IR(ir->op1)->prev;
        continue;
      }
      break;
    }
    ir->prev = REGSP_INIT;
  }
  if ((as->evenspill & 1))
    as->oddspill = as->evenspill++;
  else
    as->oddspill = 0;
}

/* -- Assembler core ------------------------------------------------------ */

/* Assemble a trace. */
void lj_asm_trace(jit_State *J, GCtrace *T)
{
  ASMState as_;
  ASMState *as = &as_;
  MCode *origtop;

  /* Ensure an initialized instruction beyond the last one for HIOP checks. */
  J->cur.nins = lj_ir_nextins(J);
  J->cur.ir[J->cur.nins].o = IR_NOP;

  /* Setup initial state. Copy some fields to reduce indirections. */
  as->J = J;
  as->T = T;
  as->ir = T->ir;
  as->flags = J->flags;
  as->loopref = J->loopref;
  as->realign = NULL;
  as->loopinv = 0;
  as->parent = J->parent ? traceref(J, J->parent) : NULL;

  /* Reserve MCode memory. */
  as->mctop = origtop = lj_mcode_reserve(J, &as->mcbot);
  as->mcp = as->mctop;
  as->mclim = as->mcbot + MCLIM_REDZONE;
  asm_setup_target(as);

  do {
    as->mcp = as->mctop;
#ifdef LUA_USE_ASSERT
    as->mcp_prev = as->mcp;
#endif
    as->curins = T->nins;
    RA_DBG_START();
    RA_DBGX((as, "===== STOP ====="));

    /* General trace setup. Emit tail of trace. */
    asm_tail_prep(as);
    as->mcloop = NULL;
    as->flagmcp = NULL;
    as->topslot = 0;
    as->gcsteps = 0;
```

```
2227      as->sectref = as->loopref;
2228      as->fuseref = (as->flags & JIT_F_OPT_FUSE) ? as->loopref : FUSE_DISABLED;
2229      asm_setup_regsp(as);
2230      if (!as->loopref)
2231        asm_tail_link(as);
2232
2233      /* Assemble a trace in linear backwards order. */
2234      for (as->curins--; as->curins > as->stopins; as->curins--) {
2235        IRIns *ir = IR(as->curins);
2236        lua_assert(!(LJ_32 && irt_isint64(ir->t)));  /* Handled by SPLIT. */
2237        if (!ra_used(ir) && !ir_sideeff(ir) && (as->flags & JIT_F_OPT_DCE))
2238          continue;  /* Dead-code elimination can be soooo easy. */
2239        if (irt_isguard(ir->t))
2240          asm_snap_prep(as);
2241        RA_DBG_REF();
2242        checkmclim(as);
2243        asm_ir(as, ir);
2244      }
2245    } while (as->realign);  /* Retry in case the MCode needs to be realigned. */
2246
2247    /* Emit head of trace. */
2248    RA_DBG_REF();
2249    checkmclim(as);
2250    if (as->gcsteps > 0) {
2251      as->curins = as->T->snap[0].ref;
2252      asm_snap_prep(as);  /* The GC check is a guard. */
2253      asm_gc_check(as);
2254    }
2255    ra_evictk(as);
2256    if (as->parent)
2257      asm_head_side(as);
2258    else
2259      asm_head_root(as);
2260    asm_phi_fixup(as);
2261
2262    RA_DBGX((as, "===== START ===="));
2263    RA_DBG_FLUSH();
2264    if (as->freeset != RSET_ALL)
2265      lj_trace_err(as->J, LJ_TRERR_BADRA);  /* Ouch! Should never happen. */
2266
2267    /* Set trace entry point before fixing up tail to allow link to self. */
2268    T->mcode = as->mcp;
2269    T->mcloop = as->mcloop ? (MSize)((char *)as->mcloop - (char *)as->mcp) : 0;
2270    if (!as->loopref)
2271      asm_tail_fixup(as, T->link);  /* Note: this may change as->mctop! */
2272    T->szmcode = (MSize)((char *)as->mctop - (char *)as->mcp);
2273    lj_mcode_sync(T->mcode, origtop);
2274  }
2275
2276  #undef IR
2277
2278  #endif
```

One Level Up Top Level

src/lj_ir.h - luajit-2.0-src

Global variables defined

- lj_ir_mode
- lj_ir_type_size

Data types defined

- IRFPMathOp
- IRFieldID
- IRIns
- IRIns
- IRMode
- IROp
- IROp1
- IROpT
- IRRef
- IRRef1
- IRRef2
- IRType
- IRType1
- IRType1
- TRef

Functions defined

- ir_sideeff
- irt_toitype_
- itype2irt

Macros defined

- FLENUM
- FLENUM
- FPMENUM
- FPMENUM
- IRBUFHDR_APPEND

- IRBUFHDR_RESET
- IRCONV_ANY
- IRCONV_CHECK
- IRCONV_CONVMASK
- IRCONV_CSH
- IRCONV_DSH
- IRCONV_DSTMASK
- IRCONV_INDEX
- IRCONV_INT_NUM
- IRCONV_MODEMASK
- IRCONV_NUM_INT
- IRCONV_SEXT
- IRCONV_SRCMASK
- IRCONV_TOBIT
- IRDEF
- IRDELTA_L2S
- IRENUM
- IRENUM
- IRFLDEF
- IRFPMDEF
- IRMODE
- IRM_A
- IRM_AW
- IRM_C
- IRM_CW
- IRM_L
- IRM_LW
- IRM_N
- IRM_NW
- IRM_R
- IRM_S
- IRM_W
- IRM___

- IRREF2
- IRSLOAD_CONVERT
- IRSLOAD_FRAME
- IRSLOAD_INHERIT
- IRSLOAD_PARENT
- IRSLOAD_READONLY
- IRSLOAD_TYPECHECK
- IRT
- IRTDEF
- IRTENUM
- IRTENUM
- IRTG
- IRTGI
- IRTI
- IRTN
- IRTOSTR_CHAR
- IRTOSTR_INT
- IRTOSTR_NUM
- IRT_IS64
- IRT_IS64
- IRT_IS64
- IRXLOAD_READONLY
- IRXLOAD_UNALIGNED
- IRXLOAD_VOLATILE
- TREF
- TREF_CONT
- TREF_FALSE
- TREF_FRAME
- TREF_NIL
- TREF_PRI
- TREF_REFMASK
- TREF_TRUE
- _LJ_IR_H

- ir_k64
- ir_kcdata
- ir_kfunc
- ir_kgc
- ir_kint64
- ir_knum
- ir_kptr
- ir_kstr
- ir_ktab
- irm_iscomm
- irm_kind
- irm_op1
- irm_op2
- irref_isk
- irt_clearmark
- irt_clearphi
- irt_is64
- irt_is64orfp
- irt_isaddr
- irt_iscdata
- irt_isfloat
- irt_isfp
- irt_isgcv
- irt_isguard
- irt_isi16
- irt_isi64
- irt_isi8
- irt_isint
- irt_isint64
- irt_isinteger
- irt_islightud
- irt_ismarked
- irt_isnil

- irt_isnum
- irt_isphi
- irt_ispri
- irt_isstr
- irt_istab
- irt_isu16
- irt_isu32
- irt_isu64
- irt_isu8
- irt_sametype
- irt_setmark
- irt_setphi
- irt_size
- irt_t
- irt_toitype
- irt_type
- irt_typerange
- irtype_ispri
- tref_isbool
- tref_iscdata
- tref_isfalse
- tref_isfunc
- tref_isgcv
- tref_isint
- tref_isinteger
- tref_isk
- tref_isk2
- tref_islightud
- tref_isnil
- tref_isnum
- tref_isnumber
- tref_isnumber_str
- tref_ispri

- tref_isstr
- tref_istab
- tref_istrue
- tref_istruecond
- tref_istype
- tref_isudata
- tref_ref
- tref_t
- tref_type
- tref_typerange

Source code

```
/*
** SSA IR (Intermediate Representation) format.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_IR_H
#define _LJ_IR_H

#include "lj_obj.h"

/* -- IR instructions ---------------------------------------------------- */

/* IR instruction definition. Order matters, see below. ORDER IR */
#define IRDEF(_) \
  /* Guarded assertions. */ \
  /* Must be properly aligned to flip opposites (^1) and (un)ordered (^4). */ \
  _(LT,                N , ref, ref) \
  _(GE,                N , ref, ref) \
  _(LE,                N , ref, ref) \
  _(GT,                N , ref, ref) \
  \
  _(ULT,         N , ref, ref) \
  _(UGE,         N , ref, ref) \
  _(ULE,         N , ref, ref) \
  _(UGT,         N , ref, ref) \
  \
  _(EQ,                C , ref, ref) \
  _(NE,                C , ref, ref) \
  \
  _(ABC,         N , ref, ref) \
  _(RETF,         S , ref, ref) \
  \
  /* Miscellaneous ops. */ \
  _(NOP,         N , ___, ___) \
  _(BASE,         N , lit, lit) \
  _(PVAL,         N , lit, ___) \
  _(GCSTEP,         S , ___, ___) \
  _(HIOP,         S , ref, ref) \
  _(LOOP,         S , ___, ___) \
  _(USE,         S , ref, ___) \
  _(PHI,         S , ref, ref) \
  _(RENAME,         S , ref, lit) \
  _(PROF,         S , ___, ___) \
  \
  /* Constants. */ \
  _(KPRI,         N , ___, ___) \
  _(KINT,         N , cst, ___) \
  _(KGC,         N , cst, ___) \
  _(KPTR,         N , cst, ___) \
  _(KKPTR,         N , cst, ___) \
```

```
  _(KNULL,        N , cst, ___) \
  _(KNUM,        N , cst, ___) \
  _(KINT64,       N , cst, ___) \
  _(KSLOT,        N , ref, lit) \
  \
  /* Bit ops. */ \
  _(BNOT,        N , ref, ___) \
  _(BSWAP,        N , ref, ___) \
  _(BAND,        C , ref, ref) \
  _(BOR,        C , ref, ref) \
  _(BXOR,        C , ref, ref) \
  _(BSHL,        N , ref, ref) \
  _(BSHR,        N , ref, ref) \
  _(BSAR,        N , ref, ref) \
  _(BROL,        N , ref, ref) \
  _(BROR,        N , ref, ref) \
  \
  /* Arithmetic ops. ORDER ARITH */ \
  _(ADD,        C , ref, ref) \
  _(SUB,        N , ref, ref) \
  _(MUL,        C , ref, ref) \
  _(DIV,        N , ref, ref) \
  _(MOD,        N , ref, ref) \
  _(POW,        N , ref, ref) \
  _(NEG,        N , ref, ref) \
  \
  _(ABS,        N , ref, ref) \
  _(ATAN2,        N , ref, ref) \
  _(LDEXP,        N , ref, ref) \
  _(MIN,        C , ref, ref) \
  _(MAX,        C , ref, ref) \
  _(FPMATH,        N , ref, lit) \
  \
  /* Overflow-checking arithmetic ops. */ \
  _(ADDOV,        CW, ref, ref) \
  _(SUBOV,        NW, ref, ref) \
  _(MULOV,        CW, ref, ref) \
  \
  /* Memory ops. A = array, H = hash, U = upvalue, F = field, S = stack. */ \
  \
  /* Memory references. */ \
  _(AREF,        R , ref, ref) \
  _(HREFK,        R , ref, ref) \
  _(HREF,        L , ref, ref) \
  _(NEWREF,        S , ref, ref) \
  _(UREFO,        LW, ref, lit) \
  _(UREFC,        LW, ref, lit) \
  _(FREF,        R , ref, lit) \
  _(STRREF,        N , ref, ref) \
   _(LREF,        L , ___, ___) \
   \
   /* Loads and Stores. These must be in the same order. */ \
   _(ALOAD,        L , ref, ___) \
   _(HLOAD,        L , ref, ___) \
   _(ULOAD,        L , ref, ___) \
   _(FLOAD,        L , ref, lit) \
   _(XLOAD,        L , ref, lit) \
   _(SLOAD,        L , lit, lit) \
   _(VLOAD,        L , ref, ___) \
   \
   _(ASTORE,        S , ref, ref) \
   _(HSTORE,        S , ref, ref) \
   _(USTORE,        S , ref, ref) \
   _(FSTORE,        S , ref, ref) \
   _(XSTORE,        S , ref, ref) \
   \
   /* Allocations. */ \
   _(SNEW,        N , ref, ref)  /* CSE is ok, not marked as A. */ \
   _(XSNEW,        A , ref, ref) \
   _(TNEW,        AW, lit, lit) \
   _(TDUP,        AW, ref, ___) \
   _(CNEW,        AW, ref, ref) \
   _(CNEWI,        NW, ref, ref)  /* CSE is ok, not marked as A. */ \
   \
   /* Buffer operations. */ \
   _(BUFHDR,        L , ref, lit) \
```

```
  _(BUFPUT,        L , ref, ref) \
  _(BUFSTR,        A , ref, ref) \
  \
  /* Barriers. */ \
  _(TBAR,        S , ref, ___) \
  _(OBAR,        S , ref, ref) \
  _(XBAR,        S , ___, ___) \
  \
  /* Type conversions. */ \
  _(CONV,        NW, ref, lit) \
  _(TOBIT,        N , ref, ref) \
  _(TOSTR,        N , ref, lit) \
  _(STRTO,        N , ref, ___) \
  \
  /* Calls. */ \
  _(CALLN,        N , ref, lit) \
  _(CALLA,        A , ref, lit) \
  _(CALLL,        L , ref, lit) \
  _(CALLS,        S , ref, lit) \
  _(CALLXS,        S , ref, ref) \
  _(CARG,        N , ref, ref) \
  \
  /* End of list. */

/* IR opcodes (max. 256). */
typedef enum {
#define IRENUM(name, m, m1, m2)        IR_##name,
IRDEF(IRENUM)
#undef IRENUM
  IR__MAX
} IROp;

/* Stored opcode. */
typedef uint8_t IROp1;

LJ_STATIC_ASSERT(((int)IR_EQ^1) == (int)IR_NE);
LJ_STATIC_ASSERT(((int)IR_LT^1) == (int)IR_GE);
LJ_STATIC_ASSERT(((int)IR_LE^1) == (int)IR_GT);
LJ_STATIC_ASSERT(((int)IR_LT^3) == (int)IR_GT);
LJ_STATIC_ASSERT(((int)IR_LT^4) == (int)IR_ULT);

/* Delta between xLOAD and xSTORE. */
#define IRDELTA_L2S                ((int)IR_ASTORE - (int)IR_ALOAD)

LJ_STATIC_ASSERT((int)IR_HLOAD + IRDELTA_L2S == (int)IR_HSTORE);
LJ_STATIC_ASSERT((int)IR_ULOAD + IRDELTA_L2S == (int)IR_USTORE);
LJ_STATIC_ASSERT((int)IR_FLOAD + IRDELTA_L2S == (int)IR_FSTORE);
LJ_STATIC_ASSERT((int)IR_XLOAD + IRDELTA_L2S == (int)IR_XSTORE);

/* -- Named IR literals ---------------------------------------------------- */

/* FPMATH sub-functions. ORDER FPM. */
#define IRFPMDEF(_) \
  _(FLOOR) _(CEIL) _(TRUNC)  /* Must be first and in this order. */ \
  _(SQRT) _(EXP) _(EXP2) _(LOG) _(LOG2) _(LOG10) \
  _(SIN) _(COS) _(TAN) \
  _(OTHER)

typedef enum {
#define FPMENUM(name)                IRFPM_##name,
IRFPMDEF(FPMENUM)
#undef FPMENUM
  IRFPM__MAX
} IRFPMathOp;

/* FLOAD fields. */
#define IRFLDEF(_) \
  _(STR_LEN,        offsetof(GCstr, len)) \
  _(FUNC_ENV,        offsetof(GCfunc, l.env)) \
  _(FUNC_PC,        offsetof(GCfunc, l.pc)) \
  _(FUNC_FFID,        offsetof(GCfunc, l.ffid)) \
  _(THREAD_ENV,        offsetof(lua_State, env)) \
  _(TAB_META,        offsetof(GCtab, metatable)) \
  _(TAB_ARRAY,        offsetof(GCtab, array)) \
  _(TAB_NODE,        offsetof(GCtab, node)) \
  _(TAB_ASIZE,        offsetof(GCtab, asize)) \
```

```
  _(TAB_HMASK,        offsetof(GCtab, hmask)) \
  _(TAB_NOMM,        offsetof(GCtab, nomm)) \
  _(UDATA_META,       offsetof(GCudata, metatable)) \
  _(UDATA_UDTYPE, offsetof(GCudata, udtype)) \
  _(UDATA_FILE,       sizeof(GCudata)) \
  _(CDATA_CTYPEID, offsetof(GCcdata, ctypeid)) \
  _(CDATA_PTR,        sizeof(GCcdata)) \
  _(CDATA_INT, sizeof(GCcdata)) \
  _(CDATA_INT64, sizeof(GCcdata)) \
  _(CDATA_INT64_4, sizeof(GCcdata) + 4)

typedef enum {
#define FLENUM(name, ofs)       IRFL_##name,
IRFLDEF(FLENUM)
#undef FLENUM
  IRFL__MAX
} IRFieldID;

/* SLOAD mode bits, stored in op2. */
#define IRSLOAD_PARENT               0x01       /* Coalesce with parent trace. */
#define IRSLOAD_FRAME               0x02       /* Load hiword of frame. */
#define IRSLOAD_TYPECHECK       0x04       /* Needs type check. */
#define IRSLOAD_CONVERT              0x08       /* Number to integer conversion. */
#define IRSLOAD_READONLY       0x10       /* Read-only, omit slot store. */
#define IRSLOAD_INHERIT              0x20       /* Inherited by exits/side traces. */

/* XLOAD mode, stored in op2. */
#define IRXLOAD_READONLY       1       /* Load from read-only data. */
#define IRXLOAD_VOLATILE       2       /* Load from volatile data. */
#define IRXLOAD_UNALIGNED       4       /* Unaligned load. */

/* BUFHDR mode, stored in op2. */
#define IRBUFHDR_RESET              0       /* Reset buffer. */
#define IRBUFHDR_APPEND              1       /* Append to buffer. */

/* CONV mode, stored in op2. */
#define IRCONV_SRCMASK              0x001f       /* Source IRType. */
#define IRCONV_DSTMASK              0x03e0       /* Dest. IRType (also in ir->t). */
#define IRCONV_DSH              5
#define IRCONV_NUM_INT              ((IRT_NUM<<IRCONV_DSH)|IRT_INT)
#define IRCONV_INT_NUM              ((IRT_INT<<IRCONV_DSH)|IRT_NUM)
#define IRCONV_SEXT              0x0800       /* Sign-extend integer to integer. */
#define IRCONV_MODEMASK              0x0fff
#define IRCONV_CONVMASK              0xf000
#define IRCONV_CSH              12
/* Number to integer conversion mode. Ordered by strength of the checks. */
#define IRCONV_TOBIT  (0<<IRCONV_CSH)       /* None. Cache only: TOBIT conv. */
#define IRCONV_ANY    (1<<IRCONV_CSH)       /* Any FP number is ok. */
#define IRCONV_INDEX  (2<<IRCONV_CSH)       /* Check + special backprop rules. */
#define IRCONV_CHECK  (3<<IRCONV_CSH)       /* Number checked for integerness. */

/* TOSTR mode, stored in op2. */
#define IRTOSTR_INT              0       /* Convert integer to string. */
#define IRTOSTR_NUM              1       /* Convert number to string. */
#define IRTOSTR_CHAR              2       /* Convert char value to string. */

/* -- IR operands ---------------------------------------------------------- */

/* IR operand mode (2 bit). */
typedef enum {
  IRMref,               /* IR reference. */
  IRMlit,               /* 16 bit unsigned literal. */
  IRMcst,               /* Constant literal: i, gcr or ptr. */
  IRMnone               /* Unused operand. */
} IRMode;
#define IRM___               IRMnone

/* Mode bits: Commutative, {Normal/Ref, Alloc, Load, Store}, Non-weak guard. */
#define IRM_C                       0x10

#define IRM_N                       0x00
#define IRM_R                       IRM_N
#define IRM_A                       0x20
#define IRM_L                       0x40
#define IRM_S                       0x60

```

```
#define IRM_W                        0x80

#define IRM_NW                        (IRM_N|IRM_W)
#define IRM_CW                        (IRM_C|IRM_W)
#define IRM_AW                        (IRM_A|IRM_W)
#define IRM_LW                        (IRM_L|IRM_W)

#define irm_op1(m)               ((IRMode)((m)&3))
#define irm_op2(m)               ((IRMode)(((m)>>2)&3))
#define irm_iscomm(m)               ((m) & IRM_C)
#define irm_kind(m)               ((m) & IRM_S)

#define IRMODE(name, m, m1, m2)        (((IRM##m1)|((IRM##m2)<<2)|(IRM_##m))^IRM_W),

LJ_DATA const uint8_t lj_ir_mode[IR__MAX+1];

/* -- IR instruction types ----------------------------------------------- */

/* Map of itypes to non-negative numbers. ORDER LJ_T.
** LJ_TUPVAL/LJ_TTRACE never appear in a TValue. Use these itypes for
** IRT_P32 and IRT_P64, which never escape the IR.
** The various integers are only used in the IR and can only escape to
** a TValue after implicit or explicit conversion. Their types must be
** contiguous and next to IRT_NUM (see the typerange macros below).
*/
#define IRTDEF(_) \
  _(NIL, 4) _(FALSE, 4) _(TRUE, 4) _(LIGHTUD, LJ_64 ? 8 : 4) _(STR, 4) \
  _(P32, 4) _(THREAD, 4) _(PROTO, 4) _(FUNC, 4) _(P64, 8) _(CDATA, 4) \
  _(TAB, 4) _(UDATA, 4) \
  _(FLOAT, 4) _(NUM, 8) _(I8, 1) _(U8, 1) _(I16, 2) _(U16, 2) \
  _(INT, 4) _(U32, 4) _(I64, 8) _(U64, 8) \
  _(SOFTFP, 4)  /* There is room for 9 more types. */

/* IR result type and flags (8 bit). */
typedef enum {
#define IRTENUM(name, size)        IRT_##name,
IRTDEF(IRTENUM)
#undef IRTENUM
  IRT__MAX,

  /* Native pointer type and the corresponding integer type. */
  IRT_PTR = LJ_64 ? IRT_P64 : IRT_P32,
  IRT_INTP = LJ_64 ? IRT_I64 : IRT_INT,
  IRT_UINTP = LJ_64 ? IRT_U64 : IRT_U32,
  /* TODO_GC64: major changes required for all uses of IRT_P32. */

  /* Additional flags. */
  IRT_MARK = 0x20,        /* Marker for misc. purposes. */
  IRT_ISPHI = 0x40,        /* Instruction is left or right PHI operand. */
  IRT_GUARD = 0x80,        /* Instruction is a guard. */

  /* Masks. */
  IRT_TYPE = 0x1f,
  IRT_T = 0xff
} IRType;

#define irtype_ispri(irt)        ((uint32_t)(irt) <= IRT_TRUE)

/* Stored IRType. */
typedef struct IRType1 { uint8_t irt; } IRType1;

#define IRT(o, t)               ((uint32_t)(((o)<<8) | (t)))
#define IRTI(o)                      (IRT((o), IRT_INT))
#define IRTN(o)                      (IRT((o), IRT_NUM))
#define IRTG(o, t)                (IRT((o), IRT_GUARD|(t)))
#define IRTGI(o)                (IRT((o), IRT_GUARD|IRT_INT))

#define irt_t(t)                ((IRType)(t).irt)
#define irt_type(t)                ((IRType)((t).irt & IRT_TYPE))
#define irt_sametype(t1, t2)        ((((t1).irt ^ (t2).irt) & IRT_TYPE) == 0)
#define irt_typerange(t, first, last) \
  ((uint32_t)((t).irt & IRT_TYPE) - (uint32_t)(first) <= (uint32_t)(last-first))

#define irt_isnil(t)                (irt_type(t) == IRT_NIL)
#define irt_ispri(t)                ((uint32_t)irt_type(t) <= IRT_TRUE)
#define irt_islightud(t)        (irt_type(t) == IRT_LIGHTUD)
```

```
#define irt_isstr(t)		(irt_type(t) == IRT_STR)
#define irt_istab(t)		(irt_type(t) == IRT_TAB)
#define irt_iscdata(t)		(irt_type(t) == IRT_CDATA)
#define irt_isfloat(t)		(irt_type(t) == IRT_FLOAT)
#define irt_isnum(t)		(irt_type(t) == IRT_NUM)
#define irt_isint(t)		(irt_type(t) == IRT_INT)
#define irt_isi8(t)		(irt_type(t) == IRT_I8)
#define irt_isu8(t)		(irt_type(t) == IRT_U8)
#define irt_isi16(t)		(irt_type(t) == IRT_I16)
#define irt_isu16(t)		(irt_type(t) == IRT_U16)
#define irt_isu32(t)		(irt_type(t) == IRT_U32)
#define irt_isi64(t)		(irt_type(t) == IRT_I64)
#define irt_isu64(t)		(irt_type(t) == IRT_U64)

#define irt_isfp(t)		(irt_isnum(t) || irt_isfloat(t))
#define irt_isinteger(t)	(irt_typerange((t), IRT_I8, IRT_INT))
#define irt_isgcv(t)		(irt_typerange((t), IRT_STR, IRT_UDATA))
#define irt_isaddr(t)		(irt_typerange((t), IRT_LIGHTUD, IRT_UDATA))
#define irt_isint64(t)		(irt_typerange((t), IRT_I64, IRT_U64))

#if LJ_GC64
#define IRT_IS64 \
  ((1u<<IRT_NUM)|(1u<<IRT_I64)|(1u<<IRT_U64)|(1u<<IRT_P64)|\
   (1u<<IRT_LIGHTUD)|(1u<<IRT_STR)|(1u<<IRT_THREAD)|(1u<<IRT_PROTO)|\
   (1u<<IRT_FUNC)|(1u<<IRT_CDATA)|(1u<<IRT_TAB)|(1u<<IRT_UDATA))
#elif LJ_64
#define IRT_IS64 \
  ((1u<<IRT_NUM)|(1u<<IRT_I64)|(1u<<IRT_U64)|(1u<<IRT_P64)|(1u<<IRT_LIGHTUD))
#else
#define IRT_IS64 \
  ((1u<<IRT_NUM)|(1u<<IRT_I64)|(1u<<IRT_U64))
#endif

#define irt_is64(t)		((IRT_IS64 >> irt_type(t)) & 1)
#define irt_is64orfp(t)		(((IRT_IS64|(1u<<IRT_FLOAT))>>irt_type(t)) & 1)

#define irt_size(t)		(lj_ir_type_size[irt_t((t))])

LJ_DATA const uint8_t lj_ir_type_size[];

static LJ_AINLINE IRType itype2irt(const TValue *tv)
{
  if (tvisint(tv))
    return IRT_INT;
  else if (tvisnum(tv))
    return IRT_NUM;
#if LJ_64 && !LJ_GC64
  else if (tvislightud(tv))
    return IRT_LIGHTUD;
#endif
  else
    return (IRType)~itype(tv);
}

static LJ_AINLINE uint32_t irt_toitype_(IRType t)
{
  lua_assert(!LJ_64 || t != IRT_LIGHTUD);
  if (LJ_DUALNUM && t > IRT_NUM) {
    return LJ_TISNUM;
  } else {
    lua_assert(t <= IRT_NUM);
    return ~(uint32_t)t;
  }
}

#define irt_toitype(t)		irt_toitype_(irt_type((t)))

#define irt_isguard(t)		((t).irt & IRT_GUARD)
#define irt_ismarked(t)		((t).irt & IRT_MARK)
#define irt_setmark(t)		((t).irt |= IRT_MARK)
#define irt_clearmark(t)	((t).irt &= ~IRT_MARK)
#define irt_isphi(t)		((t).irt & IRT_ISPHI)
#define irt_setphi(t)		((t).irt |= IRT_ISPHI)
#define irt_clearphi(t)		((t).irt &= ~IRT_ISPHI)

/* Stored combined IR opcode and type. */
```

```
typedef uint16_t IROpT;

/* -- IR references ------------------------------------------------------ */

/* IR references. */
typedef uint16_t IRRef1;        /* One stored reference. */
typedef uint32_t IRRef2;        /* Two stored references. */
typedef uint32_t IRRef;                /* Used to pass around references. */

/* Fixed references. */
enum {
  REF_BIAS =        0x8000,
  REF_TRUE =        REF_BIAS-3,
  REF_FALSE =        REF_BIAS-2,
  REF_NIL =        REF_BIAS-1,        /* \--- Constants grow downwards. */
  REF_BASE =        REF_BIAS,        /* /--- IR grows upwards. */
  REF_FIRST =        REF_BIAS+1,
  REF_DROP =        0xffff
};

/* Note: IRMlit operands must be < REF_BIAS, too!
** This allows for fast and uniform manipulation of all operands
** without looking up the operand mode in lj_ir_mode:
** - CSE calculates the maximum reference of two operands.
**   This must work with mixed reference/literal operands, too.
** - DCE marking only checks for operand >= REF_BIAS.
** - LOOP needs to substitute reference operands.
**   Constant references and literals must not be modified.
*/

#define IRREF2(lo, hi)                ((IRRef2)(lo) | ((IRRef2)(hi) << 16))

#define irref_isk(ref)                ((ref) < REF_BIAS)

/* Tagged IR references (32 bit).
**
** +-------+-------+---------------+
** |  irt  | flags |      ref      |
** +-------+-------+---------------+
**
** The tag holds a copy of the IRType and speeds up IR type checks.
*/
typedef uint32_t TRef;

#define TREF_REFMASK                0x0000ffff
#define TREF_FRAME                0x00010000
#define TREF_CONT                0x00020000

#define TREF(ref, t)                ((TRef)((ref) + ((t)<<24)))

#define tref_ref(tr)                ((IRRef1)(tr))
#define tref_t(tr)                ((IRType)((tr)>>24))
#define tref_type(tr)                ((IRType)(((tr)>>24) & IRT_TYPE))
#define tref_typerange(tr, first, last) \
  ((((tr)>>24) & IRT_TYPE) - (TRef)(first) <= (TRef)(last-first))

#define tref_istype(tr, t)        (((tr) & (IRT_TYPE<<24)) == ((t)<<24))
#define tref_isnil(tr)                (tref_istype((tr), IRT_NIL))
#define tref_isfalse(tr)        (tref_istype((tr), IRT_FALSE))
#define tref_istrue(tr)                (tref_istype((tr), IRT_TRUE))
#define tref_islightud(tr)        (tref_istype((tr), IRT_LIGHTUD))
#define tref_isstr(tr)                (tref_istype((tr), IRT_STR))
#define tref_isfunc(tr)                (tref_istype((tr), IRT_FUNC))
#define tref_iscdata(tr)        (tref_istype((tr), IRT_CDATA))
#define tref_istab(tr)                (tref_istype((tr), IRT_TAB))
#define tref_isudata(tr)        (tref_istype((tr), IRT_UDATA))
#define tref_isnum(tr)                (tref_istype((tr), IRT_NUM))
#define tref_isint(tr)                (tref_istype((tr), IRT_INT))

#define tref_isbool(tr)                (tref_typerange((tr), IRT_FALSE, IRT_TRUE))
#define tref_ispri(tr)                (tref_typerange((tr), IRT_NIL, IRT_TRUE))
#define tref_istruecond(tr)        (!tref_typerange((tr), IRT_NIL, IRT_FALSE))
#define tref_isinteger(tr)        (tref_typerange((tr), IRT_I8, IRT_INT))
#define tref_isnumber(tr)        (tref_typerange((tr), IRT_NUM, IRT_INT))
#define tref_isnumber_str(tr)        (tref_isnumber((tr)) || tref_isstr((tr)))
#define tref_isgcv(tr)                (tref_typerange((tr), IRT_STR, IRT_UDATA))
```

```
507
508 #define tref_isk(tr)               (irref_isk(tref_ref((tr))))
509 #define tref_isk2(tr1, tr2)        (irref_isk(tref_ref((tr1) | (tr2))))
510
511 #define TREF_PRI(t)               (TREF(REF_NIL-(t), (t)))
512 #define TREF_NIL                (TREF_PRI(IRT_NIL))
513 #define TREF_FALSE               (TREF_PRI(IRT_FALSE))
514 #define TREF_TRUE                (TREF_PRI(IRT_TRUE))
515
516 /* -- IR format ------------------------------------------------------------ */
517
518 /* IR instruction format (64 bit).
519 **
520 **    16      16     8   8   8   8
521 ** +-------+-------+---+---+---+---+
522 ** |  op1  |  op2  | t | o | r | s |
523 ** +-------+-------+---+---+---+---+
524 ** |  op12/i/gco   |   ot  | prev  | (alternative fields in union)
525 ** +---------------+-------+-------+
526 **        32           16      16
527 **
528 ** prev is only valid prior to register allocation and then reused for r + s.
529 */
530
531 typedef union IRIns {
532   struct {
533     LJ_ENDIAN_LOHI(
534       IRRef1 op1;         /* IR operand 1. */
535     , IRRef1 op2;         /* IR operand 2. */
536     )
537     IROpT ot;                /* IR opcode and type (overlaps t and o). */
538     IRRef1 prev;        /* Previous ins in same chain (overlaps r and s). */
539   };
540   struct {
541     IRRef2 op12;        /* IR operand 1 and 2 (overlaps op1 and op2). */
542     LJ_ENDIAN_LOHI(
543       IRType1 t;        /* IR type. */
544     , IROp1 o;                /* IR opcode. */
545     )
546     LJ_ENDIAN_LOHI(
547       uint8_t r;        /* Register allocation (overlaps prev). */
548     , uint8_t s;        /* Spill slot allocation (overlaps prev). */
549     )
550   };
551   int32_t i;               /* 32 bit signed integer literal (overlaps op12). */
552   GCRef gcr;               /* GCobj constant (overlaps op12). */
553   MRef ptr;               /* Pointer constant (overlaps op12). */
554 } IRIns;
555
556 /* TODO_GC64: major changes required. */
557 #define ir_kgc(ir)        check_exp((ir)->o == IR_KGC, gcref((ir)->gcr))
558 #define ir_kstr(ir)        (gco2str(ir_kgc((ir))))
559 #define ir_ktab(ir)        (gco2tab(ir_kgc((ir))))
560 #define ir_kfunc(ir)        (gco2func(ir_kgc((ir))))
561 #define ir_kcdata(ir)        (gco2cd(ir_kgc((ir))))
562 #define ir_knum(ir)        check_exp((ir)->o == IR_KNUM, mref((ir)->ptr, cTValue))
563 #define ir_kint64(ir)        check_exp((ir)->o == IR_KINT64, mref((ir)->ptr,cTValue))
564 #define ir_k64(ir) \
565   check_exp((ir)->o == IR_KNUM || (ir)->o == IR_KINT64, mref((ir)->ptr,cTValue))
566 #define ir_kptr(ir) \
567   check_exp((ir)->o == IR_KPTR || (ir)->o == IR_KKPTR, mref((ir)->ptr, void))
568
569 /* A store or any other op with a non-weak guard has a side-effect. */
570 static LJ_AINLINE int ir_sideeff(IRIns *ir)
571 {
572   return (((ir->t.irt | ~IRT_GUARD) & lj_ir_mode[ir->o]) >= IRM_S);
573 }
574
575 LJ_STATIC_ASSERT((int)IRT_GUARD == (int)IRM_W);
576
577 #endif
```

One Level Up Top Level

src/lj_target_arm64.h - luajit-2.0-src

Data types defined

- A64Ins
- A64Ins

Macros defined

- A64F_A
- A64F_D
- A64F_M
- A64F_N
- A64F_S19
- A64F_S26
- A64F_U16
- FPRDEF
- GPRDEF
- REGARG_FIRSTFPR
- REGARG_FIRSTGPR
- REGARG_LASTFPR
- REGARG_LASTGPR
- REGARG_NUMFPR
- REGARG_NUMGPR
- RIDENUM
- RID_MIN_KREF
- RID_NUM_KREF
- RSET_ALL
- RSET_FIXED
- RSET_FPR
- RSET_GPR
- RSET_INIT
- RSET_SCRATCH
- RSET_SCRATCH_FPR
- RSET_SCRATCH_GPR

- VRIDDEF
- _LJ_TARGET_ARM64_H

Source code

```
/*
** Definitions for ARM64 CPUs.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_TARGET_ARM64_H
#define _LJ_TARGET_ARM64_H

/* -- Registers IDs ------------------------------------------------------- */

#define GPRDEF(_) \
  _(X0) _(X1) _(X2) _(X3) _(X4) _(X5) _(X6) _(X7) \
  _(X8) _(X9) _(X10) _(X11) _(X12) _(X13) _(X14) _(X15) \
  _(X16) _(X17) _(X18) _(X19) _(X20) _(X21) _(X22) _(X23) \
  _(X24) _(X25) _(X26) _(X27) _(X28) _(FP) _(LR) _(SP)
#define FPRDEF(_) \
  _(D0) _(D1) _(D2) _(D3) _(D4) _(D5) _(D6) _(D7) \
  _(D8) _(D9) _(D10) _(D11) _(D12) _(D13) _(D14) _(D15) \
  _(D16) _(D17) _(D18) _(D19) _(D20) _(D21) _(D22) _(D23) \
  _(D24) _(D25) _(D26) _(D27) _(D28) _(D29) _(D30) _(D31)
#define VRIDDEF(_)

#define RIDENUM(name)        RID_##name,

enum {
  GPRDEF(RIDENUM)               /* General-purpose registers (GPRs). */
  FPRDEF(RIDENUM)               /* Floating-point registers (FPRs). */
  RID_MAX,
  RID_TMP = RID_LR,
  RID_ZERO = RID_SP,

  /* Calling conventions. */
  RID_RET = RID_X0,
  RID_FPRET = RID_D0,

  /* These definitions must match with the *.dasc file(s): */
  RID_BASE = RID_X19,               /* Interpreter BASE. */
  RID_LPC = RID_X21,               /* Interpreter PC. */
  RID_GL = RID_X22,               /* Interpreter GL. */
  RID_LREG = RID_X23,               /* Interpreter L. */

  /* Register ranges [min, max) and number of registers. */
  RID_MIN_GPR = RID_X0,
  RID_MAX_GPR = RID_SP+1,
  RID_MIN_FPR = RID_MAX_GPR,
  RID_MAX_FPR = RID_D31+1,
  RID_NUM_GPR = RID_MAX_GPR - RID_MIN_GPR,
  RID_NUM_FPR = RID_MAX_FPR - RID_MIN_FPR
};

#define RID_NUM_KREF               RID_NUM_GPR
#define RID_MIN_KREF               RID_X0

/* -- Register sets ------------------------------------------------------- */

/* Make use of all registers, except for x18, fp, lr and sp. */
#define RSET_FIXED \
  (RID2RSET(RID_X18)|RID2RSET(RID_FP)|RID2RSET(RID_LR)|RID2RSET(RID_SP))
#define RSET_GPR        (RSET_RANGE(RID_MIN_GPR, RID_MAX_GPR) - RSET_FIXED)
#define RSET_FPR        RSET_RANGE(RID_MIN_FPR, RID_MAX_FPR)
#define RSET_ALL        (RSET_GPR|RSET_FPR)
#define RSET_INIT        RSET_ALL

/* lr is an implicit scratch register. */
#define RSET_SCRATCH_GPR        (RSET_RANGE(RID_X0, RID_X17+1))
#define RSET_SCRATCH_FPR \
  (RSET_RANGE(RID_D0, RID_D7+1)|RSET_RANGE(RID_D16, RID_D31+1))
#define RSET_SCRATCH               (RSET_SCRATCH_GPR|RSET_SCRATCH_FPR)
```

```
#define REGARG_FIRSTGPR                RID_X0
#define REGARG_LASTGPR                 RID_X7
#define REGARG_NUMGPR                  8
#define REGARG_FIRSTFPR                RID_D0
#define REGARG_LASTFPR                 RID_D7
#define REGARG_NUMFPR                  8

/* -- Instructions ---------------------------------------------------- */

/* Instruction fields. */
#define A64F_D(r)         (r)
#define A64F_N(r)         ((r) << 5)
#define A64F_A(r)         ((r) << 10)
#define A64F_M(r)         ((r) << 16)
#define A64F_U16(x)         ((x) << 5)
#define A64F_S26(x)         (x)
#define A64F_S19(x)         ((x) << 5)

typedef enum A64Ins {
  A64I_MOVZw = 0x52800000,
  A64I_MOVZx = 0xd2800000,
  A64I_LDRLw = 0x18000000,
  A64I_LDRLx = 0x58000000,
  A64I_NOP = 0xd503201f,
  A64I_B = 0x14000000,
  A64I_BR = 0xd61f0000,
} A64Ins;

#endif
```

One Level Up Top Level

src/lj_mcode.c - luajit-2.0-src

Data types defined

- MCLink
- MCLink

Functions defined

- lj_mcode_abort
- lj_mcode_commit
- lj_mcode_free
- lj_mcode_limiterr
- lj_mcode_patch
- lj_mcode_reserve
- lj_mcode_sync
- mcode_alloc
- mcode_alloc
- mcode_alloc_at
- mcode_alloc_at
- mcode_alloc_at
- mcode_allocarea
- mcode_free
- mcode_free
- mcode_free
- mcode_protect
- mcode_protect
- mcode_protfail
- mcode_setprot
- mcode_setprot

Macros defined

- LUAJIT_UNPROTECT_MCODE
- LUA_CORE
- MAP_ANONYMOUS
- MCPROT_GEN

- MCPROT_GEN
- MCPROT_RUN
- MCPROT_RUN
- MCPROT_RW
- MCPROT_RW
- MCPROT_RW
- MCPROT_RWX
- MCPROT_RWX
- MCPROT_RWX
- MCPROT_RX
- MCPROT_RX
- MCPROT_RX
- WIN32_LEAN_AND_MEAN
- lj_mcode_c
- mcode_validptr
- mcode_validptr

Source code

```
/*
** Machine code management.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_mcode_c
#define LUA_CORE

#include "lj_obj.h"
 #if LJ_HASJIT
 #include "lj_gc.h"
 #include "lj_err.h"
 #include "lj_jit.h"
 #include "lj_mcode.h"
 #include "lj_trace.h"
 #include "lj_dispatch.h"
 #endif
 #if LJ_HASJIT || LJ_HASFFI
 #include "lj_vm.h"
 #endif

 /* -- OS-specific functions ------------------------------------------------ */

 #if LJ_HASJIT || LJ_HASFFI

 /* Define this if you want to run LuaJIT with Valgrind. */
 #ifdef LUAJIT_USE_VALGRIND
 #include <valgrind/valgrind.h>
 #endif

 #if LJ_TARGET_IOS
 void sys_icache_invalidate(void *start, size_t len);
 #endif

 /* Synchronize data/instruction cache. */
 void lj_mcode_sync(void *start, void *end)
```

```
{
#ifdef LUAJIT_USE_VALGRIND
  VALGRIND_DISCARD_TRANSLATIONS(start, (char *)end-(char *)start);
#endif
#if LJ_TARGET_X86ORX64
  UNUSED(start); UNUSED(end);
#elif LJ_TARGET_IOS
  sys_icache_invalidate(start, (char *)end-(char *)start);
#elif LJ_TARGET_PPC
  lj_vm_cachesync(start, end);
#elif defined(__GNUC__)
  __clear_cache(start, end);
#else
#error "Missing builtin to flush instruction cache"
#endif
}

#endif

#if LJ_HASJIT

#if LJ_TARGET_WINDOWS

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#define MCPROT_RW        PAGE_READWRITE
#define MCPROT_RX        PAGE_EXECUTE_READ
#define MCPROT_RWX        PAGE_EXECUTE_READWRITE

static void *mcode_alloc_at(jit_State *J, uintptr_t hint, size_t sz, DWORD prot)
{
  void *p = VirtualAlloc((void *)hint, sz,
                         MEM_RESERVE|MEM_COMMIT|MEM_TOP_DOWN, prot);
  if (!p && !hint)
    lj_trace_err(J, LJ_TRERR_MCODEAL);
  return p;
}

static void mcode_free(jit_State *J, void *p, size_t sz)
{
  UNUSED(J); UNUSED(sz);
  VirtualFree(p, 0, MEM_RELEASE);
}

static int mcode_setprot(void *p, size_t sz, DWORD prot)
{
  DWORD oprot;
  return !VirtualProtect(p, sz, prot, &oprot);
}

#elif LJ_TARGET_POSIX

#include <sys/mman.h>

#ifndef MAP_ANONYMOUS
#define MAP_ANONYMOUS        MAP_ANON
#endif

#define MCPROT_RW        (PROT_READ|PROT_WRITE)
#define MCPROT_RX        (PROT_READ|PROT_EXEC)
#define MCPROT_RWX        (PROT_READ|PROT_WRITE|PROT_EXEC)

 static void *mcode_alloc_at(jit_State *J, uintptr_t hint, size_t sz, int prot)
 {
   void *p = mmap((void *)hint, sz, prot, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
   if (p == MAP_FAILED) {
     if (!hint) lj_trace_err(J, LJ_TRERR_MCODEAL);
     p = NULL;
   }
   return p;
 }

 static void mcode_free(jit_State *J, void *p, size_t sz)
 {
   UNUSED(J);
```

```
  munmap(p, sz);
}

static int mcode_setprot(void *p, size_t sz, int prot)
{
  return mprotect(p, sz, prot);
}

#elif LJ_64

#error "Missing OS support for explicit placement of executable memory"

#else

/* Fallback allocator. This will fail if memory is not executable by default. */
#define LUAJIT_UNPROTECT_MCODE
#define MCPROT_RW        0
#define MCPROT_RX        0
#define MCPROT_RWX        0

static void *mcode_alloc_at(jit_State *J, uintptr_t hint, size_t sz, int prot)
{
  UNUSED(hint); UNUSED(prot);
  return lj_mem_new(J->L, sz);
}

static void mcode_free(jit_State *J, void *p, size_t sz)
{
  lj_mem_free(J2G(J), p, sz);
}

#endif

/* -- MCode area protection ---------------------------------------------- */

/* Define this ONLY if page protection twiddling becomes a bottleneck. */
#ifdef LUAJIT_UNPROTECT_MCODE

/* It's generally considered to be a potential security risk to have
** pages with simultaneous write *and* execute access in a process.
**
** Do not even think about using this mode for server processes or
** apps handling untrusted external data (such as a browser).
**
** The security risk is not in LuaJIT itself -- but if an adversary finds
** any *other* flaw in your C application logic, then any RWX memory page
** simplifies writing an exploit considerably.
*/
#define MCPROT_GEN        MCPROT_RWX
#define MCPROT_RUN        MCPROT_RWX

static void mcode_protect(jit_State *J, int prot)
{
  UNUSED(J); UNUSED(prot);
}

#else

/* This is the default behaviour and much safer:
**
** Most of the time the memory pages holding machine code are executable,
** but NONE of them is writable.
**
** The current memory area is marked read-write (but NOT executable) only
** during the short time window while the assembler generates machine code.
*/
#define MCPROT_GEN        MCPROT_RW
#define MCPROT_RUN        MCPROT_RX

/* Protection twiddling failed. Probably due to kernel security. */
static LJ_NOINLINE void mcode_protfail(jit_State *J)
{
  lua_CFunction panic = J2G(J)->panic;
  if (panic) {
    lua_State *L = J->L;
    setstrV(L, L->top++, lj_err_str(L, LJ_ERR_JITPROT));
```

```
    panic(L);
  }
}

/* Change protection of MCode area. */
static void mcode_protect(jit_State *J, int prot)
{
  if (J->mcprot != prot) {
    if (LJ_UNLIKELY(mcode_setprot(J->mcarea, J->szmcarea, prot)))
      mcode_protfail(J);
    J->mcprot = prot;
  }
}

#endif

/* -- MCode area allocation ------------------------------------------------ */

#if LJ_TARGET_X64
#define mcode_validptr(p)        ((p) && (uintptr_t)(p) < (uintptr_t)1<<47)
#else
#define mcode_validptr(p)        ((p) && (uintptr_t)(p) < 0xffff0000)
#endif

#ifdef LJ_TARGET_JUMPRANGE

/* Get memory within relative jump distance of our code in 64 bit mode. */
static void *mcode_alloc(jit_State *J, size_t sz)
{
  /* Target an address in the static assembler code (64K aligned).
  ** Try addresses within a distance of target-range/2+1MB..target+range/2-1MB.
  ** Use half the jump range so every address in the range can reach any other.
  */
#if LJ_TARGET_MIPS
  /* Use the middle of the 256MB-aligned region. */
  uintptr_t target = ((uintptr_t)(void *)lj_vm_exit_handler & 0xf0000000u) +
                     0x08000000u;
#else
  uintptr_t target = (uintptr_t)(void *)lj_vm_exit_handler & ~(uintptr_t)0xffff;
#endif
  const uintptr_t range = (1u << (LJ_TARGET_JUMPRANGE-1)) - (1u << 21);
  /* First try a contiguous area below the last one. */
  uintptr_t hint = J->mcarea ? (uintptr_t)J->mcarea - sz : 0;
  int i;
  for (i = 0; i < 32; i++) {  /* 32 attempts ought to be enough ... */
    if (mcode_validptr(hint)) {
      void *p = mcode_alloc_at(J, hint, sz, MCPROT_GEN);

      if (mcode_validptr(p) &&
          ((uintptr_t)p + sz - target < range || target - (uintptr_t)p < range))
        return p;
      if (p) mcode_free(J, p, sz);  /* Free badly placed area. */
    }
    /* Next try probing pseudo-random addresses. */
    do {
      hint = (0x78fb ^ LJ_PRNG_BITS(J, 15)) << 16;  /* 64K aligned. */
    } while (!(hint + sz < range));
    hint = target + hint - (range>>1);
  }
  lj_trace_err(J, LJ_TRERR_MCODEAL);  /* Give up. OS probably ignores hints? */
  return NULL;
}

#else

/* All memory addresses are reachable by relative jumps. */
static void *mcode_alloc(jit_State *J, size_t sz)
{
#ifdef __OpenBSD__
  /* Allow better executable memory allocation for OpenBSD W^X mode. */
  void *p = mcode_alloc_at(J, 0, sz, MCPROT_RUN);
  if (p && mcode_setprot(p, sz, MCPROT_GEN)) {
    mcode_free(J, p, sz);
    return NULL;
  }
  return p;
```

```
#else
  return mcode_alloc_at(J, 0, sz, MCPROT_GEN);
#endif
}

#endif

/* -- MCode area management ---------------------------------------------- */

/* Linked list of MCode areas. */
typedef struct MCLink {
  MCode *next;		/* Next area. */
  size_t size;		/* Size of current area. */
} MCLink;

/* Allocate a new MCode area. */
static void mcode_allocarea(jit_State *J)
{
  MCode *oldarea = J->mcarea;
  size_t sz = (size_t)J->param[JIT_P_sizemcode] << 10;
  sz = (sz + LJ_PAGESIZE-1) & ~(size_t)(LJ_PAGESIZE - 1);
  J->mcarea = (MCode *)mcode_alloc(J, sz);
  J->szmcarea = sz;
  J->mcprot = MCPROT_GEN;
  J->mctop = (MCode *)((char *)J->mcarea + J->szmcarea);
  J->mcbot = (MCode *)((char *)J->mcarea + sizeof(MCLink));
  ((MCLink *)J->mcarea)->next = oldarea;
  ((MCLink *)J->mcarea)->size = sz;
  J->szallmcarea += sz;
}

/* Free all MCode areas. */
void lj_mcode_free(jit_State *J)
{
  MCode *mc = J->mcarea;
  J->mcarea = NULL;
  J->szallmcarea = 0;
  while (mc) {
    MCode *next = ((MCLink *)mc)->next;
    mcode_free(J, mc, ((MCLink *)mc)->size);
    mc = next;
  }
}

/* -- MCode transactions ------------------------------------------------- */

/* Reserve the remainder of the current MCode area. */
MCode *lj_mcode_reserve(jit_State *J, MCode **lim)
{
  if (!J->mcarea)
    mcode_allocarea(J);
  else
    mcode_protect(J, MCPROT_GEN);
  *lim = J->mcbot;
  return J->mctop;
}

/* Commit the top part of the current MCode area. */
void lj_mcode_commit(jit_State *J, MCode *top)
{
  J->mctop = top;
  mcode_protect(J, MCPROT_RUN);
}

/* Abort the reservation. */
void lj_mcode_abort(jit_State *J)
{
  if (J->mcarea)
    mcode_protect(J, MCPROT_RUN);
}

/* Set/reset protection to allow patching of MCode areas. */
MCode *lj_mcode_patch(jit_State *J, MCode *ptr, int finish)
{
#ifdef LUAJIT_UNPROTECT_MCODE
  UNUSED(J); UNUSED(ptr); UNUSED(finish);
```

```
  return NULL;
#else
  if (finish) {
    if (J->mcarea == ptr)
      mcode_protect(J, MCPROT_RUN);
    else if (LJ_UNLIKELY(mcode_setprot(ptr, ((MCLink *)ptr)->size, MCPROT_RUN)))
      mcode_protfail(J);
    return NULL;
  } else {
    MCode *mc = J->mcarea;
    /* Try current area first to use the protection cache. */
    if (ptr >= mc && ptr < (MCode *)((char *)mc + J->szmcarea)) {
      mcode_protect(J, MCPROT_GEN);
      return mc;
    }
    /* Otherwise search through the list of MCode areas. */
    for (;;) {
      mc = ((MCLink *)mc)->next;
      lua_assert(mc != NULL);
      if (ptr >= mc && ptr < (MCode *)((char *)mc + ((MCLink *)mc)->size)) {
        if (LJ_UNLIKELY(mcode_setprot(mc, ((MCLink *)mc)->size, MCPROT_GEN)))
          mcode_protfail(J);
        return mc;
      }
    }
  }
#endif
}

/* Limit of MCode reservation reached. */
void lj_mcode_limiterr(jit_State *J, size_t need)
{
  size_t sizemcode, maxmcode;
  lj_mcode_abort(J);
  sizemcode = (size_t)J->param[JIT_P_sizemcode] << 10;
  sizemcode = (sizemcode + LJ_PAGESIZE-1) & ~(size_t)(LJ_PAGESIZE - 1);
  maxmcode = (size_t)J->param[JIT_P_maxmcode] << 10;
  if ((size_t)need > sizemcode)
    lj_trace_err(J, LJ_TRERR_MCODEOV);  /* Too long for any area. */
  if (J->szallmcarea + sizemcode > maxmcode)
    lj_trace_err(J, LJ_TRERR_MCODEAL);
  mcode_allocarea(J);
  lj_trace_err(J, LJ_TRERR_MCODELM);  /* Retry with new area. */
}

#endif
```

One Level Up Top Level

src/lj_emit_mips.h - luajit-2.0-src

Data types defined

- MCLabel

Functions defined

- emit_addptr
- emit_branch
- emit_call
- emit_dst
- emit_dta
- emit_fgh
- emit_jmp
- emit_kdelta1
- emit_loadi
- emit_loadofs
- emit_lsglptr
- emit_lsptr
- emit_movrr
- emit_rotr
- emit_storeofs
- emit_tsi

Macros defined

- emit_canremat
- emit_ds
- emit_fg
- emit_getgl
- emit_hsi
- emit_label
- emit_loada
- emit_loadn
- emit_move
- emit_setgl

- emit_setvmstate
- emit_spsub
- emit_tg
- emit_ti

Source code

```
/*
** MIPS instruction emitter.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

/* -- Emit basic instructions --------------------------------------------- */

static void emit_dst(ASMState *as, MIPSIns mi, Reg rd, Reg rs, Reg rt)
{
  *--as->mcp = mi | MIPSF_D(rd) | MIPSF_S(rs) | MIPSF_T(rt);
}

static void emit_dta(ASMState *as, MIPSIns mi, Reg rd, Reg rt, uint32_t a)
{
  *--as->mcp = mi | MIPSF_D(rd) | MIPSF_T(rt) | MIPSF_A(a);
}

#define emit_ds(as, mi, rd, rs)               emit_dst(as, (mi), (rd), (rs), 0)
#define emit_tg(as, mi, rt, rg)               emit_dst(as, (mi), (rg)&31, 0, (rt))

static void emit_tsi(ASMState *as, MIPSIns mi, Reg rt, Reg rs, int32_t i)
{
  *--as->mcp = mi | MIPSF_T(rt) | MIPSF_S(rs) | (i & 0xffff);
}

#define emit_ti(as, mi, rt, i)                emit_tsi(as, (mi), (rt), 0, (i))
#define emit_hsi(as, mi, rh, rs, i)        emit_tsi(as, (mi), (rh) & 31, (rs), (i))

static void emit_fgh(ASMState *as, MIPSIns mi, Reg rf, Reg rg, Reg rh)
{
  *--as->mcp = mi | MIPSF_F(rf&31) | MIPSF_G(rg&31) | MIPSF_H(rh&31);
}

#define emit_fg(as, mi, rf, rg)               emit_fgh(as, (mi), (rf), (rg), 0)

static void emit_rotr(ASMState *as, Reg dest, Reg src, Reg tmp, uint32_t shift)
{
  if ((as->flags & JIT_F_MIPS32R2)) {
    emit_dta(as, MIPSI_ROTR, dest, src, shift);
  } else {
    emit_dst(as, MIPSI_OR, dest, dest, tmp);
    emit_dta(as, MIPSI_SLL, dest, src, (-shift)&31);
    emit_dta(as, MIPSI_SRL, tmp, src, shift);
  }
}

/* -- Emit loads/stores --------------------------------------------------- */

/* Prefer rematerialization of BASE/L from global_State over spills. */
#define emit_canremat(ref)        ((ref) <= REF_BASE)

/* Try to find a one step delta relative to another constant. */
static int emit_kdelta1(ASMState *as, Reg t, int32_t i)
{
  RegSet work = ~as->freeset & RSET_GPR;
  while (work) {
    Reg r = rset_picktop(work);
    IRRef ref = regcost_ref(as->cost[r]);
    lua_assert(r != t);
    if (ref < ASMREF_L) {
      int32_t delta = i - (ra_iskref(ref) ? ra_krefk(as, ref) : IR(ref)->i);
      if (checki16(delta)) {
        emit_tsi(as, MIPSI_ADDIU, t, r, delta);
```

```
64          return 1;
65        }
66      }
67      rset_clear(work, r);
68    }
69    return 0;  /* Failed. */
70  }
71
72  /* Load a 32 bit constant into a GPR. */
73  static void emit_loadi(ASMState *as, Reg r, int32_t i)
74  {
75    if (checki16(i)) {
76      emit_ti(as, MIPSI_LI, r, i);
77    } else {
78      if ((i & 0xffff)) {
79        int32_t jgl = i32ptr(J2G(as->J));
80        if ((uint32_t)(i-jgl) < 65536) {
81          emit_tsi(as, MIPSI_ADDIU, r, RID_JGL, i-jgl-32768);
82          return;
83        } else if (emit_kdelta1(as, r, i)) {
84          return;
85        } else if ((i >> 16) == 0) {
86          emit_tsi(as, MIPSI_ORI, r, RID_ZERO, i);
87          return;
88        }
89        emit_tsi(as, MIPSI_ORI, r, r, i);
90      }
91      emit_ti(as, MIPSI_LUI, r, (i >> 16));
92    }
93  }
94
95  #define emit_loada(as, r, addr)               emit_loadi(as, (r), i32ptr((addr)))
96
97  static Reg ra_allock(ASMState *as, int32_t k, RegSet allow);
98  static void ra_allockreg(ASMState *as, int32_t k, Reg r);
99
100 /* Get/set from constant pointer. */
101 static void emit_lsptr(ASMState *as, MIPSIns mi, Reg r, void *p, RegSet allow)
102 {
103   int32_t jgl = i32ptr(J2G(as->J));
104   int32_t i = i32ptr(p);
105   Reg base;
106   if ((uint32_t)(i-jgl) < 65536) {
107     i = i-jgl-32768;
108     base = RID_JGL;
109   } else {
110     base = ra_allock(as, i-(int16_t)i, allow);
111   }
112   emit_tsi(as, mi, r, base, i);
113 }
114
115 #define emit_loadn(as, r, tv) \
116   emit_lsptr(as, MIPSI_LDC1, ((r) & 31), (void *)(tv), RSET_GPR)
117
118 /* Get/set global_State fields. */
119 static void emit_lsglptr(ASMState *as, MIPSIns mi, Reg r, int32_t ofs)
120 {
121   emit_tsi(as, mi, r, RID_JGL, ofs-32768);
122 }
123
124 #define emit_getgl(as, r, field) \
125   emit_lsglptr(as, MIPSI_LW, (r), (int32_t)offsetof(global_State, field))
126 #define emit_setgl(as, r, field) \
127   emit_lsglptr(as, MIPSI_SW, (r), (int32_t)offsetof(global_State, field))
128
129 /* Trace number is determined from per-trace exit stubs. */
130 #define emit_setvmstate(as, i)                UNUSED(i)
131
132 /* -- Emit control-flow instructions ------------------------------------- */
133
134 /* Label for internal jumps. */
135 typedef MCode *MCLabel;
136
137 /* Return label pointing to current PC. */
138 #define emit_label(as)                ((as)->mcp)
139
```

```
140 static void emit_branch(ASMState *as, MIPSIns mi, Reg rs, Reg rt, MCode *target)
141 {
142   MCode *p = as->mcp;
143   ptrdiff_t delta = target - p;
144   lua_assert(((delta + 0x8000) >> 16) == 0);
145   *--p = mi | MIPSF_S(rs) | MIPSF_T(rt) | ((uint32_t)delta & 0xffffu);
146   as->mcp = p;
147 }
148
149 static void emit_jmp(ASMState *as, MCode *target)
150 {
151   *--as->mcp = MIPSI_NOP;
152   emit_branch(as, MIPSI_B, RID_ZERO, RID_ZERO, (target));
153 }
154
155 static void emit_call(ASMState *as, void *target)
156 {
157   MCode *p = as->mcp;
158   *--p = MIPSI_NOP;
159   if ((((uintptr_t)target ^ (uintptr_t)p) >> 28) == 0)
160     *--p = MIPSI_JAL | (((uintptr_t)target >>2) & 0x03ffffffu);
161   else  /* Target out of range: need indirect call. */
162     *--p = MIPSI_JALR | MIPSF_S(RID_CFUNCADDR);
163   as->mcp = p;
164   ra_allockreg(as, i32ptr(target), RID_CFUNCADDR);
165 }
166
167 /* -- Emit generic operations ---------------------------------------- */
168
169 #define emit_move(as, dst, src) \
170   emit_ds(as, MIPSI_MOVE, (dst), (src))
171
172 /* Generic move between two regs. */
173 static void emit_movrr(ASMState *as, IRIns *ir, Reg dst, Reg src)
174 {
175   if (dst < RID_MAX_GPR)
176     emit_move(as, dst, src);
177   else
178     emit_fg(as, irt_isnum(ir->t) ? MIPSI_MOV_D : MIPSI_MOV_S, dst, src);
179 }
180
181 /* Generic load of register with base and (small) offset address. */
182 static void emit_loadofs(ASMState *as, IRIns *ir, Reg r, Reg base, int32_t ofs)
183 {
184   if (r < RID_MAX_GPR)
185     emit_tsi(as, MIPSI_LW, r, base, ofs);
186   else
187     emit_tsi(as, irt_isnum(ir->t) ? MIPSI_LDC1 : MIPSI_LWC1,
188              (r & 31), base, ofs);
189 }
190
191 /* Generic store of register with base and (small) offset address. */
192 static void emit_storeofs(ASMState *as, IRIns *ir, Reg r, Reg base, int32_t ofs)
193 {
194   if (r < RID_MAX_GPR)
195     emit_tsi(as, MIPSI_SW, r, base, ofs);
196   else
197     emit_tsi(as, irt_isnum(ir->t) ? MIPSI_SDC1 : MIPSI_SWC1,
198              (r&31), base, ofs);
199 }
200
201 /* Add offset to pointer. */
202 static void emit_addptr(ASMState *as, Reg r, int32_t ofs)
203 {
204   if (ofs) {
205     lua_assert(checki16(ofs));
206     emit_tsi(as, MIPSI_ADDIU, r, r, ofs);
207   }
208 }
209
210 #define emit_spsub(as, ofs)        emit_addptr(as, RID_SP, -(ofs))
211
```

One Level Up Top Level

src/lj_target_mips.h - luajit-2.0-src

Data types defined

- ExitState
- MIPSIns
- MIPSIns

Functions defined

- exitstub_trace_addr_

Macros defined

- EXITSTATE_CHECKEXIT
- FPRDEF
- GPRDEF
- MIPSF_A
- MIPSF_D
- MIPSF_F
- MIPSF_G
- MIPSF_H
- MIPSF_M
- MIPSF_R
- MIPSF_S
- MIPSF_T
- REGARG_FIRSTFPR
- REGARG_FIRSTGPR
- REGARG_LASTFPR
- REGARG_LASTGPR
- REGARG_NUMFPR
- REGARG_NUMGPR
- RIDENUM
- RID_MIN_KREF
- RID_NUM_KREF
- RSET_ALL
- RSET_FIXED

- RSET_FPR
- RSET_GPR
- RSET_INIT
- RSET_SCRATCH
- RSET_SCRATCH_FPR
- RSET_SCRATCH_GPR
- SPOFS_TMP
- SPS_FIRST
- SPS_FIXED
- VRIDDEF
- _LJ_TARGET_MIPS_H
- exitstub_trace_addr
- sps_align
- sps_scale

Source code

```
/*
** Definitions for MIPS CPUs.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_TARGET_MIPS_H
#define _LJ_TARGET_MIPS_H

/* -- Registers IDs ------------------------------------------------------- */

#define GPRDEF(_) \
  _(R0) _(R1) _(R2) _(R3) _(R4) _(R5) _(R6) _(R7) \
  _(R8) _(R9) _(R10) _(R11) _(R12) _(R13) _(R14) _(R15) \
  _(R16) _(R17) _(R18) _(R19) _(R20) _(R21) _(R22) _(R23) \
  _(R24) _(R25) _(SYS1) _(SYS2) _(R28) _(SP) _(R30) _(RA)
#define FPRDEF(_) \
  _(F0) _(F1) _(F2) _(F3) _(F4) _(F5) _(F6) _(F7) \
  _(F8) _(F9) _(F10) _(F11) _(F12) _(F13) _(F14) _(F15) \
  _(F16) _(F17) _(F18) _(F19) _(F20) _(F21) _(F22) _(F23) \
  _(F24) _(F25) _(F26) _(F27) _(F28) _(F29) _(F30) _(F31)
#define VRIDDEF(_)

#define RIDENUM(name)        RID_##name,

enum {
  GPRDEF(RIDENUM)               /* General-purpose registers (GPRs). */
  FPRDEF(RIDENUM)               /* Floating-point registers (FPRs). */
  RID_MAX,
  RID_ZERO = RID_R0,
  RID_TMP = RID_RA,

  /* Calling conventions. */
  RID_RET = RID_R2,
#if LJ_LE
  RID_RETHI = RID_R3,
  RID_RETLO = RID_R2,
#else
  RID_RETHI = RID_R2,
  RID_RETLO = RID_R3,
#endif
  RID_FPRET = RID_F0,
```

```
  RID_CFUNCADDR = RID_R25,

  /* These definitions must match with the *.dasc file(s): */
  RID_BASE = RID_R16,		/* Interpreter BASE. */
  RID_LPC = RID_R18,		/* Interpreter PC. */
  RID_DISPATCH = RID_R19,	/* Interpreter DISPATCH table. */
  RID_LREG = RID_R20,		/* Interpreter L. */
  RID_JGL = RID_R30,		/* On-trace: global_State + 32768. */

  /* Register ranges [min, max) and number of registers. */
  RID_MIN_GPR = RID_R0,
  RID_MAX_GPR = RID_RA+1,
  RID_MIN_FPR = RID_F0,
  RID_MAX_FPR = RID_F31+1,
  RID_NUM_GPR = RID_MAX_GPR - RID_MIN_GPR,
  RID_NUM_FPR = RID_MAX_FPR - RID_MIN_FPR	/* Only even regs are used. */
};

#define RID_NUM_KREF		RID_NUM_GPR
#define RID_MIN_KREF		RID_R0

/* -- Register sets ------------------------------------------------------ */

/* Make use of all registers, except ZERO, TMP, SP, SYS1, SYS2 and JGL. */
#define RSET_FIXED \
  (RID2RSET(RID_ZERO)|RID2RSET(RID_TMP)|RID2RSET(RID_SP)|\
   RID2RSET(RID_SYS1)|RID2RSET(RID_SYS2)|RID2RSET(RID_JGL))
#define RSET_GPR	(RSET_RANGE(RID_MIN_GPR, RID_MAX_GPR) - RSET_FIXED)
#define RSET_FPR \
  (RID2RSET(RID_F0)|RID2RSET(RID_F2)|RID2RSET(RID_F4)|RID2RSET(RID_F6)|\
   RID2RSET(RID_F8)|RID2RSET(RID_F10)|RID2RSET(RID_F12)|RID2RSET(RID_F14)|\
   RID2RSET(RID_F16)|RID2RSET(RID_F18)|RID2RSET(RID_F20)|RID2RSET(RID_F22)|\
   RID2RSET(RID_F24)|RID2RSET(RID_F26)|RID2RSET(RID_F28)|RID2RSET(RID_F30))
#define RSET_ALL	(RSET_GPR|RSET_FPR)
#define RSET_INIT	RSET_ALL

#define RSET_SCRATCH_GPR \
  (RSET_RANGE(RID_R1, RID_R15+1)|\
   RID2RSET(RID_R24)|RID2RSET(RID_R25)|RID2RSET(RID_R28))
#define RSET_SCRATCH_FPR \
  (RID2RSET(RID_F0)|RID2RSET(RID_F2)|RID2RSET(RID_F4)|RID2RSET(RID_F6)|\
   RID2RSET(RID_F8)|RID2RSET(RID_F10)|RID2RSET(RID_F12)|RID2RSET(RID_F14)|\
   RID2RSET(RID_F16)|RID2RSET(RID_F18))
#define RSET_SCRATCH		(RSET_SCRATCH_GPR|RSET_SCRATCH_FPR)
#define REGARG_FIRSTGPR		RID_R4
#define REGARG_LASTGPR		RID_R7
#define REGARG_NUMGPR		4
#define REGARG_FIRSTFPR		RID_F12
#define REGARG_LASTFPR		RID_F14
#define REGARG_NUMFPR		2

/* -- Spill slots --------------------------------------------------------- */

/* Spill slots are 32 bit wide. An even/odd pair is used for FPRs.
**
** SPS_FIXED: Available fixed spill slots in interpreter frame.
** This definition must match with the *.dasc file(s).
**
** SPS_FIRST: First spill slot for general use.
*/
#define SPS_FIXED	5
#define SPS_FIRST	4

#define SPOFS_TMP	0

#define sps_scale(slot)		(4 * (int32_t)(slot))
#define sps_align(slot)		(((slot) - SPS_FIXED + 1) & ~1)

/* -- Exit state ---------------------------------------------------------- */

/* This definition must match with the *.dasc file(s). */
typedef struct {
  lua_Number fpr[RID_NUM_FPR];	/* Floating-point registers. */
  int32_t gpr[RID_NUM_GPR];	/* General-purpose registers. */
  int32_t spill[256];		/* Spill slots. */
} ExitState;
```

```
118
119 /* Highest exit + 1 indicates stack check. */
120 #define EXITSTATE_CHECKEXIT       1
121
122 /* Return the address of a per-trace exit stub. */
123 static LJ_AINLINE uint32_t *exitstub_trace_addr_(uint32_t *p)
124 {
125   while (*p == 0x00000000) p++;  /* Skip MIPSI_NOP. */
126   return p;
127 }
128 /* Avoid dependence on lj_jit.h if only including lj_target.h. */
129 #define exitstub_trace_addr(T, exitno) \
130   exitstub_trace_addr_((MCode *)((char *)(T)->mcode + (T)->szmcode))
131
132 /* -- Instructions ---------------------------------------------------- */
133
134 /* Instruction fields. */
135 #define MIPSF_S(r)        ((r) << 21)
136 #define MIPSF_T(r)        ((r) << 16)
137 #define MIPSF_D(r)        ((r) << 11)
138 #define MIPSF_R(r)        ((r) << 21)
139 #define MIPSF_H(r)        ((r) << 16)
140 #define MIPSF_G(r)        ((r) << 11)
141 #define MIPSF_F(r)        ((r) << 6)
142 #define MIPSF_A(n)        ((n) << 6)
143 #define MIPSF_M(n)        ((n) << 11)
144
145 typedef enum MIPSIns {
146   /* Integer instructions. */
147   MIPSI_MOVE = 0x00000021,
148   MIPSI_NOP = 0x00000000,
149
150   MIPSI_LI = 0x24000000,
151   MIPSI_LU = 0x34000000,
152   MIPSI_LUI = 0x3c000000,
153
154   MIPSI_ADDIU = 0x24000000,
155   MIPSI_ANDI = 0x30000000,
156   MIPSI_ORI = 0x34000000,
157   MIPSI_XORI = 0x38000000,
158   MIPSI_SLTI = 0x28000000,
159   MIPSI_SLTIU = 0x2c000000,
160
161   MIPSI_ADDU = 0x00000021,
162   MIPSI_SUBU = 0x00000023,
163   MIPSI_MUL = 0x70000002,
164   MIPSI_AND = 0x00000024,
165   MIPSI_OR = 0x00000025,
166   MIPSI_XOR = 0x00000026,
167   MIPSI_NOR = 0x00000027,
168   MIPSI_SLT = 0x0000002a,
169   MIPSI_SLTU = 0x0000002b,
170   MIPSI_MOVZ = 0x0000000a,
171   MIPSI_MOVN = 0x0000000b,
172   MIPSI_MFHI = 0x00000010,
173   MIPSI_MFLO = 0x00000012,
174   MIPSI_MULT = 0x00000018,
175
176   MIPSI_SLL = 0x00000000,
177   MIPSI_SRL = 0x00000002,
178   MIPSI_SRA = 0x00000003,
179   MIPSI_ROTR = 0x00200002,        /* MIPS32R2 */
180   MIPSI_SLLV = 0x00000004,
181   MIPSI_SRLV = 0x00000006,
182   MIPSI_SRAV = 0x00000007,
183   MIPSI_ROTRV = 0x00000046,        /* MIPS32R2 */
184
185   MIPSI_SEB = 0x7c000420,        /* MIPS32R2 */
186   MIPSI_SEH = 0x7c000620,        /* MIPS32R2 */
187   MIPSI_WSBH = 0x7c0000a0,        /* MIPS32R2 */
188
189   MIPSI_B = 0x10000000,
190   MIPSI_J = 0x08000000,
191   MIPSI_JAL = 0x0c000000,
192   MIPSI_JR = 0x00000008,
193   MIPSI_JALR = 0x0000f809,
```

```
194
195     MIPSI_BEQ = 0x10000000,
196     MIPSI_BNE = 0x14000000,
197     MIPSI_BLEZ = 0x18000000,
198     MIPSI_BGTZ = 0x1c000000,
199     MIPSI_BLTZ = 0x04000000,
200     MIPSI_BGEZ = 0x04010000,
201
202     /* Load/store instructions. */
203     MIPSI_LW = 0x8c000000,
204     MIPSI_SW = 0xac000000,
205     MIPSI_LB = 0x80000000,
206     MIPSI_SB = 0xa0000000,
207     MIPSI_LH = 0x84000000,
208     MIPSI_SH = 0xa4000000,
209     MIPSI_LBU = 0x90000000,
210     MIPSI_LHU = 0x94000000,
211     MIPSI_LWC1 = 0xc4000000,
212     MIPSI_SWC1 = 0xe4000000,
213     MIPSI_LDC1 = 0xd4000000,
214     MIPSI_SDC1 = 0xf4000000,
215
216     /* FP instructions. */
217     MIPSI_MOV_S = 0x46000006,
218     MIPSI_MOV_D = 0x46200006,
219     MIPSI_MOVT_D = 0x46210011,
220     MIPSI_MOVF_D = 0x46200011,
221
222     MIPSI_ABS_D = 0x46200005,
223     MIPSI_NEG_D = 0x46200007,
224
225     MIPSI_ADD_D = 0x46200000,
226     MIPSI_SUB_D = 0x46200001,
227     MIPSI_MUL_D = 0x46200002,
228     MIPSI_DIV_D = 0x46200003,
229     MIPSI_SQRT_D = 0x46200004,
230
231     MIPSI_ADD_S = 0x46000000,
232     MIPSI_SUB_S = 0x46000001,
233
234     MIPSI_CVT_D_S = 0x46000021,
235     MIPSI_CVT_W_S = 0x46000024,
236     MIPSI_CVT_S_D = 0x46200020,
237     MIPSI_CVT_W_D = 0x46200024,
238     MIPSI_CVT_S_W = 0x46800020,
239     MIPSI_CVT_D_W = 0x46800021,
240
241     MIPSI_TRUNC_W_S = 0x4600000d,
242     MIPSI_TRUNC_W_D = 0x4620000d,
243     MIPSI_FLOOR_W_S = 0x4600000f,
244     MIPSI_FLOOR_W_D = 0x4620000f,
245
246     MIPSI_MFC1 = 0x44000000,
247     MIPSI_MTC1 = 0x44800000,
248
249     MIPSI_BC1F = 0x45000000,
250     MIPSI_BC1T = 0x45010000,
251
252     MIPSI_C_EQ_D = 0x46200032,
253     MIPSI_C_OLT_D = 0x46200034,
254     MIPSI_C_ULT_D = 0x46200035,
255     MIPSI_C_OLE_D = 0x46200036,
256     MIPSI_C_ULE_D = 0x46200037,
257
258 } MIPSIns;
259
260 #endif
```

One Level Up Top Level

src/lj_emit_x86.h - luajit-2.0-src

Data types defined

- MCLabel

Functions defined

- emit_addptr
- emit_call_
- emit_gmrmi
- emit_gmroi
- emit_gri
- emit_jcc
- emit_jmp
- emit_loadi
- emit_loadn
- emit_loadofs
- emit_loadu64
- emit_movmroi
- emit_movrr
- emit_mrm
- emit_op
- emit_rma
- emit_rmro
- emit_rmrxo
- emit_rr
- emit_sfixup
- emit_sjcc
- emit_sjcc_label
- emit_sjmp
- emit_storeofs
- jmprel
- ptr2addr

Macros defined

- FORCE_REX
- FORCE_REX
- MODRM
- REXRB
- REXRB
- REX_64
- REX_64
- REX_64IR
- REX_64IR
- emit_call
- emit_canremat
- emit_getgl
- emit_i32
- emit_i8
- emit_label
- emit_loada
- emit_movtomro
- emit_opgl
- emit_opm
- emit_opmx
- emit_setgl
- emit_setvmstate
- emit_shifti
- emit_spsub
- emit_u32
- emit_x87op
- ptr2addr

Source code

```
/*
** x86/x64 instruction emitter.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

/* -- Emit basic instructions ---------------------------------------- */

#define MODRM(mode, r1, r2)        ((MCode)((mode)+(((r1)&7)<<3)+((r2)&7)))

 #if LJ_64
 #define REXRB(p, rr, rb) \
```

```
    { MCode rex = 0x40 + (((rr)>>1)&4) + (((rb)>>3)&1); \
      if (rex != 0x40) *--(p) = rex; }
#define FORCE_REX		0x200
#define REX_64			(FORCE_REX|0x080000)
#else
#define REXRB(p, rr, rb)	((void)0)
#define FORCE_REX		0
#define REX_64			0
#endif

#define emit_i8(as, i)		(*--as->mcp = (MCode)(i))
#define emit_i32(as, i)		(*(int32_t *)(as->mcp-4) = (i), as->mcp -= 4)
#define emit_u32(as, u)		(*(uint32_t *)(as->mcp-4) = (u), as->mcp -= 4)

#define emit_x87op(as, xo) \
  (*(uint16_t *)(as->mcp-2) = (uint16_t)(xo), as->mcp -= 2)

/* op */
static LJ_AINLINE MCode *emit_op(x86Op xo, Reg rr, Reg rb, Reg rx,
				 MCode *p, int delta)
{
  int n = (int8_t)xo;
#if defined(__GNUC__)
  if (__builtin_constant_p(xo) && n == -2)
    p[delta-2] = (MCode)(xo >> 24);
  else if (__builtin_constant_p(xo) && n == -3)
    *(uint16_t *)(p+delta-3) = (uint16_t)(xo >> 16);
  else
#endif
    *(uint32_t *)(p+delta-5) = (uint32_t)xo;
  p += n + delta;
#if LJ_64
  {
    uint32_t rex = 0x40 + ((rr>>1)&(4+(FORCE_REX>>1)))+((rx>>2)&2)+((rb>>3)&1);
    if (rex != 0x40) {
      rex |= (rr >> 16);
      if (n == -4) { *p = (MCode)rex; rex = (MCode)(xo >> 8); }
      else if ((xo & 0xffffff) == 0x6600fd) { *p = (MCode)rex; rex = 0x66; }
      *--p = (MCode)rex;
    }
  }
#else
  UNUSED(rr); UNUSED(rb); UNUSED(rx);
#endif
  return p;
}

/* op + modrm */
#define emit_opm(xo, mode, rr, rb, p, delta) \
  (p[(delta)-1] = MODRM((mode), (rr), (rb)), \
   emit_op((xo), (rr), (rb), 0, (p), (delta)))

/* op + modrm + sib */
#define emit_opmx(xo, mode, scale, rr, rb, rx, p) \
  (p[-1] = MODRM((scale), (rx), (rb)), \
   p[-2] = MODRM((mode), (rr), RID_ESP), \
   emit_op((xo), (rr), (rb), (rx), (p), -1))

/* op r1, r2 */
static void emit_rr(ASMState *as, x86Op xo, Reg r1, Reg r2)
{
  MCode *p = as->mcp;
  as->mcp = emit_opm(xo, XM_REG, r1, r2, p, 0);
}

#if LJ_64 && defined(LUA_USE_ASSERT)
/* [addr] is sign-extended in x64 and must be in lower 2G (not 4G). */
static int32_t ptr2addr(const void *p)
{
  lua_assert((uintptr_t)p < (uintptr_t)0x80000000);
  return i32ptr(p);
}
#else
#define ptr2addr(p)		(i32ptr((p)))
#endif

```

```
/* op r, [addr] */
static void emit_rma(ASMState *as, x86Op xo, Reg rr, const void *addr)
{
  MCode *p = as->mcp;
  *(int32_t *)(p-4) = ptr2addr(addr);
#if LJ_64
  p[-5] = MODRM(XM_SCALE1, RID_ESP, RID_EBP);
  as->mcp = emit_opm(xo, XM_OFS0, rr, RID_ESP, p, -5);
#else
  as->mcp = emit_opm(xo, XM_OFS0, rr, RID_EBP, p, -4);
#endif
}

/* op r, [base+ofs] */
static void emit_rmro(ASMState *as, x86Op xo, Reg rr, Reg rb, int32_t ofs)
{
  MCode *p = as->mcp;
  x86Mode mode;
  if (ra_hasreg(rb)) {
    if (ofs == 0 && (rb&7) != RID_EBP) {
      mode = XM_OFS0;
    } else if (checki8(ofs)) {
      *--p = (MCode)ofs;
      mode = XM_OFS8;
    } else {
      p -= 4;
      *(int32_t *)p = ofs;
      mode = XM_OFS32;
    }
    if ((rb&7) == RID_ESP)
      *--p = MODRM(XM_SCALE1, RID_ESP, RID_ESP);
  } else {
    *(int32_t *)(p-4) = ofs;
#if LJ_64
    p[-5] = MODRM(XM_SCALE1, RID_ESP, RID_EBP);
    p -= 5;
    rb = RID_ESP;
#else
    p -= 4;
    rb = RID_EBP;
#endif
    mode = XM_OFS0;
  }
  as->mcp = emit_opm(xo, mode, rr, rb, p, 0);
}

/* op r, [base+idx*scale+ofs] */
static void emit_rmrxo(ASMState *as, x86Op xo, Reg rr, Reg rb, Reg rx,
                       x86Mode scale, int32_t ofs)
{
  MCode *p = as->mcp;
  x86Mode mode;
  if (ofs == 0 && (rb&7) != RID_EBP) {
    mode = XM_OFS0;
  } else if (checki8(ofs)) {
    mode = XM_OFS8;
    *--p = (MCode)ofs;
  } else {
    mode = XM_OFS32;
    p -= 4;
    *(int32_t *)p = ofs;
  }
  as->mcp = emit_opmx(xo, mode, scale, rr, rb, rx, p);
}

/* op r, i */
static void emit_gri(ASMState *as, x86Group xg, Reg rb, int32_t i)
{
  MCode *p = as->mcp;
  x86Op xo;
  if (checki8(i)) {
    *--p = (MCode)i;
    xo = XG_TOXOi8(xg);
  } else {
    p -= 4;
    *(int32_t *)p = i;
```

```
    xo = XG_TOXOi(xg);
  }
  as->mcp = emit_opm(xo, XM_REG, (Reg)(xg & 7) | (rb & REX_64), rb, p, 0);
}

/* op [base+ofs], i */
static void emit_gmroi(ASMState *as, x86Group xg, Reg rb, int32_t ofs,
                       int32_t i)
{
  x86Op xo;
  if (checki8(i)) {
    emit_i8(as, i);
    xo = XG_TOXOi8(xg);
  } else {
    emit_i32(as, i);
    xo = XG_TOXOi(xg);
  }
  emit_rmro(as, xo, (Reg)(xg & 7), rb, ofs);
}

#define emit_shifti(as, xg, r, i) \
  (emit_i8(as, (i)), emit_rr(as, XO_SHIFTi, (Reg)(xg), (r)))

/* op r, rm/mrm */
static void emit_mrm(ASMState *as, x86Op xo, Reg rr, Reg rb)
{
  MCode *p = as->mcp;
  x86Mode mode = XM_REG;
  if (rb == RID_MRM) {
    rb = as->mrm.base;
    if (rb == RID_NONE) {
      rb = RID_EBP;
      mode = XM_OFS0;
      p -= 4;
      *(int32_t *)p = as->mrm.ofs;
      if (as->mrm.idx != RID_NONE)
        goto mrmidx;
#if LJ_64
      *--p = MODRM(XM_SCALE1, RID_ESP, RID_EBP);
      rb = RID_ESP;
#endif
    } else {
      if (as->mrm.ofs == 0 && (rb&7) != RID_EBP) {
        mode = XM_OFS0;
      } else if (checki8(as->mrm.ofs)) {
        *--p = (MCode)as->mrm.ofs;
        mode = XM_OFS8;
      } else {
        p -= 4;
        *(int32_t *)p = as->mrm.ofs;
        mode = XM_OFS32;
      }
      if (as->mrm.idx != RID_NONE) {
      mrmidx:
        as->mcp = emit_opmx(xo, mode, as->mrm.scale, rr, rb, as->mrm.idx, p);
        return;
      }
      if ((rb&7) == RID_ESP)
        *--p = MODRM(XM_SCALE1, RID_ESP, RID_ESP);
    }
  }
  as->mcp = emit_opm(xo, mode, rr, rb, p, 0);
}

/* op rm/mrm, i */
static void emit_gmrmi(ASMState *as, x86Group xg, Reg rb, int32_t i)
{
  x86Op xo;
  if (checki8(i)) {
    emit_i8(as, i);
    xo = XG_TOXOi8(xg);
  } else {
    emit_i32(as, i);
    xo = XG_TOXOi(xg);
  }
  emit_mrm(as, xo, (Reg)(xg & 7) | (rb & REX_64), (rb & ~REX_64));
```

```
}

/* -- Emit loads/stores ---------------------------------------------------- */

/* mov [base+ofs], i */
static void emit_movmroi(ASMState *as, Reg base, int32_t ofs, int32_t i)
{
  emit_i32(as, i);
  emit_rmro(as, XO_MOVmi, 0, base, ofs);
}

/* mov [base+ofs], r */
#define emit_movtomro(as, r, base, ofs) \
  emit_rmro(as, XO_MOVto, (r), (base), (ofs))

/* Get/set global_State fields. */
#define emit_opgl(as, xo, r, field) \
  emit_rma(as, (xo), (r), (void *)&J2G(as->J)->field)
#define emit_getgl(as, r, field)        emit_opgl(as, XO_MOV, (r), field)
#define emit_setgl(as, r, field)        emit_opgl(as, XO_MOVto, (r), field)

#define emit_setvmstate(as, i) \
  (emit_i32(as, i), emit_opgl(as, XO_MOVmi, 0, vmstate))

/* mov r, i / xor r, r */
static void emit_loadi(ASMState *as, Reg r, int32_t i)
{
  /* XOR r,r is shorter, but modifies the flags. This is bad for HIOP. */
  if (i == 0 && !(LJ_32 && (IR(as->curins)->o == IR_HIOP ||
                            (as->curins+1 < as->T->nins &&
                             IR(as->curins+1)->o == IR_HIOP)))) {
    emit_rr(as, XO_ARITH(XOg_XOR), r, r);
  } else {
    MCode *p = as->mcp;
    *(int32_t *)(p-4) = i;
    p[-5] = (MCode)(XI_MOVri+(r&7));
    p -= 5;
    REXRB(p, 0, r);
    as->mcp = p;
  }
}

/* mov r, addr */
#define emit_loada(as, r, addr) \
  emit_loadi(as, (r), ptr2addr((addr)))

#if LJ_64
/* mov r, imm64 or shorter 32 bit extended load. */
static void emit_loadu64(ASMState *as, Reg r, uint64_t u64)
{
  if (checku32(u64)) {  /* 32 bit load clears upper 32 bits. */
    emit_loadi(as, r, (int32_t)u64);
  } else if (checki32((int64_t)u64)) {  /* Sign-extended 32 bit load. */
    MCode *p = as->mcp;
    *(int32_t *)(p-4) = (int32_t)u64;
    as->mcp = emit_opm(XO_MOVmi, XM_REG, REX_64, r, p, -4);
  } else {  /* Full-size 64 bit load. */
    MCode *p = as->mcp;
    *(uint64_t *)(p-8) = u64;
    p[-9] = (MCode)(XI_MOVri+(r&7));
    p[-10] = 0x48 + ((r>>3)&1);
    p -= 10;
    as->mcp = p;
  }
}
#endif

/* movsd r, [&tv->n] / xorps r, r */
static void emit_loadn(ASMState *as, Reg r, cTValue *tv)
{
  if (tvispzero(tv))  /* Use xor only for +0. */
    emit_rr(as, XO_XORPS, r, r);
  else
    emit_rma(as, XO_MOVSD, r, &tv->n);
}

```

```
/* -- Emit control-flow instructions ------------------------------------ */

/* Label for short jumps. */
typedef MCode *MCLabel;

#if LJ_32 && LJ_HASFFI
/* jmp short target */
static void emit_sjmp(ASMState *as, MCLabel target)
{
  MCode *p = as->mcp;
  ptrdiff_t delta = target - p;
  lua_assert(delta == (int8_t)delta);
  p[-1] = (MCode)(int8_t)delta;
  p[-2] = XI_JMPs;
  as->mcp = p - 2;
}
#endif

/* jcc short target */
static void emit_sjcc(ASMState *as, int cc, MCLabel target)
{
  MCode *p = as->mcp;
  ptrdiff_t delta = target - p;
  lua_assert(delta == (int8_t)delta);
  p[-1] = (MCode)(int8_t)delta;
  p[-2] = (MCode)(XI_JCCs+(cc&15));
  as->mcp = p - 2;
}

/* jcc short (pending target) */
static MCLabel emit_sjcc_label(ASMState *as, int cc)
{
  MCode *p = as->mcp;
  p[-1] = 0;
  p[-2] = (MCode)(XI_JCCs+(cc&15));
  as->mcp = p - 2;
  return p;
}

/* Fixup jcc short target. */
static void emit_sfixup(ASMState *as, MCLabel source)
{
  source[-1] = (MCode)(as->mcp-source);
}

/* Return label pointing to current PC. */
#define emit_label(as)               ((as)->mcp)

/* Compute relative 32 bit offset for jump and call instructions. */
static LJ_AINLINE int32_t jmprel(MCode *p, MCode *target)
{
  ptrdiff_t delta = target - p;
  lua_assert(delta == (int32_t)delta);
  return (int32_t)delta;
}

/* jcc target */
static void emit_jcc(ASMState *as, int cc, MCode *target)
{
  MCode *p = as->mcp;
  *(int32_t *)(p-4) = jmprel(p, target);
  p[-5] = (MCode)(XI_JCCn+(cc&15));
  p[-6] = 0x0f;
  as->mcp = p - 6;
}

/* jmp target */
static void emit_jmp(ASMState *as, MCode *target)
{
  MCode *p = as->mcp;
  *(int32_t *)(p-4) = jmprel(p, target);
  p[-5] = XI_JMP;
  as->mcp = p - 5;
}

/* call target */
```

```
392 static void emit_call_(ASMState *as, MCode *target)
393 {
394   MCode *p = as->mcp;
395 #if LJ_64
396   if (target-p != (int32_t)(target-p)) {
397     /* Assumes RID_RET is never an argument to calls and always clobbered. */
398     emit_rr(as, XO_GROUP5, XOg_CALL, RID_RET);
399     emit_loadu64(as, RID_RET, (uint64_t)target);
400     return;
401   }
402 #endif
403   *(int32_t *)(p-4) = jmprel(p, target);
404   p[-5] = XI_CALL;
405   as->mcp = p - 5;
406 }
407
408 #define emit_call(as, f)        emit_call_(as, (MCode *)(void *)(f))
409
410 /* -- Emit generic operations ---------------------------------------- */
411
412 /* Use 64 bit operations to handle 64 bit IR types. */
413 #if LJ_64
414 #define REX_64IR(ir, r)               ((r) + (irt_is64((ir)->t) ? REX_64 : 0))
415 #else
416 #define REX_64IR(ir, r)               (r)
417 #endif
418
419 /* Generic move between two regs. */
420 static void emit_movrr(ASMState *as, IRIns *ir, Reg dst, Reg src)
421 {
422   UNUSED(ir);
423   if (dst < RID_MAX_GPR)
424     emit_rr(as, XO_MOV, REX_64IR(ir, dst), src);
425   else
426     emit_rr(as, XO_MOVAPS, dst, src);
427 }
428
429 /* Generic load of register with base and (small) offset address. */
430 static void emit_loadofs(ASMState *as, IRIns *ir, Reg r, Reg base, int32_t ofs)
431 {
432   if (r < RID_MAX_GPR)
433     emit_rmro(as, XO_MOV, REX_64IR(ir, r), base, ofs);
434   else
435     emit_rmro(as, irt_isnum(ir->t) ? XO_MOVSD : XO_MOVSS, r, base, ofs);
436 }
437
438 /* Generic store of register with base and (small) offset address. */
439 static void emit_storeofs(ASMState *as, IRIns *ir, Reg r, Reg base, int32_t ofs)
440 {
441   if (r < RID_MAX_GPR)
442     emit_rmro(as, XO_MOVto, REX_64IR(ir, r), base, ofs);
443   else
444     emit_rmro(as, irt_isnum(ir->t) ? XO_MOVSDto : XO_MOVSSto, r, base, ofs);
445 }
446
447 /* Add offset to pointer. */
448 static void emit_addptr(ASMState *as, Reg r, int32_t ofs)
449 {
450   if (ofs) {
451     if ((as->flags & JIT_F_LEA_AGU))
452       emit_rmro(as, XO_LEA, r, r, ofs);
453     else
454       emit_gri(as, XG_ARITHi(XOg_ADD), r, ofs);
455   }
456 }
457
458 #define emit_spsub(as, ofs)        emit_addptr(as, RID_ESP|REX_64, -(ofs))
459
460 /* Prefer rematerialization of BASE/L from global_State over spills. */
461 #define emit_canremat(ref)        ((ref) <= REF_BASE)
462
```

One Level Up Top Level

src/lj_iropt.h - luajit-2.0-src

Functions defined

- lj_ir_knum
- lj_ir_nextins
- lj_ir_set_

Macros defined

- CONDFOLD
- CSEFOLD
- EMITFOLD
- INT64FOLD
- INTFOLD
- LEFTFOLD
- RIGHTFOLD
- _LJ_IROPT_H
- lj_ir_kfunc
- lj_ir_kintp
- lj_ir_kintp
- lj_ir_kkptr
- lj_ir_knum_abs
- lj_ir_knum_neg
- lj_ir_knum_one
- lj_ir_knum_tobit
- lj_ir_knum_zero
- lj_ir_kptr
- lj_ir_kstr
- lj_ir_ktab
- lj_ir_set
- lj_opt_split

Source code

```
1  /*
2  ** Common header for IR emitter and optimizations.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
```

```
5
6  #ifndef _LJ_IROPT_H
7  #define _LJ_IROPT_H
8
9  #include <stdarg.h>
10
11 #include "lj_obj.h"
12 #include "lj_jit.h"
13
14 #if LJ_HASJIT
15 /* IR emitter. */
16 LJ_FUNC void LJ_FASTCALL lj_ir_growtop(jit_State *J);
17 LJ_FUNC TRef LJ_FASTCALL lj_ir_emit(jit_State *J);
18
19 /* Save current IR in J->fold.ins, but do not emit it (yet). */
20 static LJ_AINLINE void lj_ir_set_(jit_State *J, uint16_t ot, IRRef1 a, IRRef1 b)
21 {
22   J->fold.ins.ot = ot; J->fold.ins.op1 = a; J->fold.ins.op2 = b;
23 }
24
25 #define lj_ir_set(J, ot, a, b) \
26   lj_ir_set_(J, (uint16_t)(ot), (IRRef1)(a), (IRRef1)(b))
27
28 /* Get ref of next IR instruction and optionally grow IR.
29 ** Note: this may invalidate all IRIns*!
30 */
31 static LJ_AINLINE IRRef lj_ir_nextins(jit_State *J)
32 {
33   IRRef ref = J->cur.nins;
34   if (LJ_UNLIKELY(ref >= J->irtoplim)) lj_ir_growtop(J);
35   J->cur.nins = ref + 1;
36   return ref;
37 }
38
39 /* Interning of constants. */
40 LJ_FUNC TRef LJ_FASTCALL lj_ir_kint(jit_State *J, int32_t k);
41 LJ_FUNC void lj_ir_k64_freeall(jit_State *J);
42 LJ_FUNC TRef lj_ir_k64(jit_State *J, IROp op, cTValue *tv);
43 LJ_FUNC cTValue *lj_ir_k64_find(jit_State *J, uint64_t u64);
44 LJ_FUNC TRef lj_ir_knum_u64(jit_State *J, uint64_t u64);
45 LJ_FUNC TRef lj_ir_knumint(jit_State *J, lua_Number n);
46 LJ_FUNC TRef lj_ir_kint64(jit_State *J, uint64_t u64);
47 LJ_FUNC TRef lj_ir_kgc(jit_State *J, GCobj *o, IRType t);
48 LJ_FUNC TRef lj_ir_kptr_(jit_State *J, IROp op, void *ptr);
49 LJ_FUNC TRef lj_ir_knull(jit_State *J, IRType t);
50 LJ_FUNC TRef lj_ir_kslot(jit_State *J, TRef key, IRRef slot);
51
52 #if LJ_64
53 #define lj_ir_kintp(J, k)        lj_ir_kint64(J, (uint64_t)(k))
54 #else
55 #define lj_ir_kintp(J, k)        lj_ir_kint(J, (int32_t)(k))
56 #endif
57
58 static LJ_AINLINE TRef lj_ir_knum(jit_State *J, lua_Number n)
59 {
60   TValue tv;
61   tv.n = n;
62   return lj_ir_knum_u64(J, tv.u64);
63 }
64
65 #define lj_ir_kstr(J, str)        lj_ir_kgc(J, obj2gco((str)), IRT_STR)
66 #define lj_ir_ktab(J, tab)        lj_ir_kgc(J, obj2gco((tab)), IRT_TAB)
67 #define lj_ir_kfunc(J, func)        lj_ir_kgc(J, obj2gco((func)), IRT_FUNC)
68 #define lj_ir_kptr(J, ptr)        lj_ir_kptr_(J, IR_KPTR, (ptr))
69 #define lj_ir_kkptr(J, ptr)        lj_ir_kptr_(J, IR_KKPTR, (ptr))
70
71 /* Special FP constants. */
72 #define lj_ir_knum_zero(J)        lj_ir_knum_u64(J, U64x(00000000,00000000))
73 #define lj_ir_knum_one(J)        lj_ir_knum_u64(J, U64x(3ff00000,00000000))
74 #define lj_ir_knum_tobit(J)        lj_ir_knum_u64(J, U64x(43380000,00000000))
75
76 /* Special 128 bit SIMD constants. */
77 #define lj_ir_knum_abs(J)        lj_ir_k64(J, IR_KNUM, LJ_KSIMD(J, LJ_KSIMD_ABS))
78 #define lj_ir_knum_neg(J)        lj_ir_k64(J, IR_KNUM, LJ_KSIMD(J, LJ_KSIMD_NEG))
79
80 /* Access to constants. */
```

```
LJ_FUNC void lj_ir_kvalue(lua_State *L, TValue *tv, const IRIns *ir);

/* Convert IR operand types. */
LJ_FUNC TRef LJ_FASTCALL lj_ir_tonumber(jit_State *J, TRef tr);
LJ_FUNC TRef LJ_FASTCALL lj_ir_tonum(jit_State *J, TRef tr);
LJ_FUNC TRef LJ_FASTCALL lj_ir_tostr(jit_State *J, TRef tr);

/* Miscellaneous IR ops. */
LJ_FUNC int lj_ir_numcmp(lua_Number a, lua_Number b, IROp op);
LJ_FUNC int lj_ir_strcmp(GCstr *a, GCstr *b, IROp op);
LJ_FUNC void lj_ir_rollback(jit_State *J, IRRef ref);

/* Emit IR instructions with on-the-fly optimizations. */
LJ_FUNC TRef LJ_FASTCALL lj_opt_fold(jit_State *J);
LJ_FUNC TRef LJ_FASTCALL lj_opt_cse(jit_State *J);
LJ_FUNC TRef LJ_FASTCALL lj_opt_cselim(jit_State *J, IRRef lim);

/* Special return values for the fold functions. */
enum {
  NEXTFOLD,                /* Couldn't fold, pass on. */
  RETRYFOLD,               /* Retry fold with modified fins. */
  KINTFOLD,                /* Return ref for int constant in fins->i. */
  FAILFOLD,                /* Guard would always fail. */
  DROPFOLD,                /* Guard eliminated. */
  MAX_FOLD
};

#define INTFOLD(k)        ((J->fold.ins.i = (k)), (TRef)KINTFOLD)
#define INT64FOLD(k)        (lj_ir_kint64(J, (k)))
#define CONDFOLD(cond)        ((TRef)FAILFOLD + (TRef)(cond))
#define LEFTFOLD        (J->fold.ins.op1)
#define RIGHTFOLD        (J->fold.ins.op2)
#define CSEFOLD                (lj_opt_cse(J))
#define EMITFOLD        (lj_ir_emit(J))

/* Load/store forwarding. */
LJ_FUNC TRef LJ_FASTCALL lj_opt_fwd_aload(jit_State *J);
LJ_FUNC TRef LJ_FASTCALL lj_opt_fwd_hload(jit_State *J);
LJ_FUNC TRef LJ_FASTCALL lj_opt_fwd_uload(jit_State *J);
LJ_FUNC TRef LJ_FASTCALL lj_opt_fwd_fload(jit_State *J);
LJ_FUNC TRef LJ_FASTCALL lj_opt_fwd_xload(jit_State *J);
LJ_FUNC TRef LJ_FASTCALL lj_opt_fwd_tab_len(jit_State *J);
LJ_FUNC TRef LJ_FASTCALL lj_opt_fwd_hrefk(jit_State *J);
LJ_FUNC int LJ_FASTCALL lj_opt_fwd_href_nokey(jit_State *J);
LJ_FUNC int LJ_FASTCALL lj_opt_fwd_tptr(jit_State *J, IRRef lim);
LJ_FUNC int lj_opt_fwd_wasnonnil(jit_State *J, IROpT loadop, IRRef xref);

/* Dead-store elimination. */
LJ_FUNC TRef LJ_FASTCALL lj_opt_dse_ahstore(jit_State *J);
LJ_FUNC TRef LJ_FASTCALL lj_opt_dse_ustore(jit_State *J);
LJ_FUNC TRef LJ_FASTCALL lj_opt_dse_fstore(jit_State *J);
LJ_FUNC TRef LJ_FASTCALL lj_opt_dse_xstore(jit_State *J);

/* Narrowing. */
LJ_FUNC TRef LJ_FASTCALL lj_opt_narrow_convert(jit_State *J);
LJ_FUNC TRef LJ_FASTCALL lj_opt_narrow_index(jit_State *J, TRef key);
LJ_FUNC TRef LJ_FASTCALL lj_opt_narrow_toint(jit_State *J, TRef tr);
LJ_FUNC TRef LJ_FASTCALL lj_opt_narrow_tobit(jit_State *J, TRef tr);
#if LJ_HASFFI
LJ_FUNC TRef LJ_FASTCALL lj_opt_narrow_cindex(jit_State *J, TRef key);
#endif
LJ_FUNC TRef lj_opt_narrow_arith(jit_State *J, TRef rb, TRef rc,
                                 TValue *vb, TValue *vc, IROp op);
LJ_FUNC TRef lj_opt_narrow_unm(jit_State *J, TRef rc, TValue *vc);
LJ_FUNC TRef lj_opt_narrow_mod(jit_State *J, TRef rb, TRef rc, TValue *vc);
LJ_FUNC TRef lj_opt_narrow_pow(jit_State *J, TRef rb, TRef rc, TValue *vc);
LJ_FUNC IRType lj_opt_narrow_forl(jit_State *J, cTValue *forbase);

/* Optimization passes. */
LJ_FUNC void lj_opt_dce(jit_State *J);
LJ_FUNC int lj_opt_loop(jit_State *J);
#if LJ_SOFTFP || (LJ_32 && LJ_HASFFI)
LJ_FUNC void lj_opt_split(jit_State *J);
#else
#define lj_opt_split(J)                UNUSED(J)
#endif
```

```
157 LJ_FUNC void lj_opt_sink(jit_State *J);
158
159 #endif
160
161 #endif
```

One Level Up Top Level

src/lj_ir.c - luajit-2.0-src

Global variables defined

- lj_ir_callinfo
- lj_ir_mode
- lj_ir_type_size

Data types defined

- K64Array
- K64Array

Functions defined

- ir_nextk
- lj_ir_call
- lj_ir_emit
- lj_ir_growbot
- lj_ir_growtop
- lj_ir_k64
- lj_ir_k64_find
- lj_ir_k64_freeall
- lj_ir_kgc
- lj_ir_kint
- lj_ir_kint64
- lj_ir_knull
- lj_ir_knum_u64
- lj_ir_knumint
- lj_ir_kptr_
- lj_ir_kslot
- lj_ir_kvalue
- lj_ir_numcmp
- lj_ir_rollback
- lj_ir_strcmp
- lj_ir_tonum
- lj_ir_tonumber

- lj_ir_tostr
- numistrueint

Macros defined

- IR
- IR
- IRCALLCI
- IRCALLCI
- IRTSIZE
- IRTSIZE
- LUA_CORE
- emitir
- emitir
- fins
- fins
- lj_ir_c

Source code

```
/*
** SSA IR (Intermediate Representation) emitter.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_ir_c
#define LUA_CORE

/* For pointers to libc/libm functions. */
 #include <stdio.h>
 #include <math.h>

 #include "lj_obj.h"

 #if LJ_HASJIT

 #include "lj_gc.h"
 #include "lj_buf.h"
 #include "lj_str.h"
 #include "lj_tab.h"
 #include "lj_ir.h"
 #include "lj_jit.h"
 #include "lj_ircall.h"
 #include "lj_iropt.h"
 #include "lj_trace.h"
 #if LJ_HASFFI
 #include "lj_ctype.h"
 #include "lj_cdata.h"
 #include "lj_carith.h"
 #endif
 #include "lj_vm.h"
 #include "lj_strscan.h"
 #include "lj_strfmt.h"
 #include "lj_lib.h"

 /* Some local macros to save typing. Undef'd at the end. */
 #define IR(ref)                         (&J->cur.ir[(ref)])
```

```c
#define fins                        (&J->fold.ins)

/* Pass IR on to next optimization in chain (FOLD). */
#define emitir(ot, a, b)        (lj_ir_set(J, (ot), (a), (b)), lj_opt_fold(J))

/* -- IR tables ------------------------------------------------------------ */

/* IR instruction modes. */
LJ_DATADEF const uint8_t lj_ir_mode[IR__MAX+1] = {
IRDEF(IRMODE)
  0
};

/* IR type sizes. */
LJ_DATADEF const uint8_t lj_ir_type_size[IRT__MAX+1] = {
#define IRTSIZE(name, size)        size,
IRTDEF(IRTSIZE)
#undef IRTSIZE
  0
};

/* C call info for CALL* instructions. */
LJ_DATADEF const CCallInfo lj_ir_callinfo[] = {
#define IRCALLCI(cond, name, nargs, kind, type, flags) \
  { (ASMFunction)IRCALLCOND_##cond(name), \
    (nargs)|(CCI_CALL_##kind)|(IRT_##type<<CCI_OTSHIFT)|(flags) },
IRCALLDEF(IRCALLCI)
#undef IRCALLCI
  { NULL, 0 }
};

/* -- IR emitter ----------------------------------------------------------- */

/* Grow IR buffer at the top. */
void LJ_FASTCALL lj_ir_growtop(jit_State *J)
{
  IRIns *baseir = J->irbuf + J->irbotlim;
  MSize szins = J->irtoplim - J->irbotlim;
  if (szins) {
    baseir = (IRIns *)lj_mem_realloc(J->L, baseir, szins*sizeof(IRIns),
                                     2*szins*sizeof(IRIns));
    J->irtoplim = J->irbotlim + 2*szins;
  } else {
    baseir = (IRIns *)lj_mem_realloc(J->L, NULL, 0, LJ_MIN_IRSZ*sizeof(IRIns));
    J->irbotlim = REF_BASE - LJ_MIN_IRSZ/4;
    J->irtoplim = J->irbotlim + LJ_MIN_IRSZ;
  }
  J->cur.ir = J->irbuf = baseir - J->irbotlim;
}

/* Grow IR buffer at the bottom or shift it up. */
static void lj_ir_growbot(jit_State *J)
{
  IRIns *baseir = J->irbuf + J->irbotlim;
  MSize szins = J->irtoplim - J->irbotlim;
  lua_assert(szins != 0);
  lua_assert(J->cur.nk == J->irbotlim);
  if (J->cur.nins + (szins >> 1) < J->irtoplim) {
    /* More than half of the buffer is free on top: shift up by a quarter. */
    MSize ofs = szins >> 2;
    memmove(baseir + ofs, baseir, (J->cur.nins - J->irbotlim)*sizeof(IRIns));
    J->irbotlim -= ofs;
     J->irtoplim -= ofs;
     J->cur.ir = J->irbuf = baseir - J->irbotlim;
   } else {
    /* Double the buffer size, but split the growth amongst top/bottom. */
    IRIns *newbase = lj_mem_newt(J->L, 2*szins*sizeof(IRIns), IRIns);
    MSize ofs = szins >= 256 ? 128 : (szins >> 1);  /* Limit bottom growth. */
    memcpy(newbase + ofs, baseir, (J->cur.nins - J->irbotlim)*sizeof(IRIns));
    lj_mem_free(G(J->L), baseir, szins*sizeof(IRIns));
    J->irbotlim -= ofs;
    J->irtoplim = J->irbotlim + 2*szins;
    J->cur.ir = J->irbuf = newbase - J->irbotlim;
   }
 }
```

```c
/* Emit IR without any optimizations. */
TRef LJ_FASTCALL lj_ir_emit(jit_State *J)
{
  IRRef ref = lj_ir_nextins(J);
  IRIns *ir = IR(ref);
  IROp op = fins->o;
  ir->prev = J->chain[op];
  J->chain[op] = (IRRef1)ref;
  ir->o = op;
  ir->op1 = fins->op1;
  ir->op2 = fins->op2;
  J->guardemit.irt |= fins->t.irt;
  return TREF(ref, irt_t((ir->t = fins->t)));
}

/* Emit call to a C function. */
TRef lj_ir_call(jit_State *J, IRCallID id, ...)
{
  const CCallInfo *ci = &lj_ir_callinfo[id];
  uint32_t n = CCI_NARGS(ci);
  TRef tr = TREF_NIL;
  va_list argp;
  va_start(argp, id);
  if ((ci->flags & CCI_L)) n--;
  if (n > 0)
    tr = va_arg(argp, IRRef);
  while (n-- > 1)
    tr = emitir(IRT(IR_CARG, IRT_NIL), tr, va_arg(argp, IRRef));
  va_end(argp);
  if (CCI_OP(ci) == IR_CALLS)
    J->needsnap = 1;  /* Need snapshot after call with side effect. */
  return emitir(CCI_OPTYPE(ci), tr, id);
}

/* -- Interning of constants ---------------------------------------------- */

/*
** IR instructions for constants are kept between J->cur.nk >= ref < REF_BIAS.
** They are chained like all other instructions, but grow downwards.
** The are interned (like strings in the VM) to facilitate reference
** comparisons. The same constant must get the same reference.
*/

/* Get ref of next IR constant and optionally grow IR.
** Note: this may invalidate all IRIns *!
*/
static LJ_AINLINE IRRef ir_nextk(jit_State *J)
{
  IRRef ref = J->cur.nk;
  if (LJ_UNLIKELY(ref <= J->irbotlim)) lj_ir_growbot(J);
  J->cur.nk = --ref;
  return ref;
}

/* Intern int32_t constant. */
TRef LJ_FASTCALL lj_ir_kint(jit_State *J, int32_t k)
{
  IRIns *ir, *cir = J->cur.ir;
  IRRef ref;
  for (ref = J->chain[IR_KINT]; ref; ref = cir[ref].prev)
    if (cir[ref].i == k)
      goto found;
  ref = ir_nextk(J);
  ir = IR(ref);
  ir->i = k;
  ir->t.irt = IRT_INT;
  ir->o = IR_KINT;
  ir->prev = J->chain[IR_KINT];
  J->chain[IR_KINT] = (IRRef1)ref;
found:
  return TREF(ref, IRT_INT);
}

/* The MRef inside the KNUM/KINT64 IR instructions holds the address of the
** 64 bit constant. The constants themselves are stored in a chained array
** and shared across traces.
```

```
**
** Rationale for choosing this data structure:
** - The address of the constants is embedded in the generated machine code
**   and must never move. A resizable array or hash table wouldn't work.
** - Most apps need very few non-32 bit integer constants (less than a dozen).
** - Linear search is hard to beat in terms of speed and low complexity.
*/
typedef struct K64Array {
  MRef next;                        /* Pointer to next list. */
  MSize numk;                        /* Number of used elements in this array. */
  TValue k[LJ_MIN_K64SZ];        /* Array of constants. */
} K64Array;

/* Free all chained arrays. */
void lj_ir_k64_freeall(jit_State *J)
{
  K64Array *k;
  for (k = mref(J->k64, K64Array); k; ) {
    K64Array *next = mref(k->next, K64Array);
    lj_mem_free(J2G(J), k, sizeof(K64Array));
    k = next;
  }
}

/* Find 64 bit constant in chained array or add it. */
cTValue *lj_ir_k64_find(jit_State *J, uint64_t u64)
{
  K64Array *k, *kp = NULL;
  TValue *ntv;
  MSize idx;
  /* Search for the constant in the whole chain of arrays. */
  for (k = mref(J->k64, K64Array); k; k = mref(k->next, K64Array)) {
    kp = k;  /* Remember previous element in list. */
    for (idx = 0; idx < k->numk; idx++) {  /* Search one array. */
      TValue *tv = &k->k[idx];
      if (tv->u64 == u64)  /* Needed for +-0/NaN/absmask. */
        return tv;
    }
  }
  /* Constant was not found, need to add it. */
  if (!(kp && kp->numk < LJ_MIN_K64SZ)) {  /* Allocate a new array. */
    K64Array *kn = lj_mem_newt(J->L, sizeof(K64Array), K64Array);
    setmref(kn->next, NULL);
    kn->numk = 0;
    if (kp)
      setmref(kp->next, kn);  /* Chain to the end of the list. */
    else
      setmref(J->k64, kn);  /* Link first array. */
    kp = kn;
  }
  ntv = &kp->k[kp->numk++];  /* Add to current array. */
  ntv->u64 = u64;
  return ntv;
}

/* Intern 64 bit constant, given by its address. */
TRef lj_ir_k64(jit_State *J, IROp op, cTValue *tv)
{
  IRIns *ir, *cir = J->cur.ir;
  IRRef ref;
  IRType t = op == IR_KNUM ? IRT_NUM : IRT_I64;
  for (ref = J->chain[op]; ref; ref = cir[ref].prev)
    if (ir_k64(&cir[ref]) == tv)
      goto found;
  ref = ir_nextk(J);
  ir = IR(ref);
  lua_assert(checkptrGC(tv));
  setmref(ir->ptr, tv);
  ir->t.irt = t;
  ir->o = op;
  ir->prev = J->chain[op];
  J->chain[op] = (IRRef1)ref;
found:
  return TREF(ref, t);
}
```

```c
/* Intern FP constant, given by its 64 bit pattern. */
TRef lj_ir_knum_u64(jit_State *J, uint64_t u64)
{
  return lj_ir_k64(J, IR_KNUM, lj_ir_k64_find(J, u64));
}

/* Intern 64 bit integer constant. */
TRef lj_ir_kint64(jit_State *J, uint64_t u64)
{
  return lj_ir_k64(J, IR_KINT64, lj_ir_k64_find(J, u64));
}

/* Check whether a number is int and return it. -0 is NOT considered an int. */
static int numistrueint(lua_Number n, int32_t *kp)
{
  int32_t k = lj_num2int(n);
  if (n == (lua_Number)k) {
    if (kp) *kp = k;
    if (k == 0) {  /* Special check for -0. */
      TValue tv;
      setnumV(&tv, n);
      if (tv.u32.hi != 0)
        return 0;
    }
    return 1;
  }
  return 0;
}

/* Intern number as int32_t constant if possible, otherwise as FP constant. */
TRef lj_ir_knumint(jit_State *J, lua_Number n)
{
  int32_t k;
  if (numistrueint(n, &k))
    return lj_ir_kint(J, k);
  else
    return lj_ir_knum(J, n);
}

/* Intern GC object "constant". */
TRef lj_ir_kgc(jit_State *J, GCobj *o, IRType t)
{
  IRIns *ir, *cir = J->cur.ir;
  IRRef ref;
  lua_assert(!LJ_GC64);  /* TODO_GC64: major changes required. */
  lua_assert(!isdead(J2G(J), o));
  for (ref = J->chain[IR_KGC]; ref; ref = cir[ref].prev)
    if (ir_kgc(&cir[ref]) == o)
      goto found;
  ref = ir_nextk(J);
  ir = IR(ref);
  /* NOBARRIER: Current trace is a GC root. */
  setgcref(ir->gcr, o);
  ir->t.irt = (uint8_t)t;
  ir->o = IR_KGC;
  ir->prev = J->chain[IR_KGC];
  J->chain[IR_KGC] = (IRRef1)ref;
found:
  return TREF(ref, t);
}

/* Intern 32 bit pointer constant. */
TRef lj_ir_kptr_(jit_State *J, IROp op, void *ptr)
{
  IRIns *ir, *cir = J->cur.ir;
  IRRef ref;
  lua_assert((void *)(intptr_t)i32ptr(ptr) == ptr);
  for (ref = J->chain[op]; ref; ref = cir[ref].prev)
    if (mref(cir[ref].ptr, void) == ptr)
      goto found;
  ref = ir_nextk(J);
  ir = IR(ref);
  setmref(ir->ptr, ptr);
  ir->t.irt = IRT_P32;
  ir->o = op;
  ir->prev = J->chain[op];
```

```
  J->chain[op] = (IRRef1)ref;
found:
  return TREF(ref, IRT_P32);
}

/* Intern typed NULL constant. */
TRef lj_ir_knull(jit_State *J, IRType t)
{
  IRIns *ir, *cir = J->cur.ir;
  IRRef ref;
  for (ref = J->chain[IR_KNULL]; ref; ref = cir[ref].prev)
    if (irt_t(cir[ref].t) == t)
      goto found;
  ref = ir_nextk(J);
  ir = IR(ref);
  ir->i = 0;
  ir->t.irt = (uint8_t)t;
  ir->o = IR_KNULL;
  ir->prev = J->chain[IR_KNULL];
  J->chain[IR_KNULL] = (IRRef1)ref;
found:
  return TREF(ref, t);
}

/* Intern key slot. */
TRef lj_ir_kslot(jit_State *J, TRef key, IRRef slot)
{
  IRIns *ir, *cir = J->cur.ir;
  IRRef2 op12 = IRREF2((IRRef1)key, (IRRef1)slot);
  IRRef ref;
  /* Const part is not touched by CSE/DCE, so 0-65535 is ok for IRMlit here. */
  lua_assert(tref_isk(key) && slot == (IRRef)(IRRef1)slot);
  for (ref = J->chain[IR_KSLOT]; ref; ref = cir[ref].prev)
    if (cir[ref].op12 == op12)
      goto found;
  ref = ir_nextk(J);
  ir = IR(ref);
  ir->op12 = op12;
  ir->t.irt = IRT_P32;
  ir->o = IR_KSLOT;
  ir->prev = J->chain[IR_KSLOT];
  J->chain[IR_KSLOT] = (IRRef1)ref;
found:
  return TREF(ref, IRT_P32);
}

/* -- Access to IR constants ---------------------------------------------- */

/* Copy value of IR constant. */
void lj_ir_kvalue(lua_State *L, TValue *tv, const IRIns *ir)
{
  UNUSED(L);
  lua_assert(ir->o != IR_KSLOT);  /* Common mistake. */
  switch (ir->o) {
  case IR_KPRI: setpriV(tv, irt_toitype(ir->t)); break;
  case IR_KINT: setintV(tv, ir->i); break;
  case IR_KGC: setgcV(L, tv, ir_kgc(ir), irt_toitype(ir->t)); break;
  case IR_KPTR: case IR_KKPTR: case IR_KNULL:
    setlightudV(tv, mref(ir->ptr, void));
    break;
  case IR_KNUM: setnumV(tv, ir_knum(ir)->n); break;
#if LJ_HASFFI
  case IR_KINT64: {
    GCcdata *cd = lj_cdata_new(L, CTID_INT64, 8);
    *(uint64_t *)cdataptr(cd) = ir_kint64(ir)->u64;
    setcdataV(L, tv, cd);
    break;
    }
#endif
  default: lua_assert(0); break;
  }
}

/* -- Convert IR operand types -------------------------------------------- */

/* Convert from string to number. */
```

```
TRef LJ_FASTCALL lj_ir_tonumber(jit_State *J, TRef tr)
{
  if (!tref_isnumber(tr)) {
    if (tref_isstr(tr))
      tr = emitir(IRTG(IR_STRTO, IRT_NUM), tr, 0);
    else
      lj_trace_err(J, LJ_TRERR_BADTYPE);
  }
  return tr;
}

/* Convert from integer or string to number. */
TRef LJ_FASTCALL lj_ir_tonum(jit_State *J, TRef tr)
{
  if (!tref_isnum(tr)) {
    if (tref_isinteger(tr))
      tr = emitir(IRTN(IR_CONV), tr, IRCONV_NUM_INT);
    else if (tref_isstr(tr))
      tr = emitir(IRTG(IR_STRTO, IRT_NUM), tr, 0);
    else
      lj_trace_err(J, LJ_TRERR_BADTYPE);
  }
  return tr;
}

/* Convert from integer or number to string. */
TRef LJ_FASTCALL lj_ir_tostr(jit_State *J, TRef tr)
{
  if (!tref_isstr(tr)) {
    if (!tref_isnumber(tr))
      lj_trace_err(J, LJ_TRERR_BADTYPE);
    tr = emitir(IRT(IR_TOSTR, IRT_STR), tr,
                tref_isnum(tr) ? IRTOSTR_NUM : IRTOSTR_INT);
  }
  return tr;
}

/* -- Miscellaneous IR ops ------------------------------------------------ */

/* Evaluate numeric comparison. */
int lj_ir_numcmp(lua_Number a, lua_Number b, IROp op)
{
  switch (op) {
  case IR_EQ: return (a == b);
  case IR_NE: return (a != b);
  case IR_LT: return (a < b);
  case IR_GE: return (a >= b);
  case IR_LE: return (a <= b);
  case IR_GT: return (a > b);
  case IR_ULT: return !(a >= b);
  case IR_UGE: return !(a < b);
  case IR_ULE: return !(a > b);
  case IR_UGT: return !(a <= b);
  default: lua_assert(0); return 0;
  }
}

/* Evaluate string comparison. */
int lj_ir_strcmp(GCstr *a, GCstr *b, IROp op)
{
  int res = lj_str_cmp(a, b);
  switch (op) {
  case IR_LT: return (res < 0);
  case IR_GE: return (res >= 0);
  case IR_LE: return (res <= 0);
  case IR_GT: return (res > 0);
  default: lua_assert(0); return 0;
  }
}

/* Rollback IR to previous state. */
void lj_ir_rollback(jit_State *J, IRRef ref)
{
  IRRef nins = J->cur.nins;
  while (nins > ref) {
    IRIns *ir;
```

```
494     nins--;
495     ir = IR(nins);
496     J->chain[ir->o] = ir->prev;
497   }
498   J->cur.nins = nins;
499 }
500 
501 #undef IR
502 #undef fins
503 #undef emitir
504 
505 #endif
```

One Level Up Top Level

src/lj_ircall.h - luajit-2.0-src

Global variables defined

- lj_ir_callinfo

Data types defined

- CCallInfo
- CCallInfo
- IRCallID

Macros defined

- CCI_CALL_A
- CCI_CALL_FL
- CCI_CALL_FN
- CCI_CALL_FS
- CCI_CALL_L
- CCI_CALL_N
- CCI_CALL_S
- CCI_CASTU64
- CCI_CC_CDECL
- CCI_CC_FASTCALL
- CCI_CC_MASK
- CCI_CC_SHIFT
- CCI_CC_STDCALL
- CCI_CC_THISCALL
- CCI_L
- CCI_NARGS
- CCI_NARGS_MAX
- CCI_NOFPRCLOBBER
- CCI_OP
- CCI_OPSHIFT
- CCI_OPTYPE
- CCI_OTSHIFT
- CCI_RANDFPR

- CCI_RANDFPR
- CCI_VARARG
- CCI_XA
- CCI_XARGS
- CCI_XARGS_SHIFT
- CCI_XNARGS
- CCI_XNARGS
- IRCALLCOND_ANY
- IRCALLCOND_FFI
- IRCALLCOND_FFI
- IRCALLCOND_FFI32
- IRCALLCOND_FFI32
- IRCALLCOND_FFI32
- IRCALLCOND_FP64_FFI
- IRCALLCOND_FP64_FFI
- IRCALLCOND_FPMATH
- IRCALLCOND_FPMATH
- IRCALLCOND_SOFTFP
- IRCALLCOND_SOFTFP
- IRCALLCOND_SOFTFP_FFI
- IRCALLCOND_SOFTFP_FFI
- IRCALLCOND_SOFTFP_FFI
- IRCALLDEF
- IRCALLENUM
- IRCALLENUM
- LJ_NEED_FP64
- XA2_64
- XA2_64
- XA2_FP
- XA2_FP
- XA_64
- XA_64
- XA_FP

- XA_FP
- _LJ_IRCALL_H
- fp64_d2l
- fp64_d2l
- fp64_d2l
- fp64_d2ul
- fp64_d2ul
- fp64_d2ul
- fp64_f2l
- fp64_f2l
- fp64_f2l
- fp64_f2ul
- fp64_f2ul
- fp64_f2ul
- fp64_l2d
- fp64_l2d
- fp64_l2f
- fp64_l2f
- fp64_ul2d
- fp64_ul2d
- fp64_ul2f
- fp64_ul2f
- softfp_add
- softfp_cmp
- softfp_d2f
- softfp_d2i
- softfp_d2ui
- softfp_div
- softfp_f2d
- softfp_f2i
- softfp_f2ui
- softfp_i2d
- softfp_i2f

- softfp_mul
- softfp_sub
- softfp_ui2d
- softfp_ui2f

Source code

```
/*
** IR CALL* instruction definitions.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_IRCALL_H
#define _LJ_IRCALL_H

#include "lj_obj.h"
 #include "lj_ir.h"
 #include "lj_jit.h"

 /* C call info for CALL* instructions. */
 typedef struct CCallInfo {
   ASMFunction func;               /* Function pointer. */
   uint32_t flags;               /* Number of arguments and flags. */
 } CCallInfo;

 #define CCI_NARGS(ci)               ((ci)->flags & 0xff)        /* # of args. */
 #define CCI_NARGS_MAX               32                         /* Max. # of args. */

 #define CCI_OTSHIFT                16
 #define CCI_OPTYPE(ci)                ((ci)->flags >> CCI_OTSHIFT)  /* Get op/type. */
 #define CCI_OPSHIFT                24
 #define CCI_OP(ci)                ((ci)->flags >> CCI_OPSHIFT)  /* Get op. */

 #define CCI_CALL_N                (IR_CALLN << CCI_OPSHIFT)
 #define CCI_CALL_A                (IR_CALLA << CCI_OPSHIFT)
 #define CCI_CALL_L                (IR_CALLL << CCI_OPSHIFT)
 #define CCI_CALL_S                (IR_CALLS << CCI_OPSHIFT)
 #define CCI_CALL_FN                (CCI_CALL_N|CCI_CC_FASTCALL)
 #define CCI_CALL_FL                (CCI_CALL_L|CCI_CC_FASTCALL)
 #define CCI_CALL_FS                (CCI_CALL_S|CCI_CC_FASTCALL)

 /* C call info flags. */
 #define CCI_L                        0x0100        /* Implicit L arg. */
 #define CCI_CASTU64                0x0200        /* Cast u64 result to number. */
 #define CCI_NOFPRCLOBBER        0x0400        /* Does not clobber any FPRs. */
 #define CCI_VARARG                0x0800        /* Vararg function. */

 #define CCI_CC_MASK                0x3000        /* Calling convention mask. */
 #define CCI_CC_SHIFT                12
 /* ORDER CC */
 #define CCI_CC_CDECL                0x0000        /* Default cdecl calling convention. */
 #define CCI_CC_THISCALL                0x1000        /* Thiscall calling convention. */
 #define CCI_CC_FASTCALL                0x2000        /* Fastcall calling convention. */
 #define CCI_CC_STDCALL                0x3000        /* Stdcall calling convention. */

 /* Extra args for SOFTFP, SPLIT 64 bit. */
 #define CCI_XARGS_SHIFT                14
 #define CCI_XARGS(ci)                (((ci)->flags >> CCI_XARGS_SHIFT) & 3)
 #define CCI_XA                        (1u << CCI_XARGS_SHIFT)

 #if LJ_SOFTFP || (LJ_32 && LJ_HASFFI)
 #define CCI_XNARGS(ci)                (CCI_NARGS((ci)) + CCI_XARGS((ci)))
 #else
 #define CCI_XNARGS(ci)                CCI_NARGS((ci))
 #endif

 /* Helpers for conditional function definitions. */
 #define IRCALLCOND_ANY(x)                x

 #if LJ_TARGET_X86ORX64
```

```
64  #define IRCALLCOND_FPMATH(x)                NULL
65  #else
66  #define IRCALLCOND_FPMATH(x)                x
67  #endif
68
69  #if LJ_SOFTFP
70  #define IRCALLCOND_SOFTFP(x)                x
71  #if LJ_HASFFI
72  #define IRCALLCOND_SOFTFP_FFI(x)        x
73  #else
74  #define IRCALLCOND_SOFTFP_FFI(x)        NULL
75  #endif
76  #else
77  #define IRCALLCOND_SOFTFP(x)                NULL
78  #define IRCALLCOND_SOFTFP_FFI(x)        NULL
79  #endif
80
81  #define LJ_NEED_FP64        (LJ_TARGET_ARM || LJ_TARGET_PPC || LJ_TARGET_MIPS)
82
83  #if LJ_HASFFI && (LJ_SOFTFP || LJ_NEED_FP64)
84  #define IRCALLCOND_FP64_FFI(x)                x
85  #else
86  #define IRCALLCOND_FP64_FFI(x)                NULL
87  #endif
88
89  #if LJ_HASFFI
90  #define IRCALLCOND_FFI(x)                x
91  #if LJ_32
92  #define IRCALLCOND_FFI32(x)                x
93  #else
94  #define IRCALLCOND_FFI32(x)                NULL
95  #endif
96  #else
97  #define IRCALLCOND_FFI(x)                NULL
98  #define IRCALLCOND_FFI32(x)                NULL
99  #endif
100
101 #if LJ_TARGET_X86
102 #define CCI_RANDFPR        0        /* Clang on OSX/x86 is overzealous. */
103 #else
104 #define CCI_RANDFPR        CCI_NOFPRCLOBBER
105 #endif
106
107 #if LJ_SOFTFP
108 #define XA_FP                CCI_XA
109 #define XA2_FP                (CCI_XA+CCI_XA)
110 #else
111 #define XA_FP                0
112 #define XA2_FP                0
113 #endif
114
115 #if LJ_32
116 #define XA_64                CCI_XA
117 #define XA2_64                (CCI_XA+CCI_XA)
118 #else
119 #define XA_64                0
120 #define XA2_64                0
121 #endif
122
123 /* Function definitions for CALL* instructions. */
124 #define IRCALLDEF(_) \
125   _(ANY,        lj_str_cmp,                2,  FN, INT, CCI_NOFPRCLOBBER) \
126   _(ANY,        lj_str_find,                4,   N, P32, 0) \
127   _(ANY,        lj_str_new,                3,   S, STR, CCI_L) \
128   _(ANY,        lj_strscan_num,                2,  FN, INT, 0) \
129   _(ANY,        lj_strfmt_int,                2,  FN, STR, CCI_L) \
130   _(ANY,        lj_strfmt_num,                2,  FN, STR, CCI_L) \
131   _(ANY,        lj_strfmt_char,                2,  FN, STR, CCI_L) \
132   _(ANY,        lj_strfmt_putint,        2,  FL, P32, 0) \
133   _(ANY,        lj_strfmt_putnum,        2,  FL, P32, 0) \
134   _(ANY,        lj_strfmt_putquoted,        2,  FL, P32, 0) \
135   _(ANY,        lj_strfmt_putfxint,        3,   L, P32, XA_64) \
136   _(ANY,        lj_strfmt_putfnum_int,        3,   L, P32, XA_FP) \
137   _(ANY,        lj_strfmt_putfnum_uint,        3,   L, P32, XA_FP) \
138   _(ANY,        lj_strfmt_putfnum,        3,   L, P32, XA_FP) \
139   _(ANY,        lj_strfmt_putfstr,        3,   L, P32, 0) \
```

```
  _(ANY,        lj_strfmt_putfchar,        3,   L, P32, 0) \
  _(ANY,        lj_buf_putmem,               3,   S, P32, 0) \
  _(ANY,        lj_buf_putstr,               2,  FL, P32, 0) \
  _(ANY,        lj_buf_putchar,               2,  FL, P32, 0) \
  _(ANY,        lj_buf_putstr_reverse,       2,  FL, P32, 0) \
  _(ANY,        lj_buf_putstr_lower,        2,  FL, P32, 0) \
  _(ANY,        lj_buf_putstr_upper,        2,  FL, P32, 0) \
  _(ANY,        lj_buf_putstr_rep,        3,   L, P32, 0) \
  _(ANY,        lj_buf_puttab,               5,   L, P32, 0) \
  _(ANY,        lj_buf_tostr,               1,  FL, STR, 0) \
  _(ANY,        lj_tab_new_ah,               3,   A, TAB, CCI_L) \
  _(ANY,        lj_tab_new1,                2,  FS, TAB, CCI_L) \
  _(ANY,        lj_tab_dup,                 2,  FS, TAB, CCI_L) \
  _(ANY,        lj_tab_clear,                1,  FS, NIL, 0) \
  _(ANY,        lj_tab_newkey,               3,   S, P32, CCI_L) \
  _(ANY,        lj_tab_len,                 1,  FL, INT, 0) \
  _(ANY,        lj_gc_step_jit,               2,  FS, NIL, CCI_L) \
  _(ANY,        lj_gc_barrieruv,        2,  FS, NIL, 0) \
  _(ANY,        lj_mem_newgco,               2,  FS, P32, CCI_L) \
  _(ANY,        lj_math_random_step, 1, FS, NUM, CCI_CASTU64|CCI_RANDFPR)\
  _(ANY,        lj_vm_modi,                 2,  FN, INT, 0) \
  _(ANY,        sinh,                        1,   N, NUM, XA_FP) \
  _(ANY,        cosh,                        1,   N, NUM, XA_FP) \
  _(ANY,        tanh,                        1,   N, NUM, XA_FP) \
  _(ANY,        fputc,                       2,   S, INT, 0) \
  _(ANY,        fwrite,                       4,   S, INT, 0) \
  _(ANY,        fflush,                       1,   S, INT, 0) \
  /* ORDER FPM */ \
  _(FPMATH,        lj_vm_floor,               1,   N, NUM, XA_FP) \
  _(FPMATH,        lj_vm_ceil,               1,   N, NUM, XA_FP) \
  _(FPMATH,        lj_vm_trunc,               1,   N, NUM, XA_FP) \
  _(FPMATH,        sqrt,                       1,   N, NUM, XA_FP) \
  _(ANY,        exp,                       1,   N, NUM, XA_FP) \
  _(ANY,        lj_vm_exp2,                1,   N, NUM, XA_FP) \
  _(ANY,        log,                       1,   N, NUM, XA_FP) \
  _(ANY,        lj_vm_log2,                1,   N, NUM, XA_FP) \
  _(ANY,        log10,                       1,   N, NUM, XA_FP) \
  _(ANY,        sin,                       1,   N, NUM, XA_FP) \
  _(ANY,        cos,                       1,   N, NUM, XA_FP) \
  _(ANY,        tan,                       1,   N, NUM, XA_FP) \
  _(ANY,        lj_vm_powi,                2,   N, NUM, XA_FP) \
  _(ANY,        pow,                       2,   N, NUM, XA2_FP) \
  _(ANY,        atan2,                       2,   N, NUM, XA2_FP) \
  _(ANY,        ldexp,                       2,   N, NUM, XA_FP) \
  _(SOFTFP,        lj_vm_tobit,               2,   N, INT, 0) \
  _(SOFTFP,        softfp_add,               4,   N, NUM, 0) \
  _(SOFTFP,        softfp_sub,               4,   N, NUM, 0) \
  _(SOFTFP,        softfp_mul,               4,   N, NUM, 0) \
  _(SOFTFP,        softfp_div,               4,   N, NUM, 0) \
  _(SOFTFP,        softfp_cmp,               4,   N, NIL, 0) \
  _(SOFTFP,        softfp_i2d,               1,   N, NUM, 0) \
  _(SOFTFP,        softfp_d2i,               2,   N, INT, 0) \
  _(SOFTFP_FFI,        softfp_ui2d,               1,   N, NUM, 0) \
  _(SOFTFP_FFI,        softfp_f2d,              1,   N, NUM, 0) \
  _(SOFTFP_FFI,        softfp_d2ui,               2,   N, INT, 0) \
  _(SOFTFP_FFI,        softfp_d2f,              2,   N, FLOAT, 0) \
  _(SOFTFP_FFI,        softfp_i2f,              1,   N, FLOAT, 0) \
  _(SOFTFP_FFI,        softfp_ui2f,               1,   N, FLOAT, 0) \
  _(SOFTFP_FFI,        softfp_f2i,              1,   N, INT, 0) \
  _(SOFTFP_FFI,        softfp_f2ui,               1,   N, INT, 0) \
  _(FP64_FFI,        fp64_l2d,               1,   N, NUM, XA_64) \
  _(FP64_FFI,        fp64_ul2d,               1,   N, NUM, XA_64) \
  _(FP64_FFI,        fp64_l2f,               1,   N, FLOAT, XA_64) \
  _(FP64_FFI,        fp64_ul2f,               1,   N, FLOAT, XA_64) \
  _(FP64_FFI,        fp64_d2l,               1,   N, I64, XA_FP) \
  _(FP64_FFI,        fp64_d2ul,               1,   N, U64, XA_FP) \
  _(FP64_FFI,        fp64_f2l,               1,   N, I64, 0) \
  _(FP64_FFI,        fp64_f2ul,               1,   N, U64, 0) \
  _(FFI,        lj_carith_divi64,        2,   N, I64, XA2_64|CCI_NOFPRCLOBBER) \
  _(FFI,        lj_carith_divu64,        2,   N, U64, XA2_64|CCI_NOFPRCLOBBER) \
  _(FFI,        lj_carith_modi64,        2,   N, I64, XA2_64|CCI_NOFPRCLOBBER) \
  _(FFI,        lj_carith_modu64,        2,   N, U64, XA2_64|CCI_NOFPRCLOBBER) \
  _(FFI,        lj_carith_powi64,        2,   N, I64, XA2_64|CCI_NOFPRCLOBBER) \
  _(FFI,        lj_carith_powu64,        2,   N, U64, XA2_64|CCI_NOFPRCLOBBER) \
  _(FFI,        lj_cdata_newv,               4,   S, CDATA, CCI_L) \
  _(FFI,        lj_cdata_setfin,        4,   S, NIL, CCI_L) \
```

```
  _(FFI,        strlen,                     1,   L, INTP, 0) \
  _(FFI,        memcpy,                     3,   S, PTR, 0) \
  _(FFI,        memset,                     3,   S, PTR, 0) \
  _(FFI,        lj_vm_errno,              0,   S, INT, CCI_NOFPRCLOBBER) \
  _(FFI32,        lj_carith_mul64,        2,   N, I64, XA2_64|CCI_NOFPRCLOBBER) \
  _(FFI32,        lj_carith_shl64,        2,   N, U64, XA_64|CCI_NOFPRCLOBBER) \
  _(FFI32,        lj_carith_shr64,        2,   N, U64, XA_64|CCI_NOFPRCLOBBER) \
  _(FFI32,        lj_carith_sar64,        2,   N, U64, XA_64|CCI_NOFPRCLOBBER) \
  _(FFI32,        lj_carith_rol64,        2,   N, U64, XA_64|CCI_NOFPRCLOBBER) \
  _(FFI32,        lj_carith_ror64,        2,   N, U64, XA_64|CCI_NOFPRCLOBBER) \
  \
  /* End of list. */

typedef enum {
#define IRCALLENUM(cond, name, nargs, kind, type, flags)        IRCALL_##name,
IRCALLDEF(IRCALLENUM)
#undef IRCALLENUM
  IRCALL__MAX
} IRCallID;

LJ_FUNC TRef lj_ir_call(jit_State *J, IRCallID id, ...);

LJ_DATA const CCallInfo lj_ir_callinfo[IRCALL__MAX+1];

/* Soft-float declarations. */
#if LJ_SOFTFP
#if LJ_TARGET_ARM
#define softfp_add __aeabi_dadd
#define softfp_sub __aeabi_dsub
#define softfp_mul __aeabi_dmul
#define softfp_div __aeabi_ddiv
#define softfp_cmp __aeabi_cdcmple
#define softfp_i2d __aeabi_i2d
#define softfp_d2i __aeabi_d2iz
#define softfp_ui2d __aeabi_ui2d
#define softfp_f2d __aeabi_f2d
#define softfp_d2ui __aeabi_d2uiz
#define softfp_d2f __aeabi_d2f
#define softfp_i2f __aeabi_i2f
#define softfp_ui2f __aeabi_ui2f
#define softfp_f2i __aeabi_f2iz
#define softfp_f2ui __aeabi_f2uiz
#define fp64_l2d __aeabi_l2d
#define fp64_ul2d __aeabi_ul2d
#define fp64_l2f __aeabi_l2f
#define fp64_ul2f __aeabi_ul2f
#if LJ_TARGET_IOS
#define fp64_d2l __fixdfdi
#define fp64_d2ul __fixunsdfdi
#define fp64_f2l __fixsfdi
#define fp64_f2ul __fixunssfdi
#else
#define fp64_d2l __aeabi_d2lz
#define fp64_d2ul __aeabi_d2ulz
#define fp64_f2l __aeabi_f2lz
#define fp64_f2ul __aeabi_f2ulz
#endif
#else
#error "Missing soft-float definitions for target architecture"
#endif
extern double softfp_add(double a, double b);
extern double softfp_sub(double a, double b);
extern double softfp_mul(double a, double b);
extern double softfp_div(double a, double b);
extern void softfp_cmp(double a, double b);
extern double softfp_i2d(int32_t a);
extern int32_t softfp_d2i(double a);
#if LJ_HASFFI
extern double softfp_ui2d(uint32_t a);
extern double softfp_f2d(float a);
extern uint32_t softfp_d2ui(double a);
extern float softfp_d2f(double a);
extern float softfp_i2f(int32_t a);
extern float softfp_ui2f(uint32_t a);
extern int32_t softfp_f2i(float a);
extern uint32_t softfp_f2ui(float a);
```

```
#endif
#endif

#if LJ_HASFFI && LJ_NEED_FP64 && !(LJ_TARGET_ARM && LJ_SOFTFP)
#ifdef __GNUC__
#define fp64_l2d __floatdidf
#define fp64_ul2d __floatundidf
#define fp64_l2f __floatdisf
#define fp64_ul2f __floatundisf
#define fp64_d2l __fixdfdi
#define fp64_d2ul __fixunsdfdi
#define fp64_f2l __fixsfdi
#define fp64_f2ul __fixunssfdi
#else
#error "Missing fp64 helper definitions for this compiler"
#endif
#endif

#if LJ_HASFFI && (LJ_SOFTFP || LJ_NEED_FP64)
extern double fp64_l2d(int64_t a);
extern double fp64_ul2d(uint64_t a);
extern float fp64_l2f(int64_t a);
extern float fp64_ul2f(uint64_t a);
extern int64_t fp64_d2l(double a);
extern uint64_t fp64_d2ul(double a);
extern int64_t fp64_f2l(float a);
extern uint64_t fp64_f2ul(float a);
#endif

#endif
```

One Level Up Top Level

src/lj_strscan.c - luajit-2.0-src

Functions defined

- lj_strscan_num
- lj_strscan_number
- lj_strscan_scan
- strscan_dec
- strscan_double
- strscan_hex
- strscan_oct

Macros defined

- DLEN
- DLEN
- DNEXT
- DNEXT
- DPREV
- DPREV
- LUA_CORE
- STRSCAN_DDIG
- STRSCAN_DIG
- STRSCAN_DMASK
- STRSCAN_MAXDIG
- casecmp
- lj_strscan_c

Source code

```
1  /*
2  ** String scanning.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #include <math.h>
7
8  #define lj_strscan_c
9  #define LUA_CORE
10
11  #include "lj_obj.h"
12  #include "lj_char.h"
13  #include "lj_strscan.h"
14
15  /* -- Scanning numbers ---------------------------------------------------- */
16
```

```c
/*
** Rationale for the builtin string to number conversion library:
**
** It removes a dependency on libc's strtod(), which is a true portability
** nightmare. Mainly due to the plethora of supported OS and toolchain
** combinations. Sadly, the various implementations
** a) are often buggy, incomplete (no hex floats) and/or imprecise,
** b) sometimes crash or hang on certain inputs,
** c) return non-standard NaNs that need to be filtered out, and
** d) fail if the locale-specific decimal separator is not a dot,
**    which can only be fixed with atrocious workarounds.
**
** Also, most of the strtod() implementations are hopelessly bloated,
** which is not just an I-cache hog, but a problem for static linkage
** on embedded systems, too.
**
** OTOH the builtin conversion function is very compact. Even though it
** does a lot more, like parsing long longs, octal or imaginary numbers
** and returning the result in different formats:
** a) It needs less than 3 KB (!) of machine code (on x64 with -Os),
** b) it doesn't perform any dynamic allocation and,
** c) it needs only around 600 bytes of stack space.
**
** The builtin function is faster than strtod() for typical inputs, e.g.
** "123", "1.5" or "1e6". Arguably, it's slower for very large exponents,
** which are not very common (this could be fixed, if needed).
**
** And most importantly, the builtin function is equally precise on all
** platforms. It correctly converts and rounds any input to a double.
** If this is not the case, please send a bug report -- but PLEASE verify
** that the implementation you're comparing to is not the culprit!
**
** The implementation quickly pre-scans the entire string first and
** handles simple integers on-the-fly. Otherwise, it dispatches to the
** base-specific parser. Hex and octal is straightforward.
**
** Decimal to binary conversion uses a fixed-length circular buffer in
** base 100. Some simple cases are handled directly. For other cases, the
** number in the buffer is up-scaled or down-scaled until the integer part
** is in the proper range. Then the integer part is rounded and converted
** to a double which is finally rescaled to the result. Denormals need
** special treatment to prevent incorrect 'double rounding'.
*/

/* Definitions for circular decimal digit buffer (base 100 = 2 digits/byte). */
#define STRSCAN_DIG        1024
#define STRSCAN_MAXDIG        800               /* 772 + extra are sufficient. */
#define STRSCAN_DDIG        (STRSCAN_DIG/2)
#define STRSCAN_DMASK        (STRSCAN_DDIG-1)

/* Helpers for circular buffer. */
#define DNEXT(a)        (((a)+1) & STRSCAN_DMASK)
#define DPREV(a)        (((a)-1) & STRSCAN_DMASK)
#define DLEN(lo, hi)        ((int32_t)(((lo)-(hi)) & STRSCAN_DMASK))

#define casecmp(c, k)        (((c) | 0x20) == k)

/* Final conversion to double. */
static void strscan_double(uint64_t x, TValue *o, int32_t ex2, int32_t neg)
{
  double n;

  /* Avoid double rounding for denormals. */
  if (LJ_UNLIKELY(ex2 <= -1075 && x != 0)) {
    /* NYI: all of this generates way too much code on 32 bit CPUs. */
#if defined(__GNUC__) && LJ_64
    int32_t b = (int32_t)(__builtin_clzll(x)^63);
#else
    int32_t b = (x>>32) ? 32+(int32_t)lj_fls((uint32_t)(x>>32)) :
                          (int32_t)lj_fls((uint32_t)x);
#endif
    if ((int32_t)b + ex2 <= -1023 && (int32_t)b + ex2 >= -1075) {
      uint64_t rb = (uint64_t)1 << (-1075-ex2);
      if ((x & rb) && ((x & (rb+rb+rb-1)))) x += rb+rb;
      x = (x & ~(rb+rb-1));
    }
```

```
  }

  /* Convert to double using a signed int64_t conversion, then rescale. */
  lua_assert((int64_t)x >= 0);
  n = (double)(int64_t)x;
  if (neg) n = -n;
  if (ex2) n = ldexp(n, ex2);
   o->n = n;
}

/* Parse hexadecimal number. */
static StrScanFmt strscan_hex(const uint8_t *p, TValue *o,
                              StrScanFmt fmt, uint32_t opt,
                              int32_t ex2, int32_t neg, uint32_t dig)
{
  uint64_t x = 0;
  uint32_t i;

  /* Scan hex digits. */
  for (i = dig > 16 ? 16 : dig ; i; i--, p++) {
    uint32_t d = (*p != '.' ? *p : *++p); if (d > '9') d += 9;
    x = (x << 4) + (d & 15);
  }

  /* Summarize rounding-effect of excess digits. */
  for (i = 16; i < dig; i++, p++)
    x |= ((*p != '.' ? *p : *++p) != '0'), ex2 += 4;

  /* Format-specific handling. */
  switch (fmt) {
  case STRSCAN_INT:
    if (!(opt & STRSCAN_OPT_TONUM) && x < 0x80000000u+neg) {
      o->i = neg ? -(int32_t)x : (int32_t)x;
      return STRSCAN_INT;  /* Fast path for 32 bit integers. */
    }
    if (!(opt & STRSCAN_OPT_C)) { fmt = STRSCAN_NUM; break; }
    /* fallthrough */
  case STRSCAN_U32:
    if (dig > 8) return STRSCAN_ERROR;
    o->i = neg ? -(int32_t)x : (int32_t)x;
    return STRSCAN_U32;
  case STRSCAN_I64:
  case STRSCAN_U64:
    if (dig > 16) return STRSCAN_ERROR;
    o->u64 = neg ? (uint64_t)-(int64_t)x : x;
    return fmt;
  default:
    break;
  }

  /* Reduce range then convert to double. */
  if ((x & U64x(c0000000,0000000))) { x = (x >> 2) | (x & 3); ex2 += 2; }
  strscan_double(x, o, ex2, neg);
  return fmt;
}

/* Parse octal number. */
static StrScanFmt strscan_oct(const uint8_t *p, TValue *o,
                              StrScanFmt fmt, int32_t neg, uint32_t dig)
{
  uint64_t x = 0;

  /* Scan octal digits. */
  if (dig > 22 || (dig == 22 && *p > '1')) return STRSCAN_ERROR;
  while (dig-- > 0) {
    if (!(*p >= '0' && *p <= '7')) return STRSCAN_ERROR;
    x = (x << 3) + (*p++ & 7);
  }

  /* Format-specific handling. */
  switch (fmt) {
  case STRSCAN_INT:
    if (x >= 0x80000000u+neg) fmt = STRSCAN_U32;
    /* fallthrough */
  case STRSCAN_U32:
    if ((x >> 32)) return STRSCAN_ERROR;
```

```
    o->i = neg ? -(int32_t)x : (int32_t)x;
    break;
  default:
  case STRSCAN_I64:
  case STRSCAN_U64:
    o->u64 = neg ? (uint64_t)-(int64_t)x : x;
    break;
  }
  return fmt;
}

/* Parse decimal number. */
static StrScanFmt strscan_dec(const uint8_t *p, TValue *o,
                              StrScanFmt fmt, uint32_t opt,
                              int32_t ex10, int32_t neg, uint32_t dig)
{
  uint8_t xi[STRSCAN_DDIG], *xip = xi;

  if (dig) {
    uint32_t i = dig;
    if (i > STRSCAN_MAXDIG) {
      ex10 += (int32_t)(i - STRSCAN_MAXDIG);
      i = STRSCAN_MAXDIG;
    }
    /* Scan unaligned leading digit. */
    if (((ex10^i) & 1))
      *xip++ = ((*p != '.' ? *p : *++p) & 15), i--, p++;
    /* Scan aligned double-digits. */
    for ( ; i > 1; i -= 2) {
      uint32_t d = 10 * ((*p != '.' ? *p : *++p) & 15); p++;
      *xip++ = d + ((*p != '.' ? *p : *++p) & 15); p++;
    }
    /* Scan and realign trailing digit. */
    if (i) *xip++ = 10 * ((*p != '.' ? *p : *++p) & 15), ex10--, dig++, p++;

    /* Summarize rounding-effect of excess digits. */
    if (dig > STRSCAN_MAXDIG) {
      do {
        if ((*p != '.' ? *p : *++p) != '0') { xip[-1] |= 1; break; }
        p++;
      } while (--dig > STRSCAN_MAXDIG);
      dig = STRSCAN_MAXDIG;
    } else {  /* Simplify exponent. */
      while (ex10 > 0 && dig <= 18) *xip++ = 0, ex10 -= 2, dig += 2;
    }
  } else {  /* Only got zeros. */
    ex10 = 0;
    xi[0] = 0;
  }

  /* Fast path for numbers in integer format (but handles e.g. 1e6, too). */
  if (dig <= 20 && ex10 == 0) {
    uint8_t *xis;
    uint64_t x = xi[0];
    double n;
    for (xis = xi+1; xis < xip; xis++) x = x * 100 + *xis;
    if (!(dig == 20 && (xi[0] > 18 || (int64_t)x >= 0))) {  /* No overflow? */
      /* Format-specific handling. */
      switch (fmt) {
      case STRSCAN_INT:
        if (!(opt & STRSCAN_OPT_TONUM) && x < 0x80000000u+neg) {
          o->i = neg ? -(int32_t)x : (int32_t)x;
          return STRSCAN_INT;  /* Fast path for 32 bit integers. */
        }
        if (!(opt & STRSCAN_OPT_C)) { fmt = STRSCAN_NUM; goto plainnumber; }
        /* fallthrough */
      case STRSCAN_U32:
        if ((x >> 32) != 0) return STRSCAN_ERROR;
        o->i = neg ? -(int32_t)x : (int32_t)x;
        return STRSCAN_U32;
      case STRSCAN_I64:
      case STRSCAN_U64:
        o->u64 = neg ? (uint64_t)-(int64_t)x : x;
        return fmt;
      default:
      plainnumber:  /* Fast path for plain numbers < 2^63. */
```

```
        if ((int64_t)x < 0) break;
        n = (double)(int64_t)x;
        if (neg) n = -n;
        o->n = n;
        return fmt;
      }
    }
  }

  /* Slow non-integer path. */
  if (fmt == STRSCAN_INT) {
    if ((opt & STRSCAN_OPT_C)) return STRSCAN_ERROR;
    fmt = STRSCAN_NUM;
  } else if (fmt > STRSCAN_INT) {
    return STRSCAN_ERROR;
  }
  {
    uint32_t hi = 0, lo = (uint32_t)(xip-xi);
    int32_t ex2 = 0, idig = (int32_t)lo + (ex10 >> 1);

    lua_assert(lo > 0 && (ex10 & 1) == 0);

    /* Handle simple overflow/underflow. */
    if (idig > 310/2) { if (neg) setminfV(o); else setpinfV(o); return fmt; }
    else if (idig < -326/2) { o->n = neg ? -0.0 : 0.0; return fmt; }

    /* Scale up until we have at least 17 or 18 integer part digits. */
    while (idig < 9 && idig < DLEN(lo, hi)) {
      uint32_t i, cy = 0;
      ex2 -= 6;
      for (i = DPREV(lo); ; i = DPREV(i)) {
        uint32_t d = (xi[i] << 6) + cy;
        cy = (((d >> 2) * 5243) >> 17); d = d - cy * 100;  /* Div/mod 100. */
        xi[i] = (uint8_t)d;
        if (i == hi) break;
        if (d == 0 && i == DPREV(lo)) lo = i;
      }
      if (cy) {
        hi = DPREV(hi);
        if (xi[DPREV(lo)] == 0) lo = DPREV(lo);
        else if (hi == lo) { lo = DPREV(lo); xi[DPREV(lo)] |= xi[lo]; }
        xi[hi] = (uint8_t)cy; idig++;
      }
    }

    /* Scale down until no more than 17 or 18 integer part digits remain. */
    while (idig > 9) {
      uint32_t i = hi, cy = 0;
      ex2 += 6;
      do {
        cy += xi[i];
        xi[i] = (cy >> 6);
        cy = 100 * (cy & 0x3f);
        if (xi[i] == 0 && i == hi) hi = DNEXT(hi), idig--;
        i = DNEXT(i);
      } while (i != lo);
      while (cy) {
        if (hi == lo) { xi[DPREV(lo)] |= 1; break; }
        xi[lo] = (cy >> 6); lo = DNEXT(lo);
        cy = 100 * (cy & 0x3f);
      }
    }

    /* Collect integer part digits and convert to rescaled double. */
    {
      uint64_t x = xi[hi];
      uint32_t i;
      for (i = DNEXT(hi); --idig > 0 && i != lo; i = DNEXT(i))
        x = x * 100 + xi[i];
      if (i == lo) {
        while (--idig >= 0) x = x * 100;
      } else {  /* Gather round bit from remaining digits. */
        x <<= 1; ex2--;
        do {
          if (xi[i]) { x |= 1; break; }
          i = DNEXT(i);
```

```
        } while (i != lo);
      }
      strscan_double(x, o, ex2, neg);
    }
  }
  return fmt;
}

/* Scan string containing a number. Returns format. Returns value in o. */
StrScanFmt lj_strscan_scan(const uint8_t *p, TValue *o, uint32_t opt)
{
  int32_t neg = 0;

  /* Remove leading space, parse sign and non-numbers. */
  if (LJ_UNLIKELY(!lj_char_isdigit(*p))) {
    while (lj_char_isspace(*p)) p++;
    if (*p == '+' || *p == '-') neg = (*p++ == '-');
    if (LJ_UNLIKELY(*p >= 'A')) {  /* Parse "inf", "infinity" or "nan". */
      TValue tmp;
      setnanV(&tmp);
      if (casecmp(p[0],'i') && casecmp(p[1],'n') && casecmp(p[2],'f')) {
        if (neg) setminfV(&tmp); else setpinfV(&tmp);
        p += 3;
        if (casecmp(p[0],'i') && casecmp(p[1],'n') && casecmp(p[2],'i') &&
            casecmp(p[3],'t') && casecmp(p[4],'y')) p += 5;
      } else if (casecmp(p[0],'n') && casecmp(p[1],'a') && casecmp(p[2],'n')) {
        p += 3;
      }
      while (lj_char_isspace(*p)) p++;
      if (*p) return STRSCAN_ERROR;
      o->u64 = tmp.u64;
      return STRSCAN_NUM;
    }
  }

  /* Parse regular number. */
  {
    StrScanFmt fmt = STRSCAN_INT;
    int cmask = LJ_CHAR_DIGIT;
    int base = (opt & STRSCAN_OPT_C) && *p == '0' ? 0 : 10;
    const uint8_t *sp, *dp = NULL;
    uint32_t dig = 0, hasdig = 0, x = 0;
    int32_t ex = 0;

    /* Determine base and skip leading zeros. */
    if (LJ_UNLIKELY(*p <= '0')) {
      if (*p == '0' && casecmp(p[1], 'x'))
        base = 16, cmask = LJ_CHAR_XDIGIT, p += 2;
      for ( ; ; p++) {
        if (*p == '0') {
          hasdig = 1;
        } else if (*p == '.') {
          if (dp) return STRSCAN_ERROR;
          dp = p;
        } else {
          break;
        }
      }
    }

    /* Preliminary digit and decimal point scan. */
    for (sp = p; ; p++) {
      if (LJ_LIKELY(lj_char_isa(*p, cmask))) {
        x = x * 10 + (*p & 15);  /* For fast path below. */
        dig++;
      } else if (*p == '.') {
        if (dp) return STRSCAN_ERROR;
        dp = p;
      } else {
        break;
      }
    }
    if (!(hasdig | dig)) return STRSCAN_ERROR;

    /* Handle decimal point. */
    if (dp) {
```

```
    fmt = STRSCAN_NUM;
    if (dig) {
      ex = (int32_t)(dp-(p-1)); dp = p-1;
      while (ex < 0 && *dp-- == '0') ex++, dig--;  /* Skip trailing zeros. */
      if (base == 16) ex *= 4;
    }
  }

  /* Parse exponent. */
  if (casecmp(*p, (uint32_t)(base == 16 ? 'p' : 'e'))) {
    uint32_t xx;
    int negx = 0;
    fmt = STRSCAN_NUM; p++;
    if (*p == '+' || *p == '-') negx = (*p++ == '-');
    if (!lj_char_isdigit(*p)) return STRSCAN_ERROR;
    xx = (*p++ & 15);
    while (lj_char_isdigit(*p)) {
      if (xx < 65536) xx = xx * 10 + (*p & 15);
      p++;
    }
    ex += negx ? -(int32_t)xx : (int32_t)xx;
  }

  /* Parse suffix. */
  if (*p) {
    /* I (IMAG), U (U32), LL (I64), ULL/LLU (U64), L (long), UL/LU (ulong). */
    /* NYI: f (float). Not needed until cp_number() handles non-integers. */
    if (casecmp(*p, 'i')) {
      if (!(opt & STRSCAN_OPT_IMAG)) return STRSCAN_ERROR;
      p++; fmt = STRSCAN_IMAG;
    } else if (fmt == STRSCAN_INT) {
      if (casecmp(*p, 'u')) p++, fmt = STRSCAN_U32;
      if (casecmp(*p, 'l')) {
        p++;
        if (casecmp(*p, 'l')) p++, fmt += STRSCAN_I64 - STRSCAN_INT;
        else if (!(opt & STRSCAN_OPT_C)) return STRSCAN_ERROR;
        else if (sizeof(long) == 8) fmt += STRSCAN_I64 - STRSCAN_INT;
      }
      if (casecmp(*p, 'u') && (fmt == STRSCAN_INT || fmt == STRSCAN_I64))
        p++, fmt += STRSCAN_U32 - STRSCAN_INT;
      if ((fmt == STRSCAN_U32 && !(opt & STRSCAN_OPT_C)) ||
          (fmt >= STRSCAN_I64 && !(opt & STRSCAN_OPT_LL)))
        return STRSCAN_ERROR;
    }
    while (lj_char_isspace(*p)) p++;
    if (*p) return STRSCAN_ERROR;
  }

  /* Fast path for decimal 32 bit integers. */
  if (fmt == STRSCAN_INT && base == 10 &&
      (dig < 10 || (dig == 10 && *sp <= '2' && x < 0x80000000u+neg))) {
    int32_t y = neg ? -(int32_t)x : (int32_t)x;
    if ((opt & STRSCAN_OPT_TONUM)) {
      o->n = (double)y;
      return STRSCAN_NUM;
    } else {
      o->i = y;
      return STRSCAN_INT;
    }
  }

  /* Dispatch to base-specific parser. */
  if (base == 0 && !(fmt == STRSCAN_NUM || fmt == STRSCAN_IMAG))
    return strscan_oct(sp, o, fmt, neg, dig);
  if (base == 16)
    fmt = strscan_hex(sp, o, fmt, opt, ex, neg, dig);
  else
    fmt = strscan_dec(sp, o, fmt, opt, ex, neg, dig);

  /* Try to convert number to integer, if requested. */
  if (fmt == STRSCAN_NUM && (opt & STRSCAN_OPT_TOINT)) {
    double n = o->n;
    int32_t i = lj_num2int(n);
    if (n == (lua_Number)i) { o->i = i; return STRSCAN_INT; }
  }
  return fmt;
```

```
473   }
474 }
475
476 int LJ_FASTCALL lj_strscan_num(GCstr *str, TValue *o)
477 {
478   StrScanFmt fmt = lj_strscan_scan((const uint8_t *)strdata(str), o,
479                                    STRSCAN_OPT_TONUM);
480   lua_assert(fmt == STRSCAN_ERROR || fmt == STRSCAN_NUM);
481   return (fmt != STRSCAN_ERROR);
482 }
483
484 #if LJ_DUALNUM
485 int LJ_FASTCALL lj_strscan_number(GCstr *str, TValue *o)
486 {
487   StrScanFmt fmt = lj_strscan_scan((const uint8_t *)strdata(str), o,
488                                    STRSCAN_OPT_TOINT);
489   lua_assert(fmt == STRSCAN_ERROR || fmt == STRSCAN_NUM || fmt == STRSCAN_INT);
490   if (fmt == STRSCAN_INT) setitype(o, LJ_TISNUM);
491   return (fmt != STRSCAN_ERROR);
492 }
493 #endif
494
495 #undef DNEXT
496 #undef DPREV
497 #undef DLEN
498
```

One Level Up Top Level

src/lj_strscan.h - luajit-2.0-src

Data types defined

- StrScanFmt

Functions defined

- lj_strscan_numberobj

Macros defined

- STRSCAN_OPT_C
- STRSCAN_OPT_IMAG
- STRSCAN_OPT_LL
- STRSCAN_OPT_TOINT
- STRSCAN_OPT_TONUM
- _LJ_STRSCAN_H
- lj_strscan_number

Source code

```
/*
** String scanning.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_STRSCAN_H
#define _LJ_STRSCAN_H

#include "lj_obj.h"

/* Options for accepted/returned formats. */
#define STRSCAN_OPT_TOINT	0x01  /* Convert to int32_t, if possible. */
#define STRSCAN_OPT_TONUM	0x02  /* Always convert to double. */
#define STRSCAN_OPT_IMAG	0x04
#define STRSCAN_OPT_LL		0x08
#define STRSCAN_OPT_C		0x10

/* Returned format. */
typedef enum {
  STRSCAN_ERROR,
  STRSCAN_NUM, STRSCAN_IMAG,
  STRSCAN_INT, STRSCAN_U32, STRSCAN_I64, STRSCAN_U64,
} StrScanFmt;

LJ_FUNC StrScanFmt lj_strscan_scan(const uint8_t *p, TValue *o, uint32_t opt);
LJ_FUNC int LJ_FASTCALL lj_strscan_num(GCstr *str, TValue *o);
#if LJ_DUALNUM
LJ_FUNC int LJ_FASTCALL lj_strscan_number(GCstr *str, TValue *o);
#else
#define lj_strscan_number(s, o)		lj_strscan_num((s), (o))
#endif

/* Check for number or convert string to number/int in-place (!). */
static LJ_AINLINE int lj_strscan_numberobj(TValue *o)
{
  return tvisnumber(o) || (tvisstr(o) && lj_strscan_number(strV(o), o));
}
```

```
38
39 #endif
```

One Level Up Top Level

src/lj_cparse.c - luajit-2.0-src

Global variables defined

- ctoknames

Data types defined

- CPDecl
- CPDecl
- CPDeclIdx
- CPscl

Functions defined

- cp_add
- cp_check
- cp_cleanup
- cp_comment_c
- cp_comment_cpp
- cp_decl_abstract
- cp_decl_align
- cp_decl_array
- cp_decl_asm
- cp_decl_attributes
- cp_decl_constinit
- cp_decl_enum
- cp_decl_func
- cp_decl_gccattribute
- cp_decl_intern
- cp_decl_mode
- cp_decl_msvcattribute
- cp_decl_multi
- cp_decl_reset
- cp_decl_single
- cp_decl_sizeattr
- cp_decl_spec

- cp_decl_struct
- cp_declarator
- cp_err
- cp_err_badidx
- cp_err_token
- cp_errmsg
- cp_expr_comma
- cp_expr_infix
- cp_expr_kint
- cp_expr_ksize
- cp_expr_postfix
- cp_expr_prefix
- cp_expr_sizeof
- cp_expr_sub
- cp_expr_unary
- cp_field_align
- cp_get
- cp_get_bs
- cp_ident
- cp_init
- cp_iseol
- cp_istypedecl
- cp_newline
- cp_next
- cp_next_
- cp_number
- cp_opt
- cp_param
- cp_pragma
- cp_push
- cp_push_attributes
- cp_push_type
- cp_rawpeek

- cp_save
- cp_string
- cp_struct_layout
- cp_struct_name
- cp_tok2str
- cpcparser
- lj_cparse

Macros defined

- CPNS_DEFAULT
- CPNS_STRUCT
- CTOKSTR
- CTOKSTR
- H_
- H_

Source code

```
/*
** C declaration parser.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#include "lj_obj.h"

#if LJ_HASFFI

 #include "lj_gc.h"
 #include "lj_err.h"
 #include "lj_buf.h"
 #include "lj_ctype.h"
 #include "lj_cparse.h"
 #include "lj_frame.h"
 #include "lj_vm.h"
 #include "lj_char.h"
 #include "lj_strscan.h"
 #include "lj_strfmt.h"

 /*
 ** Important note: this is NOT a validating C parser! This is a minimal
 ** C declaration parser, solely for use by the LuaJIT FFI.
 **
 ** It ought to return correct results for properly formed C declarations,
 ** but it may accept some invalid declarations, too (and return nonsense).
 ** Also, it shows rather generic error messages to avoid unnecessary bloat.
 ** If in doubt, please check the input against your favorite C compiler.
 */

 /* -- C lexer ----------------------------------------------------------- */

 /* C lexer token names. */
 static const char *const ctoknames[] = {
 #define CTOKSTR(name, str)        str,
 CTOKDEF(CTOKSTR)
 #undef CTOKSTR
   NULL
 };
```

```
40
41  /* Forward declaration. */
42  LJ_NORET static void cp_err(CPState *cp, ErrMsg em);
43
44  static const char *cp_tok2str(CPState *cp, CPToken tok)
45  {
46    lua_assert(tok < CTOK_FIRSTDECL);
47    if (tok > CTOK_OFS)
48      return ctoknames[tok-CTOK_OFS-1];
49    else if (!lj_char_iscntrl(tok))
50      return lj_strfmt_pushf(cp->L, "%c", tok);
51    else
52      return lj_strfmt_pushf(cp->L, "char(%d)", tok);
53  }
54
55  /* End-of-line? */
56  static LJ_AINLINE int cp_iseol(CPChar c)
57  {
58    return (c == '\n' || c == '\r');
59  }
60
61  /* Peek next raw character. */
62  static LJ_AINLINE CPChar cp_rawpeek(CPState *cp)
63  {
64    return (CPChar)(uint8_t)(*cp->p);
65  }
66
67  static LJ_NOINLINE CPChar cp_get_bs(CPState *cp);
68
69  /* Get next character. */
70  static LJ_AINLINE CPChar cp_get(CPState *cp)
71  {
72    cp->c = (CPChar)(uint8_t)(*cp->p++);
73    if (LJ_LIKELY(cp->c != '\\')) return cp->c;
74    return cp_get_bs(cp);
75  }
76
77  /* Transparently skip backslash-escaped line breaks. */
78  static LJ_NOINLINE CPChar cp_get_bs(CPState *cp)
79  {
80    CPChar c2, c = cp_rawpeek(cp);
81    if (!cp_iseol(c)) return cp->c;
82    cp->p++;
83    c2 = cp_rawpeek(cp);
84    if (cp_iseol(c2) && c2 != c) cp->p++;
85    cp->linenumber++;
86    return cp_get(cp);
87  }
88
89  /* Save character in buffer. */
90  static LJ_AINLINE void cp_save(CPState *cp, CPChar c)
91  {
92    lj_buf_putb(&cp->sb, c);
93  }
94
95  /* Skip line break. Handles "\n", "\r", "\r\n" or "\n\r". */
96  static void cp_newline(CPState *cp)
97  {
98    CPChar c = cp_rawpeek(cp);
99    if (cp_iseol(c) && c != cp->c) cp->p++;
100     cp->linenumber++;
101   }
102
103   LJ_NORET static void cp_errmsg(CPState *cp, CPToken tok, ErrMsg em, ...)
104   {
105     const char *msg, *tokstr;
106     lua_State *L;
107     va_list argp;
108     if (tok == 0) {
109       tokstr = NULL;
110     } else if (tok == CTOK_IDENT || tok == CTOK_INTEGER || tok == CTOK_STRING ||
111                tok >= CTOK_FIRSTDECL) {
112       if (sbufP(&cp->sb) == sbufB(&cp->sb)) cp_save(cp, '$');
113       cp_save(cp, '\0');
114       tokstr = sbufB(&cp->sb);
115     } else {
```

```
    tokstr = cp_tok2str(cp, tok);
  }
  L = cp->L;
  va_start(argp, em);
  msg = lj_strfmt_pushvf(L, err2msg(em), argp);
  va_end(argp);
  if (tokstr)
    msg = lj_strfmt_pushf(L, err2msg(LJ_ERR_XNEAR), msg, tokstr);
  if (cp->linenumber > 1)
    msg = lj_strfmt_pushf(L, "%s at line %d", msg, cp->linenumber);
  lj_err_callermsg(L, msg);
}

LJ_NORET LJ_NOINLINE static void cp_err_token(CPState *cp, CPToken tok)
{
  cp_errmsg(cp, cp->tok, LJ_ERR_XTOKEN, cp_tok2str(cp, tok));
}

LJ_NORET LJ_NOINLINE static void cp_err_badidx(CPState *cp, CType *ct)
{
  GCstr *s = lj_ctype_repr(cp->cts->L, ctype_typeid(cp->cts, ct), NULL);
  cp_errmsg(cp, 0, LJ_ERR_FFI_BADIDX, strdata(s));
}

LJ_NORET LJ_NOINLINE static void cp_err(CPState *cp, ErrMsg em)
{
  cp_errmsg(cp, 0, em);
}

/* -- Main lexical scanner ---------------------------------------------- */

/* Parse number literal. Only handles int32_t/uint32_t right now. */
static CPToken cp_number(CPState *cp)
{
  StrScanFmt fmt;
  TValue o;
  do { cp_save(cp, cp->c); } while (lj_char_isident(cp_get(cp)));
  cp_save(cp, '\0');
  fmt = lj_strscan_scan((const uint8_t *)sbufB(&cp->sb), &o, STRSCAN_OPT_C);
  if (fmt == STRSCAN_INT) cp->val.id = CTID_INT32;
  else if (fmt == STRSCAN_U32) cp->val.id = CTID_UINT32;
  else if (!(cp->mode & CPARSE_MODE_SKIP))
    cp_errmsg(cp, CTOK_INTEGER, LJ_ERR_XNUMBER);
  cp->val.u32 = (uint32_t)o.i;
  return CTOK_INTEGER;
}

/* Parse identifier or keyword. */
static CPToken cp_ident(CPState *cp)
{
  do { cp_save(cp, cp->c); } while (lj_char_isident(cp_get(cp)));
  cp->str = lj_buf_str(cp->L, &cp->sb);
  cp->val.id = lj_ctype_getname(cp->cts, &cp->ct, cp->str, cp->tmask);
  if (ctype_type(cp->ct->info) == CT_KW)
    return ctype_cid(cp->ct->info);
  return CTOK_IDENT;
}

/* Parse parameter. */
static CPToken cp_param(CPState *cp)
{
  CPChar c = cp_get(cp);
  TValue *o = cp->param;
  if (lj_char_isident(c) || c == '$')  /* Reserve $xyz for future extensions. */
    cp_errmsg(cp, c, LJ_ERR_XSYNTAX);
  if (!o || o >= cp->L->top)
    cp_err(cp, LJ_ERR_FFI_NUMPARAM);
  cp->param = o+1;
  if (tvisstr(o)) {
    cp->str = strV(o);
    cp->val.id = 0;
    cp->ct = &cp->cts->tab[0];
    return CTOK_IDENT;
  } else if (tvisnumber(o)) {
    cp->val.i32 = numberVint(o);
    cp->val.id = CTID_INT32;
```

```
    return CTOK_INTEGER;
  } else {
    GCcdata *cd;
    if (!tviscdata(o))
      lj_err_argtype(cp->L, (int)(o-cp->L->base)+1, "type parameter");
    cd = cdataV(o);
    if (cd->ctypeid == CTID_CTYPEID)
      cp->val.id = *(CTypeID *)cdataptr(cd);
    else
      cp->val.id = cd->ctypeid;
    return '$';
  }
}

/* Parse string or character constant. */
static CPToken cp_string(CPState *cp)
{
  CPChar delim = cp->c;
  cp_get(cp);
  while (cp->c != delim) {
    CPChar c = cp->c;
    if (c == '\0') cp_errmsg(cp, CTOK_EOF, LJ_ERR_XSTR);
    if (c == '\\') {
      c = cp_get(cp);
      switch (c) {
      case '\0': cp_errmsg(cp, CTOK_EOF, LJ_ERR_XSTR); break;
      case 'a': c = '\a'; break;
      case 'b': c = '\b'; break;
      case 'f': c = '\f'; break;
      case 'n': c = '\n'; break;
      case 'r': c = '\r'; break;
      case 't': c = '\t'; break;
      case 'v': c = '\v'; break;
      case 'e': c = 27; break;
      case 'x':
        c = 0;
        while (lj_char_isxdigit(cp_get(cp)))
          c = (c<<4) + (lj_char_isdigit(cp->c) ? cp->c-'0' : (cp->c&15)+9);
        cp_save(cp, (c & 0xff));
        continue;
      default:
        if (lj_char_isdigit(c)) {
          c -= '0';
          if (lj_char_isdigit(cp_get(cp))) {
            c = c*8 + (cp->c - '0');
            if (lj_char_isdigit(cp_get(cp))) {
              c = c*8 + (cp->c - '0');
              cp_get(cp);
            }
          }
          cp_save(cp, (c & 0xff));
          continue;
        }
        break;
      }
    }
    cp_save(cp, c);
    cp_get(cp);
  }
  cp_get(cp);
  if (delim == '"') {
    cp->str = lj_buf_str(cp->L, &cp->sb);
    return CTOK_STRING;
  } else {
    if (sbuflen(&cp->sb) != 1) cp_err_token(cp, '\'');
    cp->val.i32 = (int32_t)(char)*sbufB(&cp->sb);
    cp->val.id = CTID_INT32;
    return CTOK_INTEGER;
  }
}

/* Skip C comment. */
static void cp_comment_c(CPState *cp)
{
  do {
    if (cp_get(cp) == '*') {
```

```
      do {
        if (cp_get(cp) == '/') { cp_get(cp); return; }
      } while (cp->c == '*');
    }
    if (cp_iseol(cp->c)) cp_newline(cp);
  } while (cp->c != '\0');
}

/* Skip C++ comment. */
static void cp_comment_cpp(CPState *cp)
{
  while (!cp_iseol(cp_get(cp)) && cp->c != '\0')
    ;
}

/* Lexical scanner for C. Only a minimal subset is implemented. */
static CPToken cp_next_(CPState *cp)
{
  lj_buf_reset(&cp->sb);
  for (;;) {
    if (lj_char_isident(cp->c))
      return lj_char_isdigit(cp->c) ? cp_number(cp) : cp_ident(cp);
    switch (cp->c) {
    case '\n': case '\r': cp_newline(cp);  /* fallthrough. */
    case ' ': case '\t': case '\v': case '\f': cp_get(cp); break;
    case '"': case '\'': return cp_string(cp);
    case '/':
      if (cp_get(cp) == '*') cp_comment_c(cp);
      else if (cp->c == '/') cp_comment_cpp(cp);
      else return '/';
      break;
    case '|':
      if (cp_get(cp) != '|') return '|'; cp_get(cp); return CTOK_OROR;
    case '&':
      if (cp_get(cp) != '&') return '&'; cp_get(cp); return CTOK_ANDAND;
    case '=':
      if (cp_get(cp) != '=') return '='; cp_get(cp); return CTOK_EQ;
    case '!':
      if (cp_get(cp) != '=') return '!'; cp_get(cp); return CTOK_NE;
    case '<':
      if (cp_get(cp) == '=') { cp_get(cp); return CTOK_LE; }
      else if (cp->c == '<') { cp_get(cp); return CTOK_SHL; }
      return '<';
    case '>':
      if (cp_get(cp) == '=') { cp_get(cp); return CTOK_GE; }
      else if (cp->c == '>') { cp_get(cp); return CTOK_SHR; }
      return '>';
    case '-':
      if (cp_get(cp) != '>') return '-'; cp_get(cp); return CTOK_DEREF;
    case '$':
      return cp_param(cp);
    case '\0': return CTOK_EOF;
    default: { CPToken c = cp->c; cp_get(cp); return c; }
    }
  }
}

static LJ_NOINLINE CPToken cp_next(CPState *cp)
{
  return (cp->tok = cp_next_(cp));
}

/* -- C parser ------------------------------------------------------------ */

/* Namespaces for resolving identifiers. */
#define CPNS_DEFAULT \
  ((1u<<CT_KW)|(1u<<CT_TYPEDEF)|(1u<<CT_FUNC)|(1u<<CT_EXTERN)|(1u<<CT_CONSTVAL))
#define CPNS_STRUCT        ((1u<<CT_KW)|(1u<<CT_STRUCT)|(1u<<CT_ENUM))

typedef CTypeID CPDeclIdx;        /* Index into declaration stack. */
typedef uint32_t CPscl;               /* Storage class flags. */

/* Type declaration context. */
typedef struct CPDecl {
  CPDeclIdx top;        /* Top of declaration stack. */
  CPDeclIdx pos;        /* Insertion position in declaration chain. */
```

```
  CPDeclIdx specpos;        /* Saved position for declaration specifier. */
  uint32_t mode;        /* Declarator mode. */
  CPState *cp;              /* C parser state. */
  GCstr *name;              /* Name of declared identifier (if direct). */
  GCstr *redir;              /* Redirected symbol name. */
  CTypeID nameid;        /* Existing typedef for declared identifier. */
  CTInfo attr;              /* Attributes. */
  CTInfo fattr;              /* Function attributes. */
  CTInfo specattr;        /* Saved attributes. */
  CTInfo specfattr;        /* Saved function attributes. */
  CTSize bits;              /* Field size in bits (if any). */
  CType stack[CPARSE_MAX_DECLSTACK];  /* Type declaration stack. */
} CPDecl;

/* Forward declarations. */
static CPscl cp_decl_spec(CPState *cp, CPDecl *decl, CPscl scl);
static void cp_declarator(CPState *cp, CPDecl *decl);
static CTypeID cp_decl_abstract(CPState *cp);

/* Initialize C parser state. Caller must set up: L, p, srcname, mode. */
static void cp_init(CPState *cp)
{
  cp->linenumber = 1;
  cp->depth = 0;
  cp->curpack = 0;
  cp->packstack[0] = 255;
  lj_buf_init(cp->L, &cp->sb);
  lua_assert(cp->p != NULL);
  cp_get(cp);  /* Read-ahead first char. */
  cp->tok = 0;
  cp->tmask = CPNS_DEFAULT;
  cp_next(cp);  /* Read-ahead first token. */
}

/* Cleanup C parser state. */
static void cp_cleanup(CPState *cp)
{
  global_State *g = G(cp->L);
  lj_buf_free(g, &cp->sb);
}

/* Check and consume optional token. */
static int cp_opt(CPState *cp, CPToken tok)
{
  if (cp->tok == tok) { cp_next(cp); return 1; }
  return 0;
}

/* Check and consume token. */
static void cp_check(CPState *cp, CPToken tok)
{
  if (cp->tok != tok) cp_err_token(cp, tok);
  cp_next(cp);
}

/* Check if the next token may start a type declaration. */
static int cp_istypedecl(CPState *cp)
{
  if (cp->tok >= CTOK_FIRSTDECL && cp->tok <= CTOK_LASTDECL) return 1;
  if (cp->tok == CTOK_IDENT && ctype_istypedef(cp->ct->info)) return 1;
  if (cp->tok == '$') return 1;
  return 0;
}

/* -- Constant expression evaluator ---------------------------------------- */

/* Forward declarations. */
static void cp_expr_unary(CPState *cp, CPValue *k);
static void cp_expr_sub(CPState *cp, CPValue *k, int pri);

/* Please note that type handling is very weak here. Most ops simply
** assume integer operands. Accessors are only needed to compute types and
** return synthetic values. The only purpose of the expression evaluator
** is to compute the values of constant expressions one would typically
** find in C header files. And again: this is NOT a validating C parser!
*/
```

```
420
421 /* Parse comma separated expression and return last result. */
422 static void cp_expr_comma(CPState *cp, CPValue *k)
423 {
424   do { cp_expr_sub(cp, k, 0); } while (cp_opt(cp, ','));
425 }
426
427 /* Parse sizeof/alignof operator. */
428 static void cp_expr_sizeof(CPState *cp, CPValue *k, int wantsz)
429 {
430   CTSize sz;
431   CTInfo info;
432   if (cp_opt(cp, '(')) {
433     if (cp_istypedecl(cp))
434       k->id = cp_decl_abstract(cp);
435     else
436       cp_expr_comma(cp, k);
437     cp_check(cp, ')');
438   } else {
439     cp_expr_unary(cp, k);
440   }
441   info = lj_ctype_info(cp->cts, k->id, &sz);
442   if (wantsz) {
443     if (sz != CTSIZE_INVALID)
444       k->u32 = sz;
445     else if (k->id != CTID_A_CCHAR)  /* Special case for sizeof("string"). */
446       cp_err(cp, LJ_ERR_FFI_INVSIZE);
447   } else {
448     k->u32 = 1u << ctype_align(info);
449   }
450   k->id = CTID_UINT32;  /* Really size_t. */
451 }
452
453 /* Parse prefix operators. */
454 static void cp_expr_prefix(CPState *cp, CPValue *k)
455 {
456   if (cp->tok == CTOK_INTEGER) {
457     *k = cp->val; cp_next(cp);
458   } else if (cp_opt(cp, '+')) {
459     cp_expr_unary(cp, k);  /* Nothing to do (well, integer promotion). */
460   } else if (cp_opt(cp, '-')) {
461     cp_expr_unary(cp, k); k->i32 = -k->i32;
462   } else if (cp_opt(cp, '~')) {
463     cp_expr_unary(cp, k); k->i32 = ~k->i32;
464   } else if (cp_opt(cp, '!')) {
465     cp_expr_unary(cp, k); k->i32 = !k->i32; k->id = CTID_INT32;
466   } else if (cp_opt(cp, '(')) {
467     if (cp_istypedecl(cp)) {  /* Cast operator. */
468       CTypeID id = cp_decl_abstract(cp);
469       cp_check(cp, ')');
470       cp_expr_unary(cp, k);
471       k->id = id;  /* No conversion performed. */
472     } else {  /* Sub-expression. */
473       cp_expr_comma(cp, k);
474       cp_check(cp, ')');
475     }
476   } else if (cp_opt(cp, '*')) {  /* Indirection. */
477     CType *ct;
478     cp_expr_unary(cp, k);
479     ct = lj_ctype_rawref(cp->cts, k->id);
480     if (!ctype_ispointer(ct->info))
481       cp_err_badidx(cp, ct);
482     k->u32 = 0; k->id = ctype_cid(ct->info);
483   } else if (cp_opt(cp, '&')) {  /* Address operator. */
484     cp_expr_unary(cp, k);
485     k->id = lj_ctype_intern(cp->cts, CTINFO(CT_PTR, CTALIGN_PTR+k->id),
486                             CTSIZE_PTR);
487   } else if (cp_opt(cp, CTOK_SIZEOF)) {
488     cp_expr_sizeof(cp, k, 1);
489   } else if (cp_opt(cp, CTOK_ALIGNOF)) {
490     cp_expr_sizeof(cp, k, 0);
491   } else if (cp->tok == CTOK_IDENT) {
492     if (ctype_type(cp->ct->info) == CT_CONSTVAL) {
493       k->u32 = cp->ct->size; k->id = ctype_cid(cp->ct->info);
494     } else if (ctype_type(cp->ct->info) == CT_EXTERN) {
495       k->u32 = cp->val.id; k->id = ctype_cid(cp->ct->info);
```

```
    } else if (ctype_type(cp->ct->info) == CT_FUNC) {
      k->u32 = cp->val.id; k->id = cp->val.id;
    } else {
      goto err_expr;
    }
    cp_next(cp);
  } else if (cp->tok == CTOK_STRING) {
    CTSize sz = cp->str->len;
    while (cp_next(cp) == CTOK_STRING)
      sz += cp->str->len;
    k->u32 = sz + 1;
    k->id = CTID_A_CCHAR;
  } else {
  err_expr:
    cp_errmsg(cp, cp->tok, LJ_ERR_XSYMBOL);
  }
}

/* Parse postfix operators. */
static void cp_expr_postfix(CPState *cp, CPValue *k)
{
  for (;;) {
    CType *ct;
    if (cp_opt(cp, '[')) {  /* Array/pointer index. */
      CPValue k2;
      cp_expr_comma(cp, &k2);
      ct = lj_ctype_rawref(cp->cts, k->id);
      if (!ctype_ispointer(ct->info)) {
        ct = lj_ctype_rawref(cp->cts, k2.id);
        if (!ctype_ispointer(ct->info))
          cp_err_badidx(cp, ct);
      }
      cp_check(cp, ']');
      k->u32 = 0;
    } else if (cp->tok == '.' || cp->tok == CTOK_DEREF) {  /* Struct deref. */
      CTSize ofs;
      CType *fct;
      ct = lj_ctype_rawref(cp->cts, k->id);
      if (cp->tok == CTOK_DEREF) {
        if (!ctype_ispointer(ct->info))
          cp_err_badidx(cp, ct);
        ct = lj_ctype_rawref(cp->cts, ctype_cid(ct->info));
      }
      cp_next(cp);
      if (cp->tok != CTOK_IDENT) cp_err_token(cp, CTOK_IDENT);
      if (!ctype_isstruct(ct->info) || ct->size == CTSIZE_INVALID ||
          !(fct = lj_ctype_getfield(cp->cts, ct, cp->str, &ofs)) ||
          ctype_isbitfield(fct->info)) {
        GCstr *s = lj_ctype_repr(cp->cts->L, ctype_typeid(cp->cts, ct), NULL);
        cp_errmsg(cp, 0, LJ_ERR_FFI_BADMEMBER, strdata(s), strdata(cp->str));
      }
      ct = fct;
      k->u32 = ctype_isconstval(ct->info) ? ct->size : 0;
      cp_next(cp);
    } else {
      return;
    }
    k->id = ctype_cid(ct->info);
  }
}

/* Parse infix operators. */
static void cp_expr_infix(CPState *cp, CPValue *k, int pri)
{
  CPValue k2;
  k2.u32 = 0; k2.id = 0;  /* Silence the compiler. */
  for (;;) {
    switch (pri) {
    case 0:
      if (cp_opt(cp, '?')) {
        CPValue k3;
        cp_expr_comma(cp, &k2);  /* Right-associative. */
        cp_check(cp, ':');
        cp_expr_sub(cp, &k3, 0);
        k->u32 = k->u32 ? k2.u32 : k3.u32;
        k->id = k2.id > k3.id ? k2.id : k3.id;
```

```
572         continue;
573       }
574     case 1:
575       if (cp_opt(cp, CTOK_OROR)) {
576         cp_expr_sub(cp, &k2, 2); k->i32 = k->u32 || k2.u32; k->id = CTID_INT32;
577         continue;
578       }
579     case 2:
580       if (cp_opt(cp, CTOK_ANDAND)) {
581         cp_expr_sub(cp, &k2, 3); k->i32 = k->u32 && k2.u32; k->id = CTID_INT32;
582         continue;
583       }
584     case 3:
585       if (cp_opt(cp, '|')) {
586         cp_expr_sub(cp, &k2, 4); k->u32 = k->u32 | k2.u32; goto arith_result;
587       }
588     case 4:
589       if (cp_opt(cp, '^')) {
590         cp_expr_sub(cp, &k2, 5); k->u32 = k->u32 ^ k2.u32; goto arith_result;
591       }
592     case 5:
593       if (cp_opt(cp, '&')) {
594         cp_expr_sub(cp, &k2, 6); k->u32 = k->u32 & k2.u32; goto arith_result;
595       }
596     case 6:
597       if (cp_opt(cp, CTOK_EQ)) {
598         cp_expr_sub(cp, &k2, 7); k->i32 = k->u32 == k2.u32; k->id = CTID_INT32;
599         continue;
600       } else if (cp_opt(cp, CTOK_NE)) {
601         cp_expr_sub(cp, &k2, 7); k->i32 = k->u32 != k2.u32; k->id = CTID_INT32;
602         continue;
603       }
604     case 7:
605       if (cp_opt(cp, '<')) {
606         cp_expr_sub(cp, &k2, 8);
607         if (k->id == CTID_INT32 && k2.id == CTID_INT32)
608           k->i32 = k->i32 < k2.i32;
609         else
610           k->i32 = k->u32 < k2.u32;
611         k->id = CTID_INT32;
612         continue;
613       } else if (cp_opt(cp, '>')) {
614         cp_expr_sub(cp, &k2, 8);
615         if (k->id == CTID_INT32 && k2.id == CTID_INT32)
616           k->i32 = k->i32 > k2.i32;
617         else
618           k->i32 = k->u32 > k2.u32;
619         k->id = CTID_INT32;
620         continue;
621       } else if (cp_opt(cp, CTOK_LE)) {
622         cp_expr_sub(cp, &k2, 8);
623         if (k->id == CTID_INT32 && k2.id == CTID_INT32)
624           k->i32 = k->i32 <= k2.i32;
625         else
626           k->i32 = k->u32 <= k2.u32;
627         k->id = CTID_INT32;
628         continue;
629       } else if (cp_opt(cp, CTOK_GE)) {
630         cp_expr_sub(cp, &k2, 8);
631         if (k->id == CTID_INT32 && k2.id == CTID_INT32)
632           k->i32 = k->i32 >= k2.i32;
633         else
634           k->i32 = k->u32 >= k2.u32;
635         k->id = CTID_INT32;
636         continue;
637       }
638     case 8:
639       if (cp_opt(cp, CTOK_SHL)) {
640         cp_expr_sub(cp, &k2, 9); k->u32 = k->u32 << k2.u32;
641         continue;
642       } else if (cp_opt(cp, CTOK_SHR)) {
643         cp_expr_sub(cp, &k2, 9);
644         if (k->id == CTID_INT32)
645           k->i32 = k->i32 >> k2.i32;
646         else
647           k->u32 = k->u32 >> k2.u32;
```

```
        continue;
      }
    case 9:
      if (cp_opt(cp, '+')) {
        cp_expr_sub(cp, &k2, 10); k->u32 = k->u32 + k2.u32;
      arith_result:
        if (k2.id > k->id) k->id = k2.id;  /* Trivial promotion to unsigned. */
        continue;
      } else if (cp_opt(cp, '-')) {
        cp_expr_sub(cp, &k2, 10); k->u32 = k->u32 - k2.u32; goto arith_result;
      }
    case 10:
      if (cp_opt(cp, '*')) {
        cp_expr_unary(cp, &k2); k->u32 = k->u32 * k2.u32; goto arith_result;
      } else if (cp_opt(cp, '/')) {
        cp_expr_unary(cp, &k2);
        if (k2.id > k->id) k->id = k2.id;  /* Trivial promotion to unsigned. */
        if (k2.u32 == 0 ||
            (k->id == CTID_INT32 && k->u32 == 0x80000000u && k2.i32 == -1))
          cp_err(cp, LJ_ERR_BADVAL);
        if (k->id == CTID_INT32)
          k->i32 = k->i32 / k2.i32;
        else
          k->u32 = k->u32 / k2.u32;
        continue;
      } else if (cp_opt(cp, '%')) {
        cp_expr_unary(cp, &k2);
        if (k2.id > k->id) k->id = k2.id;  /* Trivial promotion to unsigned. */
        if (k2.u32 == 0 ||
            (k->id == CTID_INT32 && k->u32 == 0x80000000u && k2.i32 == -1))
          cp_err(cp, LJ_ERR_BADVAL);
        if (k->id == CTID_INT32)
          k->i32 = k->i32 % k2.i32;
        else
          k->u32 = k->u32 % k2.u32;
        continue;
      }
    default:
      return;
    }
  }
}

/* Parse and evaluate unary expression. */
static void cp_expr_unary(CPState *cp, CPValue *k)
{
  if (++cp->depth > CPARSE_MAX_DECLDEPTH) cp_err(cp, LJ_ERR_XLEVELS);
  cp_expr_prefix(cp, k);
  cp_expr_postfix(cp, k);
  cp->depth--;
}

/* Parse and evaluate sub-expression. */
static void cp_expr_sub(CPState *cp, CPValue *k, int pri)
{
  cp_expr_unary(cp, k);
  cp_expr_infix(cp, k, pri);
}

/* Parse constant integer expression. */
static void cp_expr_kint(CPState *cp, CPValue *k)
{
  CType *ct;
  cp_expr_sub(cp, k, 0);
  ct = ctype_raw(cp->cts, k->id);
  if (!ctype_isinteger(ct->info)) cp_err(cp, LJ_ERR_BADVAL);
}

/* Parse (non-negative) size expression. */
static CTSize cp_expr_ksize(CPState *cp)
{
  CPValue k;
  cp_expr_kint(cp, &k);
  if (k.u32 >= 0x80000000u) cp_err(cp, LJ_ERR_FFI_INVSIZE);
  return k.u32;
}
```

```
724
725 /* -- Type declaration stack management ----------------------------------- */
726
727 /* Add declaration element behind the insertion position. */
728 static CPDeclIdx cp_add(CPDecl *decl, CTInfo info, CTSize size)
729 {
730   CPDeclIdx top = decl->top;
731   if (top >= CPARSE_MAX_DECLSTACK) cp_err(decl->cp, LJ_ERR_XLEVELS);
732   decl->stack[top].info = info;
733   decl->stack[top].size = size;
734   decl->stack[top].sib = 0;
735   setgcrefnull(decl->stack[top].name);
736   decl->stack[top].next = decl->stack[decl->pos].next;
737   decl->stack[decl->pos].next = (CTypeID1)top;
738   decl->top = top+1;
739   return top;
740 }
741
742 /* Push declaration element before the insertion position. */
743 static CPDeclIdx cp_push(CPDecl *decl, CTInfo info, CTSize size)
744 {
745   return (decl->pos = cp_add(decl, info, size));
746 }
747
748 /* Push or merge attributes. */
749 static void cp_push_attributes(CPDecl *decl)
750 {
751   CType *ct = &decl->stack[decl->pos];
752   if (ctype_isfunc(ct->info)) {  /* Ok to modify in-place. */
753 #if LJ_TARGET_X86
754     if ((decl->fattr & CTFP_CCONV))
755       ct->info = (ct->info & (CTMASK_NUM|CTF_VARARG|CTMASK_CID)) +
756                  (decl->fattr & ~CTMASK_CID);
757 #endif
758   } else {
759     if ((decl->attr & CTFP_ALIGNED) && !(decl->mode & CPARSE_MODE_FIELD))
760       cp_push(decl, CTINFO(CT_ATTRIB, CTATTRIB(CTA_ALIGN)),
761               ctype_align(decl->attr));
762   }
763 }
764
765 /* Push unrolled type to declaration stack and merge qualifiers. */
766 static void cp_push_type(CPDecl *decl, CTypeID id)
767 {
768   CType *ct = ctype_get(decl->cp->cts, id);
769   CTInfo info = ct->info;
770   CTSize size = ct->size;
771   switch (ctype_type(info)) {
772   case CT_STRUCT: case CT_ENUM:
773     cp_push(decl, CTINFO(CT_TYPEDEF, id), 0);  /* Don't copy unique types. */
774     if ((decl->attr & CTF_QUAL)) {  /* Push unmerged qualifiers. */
775       cp_push(decl, CTINFO(CT_ATTRIB, CTATTRIB(CTA_QUAL)),
776               (decl->attr & CTF_QUAL));
777       decl->attr &= ~CTF_QUAL;
778     }
779     break;
780   case CT_ATTRIB:
781     if (ctype_isxattrib(info, CTA_QUAL))
782       decl->attr &= ~size;  /* Remove redundant qualifiers. */
783     cp_push_type(decl, ctype_cid(info));  /* Unroll. */
784     cp_push(decl, info & ~CTMASK_CID, size);  /* Copy type. */
785     break;
786   case CT_ARRAY:
787     cp_push_type(decl, ctype_cid(info));  /* Unroll. */
788     cp_push(decl, info & ~CTMASK_CID, size);  /* Copy type. */
789     decl->stack[decl->pos].sib = 1;  /* Mark as already checked and sized. */
790     /* Note: this is not copied to the ct->sib in the C type table. */
791     break;
792   case CT_FUNC:
793     /* Copy type, link parameters (shared). */
794     decl->stack[cp_push(decl, info, size)].sib = ct->sib;
795     break;
796   default:
797     /* Copy type, merge common qualifiers. */
798     cp_push(decl, info|(decl->attr & CTF_QUAL), size);
799     decl->attr &= ~CTF_QUAL;
```

```
    break;
  }
}

/* Consume the declaration element chain and intern the C type. */
static CTypeID cp_decl_intern(CPState *cp, CPDecl *decl)
{
  CTypeID id = 0;
  CPDeclIdx idx = 0;
  CTSize csize = CTSIZE_INVALID;
  CTSize cinfo = 0;
  do {
    CType *ct = &decl->stack[idx];
    CTInfo info = ct->info;
    CTInfo size = ct->size;
    /* The cid is already part of info for copies of pointers/functions. */
    idx = ct->next;
    if (ctype_istypedef(info)) {
      lua_assert(id == 0);
      id = ctype_cid(info);
      /* Always refetch info/size, since struct/enum may have been completed. */
      cinfo = ctype_get(cp->cts, id)->info;
      csize = ctype_get(cp->cts, id)->size;
      lua_assert(ctype_isstruct(cinfo) || ctype_isenum(cinfo));
    } else if (ctype_isfunc(info)) {  /* Intern function. */
      CType *fct;
      CTypeID fid;
      CTypeID sib;
      if (id) {
        CType *refct = ctype_raw(cp->cts, id);
        /* Reject function or refarray return types. */
        if (ctype_isfunc(refct->info) || ctype_isrefarray(refct->info))
          cp_err(cp, LJ_ERR_FFI_INVTYPE);
      }
      /* No intervening attributes allowed, skip forward. */
      while (idx) {
        CType *ctn = &decl->stack[idx];
        if (!ctype_isattrib(ctn->info)) break;
        idx = ctn->next;  /* Skip attribute. */
      }
      sib = ct->sib;  /* Next line may reallocate the C type table. */
      fid = lj_ctype_new(cp->cts, &fct);
      csize = CTSIZE_INVALID;
      fct->info = cinfo = info + id;
      fct->size = size;
      fct->sib = sib;
      id = fid;
    } else if (ctype_isattrib(info)) {
      if (ctype_isxattrib(info, CTA_QUAL))
        cinfo |= size;
      else if (ctype_isxattrib(info, CTA_ALIGN))
        CTF_INSERT(cinfo, ALIGN, size);
      id = lj_ctype_intern(cp->cts, info+id, size);
      /* Inherit csize/cinfo from original type. */
    } else {
      if (ctype_isnum(info)) {  /* Handle mode/vector-size attributes. */
        lua_assert(id == 0);
        if (!(info & CTF_BOOL)) {
          CTSize msize = ctype_msizeP(decl->attr);
          CTSize vsize = ctype_vsizeP(decl->attr);
          if (msize && (!(info & CTF_FP) || (msize == 4 || msize == 8))) {
            CTSize malign = lj_fls(msize);
            if (malign > 4) malign = 4;  /* Limit alignment. */
            CTF_INSERT(info, ALIGN, malign);
            size = msize;  /* Override size via mode. */
          }
          if (vsize) {  /* Vector size set? */
            CTSize esize = lj_fls(size);
            if (vsize >= esize) {
              /* Intern the element type first. */
              id = lj_ctype_intern(cp->cts, info, size);
              /* Then create a vector (array) with vsize alignment. */
              size = (1u << vsize);
              if (vsize > 4) vsize = 4;  /* Limit alignment. */
              if (ctype_align(info) > vsize) vsize = ctype_align(info);
              info = CTINFO(CT_ARRAY, (info & CTF_QUAL) + CTF_VECTOR +
```

```
                                    CTALIGN(vsize));
            }
          }
        }
      } else if (ctype_isptr(info)) {
        /* Reject pointer/ref to ref. */
        if (id && ctype_isref(ctype_raw(cp->cts, id)->info))
          cp_err(cp, LJ_ERR_FFI_INVTYPE);
        if (ctype_isref(info)) {
          info &= ~CTF_VOLATILE;  /* Refs are always const, never volatile. */
          /* No intervening attributes allowed, skip forward. */
          while (idx) {
            CType *ctn = &decl->stack[idx];
            if (!ctype_isattrib(ctn->info)) break;
            idx = ctn->next;  /* Skip attribute. */
          }
        }
      } else if (ctype_isarray(info)) {  /* Check for valid array size etc. */
        if (ct->sib == 0) {  /* Only check/size arrays not copied by unroll. */
          if (ctype_isref(cinfo))  /* Reject arrays of refs. */
            cp_err(cp, LJ_ERR_FFI_INVTYPE);
          /* Reject VLS or unknown-sized types. */
          if (ctype_isvltype(cinfo) || csize == CTSIZE_INVALID)
            cp_err(cp, LJ_ERR_FFI_INVSIZE);
          /* a[] and a[?] keep their invalid size. */
          if (size != CTSIZE_INVALID) {
            uint64_t xsz = (uint64_t)size * csize;
            if (xsz >= 0x80000000u) cp_err(cp, LJ_ERR_FFI_INVSIZE);
            size = (CTSize)xsz;
          }
        }
        if ((cinfo & CTF_ALIGN) > (info & CTF_ALIGN))  /* Find max. align. */
          info = (info & ~CTF_ALIGN) | (cinfo & CTF_ALIGN);
        info |= (cinfo & CTF_QUAL);  /* Inherit qual. */
      } else {
        lua_assert(ctype_isvoid(info));
      }
      csize = size;
      cinfo = info+id;
      id = lj_ctype_intern(cp->cts, info+id, size);
    }
  } while (idx);
  return id;
}

/* -- C declaration parser ------------------------------------------------ */

#define H_(le, be)        LJ_ENDIAN_SELECT(0x##le, 0x##be)

/* Reset declaration state to declaration specifier. */
static void cp_decl_reset(CPDecl *decl)
{
  decl->pos = decl->specpos;
  decl->top = decl->specpos+1;
  decl->stack[decl->specpos].next = 0;
  decl->attr = decl->specattr;
  decl->fattr = decl->specfattr;
  decl->name = NULL;
  decl->redir = NULL;
}

/* Parse constant initializer. */
/* NYI: FP constants and strings as initializers. */
static CTypeID cp_decl_constinit(CPState *cp, CType **ctp, CTypeID ctypeid)
{
  CType *ctt = ctype_get(cp->cts, ctypeid);
  CTInfo info;
  CTSize size;
  CPValue k;
  CTypeID constid;
  while (ctype_isattrib(ctt->info)) {  /* Skip attributes. */
    ctypeid = ctype_cid(ctt->info);  /* Update ID, too. */
    ctt = ctype_get(cp->cts, ctypeid);
  }
  info = ctt->info;
  size = ctt->size;
```

```
  if (!ctype_isinteger(info) || !(info & CTF_CONST) || size > 4)
    cp_err(cp, LJ_ERR_FFI_INVTYPE);
  cp_check(cp, '=');
  cp_expr_sub(cp, &k, 0);
  constid = lj_ctype_new(cp->cts, ctp);
  (*ctp)->info = CTINFO(CT_CONSTVAL, CTF_CONST|ctypeid);
  k.u32 <<= 8*(4-size);
  if ((info & CTF_UNSIGNED))
    k.u32 >>= 8*(4-size);
  else
    k.u32 = (uint32_t)((int32_t)k.u32 >> 8*(4-size));
  (*ctp)->size = k.u32;
  return constid;
}

/* Parse size in parentheses as part of attribute. */
static CTSize cp_decl_sizeattr(CPState *cp)
{
  CTSize sz;
  uint32_t oldtmask = cp->tmask;
  cp->tmask = CPNS_DEFAULT;  /* Required for expression evaluator. */
  cp_check(cp, '(');
  sz = cp_expr_ksize(cp);
  cp->tmask = oldtmask;
  cp_check(cp, ')');
  return sz;
}

/* Parse alignment attribute. */
static void cp_decl_align(CPState *cp, CPDecl *decl)
{
  CTSize al = 4;  /* Unspecified alignment is 16 bytes. */
  if (cp->tok == '(') {
    al = cp_decl_sizeattr(cp);
    al = al ? lj_fls(al) : 0;
  }
  CTF_INSERT(decl->attr, ALIGN, al);
  decl->attr |= CTFP_ALIGNED;
}

/* Parse GCC asm("name") redirect. */
static void cp_decl_asm(CPState *cp, CPDecl *decl)
{
  UNUSED(decl);
  cp_next(cp);
  cp_check(cp, '(');
  if (cp->tok == CTOK_STRING) {
    GCstr *str = cp->str;
    while (cp_next(cp) == CTOK_STRING) {
      lj_strfmt_pushf(cp->L, "%s%s", strdata(str), strdata(cp->str));
      cp->L->top--;
      str = strV(cp->L->top);
    }
    decl->redir = str;
  }
  cp_check(cp, ')');
}

/* Parse GCC __attribute__((mode(...))). */
static void cp_decl_mode(CPState *cp, CPDecl *decl)
{
  cp_check(cp, '(');
  if (cp->tok == CTOK_IDENT) {
    const char *s = strdata(cp->str);
    CTSize sz = 0, vlen = 0;
    if (s[0] == '_' && s[1] == '_') s += 2;
    if (*s == 'V') {
      s++;
      vlen = *s++ - '0';
      if (*s >= '0' && *s <= '9')
        vlen = vlen*10 + (*s++ - '0');
    }
    switch (*s++) {
    case 'Q': sz = 1; break;
    case 'H': sz = 2; break;
    case 'S': sz = 4; break;
```

```
    case 'D': sz = 8; break;
    case 'T': sz = 16; break;
    case 'O': sz = 32; break;
    default: goto bad_size;
    }
    if (*s == 'I' || *s == 'F') {
      CTF_INSERT(decl->attr, MSIZEP, sz);
      if (vlen) CTF_INSERT(decl->attr, VSIZEP, lj_fls(vlen*sz));
    }
  bad_size:
    cp_next(cp);
  }
  cp_check(cp, ')');
}

/* Parse GCC __attribute__((...)). */
static void cp_decl_gccattribute(CPState *cp, CPDecl *decl)
{
  cp_next(cp);
  cp_check(cp, '(');
  cp_check(cp, '(');
  while (cp->tok != ')') {
    if (cp->tok == CTOK_IDENT) {
      GCstr *attrstr = cp->str;
      cp_next(cp);
      switch (attrstr->hash) {
      case H_(64a9208e,8ce14319): case H_(8e6331b2,95a282af):  /* aligned */
        cp_decl_align(cp, decl);
        break;
      case H_(42eb47de,f0ede26c): case H_(29f48a09,cf383e0c):  /* packed */
        decl->attr |= CTFP_PACKED;
        break;
      case H_(0a84eef6,8dfab04c): case H_(995cf92c,d5696591):  /* mode */
        cp_decl_mode(cp, decl);
        break;
      case H_(0ab31997,2d5213fa): case H_(bf875611,200e9990):  /* vector_size */
        {
          CTSize vsize = cp_decl_sizeattr(cp);
          if (vsize) CTF_INSERT(decl->attr, VSIZEP, lj_fls(vsize));
        }
        break;
#if LJ_TARGET_X86
      case H_(5ad22db8,c689b848): case H_(439150fa,65ea78cb):  /* regparm */
        CTF_INSERT(decl->fattr, REGPARM, cp_decl_sizeattr(cp));
        decl->fattr |= CTFP_CCONV;
        break;
      case H_(18fc0b98,7ff4c074): case H_(4e62abed,0a747424):  /* cdecl */
        CTF_INSERT(decl->fattr, CCONV, CTCC_CDECL);
        decl->fattr |= CTFP_CCONV;
        break;
      case H_(72b2e41b,494c5a44): case H_(f2356d59,f25fc9bd):  /* thiscall */
        CTF_INSERT(decl->fattr, CCONV, CTCC_THISCALL);
        decl->fattr |= CTFP_CCONV;
        break;
      case H_(0d0ffc42,ab746f88): case H_(21c54ba1,7f0ca7e3):  /* fastcall */
        CTF_INSERT(decl->fattr, CCONV, CTCC_FASTCALL);
        decl->fattr |= CTFP_CCONV;
        break;
      case H_(ef76b040,9412e06a): case H_(de56697b,c750e6e1):  /* stdcall */
        CTF_INSERT(decl->fattr, CCONV, CTCC_STDCALL);
        decl->fattr |= CTFP_CCONV;
        break;
      case H_(ea78b622,f234bd8e): case H_(252ffb06,8d50f34b):  /* sseregparm */
        decl->fattr |= CTF_SSEREGPARM;
        decl->fattr |= CTFP_CCONV;
        break;
#endif
      default:  /* Skip all other attributes. */
        goto skip_attr;
      }
    } else if (cp->tok >= CTOK_FIRSTDECL) {  /* For __attribute((const)) etc. */
      cp_next(cp);
    skip_attr:
      if (cp_opt(cp, '(')) {
        while (cp->tok != ')' && cp->tok != CTOK_EOF) cp_next(cp);
        cp_check(cp, ')');
```

```
      }
    } else {
      break;
    }
    if (!cp_opt(cp, ',')) break;
  }
  cp_check(cp, ')');
  cp_check(cp, ')');
}

/* Parse MSVC __declspec(...). */
static void cp_decl_msvcattribute(CPState *cp, CPDecl *decl)
{
  cp_next(cp);
  cp_check(cp, '(');
  while (cp->tok == CTOK_IDENT) {
    GCstr *attrstr = cp->str;
    cp_next(cp);
    switch (attrstr->hash) {
    case H_(bc2395fa,98f267f8):  /* align */
      cp_decl_align(cp, decl);
      break;
    default:  /* Ignore all other attributes. */
      if (cp_opt(cp, '(')) {
        while (cp->tok != ')' && cp->tok != CTOK_EOF) cp_next(cp);
        cp_check(cp, ')');
      }
      break;
    }
  }
  cp_check(cp, ')');
}

/* Parse declaration attributes (and common qualifiers). */
static void cp_decl_attributes(CPState *cp, CPDecl *decl)
{
  for (;;) {
    switch (cp->tok) {
    case CTOK_CONST: decl->attr |= CTF_CONST; break;
    case CTOK_VOLATILE: decl->attr |= CTF_VOLATILE; break;
    case CTOK_RESTRICT: break;  /* Ignore. */
    case CTOK_EXTENSION: break;  /* Ignore. */
    case CTOK_ATTRIBUTE: cp_decl_gccattribute(cp, decl); continue;
    case CTOK_ASM: cp_decl_asm(cp, decl); continue;
    case CTOK_DECLSPEC: cp_decl_msvcattribute(cp, decl); continue;
    case CTOK_CCDECL:
#if LJ_TARGET_X86
      CTF_INSERT(decl->fattr, CCONV, cp->ct->size);
      decl->fattr |= CTFP_CCONV;
#endif
      break;
    case CTOK_PTRSZ:
#if LJ_64
      CTF_INSERT(decl->attr, MSIZEP, cp->ct->size);
#endif
      break;
    default: return;
    }
    cp_next(cp);
  }
}

/* Parse struct/union/enum name. */
static CTypeID cp_struct_name(CPState *cp, CPDecl *sdecl, CTInfo info)
{
  CTypeID sid;
  CType *ct;
  cp->tmask = CPNS_STRUCT;
  cp_next(cp);
  cp_decl_attributes(cp, sdecl);
  cp->tmask = CPNS_DEFAULT;
  if (cp->tok != '{') {
    if (cp->tok != CTOK_IDENT) cp_err_token(cp, CTOK_IDENT);
    if (cp->val.id) {  /* Name of existing struct/union/enum. */
      sid = cp->val.id;
      ct = cp->ct;
```

```
      if ((ct->info ^ info) & (CTMASK_NUM|CTF_UNION))  /* Wrong type. */
        cp_errmsg(cp, 0, LJ_ERR_FFI_REDEF, strdata(gco2str(gcref(ct->name))));
    } else {  /* Create named, incomplete struct/union/enum. */
      if ((cp->mode & CPARSE_MODE_NOIMPLICIT))
        cp_errmsg(cp, 0, LJ_ERR_FFI_BADTAG, strdata(cp->str));
      sid = lj_ctype_new(cp->cts, &ct);
      ct->info = info;
      ct->size = CTSIZE_INVALID;
      ctype_setname(ct, cp->str);
      lj_ctype_addname(cp->cts, ct, sid);
    }
    cp_next(cp);
  } else {  /* Create anonymous, incomplete struct/union/enum. */
    sid = lj_ctype_new(cp->cts, &ct);
    ct->info = info;
    ct->size = CTSIZE_INVALID;
  }
  if (cp->tok == '{') {
    if (ct->size != CTSIZE_INVALID || ct->sib)
      cp_errmsg(cp, 0, LJ_ERR_FFI_REDEF, strdata(gco2str(gcref(ct->name))));
    ct->sib = 1;  /* Indicate the type is currently being defined. */
  }
  return sid;
}

/* Determine field alignment. */
static CTSize cp_field_align(CPState *cp, CType *ct, CTInfo info)
{
  CTSize align = ctype_align(info);
  UNUSED(cp); UNUSED(ct);
#if (LJ_TARGET_X86 && !LJ_ABI_WIN) || (LJ_TARGET_ARM && __APPLE__)
  /* The SYSV i386 and iOS ABIs limit alignment of non-vector fields to 2^2. */
  if (align > 2 && !(info & CTFP_ALIGNED)) {
    if (ctype_isarray(info) && !(info & CTF_VECTOR)) {
      do {
        ct = ctype_rawchild(cp->cts, ct);
        info = ct->info;
      } while (ctype_isarray(info) && !(info & CTF_VECTOR));
    }
    if (ctype_isnum(info) || ctype_isenum(info))
      align = 2;
  }
#endif
  return align;
}

/* Layout struct/union fields. */
static void cp_struct_layout(CPState *cp, CTypeID sid, CTInfo sattr)
{
  CTSize bofs = 0, bmaxofs = 0;  /* Bit offset and max. bit offset. */
  CTSize maxalign = ctype_align(sattr);
  CType *sct = ctype_get(cp->cts, sid);
  CTInfo sinfo = sct->info;
  CTypeID fieldid = sct->sib;
  while (fieldid) {
    CType *ct = ctype_get(cp->cts, fieldid);
    CTInfo attr = ct->size;  /* Field declaration attributes (temp.). */

    if (ctype_isfield(ct->info) ||
        (ctype_isxattrib(ct->info, CTA_SUBTYPE) && attr)) {
      CTSize align, amask;  /* Alignment (pow2) and alignment mask (bits). */
      CTSize sz;
      CTInfo info = lj_ctype_info(cp->cts, ctype_cid(ct->info), &sz);
      CTSize bsz, csz = 8*sz;  /* Field size and container size (in bits). */
      sinfo |= (info & (CTF_QUAL|CTF_VLA));  /* Merge pseudo-qualifiers. */

      /* Check for size overflow and determine alignment. */
      if (sz >= 0x20000000u || bofs + csz < bofs || (info & CTF_VLA)) {
        if (!(sz == CTSIZE_INVALID && ctype_isarray(info) &&
              !(sinfo & CTF_UNION)))
          cp_err(cp, LJ_ERR_FFI_INVSIZE);
        csz = sz = 0;  /* Treat a[] and a[?] as zero-sized. */
      }
      align = cp_field_align(cp, ct, info);
      if (((attr|sattr) & CTFP_PACKED) ||
          ((attr & CTFP_ALIGNED) && ctype_align(attr) > align))
```

```
1256          align = ctype_align(attr);
1257        if (cp->packstack[cp->curpack] < align)
1258          align = cp->packstack[cp->curpack];
1259        if (align > maxalign) maxalign = align;
1260        amask = (8u << align) - 1;
1261
1262        bsz = ctype_bitcsz(ct->info);  /* Bitfield size (temp.). */
1263        if (bsz == CTBSZ_FIELD || !ctype_isfield(ct->info)) {
1264          bsz = csz;  /* Regular fields or subtypes always fill the container. */
1265          bofs = (bofs + amask) & ~amask;  /* Start new aligned field. */
1266          ct->size = (bofs >> 3);  /* Store field offset. */
1267        } else {  /* Bitfield. */
1268          if (bsz == 0 || (attr & CTFP_ALIGNED) ||
1269              (!((attr|sattr) & CTFP_PACKED) && (bofs & amask) + bsz > csz))
1270            bofs = (bofs + amask) & ~amask;  /* Start new aligned field. */
1271
1272          /* Prefer regular field over bitfield. */
1273          if (bsz == csz && (bofs & amask) == 0) {
1274            ct->info = CTINFO(CT_FIELD, ctype_cid(ct->info));
1275            ct->size = (bofs >> 3);  /* Store field offset. */
1276          } else {
1277            ct->info = CTINFO(CT_BITFIELD,
1278              (info & (CTF_QUAL|CTF_UNSIGNED|CTF_BOOL)) +
1279              (csz << (CTSHIFT_BITCSZ-3)) + (bsz << CTSHIFT_BITBSZ));
1280 #if LJ_BE
1281            ct->info += ((csz - (bofs & (csz-1)) - bsz) << CTSHIFT_BITPOS);
1282 #else
1283            ct->info += ((bofs & (csz-1)) << CTSHIFT_BITPOS);
1284 #endif
1285            ct->size = ((bofs & ~(csz-1)) >> 3);  /* Store container offset. */
1286          }
1287        }
1288
1289        /* Determine next offset or max. offset. */
1290        if ((sinfo & CTF_UNION)) {
1291          if (bsz > bmaxofs) bmaxofs = bsz;
1292        } else {
1293          bofs += bsz;
1294        }
1295      }  /* All other fields in the chain are already set up. */
1296
1297      fieldid = ct->sib;
1298    }
1299
1300    /* Complete struct/union. */
1301    sct->info = sinfo + CTALIGN(maxalign);
1302    bofs = (sinfo & CTF_UNION) ? bmaxofs : bofs;
1303    maxalign = (8u << maxalign) - 1;
1304    sct->size = (((bofs + maxalign) & ~maxalign) >> 3);
1305  }
1306
1307  /* Parse struct/union declaration. */
1308  static CTypeID cp_decl_struct(CPState *cp, CPDecl *sdecl, CTInfo sinfo)
1309  {
1310    CTypeID sid = cp_struct_name(cp, sdecl, sinfo);
1311    if (cp_opt(cp, '{')) {  /* Struct/union definition. */
1312      CTypeID lastid = sid;
1313      int lastdecl = 0;
1314      while (cp->tok != '}') {
1315        CPDecl decl;
1316        CPscl scl = cp_decl_spec(cp, &decl, CDF_STATIC);
1317        decl.mode = scl ? CPARSE_MODE_DIRECT :
1318          CPARSE_MODE_DIRECT|CPARSE_MODE_ABSTRACT|CPARSE_MODE_FIELD;
1319
1320        for (;;) {
1321          CTypeID ctypeid;
1322
1323          if (lastdecl) cp_err_token(cp, '}');
1324
1325          /* Parse field declarator. */
1326          decl.bits = CTSIZE_INVALID;
1327          cp_declarator(cp, &decl);
1328          ctypeid = cp_decl_intern(cp, &decl);
1329
1330          if ((scl & CDF_STATIC)) {  /* Static constant in struct namespace. */
1331            CType *ct;
```

```
1332          CTypeID fieldid = cp_decl_constinit(cp, &ct, ctypeid);
1333          ctype_get(cp->cts, lastid)->sib = fieldid;
1334          lastid = fieldid;
1335          ctype_setname(ct, decl.name);
1336        } else {
1337          CTSize bsz = CTBSZ_FIELD;  /* Temp. for layout phase. */
1338          CType *ct;
1339          CTypeID fieldid = lj_ctype_new(cp->cts, &ct);  /* Do this first. */
1340          CType *tct = ctype_raw(cp->cts, ctypeid);
1341
1342          if (decl.bits == CTSIZE_INVALID) {  /* Regular field. */
1343            if (ctype_isarray(tct->info) && tct->size == CTSIZE_INVALID)
1344              lastdecl = 1;  /* a[] or a[?] must be the last declared field. */
1345
1346            /* Accept transparent struct/union/enum. */
1347            if (!decl.name) {
1348              if (!((ctype_isstruct(tct->info) && !(tct->info & CTF_VLA)) ||
1349                    ctype_isenum(tct->info)))
1350                cp_err_token(cp, CTOK_IDENT);
1351              ct->info = CTINFO(CT_ATTRIB, CTATTRIB(CTA_SUBTYPE) + ctypeid);
1352              ct->size = ctype_isstruct(tct->info) ?
1353                         (decl.attr|0x80000000u) : 0;  /* For layout phase. */
1354              goto add_field;
1355            }
1356          } else {  /* Bitfield. */
1357            bsz = decl.bits;
1358            if (!ctype_isinteger_or_bool(tct->info) ||
1359                (bsz == 0 && decl.name) || 8*tct->size > CTBSZ_MAX ||
1360                bsz > ((tct->info & CTF_BOOL) ? 1 : 8*tct->size))
1361              cp_errmsg(cp, ':', LJ_ERR_BADVAL);
1362          }
1363
1364          /* Create temporary field for layout phase. */
1365          ct->info = CTINFO(CT_FIELD, ctypeid + (bsz << CTSHIFT_BITCSZ));
1366          ct->size = decl.attr;
1367          if (decl.name) ctype_setname(ct, decl.name);
1368
1369        add_field:
1370          ctype_get(cp->cts, lastid)->sib = fieldid;
1371          lastid = fieldid;
1372        }
1373        if (!cp_opt(cp, ',')) break;
1374        cp_decl_reset(&decl);
1375      }
1376      cp_check(cp, ';');
1377    }
1378    cp_check(cp, '}');
1379    ctype_get(cp->cts, lastid)->sib = 0;  /* Drop sib = 1 for empty structs. */
1380    cp_decl_attributes(cp, sdecl);  /* Layout phase needs postfix attributes. */
1381    cp_struct_layout(cp, sid, sdecl->attr);
1382  }
1383  return sid;
1384 }
1385
1386 /* Parse enum declaration. */
1387 static CTypeID cp_decl_enum(CPState *cp, CPDecl *sdecl)
1388 {
1389   CTypeID eid = cp_struct_name(cp, sdecl, CTINFO(CT_ENUM, CTID_VOID));
1390   CTInfo einfo = CTINFO(CT_ENUM, CTALIGN(2) + CTID_UINT32);
1391   CTSize esize = 4;  /* Only 32 bit enums are supported. */
1392   if (cp_opt(cp, '{')) {  /* Enum definition. */
1393     CPValue k;
1394     CTypeID lastid = eid;
1395     k.u32 = 0;
1396     k.id = CTID_INT32;
1397     do {
1398       GCstr *name = cp->str;
1399       if (cp->tok != CTOK_IDENT) cp_err_token(cp, CTOK_IDENT);
1400       if (cp->val.id) cp_errmsg(cp, 0, LJ_ERR_FFI_REDEF, strdata(name));
1401       cp_next(cp);
1402       if (cp_opt(cp, '=')) {
1403         cp_expr_kint(cp, &k);
1404         if (k.id == CTID_UINT32) {
1405           /* C99 says that enum constants are always (signed) integers.
1406           ** But since unsigned constants like 0x80000000 are quite common,
1407           ** those are left as uint32_t.
```

```
1408            */
1409            if (k.i32 >= 0) k.id = CTID_INT32;
1410          } else {
1411            /* OTOH it's common practice and even mandated by some ABIs
1412            ** that the enum type itself is unsigned, unless there are any
1413            ** negative constants.
1414            */
1415            k.id = CTID_INT32;
1416            if (k.i32 < 0) einfo = CTINFO(CT_ENUM, CTALIGN(2) + CTID_INT32);
1417          }
1418        }
1419        /* Add named enum constant. */
1420        {
1421          CType *ct;
1422          CTypeID constid = lj_ctype_new(cp->cts, &ct);
1423          ctype_get(cp->cts, lastid)->sib = constid;
1424          lastid = constid;
1425          ctype_setname(ct, name);
1426          ct->info = CTINFO(CT_CONSTVAL, CTF_CONST|k.id);
1427          ct->size = k.u32++;
1428          if (k.u32 == 0x80000000u) k.id = CTID_UINT32;
1429          lj_ctype_addname(cp->cts, ct, constid);
1430        }
1431        if (!cp_opt(cp, ',')) break;
1432      } while (cp->tok != '}');  /* Trailing ',' is ok. */
1433      cp_check(cp, '}');
1434      /* Complete enum. */
1435      ctype_get(cp->cts, eid)->info = einfo;
1436      ctype_get(cp->cts, eid)->size = esize;
1437    }
1438    return eid;
1439  }
1440
1441  /* Parse declaration specifiers. */
1442  static CPscl cp_decl_spec(CPState *cp, CPDecl *decl, CPscl scl)
1443  {
1444    uint32_t cds = 0, sz = 0;
1445    CTypeID tdef = 0;
1446
1447    decl->cp = cp;
1448    decl->mode = cp->mode;
1449    decl->name = NULL;
1450    decl->redir = NULL;
1451    decl->attr = 0;
1452    decl->fattr = 0;
1453    decl->pos = decl->top = 0;
1454    decl->stack[0].next = 0;
1455
1456    for (;;) {  /* Parse basic types. */
1457      cp_decl_attributes(cp, decl);
1458      if (cp->tok >= CTOK_FIRSTDECL && cp->tok <= CTOK_LASTDECLFLAG) {
1459        uint32_t cbit;
1460        if (cp->ct->size) {
1461          if (sz) goto end_decl;
1462          sz = cp->ct->size;
1463        }
1464        cbit = (1u << (cp->tok - CTOK_FIRSTDECL));
1465        cds = cds | cbit | ((cbit & cds & CDF_LONG) << 1);
1466        if (cp->tok >= CTOK_FIRSTSCL) {
1467          if (!(scl & cbit)) cp_errmsg(cp, cp->tok, LJ_ERR_FFI_BADSCL);
1468        } else if (tdef) {
1469          goto end_decl;
1470        }
1471        cp_next(cp);
1472        continue;
1473      }
1474      if (sz || tdef ||
1475          (cds & (CDF_SHORT|CDF_LONG|CDF_SIGNED|CDF_UNSIGNED|CDF_COMPLEX)))
1476        break;
1477      switch (cp->tok) {
1478      case CTOK_STRUCT:
1479        tdef = cp_decl_struct(cp, decl, CTINFO(CT_STRUCT, 0));
1480        continue;
1481      case CTOK_UNION:
1482        tdef = cp_decl_struct(cp, decl, CTINFO(CT_STRUCT, CTF_UNION));
1483        continue;
```

```
    case CTOK_ENUM:
      tdef = cp_decl_enum(cp, decl);
      continue;
    case CTOK_IDENT:
      if (ctype_istypedef(cp->ct->info)) {
        tdef = ctype_cid(cp->ct->info);  /* Get typedef. */
        cp_next(cp);
        continue;
      }
      break;
    case '$':
      tdef = cp->val.id;
      cp_next(cp);
      continue;
    default:
      break;
    }
    break;
  }
end_decl:

  if ((cds & CDF_COMPLEX))  /* Use predefined complex types. */
    tdef = sz == 4 ? CTID_COMPLEX_FLOAT : CTID_COMPLEX_DOUBLE;

  if (tdef) {
    cp_push_type(decl, tdef);
  } else if ((cds & CDF_VOID)) {
    cp_push(decl, CTINFO(CT_VOID, (decl->attr & CTF_QUAL)), CTSIZE_INVALID);
    decl->attr &= ~CTF_QUAL;
  } else {
    /* Determine type info and size. */
    CTInfo info = CTINFO(CT_NUM, (cds & CDF_UNSIGNED) ? CTF_UNSIGNED : 0);
    if ((cds & CDF_BOOL)) {
      if ((cds & ~(CDF_SCL|CDF_BOOL|CDF_INT|CDF_SIGNED|CDF_UNSIGNED)))
        cp_errmsg(cp, 0, LJ_ERR_FFI_INVTYPE);
      info |= CTF_BOOL;
      if (!(cds & CDF_SIGNED)) info |= CTF_UNSIGNED;
      if (!sz) {
        sz = 1;
      }
    } else if ((cds & CDF_FP)) {
      info = CTINFO(CT_NUM, CTF_FP);
      if ((cds & CDF_LONG)) sz = sizeof(long double);
    } else if ((cds & CDF_CHAR)) {
      if ((cds & (CDF_CHAR|CDF_SIGNED|CDF_UNSIGNED)) == CDF_CHAR)
        info |= CTF_UCHAR;  /* Handle platforms where char is unsigned. */
    } else if ((cds & CDF_SHORT)) {
      sz = sizeof(short);
    } else if ((cds & CDF_LONGLONG)) {
      sz = 8;
    } else if ((cds & CDF_LONG)) {
      info |= CTF_LONG;
      sz = sizeof(long);
    } else if (!sz) {
      if (!(cds & (CDF_SIGNED|CDF_UNSIGNED)))
        cp_errmsg(cp, cp->tok, LJ_ERR_FFI_DECLSPEC);
      sz = sizeof(int);
    }
    lua_assert(sz != 0);
    info += CTALIGN(lj_fls(sz));  /* Use natural alignment. */
    info += (decl->attr & CTF_QUAL);  /* Merge qualifiers. */
    cp_push(decl, info, sz);
    decl->attr &= ~CTF_QUAL;
  }
  decl->specpos = decl->pos;
  decl->specattr = decl->attr;
  decl->specfattr = decl->fattr;
  return (cds & CDF_SCL);  /* Return storage class. */
}

/* Parse array declaration. */
static void cp_decl_array(CPState *cp, CPDecl *decl)
{
  CTInfo info = CTINFO(CT_ARRAY, 0);
  CTSize nelem = CTSIZE_INVALID;  /* Default size for a[] or a[?]. */
  cp_decl_attributes(cp, decl);
```

```
  if (cp_opt(cp, '?'))
    info |= CTF_VLA;  /* Create variable-length array a[?]. */
  else if (cp->tok != ']')
    nelem = cp_expr_ksize(cp);
  cp_check(cp, ']');
  cp_add(decl, info, nelem);
}

/* Parse function declaration. */
static void cp_decl_func(CPState *cp, CPDecl *fdecl)
{
  CTSize nargs = 0;
  CTInfo info = CTINFO(CT_FUNC, 0);
  CTypeID lastid = 0, anchor = 0;
  if (cp->tok != ')') {
    do {
      CPDecl decl;
      CTypeID ctypeid, fieldid;
      CType *ct;
      if (cp_opt(cp, '.')) {  /* Vararg function. */
        cp_check(cp, '.');  /* Workaround for the minimalistic lexer. */
        cp_check(cp, '.');
        info |= CTF_VARARG;
        break;
      }
      cp_decl_spec(cp, &decl, CDF_REGISTER);
      decl.mode = CPARSE_MODE_DIRECT|CPARSE_MODE_ABSTRACT;
      cp_declarator(cp, &decl);
      ctypeid = cp_decl_intern(cp, &decl);
      ct = ctype_raw(cp->cts, ctypeid);
      if (ctype_isvoid(ct->info))
        break;
      else if (ctype_isrefarray(ct->info))
        ctypeid = lj_ctype_intern(cp->cts,
          CTINFO(CT_PTR, CTALIGN_PTR|ctype_cid(ct->info)), CTSIZE_PTR);
      else if (ctype_isfunc(ct->info))
        ctypeid = lj_ctype_intern(cp->cts,
          CTINFO(CT_PTR, CTALIGN_PTR|ctypeid), CTSIZE_PTR);
      /* Add new parameter. */
      fieldid = lj_ctype_new(cp->cts, &ct);
      if (anchor)
        ctype_get(cp->cts, lastid)->sib = fieldid;
      else
        anchor = fieldid;
      lastid = fieldid;
      if (decl.name) ctype_setname(ct, decl.name);
      ct->info = CTINFO(CT_FIELD, ctypeid);
      ct->size = nargs++;
    } while (cp_opt(cp, ','));
  }
  cp_check(cp, ')');
  if (cp_opt(cp, '{')) {  /* Skip function definition. */
    int level = 1;
    cp->mode |= CPARSE_MODE_SKIP;
    for (;;) {
      if (cp->tok == '{') level++;
      else if (cp->tok == '}' && --level == 0) break;
      else if (cp->tok == CTOK_EOF) cp_err_token(cp, '}');
      cp_next(cp);
    }
    cp->mode &= ~CPARSE_MODE_SKIP;
    cp->tok = ';';  /* Ok for cp_decl_multi(), error in cp_decl_single(). */
  }
  info |= (fdecl->fattr & ~CTMASK_CID);
  fdecl->fattr = 0;
  fdecl->stack[cp_add(fdecl, info, nargs)].sib = anchor;
}

/* Parse declarator. */
static void cp_declarator(CPState *cp, CPDecl *decl)
{
  if (++cp->depth > CPARSE_MAX_DECLDEPTH) cp_err(cp, LJ_ERR_XLEVELS);

  for (;;) {  /* Head of declarator. */
    if (cp_opt(cp, '*')) {  /* Pointer. */
      CTSize sz;
```

```
      CTInfo info;
      cp_decl_attributes(cp, decl);
      sz = CTSIZE_PTR;
      info = CTINFO(CT_PTR, CTALIGN_PTR);
#if LJ_64
      if (ctype_msizeP(decl->attr) == 4) {
        sz = 4;
        info = CTINFO(CT_PTR, CTALIGN(2));
      }
#endif
      info += (decl->attr & (CTF_QUAL|CTF_REF));
      decl->attr &= ~(CTF_QUAL|(CTMASK_MSIZEP<<CTSHIFT_MSIZEP));
      cp_push(decl, info, sz);
    } else if (cp_opt(cp, '&') || cp_opt(cp, CTOK_ANDAND)) {  /* Reference. */
      decl->attr &= ~(CTF_QUAL|(CTMASK_MSIZEP<<CTSHIFT_MSIZEP));
      cp_push(decl, CTINFO_REF(0), CTSIZE_PTR);
    } else {
      break;
    }
  }

  if (cp_opt(cp, '(')) {  /* Inner declarator. */
    CPDeclIdx pos;
    cp_decl_attributes(cp, decl);
    /* Resolve ambiguity between inner declarator and 1st function parameter. */
    if ((decl->mode & CPARSE_MODE_ABSTRACT) &&
        (cp->tok == ')' || cp_istypedecl(cp))) goto func_decl;
    pos = decl->pos;
    cp_declarator(cp, decl);
    cp_check(cp, ')');
    decl->pos = pos;
  } else if (cp->tok == CTOK_IDENT) {  /* Direct declarator. */
    if (!(decl->mode & CPARSE_MODE_DIRECT)) cp_err_token(cp, CTOK_EOF);
    decl->name = cp->str;
    decl->nameid = cp->val.id;
    cp_next(cp);
  } else {  /* Abstract declarator. */
    if (!(decl->mode & CPARSE_MODE_ABSTRACT)) cp_err_token(cp, CTOK_IDENT);
  }

  for (;;) {  /* Tail of declarator. */
    if (cp_opt(cp, '[')) {  /* Array. */
      cp_decl_array(cp, decl);
    } else if (cp_opt(cp, '(')) {  /* Function. */
    func_decl:
      cp_decl_func(cp, decl);
    } else {
      break;
    }
  }

  if ((decl->mode & CPARSE_MODE_FIELD) && cp_opt(cp, ':'))  /* Field width. */
    decl->bits = cp_expr_ksize(cp);

  /* Process postfix attributes. */
  cp_decl_attributes(cp, decl);
  cp_push_attributes(decl);

  cp->depth--;
}

/* Parse an abstract type declaration and return it's C type ID. */
static CTypeID cp_decl_abstract(CPState *cp)
{
  CPDecl decl;
  cp_decl_spec(cp, &decl, 0);
  decl.mode = CPARSE_MODE_ABSTRACT;
  cp_declarator(cp, &decl);
  return cp_decl_intern(cp, &decl);
}

/* Handle pragmas. */
static void cp_pragma(CPState *cp, BCLine pragmaline)
{
  cp_next(cp);
  if (cp->tok == CTOK_IDENT &&
```

```
1712        cp->str->hash == H_(e79b999f,42ca3e85))  {  /* pack */
1713      cp_next(cp);
1714      cp_check(cp, '(');
1715      if (cp->tok == CTOK_IDENT) {
1716        if (cp->str->hash == H_(738e923c,a1b65954)) {  /* push */
1717          if (cp->curpack < CPARSE_MAX_PACKSTACK) {
1718            cp->packstack[cp->curpack+1] = cp->packstack[cp->curpack];
1719            cp->curpack++;
1720          }
1721        } else if (cp->str->hash == H_(6c71cf27,6c71cf27)) {  /* pop */
1722          if (cp->curpack > 0) cp->curpack--;
1723        } else {
1724          cp_errmsg(cp, cp->tok, LJ_ERR_XSYMBOL);
1725        }
1726        cp_next(cp);
1727        if (!cp_opt(cp, ',')) goto end_pack;
1728      }
1729      if (cp->tok == CTOK_INTEGER) {
1730        cp->packstack[cp->curpack] = cp->val.u32 ? lj_fls(cp->val.u32) : 0;
1731        cp_next(cp);
1732      } else {
1733        cp->packstack[cp->curpack] = 255;
1734      }
1735    end_pack:
1736      cp_check(cp, ')');
1737    } else {  /* Ignore all other pragmas. */
1738      while (cp->tok != CTOK_EOF && cp->linenumber == pragmaline)
1739        cp_next(cp);
1740    }
1741  }
1742
1743  /* Parse multiple C declarations of types or extern identifiers. */
1744  static void cp_decl_multi(CPState *cp)
1745  {
1746    int first = 1;
1747    while (cp->tok != CTOK_EOF) {
1748      CPDecl decl;
1749      CPscl scl;
1750      if (cp_opt(cp, ';')) {  /* Skip empty statements. */
1751        first = 0;
1752        continue;
1753      }
1754      if (cp->tok == '#') {  /* Workaround, since we have no preprocessor, yet. */
1755        BCLine pragmaline = cp->linenumber;
1756        if (!(cp_next(cp) == CTOK_IDENT &&
1757              cp->str->hash == H_(f5e6b4f8,1d509107)))  /* pragma */
1758          cp_errmsg(cp, cp->tok, LJ_ERR_XSYMBOL);
1759        cp_pragma(cp, pragmaline);
1760        continue;
1761      }
1762      scl = cp_decl_spec(cp, &decl, CDF_TYPEDEF|CDF_EXTERN|CDF_STATIC);
1763      if ((cp->tok == ';' || cp->tok == CTOK_EOF) &&
1764          ctype_istypedef(decl.stack[0].info)) {
1765        CTInfo info = ctype_rawchild(cp->cts, &decl.stack[0])->info;
1766        if (ctype_isstruct(info) || ctype_isenum(info))
1767          goto decl_end;  /* Accept empty declaration of struct/union/enum. */
1768      }
1769      for (;;) {
1770        CTypeID ctypeid;
1771        cp_declarator(cp, &decl);
1772        ctypeid = cp_decl_intern(cp, &decl);
1773        if (decl.name && !decl.nameid) {  /* NYI: redeclarations are ignored. */
1774          CType *ct;
1775          CTypeID id;
1776          if ((scl & CDF_TYPEDEF)) {  /* Create new typedef. */
1777            id = lj_ctype_new(cp->cts, &ct);
1778            ct->info = CTINFO(CT_TYPEDEF, ctypeid);
1779            goto noredir;
1780          } else if (ctype_isfunc(ctype_get(cp->cts, ctypeid)->info)) {
1781            /* Treat both static and extern function declarations as extern. */
1782            ct = ctype_get(cp->cts, ctypeid);
1783            /* We always get new anonymous functions (typedefs are copied). */
1784            lua_assert(gcref(ct->name) == NULL);
1785            id = ctypeid;  /* Just name it. */
1786          } else if ((scl & CDF_STATIC)) {  /* Accept static constants. */
1787            id = cp_decl_constinit(cp, &ct, ctypeid);
```

```
1788            goto noredir;
1789          } else {  /* External references have extern or no storage class. */
1790            id = lj_ctype_new(cp->cts, &ct);
1791            ct->info = CTINFO(CT_EXTERN, ctypeid);
1792          }
1793          if (decl.redir) {  /* Add attribute for redirected symbol name. */
1794            CType *cta;
1795            CTypeID aid = lj_ctype_new(cp->cts, &cta);
1796            ct = ctype_get(cp->cts, id);  /* Table may have been reallocated. */
1797            cta->info = CTINFO(CT_ATTRIB, CTATTRIB(CTA_REDIR));
1798            cta->sib = ct->sib;
1799            ct->sib = aid;
1800            ctype_setname(cta, decl.redir);
1801          }
1802        noredir:
1803          ctype_setname(ct, decl.name);
1804          lj_ctype_addname(cp->cts, ct, id);
1805        }
1806        if (!cp_opt(cp, ',')) break;
1807        cp_decl_reset(&decl);
1808      }
1809    decl_end:
1810      if (cp->tok == CTOK_EOF && first) break;  /* May omit ';' for 1 decl. */
1811      first = 0;
1812      cp_check(cp, ';');
1813    }
1814  }
1815
1816  /* Parse a single C type declaration. */
1817  static void cp_decl_single(CPState *cp)
1818  {
1819    CPDecl decl;
1820    cp_decl_spec(cp, &decl, 0);
1821    cp_declarator(cp, &decl);
1822    cp->val.id = cp_decl_intern(cp, &decl);
1823    if (cp->tok != CTOK_EOF) cp_err_token(cp, CTOK_EOF);
1824  }
1825
1826  #undef H_
1827
1828  /* ------------------------------------------------------------------------ */
1829
1830  /* Protected callback for C parser. */
1831  static TValue *cpcparser(lua_State *L, lua_CFunction dummy, void *ud)
1832  {
1833    CPState *cp = (CPState *)ud;
1834    UNUSED(dummy);
1835    cframe_errfunc(L->cframe) = -1;  /* Inherit error function. */
1836    cp_init(cp);
1837    if ((cp->mode & CPARSE_MODE_MULTI))
1838      cp_decl_multi(cp);
1839    else
1840      cp_decl_single(cp);
1841    if (cp->param && cp->param != cp->L->top)
1842      cp_err(cp, LJ_ERR_FFI_NUMPARAM);
1843    lua_assert(cp->depth == 0);
1844    return NULL;
1845  }
1846
1847  /* C parser. */
1848  int lj_cparse(CPState *cp)
1849  {
1850    LJ_CTYPE_SAVE(cp->cts);
1851    int errcode = lj_vm_cpcall(cp->L, NULL, cp, cpcparser);
1852    if (errcode)
1853      LJ_CTYPE_RESTORE(cp->cts);
1854    cp_cleanup(cp);
1855    return errcode;
1856  }
1857
1858  #endif
```

One Level Up Top Level

src/lj_cparse.h - luajit-2.0-src

Data types defined

- CPChar
- CPState
- CPState
- CPToken
- CPValue
- CPValue

Macros defined

- CPARSE_MAX_BUF
- CPARSE_MAX_DECLDEPTH
- CPARSE_MAX_DECLSTACK
- CPARSE_MAX_PACKSTACK
- CPARSE_MODE_ABSTRACT
- CPARSE_MODE_DIRECT
- CPARSE_MODE_FIELD
- CPARSE_MODE_MULTI
- CPARSE_MODE_NOIMPLICIT
- CPARSE_MODE_SKIP
- _LJ_CPARSE_H

Source code

```
/*
** C declaration parser.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_CPARSE_H
#define _LJ_CPARSE_H

#include "lj_obj.h"
 #include "lj_ctype.h"

 #if LJ_HASFFI

 /* C parser limits. */
 #define CPARSE_MAX_BUF               32768        /* Max. token buffer size. */
 #define CPARSE_MAX_DECLSTACK        100        /* Max. declaration stack depth. */
 #define CPARSE_MAX_DECLDEPTH        20        /* Max. recursive declaration depth. */
 #define CPARSE_MAX_PACKSTACK        7        /* Max. pack pragma stack depth. */

 /* Flags for C parser mode. */
 #define CPARSE_MODE_MULTI        1        /* Process multiple declarations. */
 #define CPARSE_MODE_ABSTRACT        2        /* Accept abstract declarators. */
 #define CPARSE_MODE_DIRECT        4        /* Accept direct declarators. */
```

```
#define CPARSE_MODE_FIELD        8        /* Accept field width in bits, too. */
#define CPARSE_MODE_NOIMPLICIT       16       /* Reject implicit declarations. */
#define CPARSE_MODE_SKIP        32        /* Skip definitions, ignore errors. */

typedef int CPChar;        /* C parser character. Unsigned ext. from char. */
typedef int CPToken;        /* C parser token. */

/* C parser internal value representation. */
typedef struct CPValue {
  union {
    int32_t i32;        /* Value for CTID_INT32. */
    uint32_t u32;        /* Value for CTID_UINT32. */
  };
  CTypeID id;               /* C Type ID of the value. */
} CPValue;

/* C parser state. */
typedef struct CPState {
  CPChar c;               /* Current character. */
  CPToken tok;               /* Current token. */
  CPValue val;               /* Token value. */
  GCstr *str;               /* Interned string of identifier/keyword. */
  CType *ct;               /* C type table entry. */
  const char *p;        /* Current position in input buffer. */
  SBuf sb;               /* String buffer for tokens. */
  lua_State *L;               /* Lua state. */
  CTState *cts;               /* C type state. */
  TValue *param;        /* C type parameters. */
  const char *srcname;        /* Current source name. */
  BCLine linenumber;        /* Input line counter. */
  int depth;               /* Recursive declaration depth. */
  uint32_t tmask;        /* Type mask for next identifier. */
  uint32_t mode;        /* C parser mode. */
  uint8_t packstack[CPARSE_MAX_PACKSTACK];  /* Stack for pack pragmas. */
  uint8_t curpack;        /* Current position in pack pragma stack. */
} CPState;

LJ_FUNC int lj_cparse(CPState *cp);

#endif

#endif
```

One Level Up Top Level

src/lj_err.h - luajit-2.0-src

Global variables defined

- lj_err_allmsg

Data types defined

- ErrMsg

Macros defined

- ERRDEF
- _LJ_ERR_H
- err2msg

Source code

```
1  /*
2  ** Error handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_ERR_H
7  #define _LJ_ERR_H
8
9  #include <stdarg.h>
10
11  #include "lj_obj.h"
12
13  typedef enum {
14  #define ERRDEF(name, msg) \
15    LJ_ERR_##name, LJ_ERR_##name##_ = LJ_ERR_##name + sizeof(msg)-1,
16  #include "lj_errmsg.h"
17    LJ_ERR__MAX
18  } ErrMsg;
19
20  LJ_DATA const char *lj_err_allmsg;
21  #define err2msg(em)        (lj_err_allmsg+(int)(em))
22
23  LJ_FUNC GCstr *lj_err_str(lua_State *L, ErrMsg em);
24  LJ_FUNCA_NORET void LJ_FASTCALL lj_err_throw(lua_State *L, int errcode);
25  LJ_FUNC_NORET void lj_err_mem(lua_State *L);
26  LJ_FUNC_NORET void lj_err_run(lua_State *L);
27  LJ_FUNC_NORET void lj_err_msg(lua_State *L, ErrMsg em);
28  LJ_FUNC_NORET void lj_err_lex(lua_State *L, GCstr *src, const char *tok,
29                                BCLine line, ErrMsg em, va_list argp);
30  LJ_FUNC_NORET void lj_err_optype(lua_State *L, cTValue *o, ErrMsg opm);
31  LJ_FUNC_NORET void lj_err_comp(lua_State *L, cTValue *o1, cTValue *o2);
32  LJ_FUNC_NORET void lj_err_optype_call(lua_State *L, TValue *o);
33  LJ_FUNC_NORET void lj_err_callermsg(lua_State *L, const char *msg);
34  LJ_FUNC_NORET void lj_err_callerv(lua_State *L, ErrMsg em, ...);
35  LJ_FUNC_NORET void lj_err_caller(lua_State *L, ErrMsg em);
36  LJ_FUNC_NORET void lj_err_arg(lua_State *L, int narg, ErrMsg em);
37  LJ_FUNC_NORET void lj_err_argv(lua_State *L, int narg, ErrMsg em, ...);
38  LJ_FUNC_NORET void lj_err_argtype(lua_State *L, int narg, const char *xname);
39  LJ_FUNC_NORET void lj_err_argt(lua_State *L, int narg, int tt);
40
41  #endif
```

src/lib_math.c - luajit-2.0-src

Data types defined

- U64double

Functions defined

- LJLIB_ASM(math_abs) LJLIB_REC(.)
- LJLIB_ASM(math_min) LJLIB_REC(math_minmax IR_MIN)
- LJLIB_ASM_(math_floor) LJLIB_REC(math_round IRFPM_FLOOR)
- LJLIB_ASM_(math_log10) LJLIB_REC(math_unary IRFPM_LOG10)
- LJLIB_ASM_(math_pow) LJLIB_REC(.)
- LJLIB_ASM_(math_max) LJLIB_REC(math_minmax IR_MAX)
- LJLIB_CF(math_random) LJLIB_REC(.)
- LJLIB_CF(math_randomseed)
- LJLIB_LUA(math_deg) /* function(x) return x * 57.29577951308232 end */
- lj_math_random_step
- luaopen_math
- random_init

Macros defined

- LJLIB_MODULE_math
- LUA_LIB
- TW223_GEN
- lib_math_c

Source code

```
/*
** Math library.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#include <math.h>

#define lib_math_c
#define LUA_LIB

 #include "lua.h"
 #include "lauxlib.h"
 #include "lualib.h"

 #include "lj_obj.h"
 #include "lj_lib.h"
 #include "lj_vm.h"

 /* ------------------------------------------------------------------------ */
```

```
#define LJLIB_MODULE_math

LJLIB_ASM(math_abs)               LJLIB_REC(.)
{
  lj_lib_checknumber(L, 1);
  return FFH_RETRY;
}
LJLIB_ASM_(math_floor)              LJLIB_REC(math_round IRFPM_FLOOR)
LJLIB_ASM_(math_ceil)               LJLIB_REC(math_round IRFPM_CEIL)

LJLIB_ASM(math_sqrt)               LJLIB_REC(math_unary IRFPM_SQRT)
{
  lj_lib_checknum(L, 1);
  return FFH_RETRY;
}
LJLIB_ASM_(math_log10)              LJLIB_REC(math_unary IRFPM_LOG10)
LJLIB_ASM_(math_exp)               LJLIB_REC(math_unary IRFPM_EXP)
LJLIB_ASM_(math_sin)               LJLIB_REC(math_unary IRFPM_SIN)
LJLIB_ASM_(math_cos)               LJLIB_REC(math_unary IRFPM_COS)
LJLIB_ASM_(math_tan)               LJLIB_REC(math_unary IRFPM_TAN)
LJLIB_ASM_(math_asin)               LJLIB_REC(math_atrig FF_math_asin)
LJLIB_ASM_(math_acos)               LJLIB_REC(math_atrig FF_math_acos)
LJLIB_ASM_(math_atan)               LJLIB_REC(math_atrig FF_math_atan)
LJLIB_ASM_(math_sinh)               LJLIB_REC(math_htrig IRCALL_sinh)
LJLIB_ASM_(math_cosh)               LJLIB_REC(math_htrig IRCALL_cosh)
LJLIB_ASM_(math_tanh)               LJLIB_REC(math_htrig IRCALL_tanh)
LJLIB_ASM_(math_frexp)
LJLIB_ASM_(math_modf)               LJLIB_REC(.)

LJLIB_ASM(math_log)               LJLIB_REC(math_log)
{
  double x = lj_lib_checknum(L, 1);
  if (L->base+1 < L->top) {
    double y = lj_lib_checknum(L, 2);
#ifdef LUAJIT_NO_LOG2
    x = log(x); y = 1.0 / log(y);
#else
    x = lj_vm_log2(x); y = 1.0 / lj_vm_log2(y);
#endif
    setnumV(L->base-1-LJ_FR2, x*y);  /* Do NOT join the expression to x / y. */
    return FFH_RES(1);
  }
  return FFH_RETRY;
}

LJLIB_LUA(math_deg) /* function(x) return x * 57.29577951308232 end */
LJLIB_LUA(math_rad) /* function(x) return x * 0.017453292519943295 end */

LJLIB_ASM(math_atan2)               LJLIB_REC(.)
{
  lj_lib_checknum(L, 1);
  lj_lib_checknum(L, 2);
  return FFH_RETRY;
}
LJLIB_ASM_(math_pow)               LJLIB_REC(.)
LJLIB_ASM_(math_fmod)

LJLIB_ASM(math_ldexp)               LJLIB_REC(.)
{
  lj_lib_checknum(L, 1);
#if LJ_DUALNUM && !LJ_TARGET_X86ORX64
  lj_lib_checkint(L, 2);
#else
  lj_lib_checknum(L, 2);
#endif
  return FFH_RETRY;
}

LJLIB_ASM(math_min)               LJLIB_REC(math_minmax IR_MIN)
{
  int i = 0;
  do { lj_lib_checknumber(L, ++i); } while (L->base+i < L->top);
  return FFH_RETRY;
}
LJLIB_ASM_(math_max)               LJLIB_REC(math_minmax IR_MAX)
```

```
96
97  LJLIB_PUSH(3.14159265358979323846) LJLIB_SET(pi)
98  LJLIB_PUSH(1e310) LJLIB_SET(huge)
99
100 /* ------------------------------------------------------------------------ */
101
102 /* This implements a Tausworthe PRNG with period 2^223. Based on:
103 **   Tables of maximally-equidistributed combined LFSR generators,
104 **   Pierre L'Ecuyer, 1991, table 3, 1st entry.
105 ** Full-period ME-CF generator with L=64, J=4, k=223, N1=49.
106 */
107
108 /* PRNG state. */
109 struct RandomState {
110   uint64_t gen[4];        /* State of the 4 LFSR generators. */
111   int valid;                /* State is valid. */
112 };
113
114 /* Union needed for bit-pattern conversion between uint64_t and double. */
115 typedef union { uint64_t u64; double d; } U64double;
116
117 /* Update generator i and compute a running xor of all states. */
118 #define TW223_GEN(i, k, q, s) \
119   z = rs->gen[i]; \
120   z = (((z<<q)^z) >> (k-s)) ^ ((z&((uint64_t)(int64_t)-1 << (64-k)))<<s); \
121   r ^= z; rs->gen[i] = z;
122
123 /* PRNG step function. Returns a double in the range 1.0 <= d < 2.0. */
124 LJ_NOINLINE uint64_t LJ_FASTCALL lj_math_random_step(RandomState *rs)
125 {
126   uint64_t z, r = 0;
127   TW223_GEN(0, 63, 31, 18)
128   TW223_GEN(1, 58, 19, 28)
129   TW223_GEN(2, 55, 24,  7)
130   TW223_GEN(3, 47, 21,  8)
131   return (r & U64x(000fffff,ffffffff)) | U64x(3ff00000,00000000);
132 }
133
134 /* PRNG initialization function. */
135 static void random_init(RandomState *rs, double d)
136 {
137   uint32_t r = 0x11090601;  /* 64-k[i] as four 8 bit constants. */
138   int i;
139   for (i = 0; i < 4; i++) {
140     U64double u;
141     uint32_t m = 1u << (r&255);
142     r >>= 8;
143     u.d = d = d * 3.14159265358979323846 + 2.7182818284590452354;
144     if (u.u64 < m) u.u64 += m;  /* Ensure k[i] MSB of gen[i] are non-zero. */
145     rs->gen[i] = u.u64;
146   }
147   rs->valid = 1;
148   for (i = 0; i < 10; i++)
149     lj_math_random_step(rs);
150 }
151
152 /* PRNG extract function. */
153 LJLIB_PUSH(top-2)  /* Upvalue holds userdata with RandomState. */
154 LJLIB_CF(math_random)               LJLIB_REC(.)
155 {
156   int n = (int)(L->top - L->base);
157   RandomState *rs = (RandomState *)(uddata(udataV(lj_lib_upvalue(L, 1))));
158   U64double u;
159   double d;
160   if (LJ_UNLIKELY(!rs->valid)) random_init(rs, 0.0);
161   u.u64 = lj_math_random_step(rs);
162   d = u.d - 1.0;
163   if (n > 0) {
164 #if LJ_DUALNUM
165     int isint = 1;
166     double r1;
167     lj_lib_checknumber(L, 1);
168     if (tvisint(L->base)) {
169       r1 = (lua_Number)intV(L->base);
170     } else {
171       isint = 0;
```

```
      r1 = numV(L->base);
    }
#else
    double r1 = lj_lib_checknum(L, 1);
#endif
    if (n == 1) {
      d = lj_vm_floor(d*r1) + 1.0;  /* d is an int in range [1, r1] */
    } else {
#if LJ_DUALNUM
      double r2;
      lj_lib_checknumber(L, 2);
      if (tvisint(L->base+1)) {
        r2 = (lua_Number)intV(L->base+1);
      } else {
        isint = 0;
        r2 = numV(L->base+1);
      }
#else
      double r2 = lj_lib_checknum(L, 2);
#endif
      d = lj_vm_floor(d*(r2-r1+1.0)) + r1;  /* d is an int in range [r1, r2] */
    }
#if LJ_DUALNUM
    if (isint) {
      setintV(L->top-1, lj_num2int(d));
      return 1;
    }
#endif
  }  /* else: d is a double in range [0, 1] */
  setnumV(L->top++, d);
  return 1;
}

/* PRNG seed function. */
LJLIB_PUSH(top-2)  /* Upvalue holds userdata with RandomState. */
LJLIB_CF(math_randomseed)
{
  RandomState *rs = (RandomState *)(uddata(udataV(lj_lib_upvalue(L, 1))));
  random_init(rs, lj_lib_checknum(L, 1));
  return 0;
}

/* ------------------------------------------------------------------------ */

#include "lj_libdef.h"

LUALIB_API int luaopen_math(lua_State *L)
{
  RandomState *rs;
  rs = (RandomState *)lua_newuserdata(L, sizeof(RandomState));
  rs->valid = 0;  /* Use lazy initialization to save some time on startup. */
  LJ_LIB_REG(L, LUA_MATHLIBNAME, math);
#if defined(LUA_COMPAT_MOD) && !LJ_52
  lua_getfield(L, -1, "fmod");
  lua_setfield(L, -2, "mod");
#endif
  return 1;
}

```

One Level Up Top Level

src/lj_vmmath.c - luajit-2.0-src

Functions defined

Macros defined

Source code

```
1  /*
2  ** Math helper functions for assembler VM.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
```

```c

#define lj_vmmath_c
#define LUA_CORE

#include <errno.h>
 #include <math.h>

 #include "lj_obj.h"
 #include "lj_ir.h"
 #include "lj_vm.h"

 /* -- Wrapper functions ----------------------------------------------------- */

 #if LJ_TARGET_X86 && __ELF__ && __PIC__
 /* Wrapper functions to deal with the ELF/x86 PIC disaster. */
 LJ_FUNCA double lj_wrap_log(double x) { return log(x); }
 LJ_FUNCA double lj_wrap_log10(double x) { return log10(x); }
 LJ_FUNCA double lj_wrap_exp(double x) { return exp(x); }
 LJ_FUNCA double lj_wrap_sin(double x) { return sin(x); }
 LJ_FUNCA double lj_wrap_cos(double x) { return cos(x); }
 LJ_FUNCA double lj_wrap_tan(double x) { return tan(x); }
 LJ_FUNCA double lj_wrap_asin(double x) { return asin(x); }
 LJ_FUNCA double lj_wrap_acos(double x) { return acos(x); }
 LJ_FUNCA double lj_wrap_atan(double x) { return atan(x); }
 LJ_FUNCA double lj_wrap_sinh(double x) { return sinh(x); }
 LJ_FUNCA double lj_wrap_cosh(double x) { return cosh(x); }
 LJ_FUNCA double lj_wrap_tanh(double x) { return tanh(x); }
 LJ_FUNCA double lj_wrap_atan2(double x, double y) { return atan2(x, y); }
 LJ_FUNCA double lj_wrap_pow(double x, double y) { return pow(x, y); }
 LJ_FUNCA double lj_wrap_fmod(double x, double y) { return fmod(x, y); }
 #endif

 /* -- Helper functions for generated machine code --------------------------- */

 double lj_vm_foldarith(double x, double y, int op)
 {
   switch (op) {
   case IR_ADD - IR_ADD: return x+y; break;
   case IR_SUB - IR_ADD: return x-y; break;
   case IR_MUL - IR_ADD: return x*y; break;
   case IR_DIV - IR_ADD: return x/y; break;
   case IR_MOD - IR_ADD: return x-lj_vm_floor(x/y)*y; break;
   case IR_POW - IR_ADD: return pow(x, y); break;
   case IR_NEG - IR_ADD: return -x; break;
   case IR_ABS - IR_ADD: return fabs(x); break;
 #if LJ_HASJIT
   case IR_ATAN2 - IR_ADD: return atan2(x, y); break;
   case IR_LDEXP - IR_ADD: return ldexp(x, (int)y); break;
   case IR_MIN - IR_ADD: return x > y ? y : x; break;
   case IR_MAX - IR_ADD: return x < y ? y : x; break;
 #endif
   default: return x;
   }
 }

 #if LJ_HASJIT

 #ifdef LUAJIT_NO_LOG2
 double lj_vm_log2(double a)
 {
   return log(a) * 1.4426950408889634074;
 }
 #endif

 #ifdef LUAJIT_NO_EXP2
 double lj_vm_exp2(double a)
 {
   return exp(a * 0.6931471805599453);
 }
 #endif

 #if !(LJ_TARGET_ARM || LJ_TARGET_ARM64 || LJ_TARGET_PPC)
 int32_t LJ_FASTCALL lj_vm_modi(int32_t a, int32_t b)
 {
   uint32_t y, ua, ub;
   lua_assert(b != 0);  /* This must be checked before using this function. */
```

```
  ua = a < 0 ? (uint32_t)-a : (uint32_t)a;
  ub = b < 0 ? (uint32_t)-b : (uint32_t)b;
  y = ua % ub;
  if (y != 0 && (a^b) < 0) y = y - ub;
  if (((int32_t)y^b) < 0) y = (uint32_t)-(int32_t)y;
  return (int32_t)y;
}
#endif

#if !LJ_TARGET_X86ORX64
/* Unsigned x^k. */
static double lj_vm_powui(double x, uint32_t k)
{
  double y;
  lua_assert(k != 0);
  for (; (k & 1) == 0; k >>= 1) x *= x;
  y = x;
  if ((k >>= 1) != 0) {
    for (;;) {
      x *= x;
      if (k == 1) break;
      if (k & 1) y *= x;
      k >>= 1;
    }
    y *= x;
  }
  return y;
}

/* Signed x^k. */
double lj_vm_powi(double x, int32_t k)
{
  if (k > 1)
    return lj_vm_powui(x, (uint32_t)k);
  else if (k == 1)
    return x;
  else if (k == 0)
    return 1.0;
  else
    return 1.0 / lj_vm_powui(x, (uint32_t)-k);
}
#endif

/* Computes fpm(x) for extended math functions. */
double lj_vm_foldfpm(double x, int fpm)
{
  switch (fpm) {
  case IRFPM_FLOOR: return lj_vm_floor(x);
  case IRFPM_CEIL: return lj_vm_ceil(x);
  case IRFPM_TRUNC: return lj_vm_trunc(x);
  case IRFPM_SQRT: return sqrt(x);
  case IRFPM_EXP: return exp(x);
  case IRFPM_EXP2: return lj_vm_exp2(x);
  case IRFPM_LOG: return log(x);
  case IRFPM_LOG2: return lj_vm_log2(x);
  case IRFPM_LOG10: return log10(x);
  case IRFPM_SIN: return sin(x);
  case IRFPM_COS: return cos(x);
  case IRFPM_TAN: return tan(x);
  default: lua_assert(0);
  }
  return 0;
}

#if LJ_HASFFI
int lj_vm_errno(void)
{
  return errno;
}
#endif

#endif
```

One Level Up Top Level

src/lj_vm.h - luajit-2.0-src

Global variables defined

- lj_vm_asm_begin

Data types defined

- lua_CPFunction

Macros defined

- _LJ_VM_H
- lj_vm_ceil
- lj_vm_exp2
- lj_vm_floor
- lj_vm_log2
- lj_vm_powi
- lj_vm_trunc
- makeasmfunc

Source code

```
1  /*
2  ** Assembler VM interface definitions.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_VM_H
7  #define _LJ_VM_H
8
9  #include "lj_obj.h"
10
11 /* Entry points for ASM parts of VM. */
12 LJ_ASMF void lj_vm_call(lua_State *L, TValue *base, int nres1);
13 LJ_ASMF int lj_vm_pcall(lua_State *L, TValue *base, int nres1, ptrdiff_t ef);
14 typedef TValue *(*lua_CPFunction)(lua_State *L, lua_CFunction func, void *ud);
15 LJ_ASMF int lj_vm_cpcall(lua_State *L, lua_CFunction func, void *ud,
16                          lua_CPFunction cp);
17 LJ_ASMF int lj_vm_resume(lua_State *L, TValue *base, int nres1, ptrdiff_t ef);
18 LJ_ASMF_NORET void LJ_FASTCALL lj_vm_unwind_c(void *cframe, int errcode);
19 LJ_ASMF_NORET void LJ_FASTCALL lj_vm_unwind_ff(void *cframe);
20 LJ_ASMF void lj_vm_unwind_c_eh(void);
21 LJ_ASMF void lj_vm_unwind_ff_eh(void);
22 #if LJ_TARGET_X86ORX64
23 LJ_ASMF void lj_vm_unwind_rethrow(void);
24 #endif
25
26 /* Miscellaneous functions. */
27 #if LJ_TARGET_X86ORX64
28 LJ_ASMF int lj_vm_cpuid(uint32_t f, uint32_t res[4]);
29 #endif
30 #if LJ_TARGET_PPC
31 void lj_vm_cachesync(void *start, void *end);
32 #endif
33 LJ_ASMF double lj_vm_foldarith(double x, double y, int op);
34 #if LJ_HASJIT
35 LJ_ASMF double lj_vm_foldfpm(double x, int op);
```

```
#endif
#if !LJ_ARCH_HASFPU
/* Declared in lj_obj.h: LJ_ASMF int32_t lj_vm_tobit(double x); */
#endif

/* Dispatch targets for recording and hooks. */
LJ_ASMF void lj_vm_record(void);
LJ_ASMF void lj_vm_inshook(void);
LJ_ASMF void lj_vm_rethook(void);
LJ_ASMF void lj_vm_callhook(void);
LJ_ASMF void lj_vm_profhook(void);

/* Trace exit handling. */
LJ_ASMF void lj_vm_exit_handler(void);
LJ_ASMF void lj_vm_exit_interp(void);

/* Internal math helper functions. */
#if LJ_TARGET_PPC || LJ_TARGET_ARM64
#define lj_vm_floor		floor
#define lj_vm_ceil		ceil
#else
LJ_ASMF double lj_vm_floor(double);
LJ_ASMF double lj_vm_ceil(double);
#if LJ_TARGET_ARM
LJ_ASMF double lj_vm_floor_sf(double);
LJ_ASMF double lj_vm_ceil_sf(double);
#endif
#endif
#ifdef LUAJIT_NO_LOG2
LJ_ASMF double lj_vm_log2(double);
#else
#define lj_vm_log2		log2
#endif

#if LJ_HASJIT
#if LJ_TARGET_X86ORX64
LJ_ASMF void lj_vm_floor_sse(void);
LJ_ASMF void lj_vm_ceil_sse(void);
LJ_ASMF void lj_vm_trunc_sse(void);
LJ_ASMF void lj_vm_powi_sse(void);
#define lj_vm_powi		NULL
#else
LJ_ASMF double lj_vm_powi(double, int32_t);
#endif
#if LJ_TARGET_PPC || LJ_TARGET_ARM64
#define lj_vm_trunc		trunc
#else
LJ_ASMF double lj_vm_trunc(double);
#if LJ_TARGET_ARM
LJ_ASMF double lj_vm_trunc_sf(double);
#endif
#endif
#ifdef LUAJIT_NO_EXP2
LJ_ASMF double lj_vm_exp2(double);
#else
#define lj_vm_exp2		exp2
#endif
LJ_ASMF int32_t LJ_FASTCALL lj_vm_modi(int32_t, int32_t);
#if LJ_HASFFI
LJ_ASMF int lj_vm_errno(void);
#endif
#endif

/* Continuations for metamethods. */
LJ_ASMF void lj_cont_cat(void);  /* Continue with concatenation. */
LJ_ASMF void lj_cont_ra(void);  /* Store result in RA from instruction. */
LJ_ASMF void lj_cont_nop(void);  /* Do nothing, just continue execution. */
LJ_ASMF void lj_cont_condt(void);  /* Branch if result is true. */
LJ_ASMF void lj_cont_condf(void);  /* Branch if result is false. */
LJ_ASMF void lj_cont_hook(void);  /* Continue from hook yield. */
LJ_ASMF void lj_cont_stitch(void);  /* Trace stitching. */

/* Start of the ASM code. */
LJ_ASMF char lj_vm_asm_begin[];

/* Bytecode offsets are relative to lj_vm_asm_begin. */
```

```
112 #define makeasmfunc(ofs)        ((ASMFunction)(lj_vm_asm_begin + (ofs)))
113
114 #endif
```

One Level Up Top Level

src/lib_io.c - luajit-2.0-src

Data types defined

- IOFileUD
- IOFileUD

Functions defined

- LJLIB_CF(io_method_close)
- LJLIB_CF(io_method_read)
- LJLIB_CF(io_method_write) LJLIB_REC(io_write 0)
- LJLIB_CF(io_method_flush) LJLIB_REC(io_flush 0)
- LJLIB_CF(io_method_seek)
- LJLIB_CF(io_method_setvbuf)
- LJLIB_CF(io_method_lines)
- LJLIB_CF(io_method___gc)
- LJLIB_CF(io_method___tostring)
- LJLIB_CF(io_popen)
- LJLIB_CF(io_tmpfile)
- LJLIB_CF(io_close)
- LJLIB_CF(io_read)
- LJLIB_CF(io_write) LJLIB_REC(io_write GCROOT_IO_OUTPUT)
- LJLIB_CF(io_flush) LJLIB_REC(io_flush GCROOT_IO_OUTPUT)
- LJLIB_CF(io_input)
- LJLIB_CF(io_output)
- LJLIB_CF(io_lines)
- LJLIB_CF(io_type)
- LJLIB_PUSH(top-1) LJLIB_SET(__index)
- io_file_close
- io_file_iter
- io_file_new
- io_file_open
- io_file_read
- io_file_readall

- io_file_readlen
- io_file_readline
- io_file_readnum
- io_file_write
- io_std_getset
- io_std_new
- io_stdfile
- io_tofile
- io_tofilep
- luaopen_io

Macros defined

- IOFILE_FLAG_CLOSE
- IOFILE_TYPE_FILE
- IOFILE_TYPE_MASK
- IOFILE_TYPE_PIPE
- IOFILE_TYPE_STDF
- IOSTDF_IOF
- IOSTDF_UD
- LJLIB_MODULE_io
- LJLIB_MODULE_io_method
- LUA_LIB
- lib_io_c

Source code

```
/*
** I/O library.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Major portions taken verbatim or adapted from the Lua interpreter.
** Copyright (C) 1994-2011 Lua.org, PUC-Rio. See Copyright Notice in lua.h
*/

#include <errno.h>
 #include <stdio.h>

 #define lib_io_c
 #define LUA_LIB

 #include "lua.h"
 #include "lauxlib.h"
 #include "lualib.h"

 #include "lj_obj.h"
 #include "lj_gc.h"
 #include "lj_err.h"
```

```c
#include "lj_buf.h"
#include "lj_str.h"
#include "lj_state.h"
#include "lj_strfmt.h"
#include "lj_ff.h"
#include "lj_lib.h"

/* Userdata payload for I/O file. */
typedef struct IOFileUD {
  FILE *fp;               /* File handle. */
  uint32_t type;        /* File type. */
} IOFileUD;

#define IOFILE_TYPE_FILE        0       /* Regular file. */
#define IOFILE_TYPE_PIPE        1       /* Pipe. */
#define IOFILE_TYPE_STDF        2       /* Standard file handle. */
#define IOFILE_TYPE_MASK        3

#define IOFILE_FLAG_CLOSE        4       /* Close after io.lines() iterator. */

#define IOSTDF_UD(L, id)        (&gcref(G(L)->gcroot[(id)])->ud)
#define IOSTDF_IOF(L, id)       ((IOFileUD *)uddata(IOSTDF_UD(L, (id))))

/* -- Open/close helpers ------------------------------------------------- */

static IOFileUD *io_tofilep(lua_State *L)
{
  if (!(L->base < L->top && tvisudata(L->base) &&
        udataV(L->base)->udtype == UDTYPE_IO_FILE))
    lj_err_argtype(L, 1, "FILE*");
  return (IOFileUD *)uddata(udataV(L->base));
}

static IOFileUD *io_tofile(lua_State *L)
{
  IOFileUD *iof = io_tofilep(L);
  if (iof->fp == NULL)
    lj_err_caller(L, LJ_ERR_IOCLFL);
  return iof;
}

static FILE *io_stdfile(lua_State *L, ptrdiff_t id)
{
  IOFileUD *iof = IOSTDF_IOF(L, id);
  if (iof->fp == NULL)
    lj_err_caller(L, LJ_ERR_IOSTDCL);
  return iof->fp;
}

static IOFileUD *io_file_new(lua_State *L)
{
  IOFileUD *iof = (IOFileUD *)lua_newuserdata(L, sizeof(IOFileUD));
  GCudata *ud = udataV(L->top-1);
  ud->udtype = UDTYPE_IO_FILE;
  /* NOBARRIER: The GCudata is new (marked white). */
  setgcrefr(ud->metatable, curr_func(L)->c.env);
  iof->fp = NULL;
  iof->type = IOFILE_TYPE_FILE;
  return iof;
}

static IOFileUD *io_file_open(lua_State *L, const char *mode)
{
  const char *fname = strdata(lj_lib_checkstr(L, 1));
  IOFileUD *iof = io_file_new(L);
  iof->fp = fopen(fname, mode);
  if (iof->fp == NULL)
    luaL_argerror(L, 1, lj_strfmt_pushf(L, "%s: %s", fname, strerror(errno)));
  return iof;
}

static int io_file_close(lua_State *L, IOFileUD *iof)
{
  int ok;
  if ((iof->type & IOFILE_TYPE_MASK) == IOFILE_TYPE_FILE) {
    ok = (fclose(iof->fp) == 0);
```

```
 } else if ((iof->type & IOFILE_TYPE_MASK) == IOFILE_TYPE_PIPE) {
   int stat = -1;
#if LJ_TARGET_POSIX
    stat = pclose(iof->fp);
#elif LJ_TARGET_WINDOWS
    stat = _pclose(iof->fp);
#else
    lua_assert(0);
    return 0;
#endif
#if LJ_52
    iof->fp = NULL;
    return luaL_execresult(L, stat);
#else
    ok = (stat != -1);
#endif
  } else {
    lua_assert((iof->type & IOFILE_TYPE_MASK) == IOFILE_TYPE_STDF);
    setnilV(L->top++);
    lua_pushliteral(L, "cannot close standard file");
    return 2;
  }
  iof->fp = NULL;
  return luaL_fileresult(L, ok, NULL);
}

/* -- Read/write helpers ---------------------------------------------------- */

static int io_file_readnum(lua_State *L, FILE *fp)
{
  lua_Number d;
  if (fscanf(fp, LUA_NUMBER_SCAN, &d) == 1) {
    if (LJ_DUALNUM) {
      int32_t i = lj_num2int(d);
      if (d == (lua_Number)i && !tvismzero((cTValue *)&d)) {
        setintV(L->top++, i);
        return 1;
      }
    }
    setnumV(L->top++, d);
    return 1;
  } else {
    setnilV(L->top++);
    return 0;
  }
}

static int io_file_readline(lua_State *L, FILE *fp, MSize chop)
{
  MSize m = LUAL_BUFFERSIZE, n = 0, ok = 0;
  char *buf;
  for (;;) {
    buf = lj_buf_tmp(L, m);
    if (fgets(buf+n, m-n, fp) == NULL) break;
    n += (MSize)strlen(buf+n);
    ok |= n;
    if (n && buf[n-1] == '\n') { n -= chop; break; }
    if (n >= m - 64) m += m;
  }
  setstrV(L, L->top++, lj_str_new(L, buf, (size_t)n));
  lj_gc_check(L);
  return (int)ok;
}

static void io_file_readall(lua_State *L, FILE *fp)
{
  MSize m, n;
  for (m = LUAL_BUFFERSIZE, n = 0; ; m += m) {
    char *buf = lj_buf_tmp(L, m);
    n += (MSize)fread(buf+n, 1, m-n, fp);
    if (n != m) {
      setstrV(L, L->top++, lj_str_new(L, buf, (size_t)n));
      lj_gc_check(L);
      return;
    }
  }
```

```
}

static int io_file_readlen(lua_State *L, FILE *fp, MSize m)
{
  if (m) {
    char *buf = lj_buf_tmp(L, m);
    MSize n = (MSize)fread(buf, 1, m, fp);
    setstrV(L, L->top++, lj_str_new(L, buf, (size_t)n));
    lj_gc_check(L);
    return (n > 0 || m == 0);
  } else {
    int c = getc(fp);
    ungetc(c, fp);
    setstrV(L, L->top++, &G(L)->strempty);
    return (c != EOF);
  }
}

static int io_file_read(lua_State *L, FILE *fp, int start)
{
  int ok, n, nargs = (int)(L->top - L->base) - start;
  clearerr(fp);
  if (nargs == 0) {
    ok = io_file_readline(L, fp, 1);
    n = start+1;  /* Return 1 result. */
  } else {
    /* The results plus the buffers go on top of the args. */
    luaL_checkstack(L, nargs+LUA_MINSTACK, "too many arguments");
    ok = 1;
    for (n = start; nargs-- && ok; n++) {
      if (tvisstr(L->base+n)) {
        const char *p = strVdata(L->base+n);
        if (p[0] != '*')
          lj_err_arg(L, n+1, LJ_ERR_INVOPT);
        if (p[1] == 'n')
          ok = io_file_readnum(L, fp);
        else if ((p[1] & ~0x20) == 'L')
          ok = io_file_readline(L, fp, (p[1] == 'l'));
        else if (p[1] == 'a')
          io_file_readall(L, fp);
        else
          lj_err_arg(L, n+1, LJ_ERR_INVFMT);
      } else if (tvisnumber(L->base+n)) {
        ok = io_file_readlen(L, fp, (MSize)lj_lib_checkint(L, n+1));
      } else {
        lj_err_arg(L, n+1, LJ_ERR_INVOPT);
      }
    }
  }
  if (ferror(fp))
    return luaL_fileresult(L, 0, NULL);
  if (!ok)
    setnilV(L->top-1);  /* Replace last result with nil. */
  return n - start;
}

static int io_file_write(lua_State *L, FILE *fp, int start)
{
  cTValue *tv;
  int status = 1;
  for (tv = L->base+start; tv < L->top; tv++) {
    char buf[STRFMT_MAXBUF_NUM];
    MSize len;
    const char *p = lj_strfmt_wstrnum(buf, tv, &len);
    if (!p)
      lj_err_argt(L, (int)(tv - L->base) + 1, LUA_TSTRING);
    status = status && (fwrite(p, 1, len, fp) == len);
  }
  if (LJ_52 && status) {
    L->top = L->base+1;
    if (start == 0)
      setudataV(L, L->base, IOSTDF_UD(L, GCROOT_IO_OUTPUT));
    return 1;
  }
  return luaL_fileresult(L, status, NULL);
}
```

```
250
251 static int io_file_iter(lua_State *L)
252 {
253   GCfunc *fn = curr_func(L);
254   IOFileUD *iof = uddata(udataV(&fn->c.upvalue[0]));
255   int n = fn->c.nupvalues - 1;
256   if (iof->fp == NULL)
257     lj_err_caller(L, LJ_ERR_IOCLFL);
258   L->top = L->base;
259   if (n) {  /* Copy upvalues with options to stack. */
260     if (n > LUAI_MAXCSTACK)
261       lj_err_caller(L, LJ_ERR_STKOV);
262     lj_state_checkstack(L, (MSize)n);
263     memcpy(L->top, &fn->c.upvalue[1], n*sizeof(TValue));
264     L->top += n;
265   }
266   n = io_file_read(L, iof->fp, 0);
267   if (ferror(iof->fp))
268     lj_err_callermsg(L, strVdata(L->top-2));
269   if (tvisnil(L->base) && (iof->type & IOFILE_FLAG_CLOSE)) {
270     io_file_close(L, iof);  /* Return values are ignored. */
271     return 0;
272   }
273   return n;
274 }
275
276 /* -- I/O file methods ----------------------------------------------------- */
277
278 #define LJLIB_MODULE_io_method
279
280 LJLIB_CF(io_method_close)
281 {
282   IOFileUD *iof = L->base < L->top ? io_tofile(L) :
283                   IOSTDF_IOF(L, GCROOT_IO_OUTPUT);
284   return io_file_close(L, iof);
285 }
286
287 LJLIB_CF(io_method_read)
288 {
289   return io_file_read(L, io_tofile(L)->fp, 1);
290 }
291
292 LJLIB_CF(io_method_write)               LJLIB_REC(io_write 0)
293 {
294   return io_file_write(L, io_tofile(L)->fp, 1);
295 }
296
297 LJLIB_CF(io_method_flush)               LJLIB_REC(io_flush 0)
298 {
299   return luaL_fileresult(L, fflush(io_tofile(L)->fp) == 0, NULL);
300 }
301
302 LJLIB_CF(io_method_seek)
303 {
304   FILE *fp = io_tofile(L)->fp;
305   int opt = lj_lib_checkopt(L, 2, 1, "\3set\3cur\3end");
306   int64_t ofs = 0;
307   cTValue *o;
308   int res;
309   if (opt == 0) opt = SEEK_SET;
310   else if (opt == 1) opt = SEEK_CUR;
311   else if (opt == 2) opt = SEEK_END;
312   o = L->base+2;
313   if (o < L->top) {
314     if (tvisint(o))
315       ofs = (int64_t)intV(o);
316     else if (tvisnum(o))
317       ofs = (int64_t)numV(o);
318     else if (!tvisnil(o))
319       lj_err_argt(L, 3, LUA_TNUMBER);
320   }
321 #if LJ_TARGET_POSIX
322   res = fseeko(fp, ofs, opt);
323 #elif _MSC_VER >= 1400
324   res = _fseeki64(fp, ofs, opt);
325 #elif defined(__MINGW32__)
```

```
  res = fseeko64(fp, ofs, opt);
#else
  res = fseek(fp, (long)ofs, opt);
#endif
  if (res)
    return luaL_fileresult(L, 0, NULL);
#if LJ_TARGET_POSIX
  ofs = ftello(fp);
#elif _MSC_VER >= 1400
  ofs = _ftelli64(fp);
#elif defined(__MINGW32__)
  ofs = ftello64(fp);
#else
  ofs = (int64_t)ftell(fp);
#endif
  setint64V(L->top-1, ofs);
  return 1;
}

LJLIB_CF(io_method_setvbuf)
{
  FILE *fp = io_tofile(L)->fp;
  int opt = lj_lib_checkopt(L, 2, -1, "\4full\4line\2no");
  size_t sz = (size_t)lj_lib_optint(L, 3, LUAL_BUFFERSIZE);
  if (opt == 0) opt = _IOFBF;
  else if (opt == 1) opt = _IOLBF;
  else if (opt == 2) opt = _IONBF;
  return luaL_fileresult(L, setvbuf(fp, NULL, opt, sz) == 0, NULL);
}

LJLIB_CF(io_method_lines)
{
  io_tofile(L);
  lua_pushcclosure(L, io_file_iter, (int)(L->top - L->base));
  return 1;
}

LJLIB_CF(io_method___gc)
{
  IOFileUD *iof = io_tofilep(L);
  if (iof->fp != NULL && (iof->type & IOFILE_TYPE_MASK) != IOFILE_TYPE_STDF)
    io_file_close(L, iof);
  return 0;
}

LJLIB_CF(io_method___tostring)
{
  IOFileUD *iof = io_tofilep(L);
  if (iof->fp != NULL)
    lua_pushfstring(L, "file (%p)", iof->fp);
  else
    lua_pushliteral(L, "file (closed)");
  return 1;
}

LJLIB_PUSH(top-1) LJLIB_SET(__index)

#include "lj_libdef.h"

/* -- I/O library functions ----------------------------------------------- */

#define LJLIB_MODULE_io

LJLIB_PUSH(top-2) LJLIB_SET(!)  /* Set environment. */

LJLIB_CF(io_open)
{
  const char *fname = strdata(lj_lib_checkstr(L, 1));
  GCstr *s = lj_lib_optstr(L, 2);
  const char *mode = s ? strdata(s) : "r";
  IOFileUD *iof = io_file_new(L);
  iof->fp = fopen(fname, mode);
  return iof->fp != NULL ? 1 : luaL_fileresult(L, 0, fname);
}

LJLIB_CF(io_popen)
```

```
{
#if LJ_TARGET_POSIX || LJ_TARGET_WINDOWS
  const char *fname = strdata(lj_lib_checkstr(L, 1));
  GCstr *s = lj_lib_optstr(L, 2);
  const char *mode = s ? strdata(s) : "r";
  IOFileUD *iof = io_file_new(L);
  iof->type = IOFILE_TYPE_PIPE;
#if LJ_TARGET_POSIX
  fflush(NULL);
  iof->fp = popen(fname, mode);
#else
  iof->fp = _popen(fname, mode);
#endif
  return iof->fp != NULL ? 1 : luaL_fileresult(L, 0, fname);
#else
  return luaL_error(L, LUA_QL("popen") " not supported");
#endif
}

LJLIB_CF(io_tmpfile)
{
  IOFileUD *iof = io_file_new(L);
#if LJ_TARGET_PS3 || LJ_TARGET_PS4 || LJ_TARGET_PSVITA
  iof->fp = NULL; errno = ENOSYS;
#else
  iof->fp = tmpfile();
#endif
  return iof->fp != NULL ? 1 : luaL_fileresult(L, 0, NULL);
}

LJLIB_CF(io_close)
{
  return lj_cf_io_method_close(L);
}

LJLIB_CF(io_read)
{
  return io_file_read(L, io_stdfile(L, GCROOT_IO_INPUT), 0);
}

LJLIB_CF(io_write)               LJLIB_REC(io_write GCROOT_IO_OUTPUT)
{
  return io_file_write(L, io_stdfile(L, GCROOT_IO_OUTPUT), 0);
}

LJLIB_CF(io_flush)               LJLIB_REC(io_flush GCROOT_IO_OUTPUT)
{
  return luaL_fileresult(L, fflush(io_stdfile(L, GCROOT_IO_OUTPUT)) == 0, NULL);
}

static int io_std_getset(lua_State *L, ptrdiff_t id, const char *mode)
{
  if (L->base < L->top && !tvisnil(L->base)) {
    if (tvisudata(L->base)) {
      io_tofile(L);
      L->top = L->base+1;
    } else {
      io_file_open(L, mode);
    }
    /* NOBARRIER: The standard I/O handles are GC roots. */
    setgcref(G(L)->gcroot[id], gcV(L->top-1));
  } else {
    setudataV(L, L->top++, IOSTDF_UD(L, id));
  }
  return 1;
}

LJLIB_CF(io_input)
{
  return io_std_getset(L, GCROOT_IO_INPUT, "r");
}

LJLIB_CF(io_output)
{
  return io_std_getset(L, GCROOT_IO_OUTPUT, "w");
}
```

```
478
479 LJLIB_CF(io_lines)
480 {
481   if (L->base == L->top) setnilV(L->top++);
482   if (!tvisnil(L->base)) {  /* io.lines(fname) */
483     IOFileUD *iof = io_file_open(L, "r");
484     iof->type = IOFILE_TYPE_FILE|IOFILE_FLAG_CLOSE;
485     L->top--;
486     setudataV(L, L->base, udataV(L->top));
487   } else {  /* io.lines() iterates over stdin. */
488     setudataV(L, L->base, IOSTDF_UD(L, GCROOT_IO_INPUT));
489   }
490   lua_pushcclosure(L, io_file_iter, (int)(L->top - L->base));
491   return 1;
492 }
493
494 LJLIB_CF(io_type)
495 {
496   cTValue *o = lj_lib_checkany(L, 1);
497   if (!(tvisudata(o) && udataV(o)->udtype == UDTYPE_IO_FILE))
498     setnilV(L->top++);
499   else if (((IOFileUD *)uddata(udataV(o)))->fp != NULL)
500     lua_pushliteral(L, "file");
501   else
502     lua_pushliteral(L, "closed file");
503   return 1;
504 }
505
506 #include "lj_libdef.h"
507
508 /* ------------------------------------------------------------------------ */
509
510 static GCobj *io_std_new(lua_State *L, FILE *fp, const char *name)
511 {
512   IOFileUD *iof = (IOFileUD *)lua_newuserdata(L, sizeof(IOFileUD));
513   GCudata *ud = udataV(L->top-1);
514   ud->udtype = UDTYPE_IO_FILE;
515   /* NOBARRIER: The GCudata is new (marked white). */
516   setgcref(ud->metatable, gcV(L->top-3));
517   iof->fp = fp;
518   iof->type = IOFILE_TYPE_STDF;
519   lua_setfield(L, -2, name);
520   return obj2gco(ud);
521 }
522
523 LUALIB_API int luaopen_io(lua_State *L)
524 {
525   LJ_LIB_REG(L, NULL, io_method);
526   copyTV(L, L->top, L->top-1); L->top++;
527   lua_setfield(L, LUA_REGISTRYINDEX, LUA_FILEHANDLE);
528   LJ_LIB_REG(L, LUA_IOLIBNAME, io);
529   setgcref(G(L)->gcroot[GCROOT_IO_INPUT], io_std_new(L, stdin, "stdin"));
530   setgcref(G(L)->gcroot[GCROOT_IO_OUTPUT], io_std_new(L, stdout, "stdout"));
531   io_std_new(L, stderr, "stderr");
532   return 1;
533 }
534
```

One Level Up Top Level

src/lj_state.h - luajit-2.0-src

Functions defined

- lj_state_checkstack

Macros defined

- _LJ_STATE_H
- incr_top
- restorestack
- savestack

Source code

```
1  /*
2  ** State and stack handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_STATE_H
7  #define _LJ_STATE_H
8
9  #include "lj_obj.h"
10
11 #define incr_top(L) \
12   (++L->top >= tvref(L->maxstack) && (lj_state_growstack1(L), 0))
13
14 #define savestack(L, p)               ((char *)(p) - mref(L->stack, char))
15 #define restorestack(L, n)        ((TValue *)(mref(L->stack, char) + (n)))
16
17 LJ_FUNC void lj_state_relimitstack(lua_State *L);
18 LJ_FUNC void lj_state_shrinkstack(lua_State *L, MSize used);
19 LJ_FUNCA void LJ_FASTCALL lj_state_growstack(lua_State *L, MSize need);
20 LJ_FUNC void LJ_FASTCALL lj_state_growstack1(lua_State *L);
21
22 static LJ_AINLINE void lj_state_checkstack(lua_State *L, MSize need)
23 {
24   if ((mref(L->maxstack, char) - (char *)L->top) <=
25       (ptrdiff_t)need*(ptrdiff_t)sizeof(TValue))
26     lj_state_growstack(L, need);
27 }
28
29 LJ_FUNC lua_State *lj_state_new(lua_State *L);
30 LJ_FUNC void LJ_FASTCALL lj_state_free(global_State *g, lua_State *L);
31 #if LJ_64
32 LJ_FUNC lua_State *lj_state_newstate(lua_Alloc f, void *ud);
33 #endif
34
35 #endif
```

src/lualib.h - luajit-2.0-src

Macros defined

- LUA_BITLIBNAME
- LUA_COLIBNAME
- LUA_DBLIBNAME
- LUA_FFILIBNAME
- LUA_FILEHANDLE
- LUA_IOLIBNAME
- LUA_JITLIBNAME
- LUA_LOADLIBNAME
- LUA_MATHLIBNAME
- LUA_OSLIBNAME
- LUA_STRLIBNAME
- LUA_TABLIBNAME
- _LUALIB_H
- lua_assert

Source code

```
1  /*
2  ** Standard library header.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LUALIB_H
7  #define _LUALIB_H
8
9  #include "lua.h"
10
11  #define LUA_FILEHANDLE        "FILE*"
12
13  #define LUA_COLIBNAME        "coroutine"
14  #define LUA_MATHLIBNAME        "math"
15  #define LUA_STRLIBNAME        "string"
16  #define LUA_TABLIBNAME        "table"
17  #define LUA_IOLIBNAME        "io"
18  #define LUA_OSLIBNAME        "os"
19  #define LUA_LOADLIBNAME        "package"
20  #define LUA_DBLIBNAME        "debug"
21  #define LUA_BITLIBNAME        "bit"
22  #define LUA_JITLIBNAME        "jit"
23  #define LUA_FFILIBNAME        "ffi"
24
25  LUALIB_API int luaopen_base(lua_State *L);
26  LUALIB_API int luaopen_math(lua_State *L);
27  LUALIB_API int luaopen_string(lua_State *L);
28  LUALIB_API int luaopen_table(lua_State *L);
29  LUALIB_API int luaopen_io(lua_State *L);
30  LUALIB_API int luaopen_os(lua_State *L);
31  LUALIB_API int luaopen_package(lua_State *L);
32  LUALIB_API int luaopen_debug(lua_State *L);
33  LUALIB_API int luaopen_bit(lua_State *L);
```

```
34 LUALIB_API int luaopen_jit(lua_State *L);
35 LUALIB_API int luaopen_ffi(lua_State *L);
36
37 LUALIB_API void luaL_openlibs(lua_State *L);
38
39 #ifndef lua_assert
40 #define lua_assert(x)        ((void)0)
41 #endif
42
43 #endif
```

One Level Up Top Level

src/lib_package.c - luajit-2.0-src

Global variables defined

- package_global
- package_lib
- package_loaders
- sentinel_

Functions defined

- dooptions
- findfile
- lj_cf_package_loader_c
- lj_cf_package_loader_croot
- lj_cf_package_loader_lua
- lj_cf_package_loader_preload
- lj_cf_package_loadlib
- lj_cf_package_module
- lj_cf_package_require
- lj_cf_package_searchpath
- lj_cf_package_seeall
- lj_cf_package_unloadlib
- ll_bcsym
- ll_bcsym
- ll_bcsym
- ll_load
- ll_load
- ll_load
- ll_loadfunc
- ll_register
- ll_sym
- ll_sym
- ll_sym
- ll_unloadlib

- ll_unloadlib
- ll_unloadlib
- loaderror
- luaopen_package
- mksymname
- modinit
- pusherror
- pushnexttemplate
- readable
- searchpath
- setfenv
- setpath
- setprogdir

Macros defined

- AUXMARK
- DLMSG
- GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS
- GET_MODULE_HANDLE_EX_FLAG_UNCHANGED_REFCOUNT
- LUA_LIB
- PACKAGE_ERR_FUNC
- PACKAGE_ERR_LIB
- PACKAGE_ERR_LOAD
- PACKAGE_LIB_FAIL
- PACKAGE_LIB_FAIL
- PACKAGE_LIB_FAIL
- SYMPREFIX_BC
- SYMPREFIX_CF
- WIN32_LEAN_AND_MEAN
- lib_package_c
- sentinel
- setprogdir
- setprogdir

Source code

```
/*
** Package library.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Major portions taken verbatim or adapted from the Lua interpreter.
** Copyright (C) 1994-2012 Lua.org, PUC-Rio. See Copyright Notice in lua.h
*/

#define lib_package_c
 #define LUA_LIB

 #include "lua.h"
 #include "lauxlib.h"
 #include "lualib.h"

 #include "lj_obj.h"
 #include "lj_err.h"
 #include "lj_lib.h"

 /* ------------------------------------------------------------------------ */

 /* Error codes for ll_loadfunc. */
 #define PACKAGE_ERR_LIB                1
 #define PACKAGE_ERR_FUNC        2
 #define PACKAGE_ERR_LOAD        3

 /* Redefined in platform specific part. */
 #define PACKAGE_LIB_FAIL        "open"
 #define setprogdir(L)                ((void)0)

 /* Symbol name prefixes. */
 #define SYMPREFIX_CF                "luaopen_%s"
 #define SYMPREFIX_BC                "luaJIT_BC_%s"

 #if LJ_TARGET_DLOPEN

 #include <dlfcn.h>

 static void ll_unloadlib(void *lib)
 {
   dlclose(lib);
 }

 static void *ll_load(lua_State *L, const char *path, int gl)
 {
   void *lib = dlopen(path, RTLD_NOW | (gl ? RTLD_GLOBAL : RTLD_LOCAL));
   if (lib == NULL) lua_pushstring(L, dlerror());
   return lib;
 }

 static lua_CFunction ll_sym(lua_State *L, void *lib, const char *sym)
 {
   lua_CFunction f = (lua_CFunction)dlsym(lib, sym);
   if (f == NULL) lua_pushstring(L, dlerror());
   return f;
 }

 static const char *ll_bcsym(void *lib, const char *sym)
 {
 #if defined(RTLD_DEFAULT)
   if (lib == NULL) lib = RTLD_DEFAULT;
 #elif LJ_TARGET_OSX || LJ_TARGET_BSD
   if (lib == NULL) lib = (void *)(intptr_t)-2;
 #endif
   return (const char *)dlsym(lib, sym);
 }

 #elif LJ_TARGET_WINDOWS

 #define WIN32_LEAN_AND_MEAN
 #include <windows.h>

 #ifndef GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS
```

```
#define GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS  4
#define GET_MODULE_HANDLE_EX_FLAG_UNCHANGED_REFCOUNT  2
BOOL WINAPI GetModuleHandleExA(DWORD, LPCSTR, HMODULE*);
#endif

#undef setprogdir

static void setprogdir(lua_State *L)
{
  char buff[MAX_PATH + 1];
  char *lb;
  DWORD nsize = sizeof(buff);
  DWORD n = GetModuleFileNameA(NULL, buff, nsize);
  if (n == 0 || n == nsize || (lb = strrchr(buff, '\\')) == NULL) {
    luaL_error(L, "unable to get ModuleFileName");
  } else {
    *lb = '\0';
    luaL_gsub(L, lua_tostring(L, -1), LUA_EXECDIR, buff);
    lua_remove(L, -2);  /* remove original string */
  }
}

static void pusherror(lua_State *L)
{
  DWORD error = GetLastError();
  char buffer[128];
   if (FormatMessageA(FORMAT_MESSAGE_IGNORE_INSERTS | FORMAT_MESSAGE_FROM_SYSTEM,
       NULL, error, 0, buffer, sizeof(buffer), NULL))
     lua_pushstring(L, buffer);
   else
     lua_pushfstring(L, "system error %d\n", error);
 }

 static void ll_unloadlib(void *lib)
 {
   FreeLibrary((HINSTANCE)lib);
 }

 static void *ll_load(lua_State *L, const char *path, int gl)
 {
   HINSTANCE lib = LoadLibraryA(path);
   if (lib == NULL) pusherror(L);
   UNUSED(gl);
   return lib;
 }

 static lua_CFunction ll_sym(lua_State *L, void *lib, const char *sym)
 {
   lua_CFunction f = (lua_CFunction)GetProcAddress((HINSTANCE)lib, sym);
   if (f == NULL) pusherror(L);
   return f;
 }

 static const char *ll_bcsym(void *lib, const char *sym)
 {
   if (lib) {
     return (const char *)GetProcAddress((HINSTANCE)lib, sym);
   } else {
     HINSTANCE h = GetModuleHandleA(NULL);
     const char *p = (const char *)GetProcAddress(h, sym);
     if (p == NULL &&
GetModuleHandleExA(GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS|GET_MODULE_HANDLE_EX_FLAG_UNCHANGED_REFCOUNT,
                                        (const char *)ll_bcsym, &h))
       p = (const char *)GetProcAddress(h, sym);
     return p;
   }
 }

 #else

 #undef PACKAGE_LIB_FAIL
 #define PACKAGE_LIB_FAIL        "absent"

 #define DLMSG        "dynamic libraries not enabled; no support for target OS"

 static void ll_unloadlib(void *lib)
```

```c
{
  UNUSED(lib);
}

static void *ll_load(lua_State *L, const char *path, int gl)
{
  UNUSED(path); UNUSED(gl);
  lua_pushliteral(L, DLMSG);
  return NULL;
}

static lua_CFunction ll_sym(lua_State *L, void *lib, const char *sym)
{
  UNUSED(lib); UNUSED(sym);
  lua_pushliteral(L, DLMSG);
  return NULL;
}

static const char *ll_bcsym(void *lib, const char *sym)
{
  UNUSED(lib); UNUSED(sym);
  return NULL;
}

#endif

/* ------------------------------------------------------------------------ */

static void **ll_register(lua_State *L, const char *path)
{
  void **plib;
  lua_pushfstring(L, "LOADLIB: %s", path);
  lua_gettable(L, LUA_REGISTRYINDEX);  /* check library in registry? */
  if (!lua_isnil(L, -1)) {  /* is there an entry? */
    plib = (void **)lua_touserdata(L, -1);
  } else {  /* no entry yet; create one */
    lua_pop(L, 1);
    plib = (void **)lua_newuserdata(L, sizeof(void *));
    *plib = NULL;
    luaL_getmetatable(L, "_LOADLIB");
    lua_setmetatable(L, -2);
    lua_pushfstring(L, "LOADLIB: %s", path);
    lua_pushvalue(L, -2);
    lua_settable(L, LUA_REGISTRYINDEX);
  }
  return plib;
}

static const char *mksymname(lua_State *L, const char *modname,
                             const char *prefix)
{
  const char *funcname;
  const char *mark = strchr(modname, *LUA_IGMARK);
  if (mark) modname = mark + 1;
  funcname = luaL_gsub(L, modname, ".", "_");
  funcname = lua_pushfstring(L, prefix, funcname);
  lua_remove(L, -2);  /* remove 'gsub' result */
  return funcname;
}

static int ll_loadfunc(lua_State *L, const char *path, const char *name, int r)
{
  void **reg = ll_register(L, path);
  if (*reg == NULL) *reg = ll_load(L, path, (*name == '*'));
  if (*reg == NULL) {
    return PACKAGE_ERR_LIB;  /* Unable to load library. */
  } else if (*name == '*') {  /* Only load library into global namespace. */
    lua_pushboolean(L, 1);
    return 0;
  } else {
    const char *sym = r ? name : mksymname(L, name, SYMPREFIX_CF);
    lua_CFunction f = ll_sym(L, *reg, sym);
    if (f) {
      lua_pushcfunction(L, f);
      return 0;
    }
```

```
    if (!r) {
      const char *bcdata = ll_bcsym(*reg, mksymname(L, name, SYMPREFIX_BC));
      lua_pop(L, 1);
      if (bcdata) {
        if (luaL_loadbuffer(L, bcdata, LJ_MAX_BUF, name) != 0)
          return PACKAGE_ERR_LOAD;
        return 0;
      }
    }
    return PACKAGE_ERR_FUNC;  /* Unable to find function. */
  }
}

static int lj_cf_package_loadlib(lua_State *L)
{
  const char *path = luaL_checkstring(L, 1);
  const char *init = luaL_checkstring(L, 2);
  int st = ll_loadfunc(L, path, init, 1);
  if (st == 0) {  /* no errors? */
    return 1;  /* return the loaded function */
  } else {  /* error; error message is on stack top */
    lua_pushnil(L);
    lua_insert(L, -2);
    lua_pushstring(L, (st == PACKAGE_ERR_LIB) ?  PACKAGE_LIB_FAIL : "init");
    return 3;  /* return nil, error message, and where */
  }
}

static int lj_cf_package_unloadlib(lua_State *L)
{
  void **lib = (void **)luaL_checkudata(L, 1, "_LOADLIB");
  if (*lib) ll_unloadlib(*lib);
  *lib = NULL;  /* mark library as closed */
  return 0;
}

/* ------------------------------------------------------------------------ */

static int readable(const char *filename)
{
  FILE *f = fopen(filename, "r");  /* try to open file */
  if (f == NULL) return 0;  /* open failed */
  fclose(f);
  return 1;
}

static const char *pushnexttemplate(lua_State *L, const char *path)
{
  const char *l;
  while (*path == *LUA_PATHSEP) path++;  /* skip separators */
  if (*path == '\0') return NULL;  /* no more templates */
  l = strchr(path, *LUA_PATHSEP);  /* find next separator */
  if (l == NULL) l = path + strlen(path);
  lua_pushlstring(L, path, (size_t)(l - path));  /* template */
  return l;
}

static const char *searchpath (lua_State *L, const char *name,
                               const char *path, const char *sep,
                               const char *dirsep)
{
  luaL_Buffer msg;  /* to build error message */
  luaL_buffinit(L, &msg);
  if (*sep != '\0')  /* non-empty separator? */
    name = luaL_gsub(L, name, sep, dirsep);  /* replace it by 'dirsep' */
  while ((path = pushnexttemplate(L, path)) != NULL) {
    const char *filename = luaL_gsub(L, lua_tostring(L, -1),
                                     LUA_PATH_MARK, name);
    lua_remove(L, -2);  /* remove path template */
    if (readable(filename))  /* does file exist and is readable? */
      return filename;  /* return that file name */
    lua_pushfstring(L, "\n\tno file " LUA_QS, filename);
    lua_remove(L, -2);  /* remove file name */
    luaL_addvalue(&msg);  /* concatenate error msg. entry */
  }
  luaL_pushresult(&msg);  /* create error message */
```

```
  return NULL;  /* not found */
}

static int lj_cf_package_searchpath(lua_State *L)
{
  const char *f = searchpath(L, luaL_checkstring(L, 1),
                                luaL_checkstring(L, 2),
                                luaL_optstring(L, 3, "."),
                                luaL_optstring(L, 4, LUA_DIRSEP));
  if (f != NULL) {
    return 1;
  } else {  /* error message is on top of the stack */
    lua_pushnil(L);
    lua_insert(L, -2);
    return 2;  /* return nil + error message */
  }
}

static const char *findfile(lua_State *L, const char *name,
                            const char *pname)
{
  const char *path;
  lua_getfield(L, LUA_ENVIRONINDEX, pname);
  path = lua_tostring(L, -1);
  if (path == NULL)
    luaL_error(L, LUA_QL("package.%s") " must be a string", pname);
  return searchpath(L, name, path, ".", LUA_DIRSEP);
}

static void loaderror(lua_State *L, const char *filename)
{
  luaL_error(L, "error loading module " LUA_QS " from file " LUA_QS ":\n\t%s",
             lua_tostring(L, 1), filename, lua_tostring(L, -1));
}

static int lj_cf_package_loader_lua(lua_State *L)
{
  const char *filename;
  const char *name = luaL_checkstring(L, 1);
  filename = findfile(L, name, "path");
  if (filename == NULL) return 1;  /* library not found in this path */
  if (luaL_loadfile(L, filename) != 0)
    loaderror(L, filename);
  return 1;  /* library loaded successfully */
}

static int lj_cf_package_loader_c(lua_State *L)
{
  const char *name = luaL_checkstring(L, 1);
  const char *filename = findfile(L, name, "cpath");
  if (filename == NULL) return 1;  /* library not found in this path */
  if (ll_loadfunc(L, filename, name, 0) != 0)
    loaderror(L, filename);
  return 1;  /* library loaded successfully */
}

static int lj_cf_package_loader_croot(lua_State *L)
{
  const char *filename;
  const char *name = luaL_checkstring(L, 1);
  const char *p = strchr(name, '.');
  int st;
  if (p == NULL) return 0;  /* is root */
  lua_pushlstring(L, name, (size_t)(p - name));
  filename = findfile(L, lua_tostring(L, -1), "cpath");
  if (filename == NULL) return 1;  /* root not found */
  if ((st = ll_loadfunc(L, filename, name, 0)) != 0) {
    if (st != PACKAGE_ERR_FUNC) loaderror(L, filename);  /* real error */
    lua_pushfstring(L, "\n\tno module " LUA_QS " in file " LUA_QS,
                    name, filename);
    return 1;  /* function not found */
  }
  return 1;
}

static int lj_cf_package_loader_preload(lua_State *L)
```

```
{
  const char *name = luaL_checkstring(L, 1);
  lua_getfield(L, LUA_ENVIRONINDEX, "preload");
  if (!lua_istable(L, -1))
    luaL_error(L, LUA_QL("package.preload") " must be a table");
  lua_getfield(L, -1, name);
  if (lua_isnil(L, -1)) {  /* Not found? */
    const char *bcname = mksymname(L, name, SYMPREFIX_BC);
    const char *bcdata = ll_bcsym(NULL, bcname);
    if (bcdata == NULL || luaL_loadbuffer(L, bcdata, LJ_MAX_BUF, name) != 0)
      lua_pushfstring(L, "\n\tno field package.preload['%s']", name);
  }
  return 1;
}

/* ------------------------------------------------------------------------ */

static const int sentinel_ = 0;
#define sentinel        ((void *)&sentinel_)

static int lj_cf_package_require(lua_State *L)
{
  const char *name = luaL_checkstring(L, 1);
  int i;
  lua_settop(L, 1);  /* _LOADED table will be at index 2 */
  lua_getfield(L, LUA_REGISTRYINDEX, "_LOADED");
  lua_getfield(L, 2, name);
  if (lua_toboolean(L, -1)) {  /* is it there? */
    if (lua_touserdata(L, -1) == sentinel)  /* check loops */
      luaL_error(L, "loop or previous error loading module " LUA_QS, name);
    return 1;  /* package is already loaded */
  }
  /* else must load it; iterate over available loaders */
  lua_getfield(L, LUA_ENVIRONINDEX, "loaders");
  if (!lua_istable(L, -1))
    luaL_error(L, LUA_QL("package.loaders") " must be a table");
  lua_pushliteral(L, "");  /* error message accumulator */
  for (i = 1; ; i++) {
    lua_rawgeti(L, -2, i);  /* get a loader */
    if (lua_isnil(L, -1))
      luaL_error(L, "module " LUA_QS " not found:%s",
                 name, lua_tostring(L, -2));
    lua_pushstring(L, name);
    lua_call(L, 1, 1);  /* call it */
    if (lua_isfunction(L, -1))  /* did it find module? */
      break;  /* module loaded successfully */
    else if (lua_isstring(L, -1))  /* loader returned error message? */
      lua_concat(L, 2);  /* accumulate it */
    else
      lua_pop(L, 1);
  }
  lua_pushlightuserdata(L, sentinel);
  lua_setfield(L, 2, name);  /* _LOADED[name] = sentinel */
  lua_pushstring(L, name);  /* pass name as argument to module */
  lua_call(L, 1, 1);  /* run loaded module */
  if (!lua_isnil(L, -1))  /* non-nil return? */
    lua_setfield(L, 2, name);  /* _LOADED[name] = returned value */
  lua_getfield(L, 2, name);
  if (lua_touserdata(L, -1) == sentinel) {   /* module did not set a value? */
    lua_pushboolean(L, 1);  /* use true as result */
    lua_pushvalue(L, -1);  /* extra copy to be returned */
    lua_setfield(L, 2, name);  /* _LOADED[name] = true */
  }
  lj_lib_checkfpu(L);
  return 1;
}

/* ------------------------------------------------------------------------ */

static void setfenv(lua_State *L)
{
  lua_Debug ar;
  if (lua_getstack(L, 1, &ar) == 0 ||
      lua_getinfo(L, "f", &ar) == 0 ||  /* get calling function */
      lua_iscfunction(L, -1))
    luaL_error(L, LUA_QL("module") " not called from a Lua function");
```

```
  lua_pushvalue(L, -2);
  lua_setfenv(L, -2);
  lua_pop(L, 1);
}

static void dooptions(lua_State *L, int n)
{
  int i;
  for (i = 2; i <= n; i++) {
    lua_pushvalue(L, i);  /* get option (a function) */
    lua_pushvalue(L, -2);  /* module */
    lua_call(L, 1, 0);
  }
}

static void modinit(lua_State *L, const char *modname)
{
  const char *dot;
  lua_pushvalue(L, -1);
  lua_setfield(L, -2, "_M");  /* module._M = module */
  lua_pushstring(L, modname);
  lua_setfield(L, -2, "_NAME");
  dot = strrchr(modname, '.');  /* look for last dot in module name */
  if (dot == NULL) dot = modname; else dot++;
  /* set _PACKAGE as package name (full module name minus last part) */
  lua_pushlstring(L, modname, (size_t)(dot - modname));
  lua_setfield(L, -2, "_PACKAGE");
}

static int lj_cf_package_module(lua_State *L)
{
  const char *modname = luaL_checkstring(L, 1);
  int loaded = lua_gettop(L) + 1;  /* index of _LOADED table */
  lua_getfield(L, LUA_REGISTRYINDEX, "_LOADED");
  lua_getfield(L, loaded, modname);  /* get _LOADED[modname] */
  if (!lua_istable(L, -1)) {  /* not found? */
    lua_pop(L, 1);  /* remove previous result */
    /* try global variable (and create one if it does not exist) */
    if (luaL_findtable(L, LUA_GLOBALSINDEX, modname, 1) != NULL)
      lj_err_callerv(L, LJ_ERR_BADMODN, modname);
    lua_pushvalue(L, -1);
    lua_setfield(L, loaded, modname);  /* _LOADED[modname] = new table */
  }
  /* check whether table already has a _NAME field */
  lua_getfield(L, -1, "_NAME");
  if (!lua_isnil(L, -1)) {  /* is table an initialized module? */
    lua_pop(L, 1);
  } else {  /* no; initialize it */
    lua_pop(L, 1);
    modinit(L, modname);
  }
  lua_pushvalue(L, -1);
  setfenv(L);
  dooptions(L, loaded - 1);
  return 0;
}

static int lj_cf_package_seeall(lua_State *L)
{
  luaL_checktype(L, 1, LUA_TTABLE);
  if (!lua_getmetatable(L, 1)) {
    lua_createtable(L, 0, 1); /* create new metatable */
    lua_pushvalue(L, -1);
    lua_setmetatable(L, 1);
  }
  lua_pushvalue(L, LUA_GLOBALSINDEX);
  lua_setfield(L, -2, "__index");  /* mt.__index = _G */
  return 0;
}

/* ------------------------------------------------------------------------ */

#define AUXMARK               "\1"

static void setpath(lua_State *L, const char *fieldname, const char *envname,
                    const char *def, int noenv)
```

```
{
#if LJ_TARGET_CONSOLE
  const char *path = NULL;
  UNUSED(envname);
#else
  const char *path = getenv(envname);
#endif
  if (path == NULL || noenv) {
    lua_pushstring(L, def);
  } else {
    path = luaL_gsub(L, path, LUA_PATHSEP LUA_PATHSEP,
                              LUA_PATHSEP AUXMARK LUA_PATHSEP);
    luaL_gsub(L, path, AUXMARK, def);
    lua_remove(L, -2);
  }
  setprogdir(L);
  lua_setfield(L, -2, fieldname);
}

static const luaL_Reg package_lib[] = {
  { "loadlib",        lj_cf_package_loadlib },
  { "searchpath",  lj_cf_package_searchpath },
  { "seeall",        lj_cf_package_seeall },
  { NULL, NULL }
};

static const luaL_Reg package_global[] = {
  { "module",        lj_cf_package_module },
  { "require",        lj_cf_package_require },
  { NULL, NULL }
};

static const lua_CFunction package_loaders[] =
{
  lj_cf_package_loader_preload,
  lj_cf_package_loader_lua,
  lj_cf_package_loader_c,
  lj_cf_package_loader_croot,
  NULL
};

LUALIB_API int luaopen_package(lua_State *L)
{
  int i;
  int noenv;
  luaL_newmetatable(L, "_LOADLIB");
  lj_lib_pushcf(L, lj_cf_package_unloadlib, 1);
  lua_setfield(L, -2, "__gc");
  luaL_register(L, LUA_LOADLIBNAME, package_lib);
  lua_pushvalue(L, -1);
  lua_replace(L, LUA_ENVIRONINDEX);
  lua_createtable(L, sizeof(package_loaders)/sizeof(package_loaders[0])-1, 0);
  for (i = 0; package_loaders[i] != NULL; i++) {
    lj_lib_pushcf(L, package_loaders[i], 1);
    lua_rawseti(L, -2, i+1);
  }
  lua_setfield(L, -2, "loaders");
  lua_getfield(L, LUA_REGISTRYINDEX, "LUA_NOENV");
  noenv = lua_toboolean(L, -1);
  lua_pop(L, 1);
  setpath(L, "path", LUA_PATH, LUA_PATH_DEFAULT, noenv);
  setpath(L, "cpath", LUA_CPATH, LUA_CPATH_DEFAULT, noenv);
  lua_pushliteral(L, LUA_PATH_CONFIG);
  lua_setfield(L, -2, "config");
  luaL_findtable(L, LUA_REGISTRYINDEX, "_LOADED", 16);
  lua_setfield(L, -2, "loaded");
  luaL_findtable(L, LUA_REGISTRYINDEX, "_PRELOAD", 4);
  lua_setfield(L, -2, "preload");
  lua_pushvalue(L, LUA_GLOBALSINDEX);
  luaL_register(L, NULL, package_global);
  lua_pop(L, 1);
  return 1;
}

```

One Level Up Top Level

src/lj_clib.c - luajit-2.0-src

Global variables defined

- clib_def_handle

Functions defined

- clib_check_lds
- clib_error
- clib_error
- clib_error_
- clib_extname
- clib_extname
- clib_extsym
- clib_func_argsize
- clib_getsym
- clib_getsym
- clib_getsym
- clib_loadlib
- clib_loadlib
- clib_loadlib
- clib_needext
- clib_new
- clib_resolve_lds
- clib_unloadlib
- clib_unloadlib
- clib_unloadlib
- lj_clib_default
- lj_clib_index
- lj_clib_load
- lj_clib_unload

Macros defined

- CLIB_DEFHANDLE
- CLIB_DEFHANDLE

- CLIB_DEFHANDLE
- CLIB_DEFHANDLE
- CLIB_DEFHANDLE
- CLIB_SOEXT
- CLIB_SOEXT
- CLIB_SOEXT
- CLIB_SOPREFIX
- CLIB_SOPREFIX
- GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS
- GET_MODULE_HANDLE_EX_FLAG_UNCHANGED_REFCOUNT
- WIN32_LEAN_AND_MEAN
- clib_error

Source code

```
/*
** FFI C library loader.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#include "lj_obj.h"

#if LJ_HASFFI

 #include "lj_gc.h"
 #include "lj_err.h"
 #include "lj_tab.h"
 #include "lj_str.h"
 #include "lj_udata.h"
 #include "lj_ctype.h"
 #include "lj_cconv.h"
 #include "lj_cdata.h"
 #include "lj_clib.h"
 #include "lj_strfmt.h"

 /* -- OS-specific functions ------------------------------------------------ */

 #if LJ_TARGET_DLOPEN

 #include <dlfcn.h>
 #include <stdio.h>

 #if defined(RTLD_DEFAULT)
 #define CLIB_DEFHANDLE        RTLD_DEFAULT
 #elif LJ_TARGET_OSX || LJ_TARGET_BSD
 #define CLIB_DEFHANDLE        ((void *)(intptr_t)-2)
 #else
 #define CLIB_DEFHANDLE        NULL
 #endif

 LJ_NORET LJ_NOINLINE static void clib_error_(lua_State *L)
 {
   lj_err_callermsg(L, dlerror());
 }

 #define clib_error(L, fmt, name)        clib_error_(L)

 #if defined(__CYGWIN__)
 #define CLIB_SOPREFIX        "cyg"
 #else
```

```
#define CLIB_SOPREFIX        "lib"
#endif

#if LJ_TARGET_OSX
#define CLIB_SOEXT        "%s.dylib"
#elif defined(__CYGWIN__)
#define CLIB_SOEXT        "%s.dll"
#else
#define CLIB_SOEXT        "%s.so"
#endif

static const char *clib_extname(lua_State *L, const char *name)
{
  if (!strchr(name, '/')
#ifdef __CYGWIN__
      && !strchr(name, '\\')
#endif
     ) {
    if (!strchr(name, '.')) {
      name = lj_strfmt_pushf(L, CLIB_SOEXT, name);
      L->top--;
#ifdef __CYGWIN__
    } else {
      return name;
#endif
    }
    if (!(name[0] == CLIB_SOPREFIX[0] && name[1] == CLIB_SOPREFIX[1] &&
          name[2] == CLIB_SOPREFIX[2])) {
      name = lj_strfmt_pushf(L, CLIB_SOPREFIX "%s", name);
      L->top--;
    }
  }
  return name;
}

/* Check for a recognized ld script line. */
static const char *clib_check_lds(lua_State *L, const char *buf)
{
  char *p, *e;
  if ((!strncmp(buf, "GROUP", 5) || !strncmp(buf, "INPUT", 5)) &&
      (p = strchr(buf, '('))) {
    while (*++p == ' ') ;
    for (e = p; *e && *e != ' ' && *e != ')'; e++) ;
    return strdata(lj_str_new(L, p, e-p));
  }
  return NULL;
}

/* Quick and dirty solution to resolve shared library name from ld script. */
static const char *clib_resolve_lds(lua_State *L, const char *name)
{
  FILE *fp = fopen(name, "r");
  const char *p = NULL;
  if (fp) {
    char buf[256];
    if (fgets(buf, sizeof(buf), fp)) {
      if (!strncmp(buf, "/* GNU ld script", 16)) {  /* ld script magic? */
        while (fgets(buf, sizeof(buf), fp)) {  /* Check all lines. */
          p = clib_check_lds(L, buf);
          if (p) break;
        }
      } else {  /* Otherwise check only the first line. */
        p = clib_check_lds(L, buf);
      }
    }
    fclose(fp);
  }
  return p;
}

static void *clib_loadlib(lua_State *L, const char *name, int global)
{
  void *h = dlopen(clib_extname(L, name),
                   RTLD_LAZY | (global?RTLD_GLOBAL:RTLD_LOCAL));
  if (!h) {
    const char *e, *err = dlerror();
```

```
    if (*err == '/' && (e = strchr(err, ':')) &&
        (name = clib_resolve_lds(L, strdata(lj_str_new(L, err, e-err))))) {
      h = dlopen(name, RTLD_LAZY | (global?RTLD_GLOBAL:RTLD_LOCAL));
      if (h) return h;
      err = dlerror();
    }
    lj_err_callermsg(L, err);
  }
  return h;
}

static void clib_unloadlib(CLibrary *cl)
{
  if (cl->handle && cl->handle != CLIB_DEFHANDLE)
    dlclose(cl->handle);
}

static void *clib_getsym(CLibrary *cl, const char *name)
{
  void *p = dlsym(cl->handle, name);
  return p;
}

#elif LJ_TARGET_WINDOWS

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#ifndef GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS
#define GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS        4
#define GET_MODULE_HANDLE_EX_FLAG_UNCHANGED_REFCOUNT        2
BOOL WINAPI GetModuleHandleExA(DWORD, LPCSTR, HMODULE*);
#endif

#define CLIB_DEFHANDLE        ((void *)-1)

/* Default libraries. */
enum {
  CLIB_HANDLE_EXE,
  CLIB_HANDLE_DLL,
  CLIB_HANDLE_CRT,
  CLIB_HANDLE_KERNEL32,
  CLIB_HANDLE_USER32,
  CLIB_HANDLE_GDI32,
  CLIB_HANDLE_MAX
};

static void *clib_def_handle[CLIB_HANDLE_MAX];

LJ_NORET LJ_NOINLINE static void clib_error(lua_State *L, const char *fmt,
                                            const char *name)
{
  DWORD err = GetLastError();
  char buf[128];
  if (!FormatMessageA(FORMAT_MESSAGE_IGNORE_INSERTS|FORMAT_MESSAGE_FROM_SYSTEM,
                      NULL, err, 0, buf, sizeof(buf), NULL))
    buf[0] = '\0';
  lj_err_callermsg(L, lj_strfmt_pushf(L, fmt, name, buf));
}

static int clib_needext(const char *s)
{
  while (*s) {
    if (*s == '/' || *s == '\\' || *s == '.') return 0;
    s++;
  }
  return 1;
}

static const char *clib_extname(lua_State *L, const char *name)
{
  if (clib_needext(name)) {
    name = lj_strfmt_pushf(L, "%s.dll", name);
    L->top--;
  }
  return name;
```

```
}

static void *clib_loadlib(lua_State *L, const char *name, int global)
{
  DWORD oldwerr = GetLastError();
  void *h = (void *)LoadLibraryA(clib_extname(L, name));
  if (!h) clib_error(L, "cannot load module " LUA_QS ": %s", name);
  SetLastError(oldwerr);
  UNUSED(global);
  return h;
}

static void clib_unloadlib(CLibrary *cl)
{
  if (cl->handle == CLIB_DEFHANDLE) {
    MSize i;
    for (i = CLIB_HANDLE_KERNEL32; i < CLIB_HANDLE_MAX; i++) {
      void *h = clib_def_handle[i];
      if (h) {
        clib_def_handle[i] = NULL;
        FreeLibrary((HINSTANCE)h);
      }
    }
  } else if (cl->handle) {
    FreeLibrary((HINSTANCE)cl->handle);
  }
}

static void *clib_getsym(CLibrary *cl, const char *name)
{
  void *p = NULL;
  if (cl->handle == CLIB_DEFHANDLE) {  /* Search default libraries. */
    MSize i;
    for (i = 0; i < CLIB_HANDLE_MAX; i++) {
      HINSTANCE h = (HINSTANCE)clib_def_handle[i];
      if (!(void *)h) {  /* Resolve default library handles (once). */
        switch (i) {
        case CLIB_HANDLE_EXE: GetModuleHandleExA(GET_MODULE_HANDLE_EX_FLAG_UNCHANGED_REFCOUNT, NULL, &h);
break;
        case CLIB_HANDLE_DLL:

GetModuleHandleExA(GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS|GET_MODULE_HANDLE_EX_FLAG_UNCHANGED_REFCOUNT,
                           (const char *)clib_def_handle, &h);
          break;
        case CLIB_HANDLE_CRT:

GetModuleHandleExA(GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS|GET_MODULE_HANDLE_EX_FLAG_UNCHANGED_REFCOUNT,
                           (const char *)&_fmode, &h);
          break;
        case CLIB_HANDLE_KERNEL32: h = LoadLibraryA("kernel32.dll"); break;
        case CLIB_HANDLE_USER32: h = LoadLibraryA("user32.dll"); break;
        case CLIB_HANDLE_GDI32: h = LoadLibraryA("gdi32.dll"); break;
        }
        if (!h) continue;
        clib_def_handle[i] = (void *)h;
      }
      p = (void *)GetProcAddress(h, name);
      if (p) break;
    }
  } else {
    p = (void *)GetProcAddress((HINSTANCE)cl->handle, name);
  }
  return p;
}

#else

#define CLIB_DEFHANDLE        NULL

LJ_NORET LJ_NOINLINE static void clib_error(lua_State *L, const char *fmt,
                                            const char *name)
{
  lj_err_callermsg(L, lj_strfmt_pushf(L, fmt, name, "no support for this OS"));
}

static void *clib_loadlib(lua_State *L, const char *name, int global)
```

```
{
  lj_err_callermsg(L, "no support for loading dynamic libraries for this OS");
  UNUSED(name); UNUSED(global);
  return NULL;
}

static void clib_unloadlib(CLibrary *cl)
{
  UNUSED(cl);
}

static void *clib_getsym(CLibrary *cl, const char *name)
{
  UNUSED(cl); UNUSED(name);
  return NULL;
}

#endif

/* -- C library indexing ---------------------------------------------- */

#if LJ_TARGET_X86 && LJ_ABI_WIN
/* Compute argument size for fastcall/stdcall functions. */
static CTSize clib_func_argsize(CTState *cts, CType *ct)
{
  CTSize n = 0;
  while (ct->sib) {
    CType *d;
    ct = ctype_get(cts, ct->sib);
    if (ctype_isfield(ct->info)) {
      d = ctype_rawchild(cts, ct);
      n += ((d->size + 3) & ~3);
    }
  }
  return n;
}
#endif

/* Get redirected or mangled external symbol. */
static const char *clib_extsym(CTState *cts, CType *ct, GCstr *name)
{
  if (ct->sib) {
    CType *ctf = ctype_get(cts, ct->sib);
    if (ctype_isxattrib(ctf->info, CTA_REDIR))
      return strdata(gco2str(gcref(ctf->name)));
  }
  return strdata(name);
}

/* Index a C library by name. */
TValue *lj_clib_index(lua_State *L, CLibrary *cl, GCstr *name)
{
  TValue *tv = lj_tab_setstr(L, cl->cache, name);
  if (LJ_UNLIKELY(tvisnil(tv))) {
    CTState *cts = ctype_cts(L);
    CType *ct;
    CTypeID id = lj_ctype_getname(cts, &ct, name, CLNS_INDEX);
    if (!id)
      lj_err_callerv(L, LJ_ERR_FFI_NODECL, strdata(name));
    if (ctype_isconstval(ct->info)) {
      CType *ctt = ctype_child(cts, ct);
      lua_assert(ctype_isinteger(ctt->info) && ctt->size <= 4);
      if ((ctt->info & CTF_UNSIGNED) && (int32_t)ct->size < 0)
        setnumV(tv, (lua_Number)(uint32_t)ct->size);
      else
        setintV(tv, (int32_t)ct->size);
    } else {
      const char *sym = clib_extsym(cts, ct, name);
#if LJ_TARGET_WINDOWS
      DWORD oldwerr = GetLastError();
#endif
      void *p = clib_getsym(cl, sym);
      GCcdata *cd;
      lua_assert(ctype_isfunc(ct->info) || ctype_isextern(ct->info));
#if LJ_TARGET_X86 && LJ_ABI_WIN
      /* Retry with decorated name for fastcall/stdcall functions. */
```

```
      if (!p && ctype_isfunc(ct->info)) {
        CTInfo cconv = ctype_cconv(ct->info);
        if (cconv == CTCC_FASTCALL || cconv == CTCC_STDCALL) {
          CTSize sz = clib_func_argsize(cts, ct);
          const char *symd = lj_strfmt_pushf(L,
                                cconv == CTCC_FASTCALL ? "@%s@%d" : "_%s@%d",
                                sym, sz);
          L->top--;
          p = clib_getsym(cl, symd);
        }
      }
#endif
      if (!p)
        clib_error(L, "cannot resolve symbol " LUA_QS ": %s", sym);
#if LJ_TARGET_WINDOWS
      SetLastError(oldwerr);
#endif
      cd = lj_cdata_new(cts, id, CTSIZE_PTR);
      *(void **)cdataptr(cd) = p;
      setcdataV(L, tv, cd);
    }
  }
  return tv;
}

/* -- C library management ------------------------------------------------ */

/* Create a new CLibrary object and push it on the stack. */
static CLibrary *clib_new(lua_State *L, GCtab *mt)
{
  GCtab *t = lj_tab_new(L, 0, 0);
  GCudata *ud = lj_udata_new(L, sizeof(CLibrary), t);
  CLibrary *cl = (CLibrary *)uddata(ud);
  cl->cache = t;
  ud->udtype = UDTYPE_FFI_CLIB;
  /* NOBARRIER: The GCudata is new (marked white). */
  setgcref(ud->metatable, obj2gco(mt));
  setudataV(L, L->top++, ud);
  return cl;
}

/* Load a C library. */
void lj_clib_load(lua_State *L, GCtab *mt, GCstr *name, int global)
{
  void *handle = clib_loadlib(L, strdata(name), global);
  CLibrary *cl = clib_new(L, mt);
  cl->handle = handle;
}

/* Unload a C library. */
void lj_clib_unload(CLibrary *cl)
{
  clib_unloadlib(cl);
  cl->handle = NULL;
}

/* Create the default C library object. */
void lj_clib_default(lua_State *L, GCtab *mt)
{
  CLibrary *cl = clib_new(L, mt);
  cl->handle = CLIB_DEFHANDLE;
}

#endif
```

One Level Up Top Level

src/lj_clib.h - luajit-2.0-src

Data types defined

- CLibrary
- CLibrary

Macros defined

- CLNS_INDEX
- _LJ_CLIB_H

Source code

```
1  /*
2  ** FFI C library loader.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_CLIB_H
7  #define _LJ_CLIB_H
8
9  #include "lj_obj.h"
10
11 #if LJ_HASFFI
12
13 /* Namespace for C library indexing. */
14 #define CLNS_INDEX        ((1u<<CT_FUNC)|(1u<<CT_EXTERN)|(1u<<CT_CONSTVAL))
15
16 /* C library namespace. */
17 typedef struct CLibrary {
18   void *handle;               /* Opaque handle for dynamic library loader. */
19   GCtab *cache;               /* Cache for resolved symbols. Anchored in ud->env. */
20 } CLibrary;
21
22 LJ_FUNC TValue *lj_clib_index(lua_State *L, CLibrary *cl, GCstr *name);
23 LJ_FUNC void lj_clib_load(lua_State *L, GCtab *mt, GCstr *name, int global);
24 LJ_FUNC void lj_clib_unload(CLibrary *cl);
25 LJ_FUNC void lj_clib_default(lua_State *L, GCtab *mt);
26
27 #endif
28
29 #endif
```

src/lj_cdata.h - luajit-2.0-src

Functions defined

- cdata_getptr
- cdata_setptr
- lj_cdata_new
- lj_cdata_new_

Macros defined

- _LJ_CDATA_H

Source code

```
1  /*
2  ** C data management.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_CDATA_H
7  #define _LJ_CDATA_H
8
9  #include "lj_obj.h"
10 #include "lj_gc.h"
11 #include "lj_ctype.h"
12
13 #if LJ_HASFFI
14
15 /* Get C data pointer. */
16 static LJ_AINLINE void *cdata_getptr(void *p, CTSize sz)
17 {
18   if (LJ_64 && sz == 4) {  /* Support 32 bit pointers on 64 bit targets. */
19     return ((void *)(uintptr_t)*(uint32_t *)p);
20   } else {
21     lua_assert(sz == CTSIZE_PTR);
22     return *(void **)p;
23   }
24 }
25
26 /* Set C data pointer. */
27 static LJ_AINLINE void cdata_setptr(void *p, CTSize sz, const void *v)
28 {
29   if (LJ_64 && sz == 4) {  /* Support 32 bit pointers on 64 bit targets. */
30     *(uint32_t *)p = (uint32_t)(uintptr_t)v;
31   } else {
32     lua_assert(sz == CTSIZE_PTR);
33     *(void **)p = (void *)v;
34   }
35 }
36
37 /* Allocate fixed-size C data object. */
38 static LJ_AINLINE GCcdata *lj_cdata_new(CTState *cts, CTypeID id, CTSize sz)
39 {
40   GCcdata *cd;
41 #ifdef LUA_USE_ASSERT
42   CType *ct = ctype_raw(cts, id);
43   lua_assert((ctype_hassize(ct->info) ? ct->size : CTSIZE_PTR) == sz);
44 #endif
45   cd = (GCcdata *)lj_mem_newgco(cts->L, sizeof(GCcdata) + sz);
46   cd->gct = ~LJ_TCDATA;
47   cd->ctypeid = ctype_check(cts, id);
48   return cd;
49 }
50
```

```
/* Variant which works without a valid CTState. */
static LJ_AINLINE GCcdata *lj_cdata_new_(lua_State *L, CTypeID id, CTSize sz)
{
  GCcdata *cd = (GCcdata *)lj_mem_newgco(L, sizeof(GCcdata) + sz);
  cd->gct = ~LJ_TCDATA;
  cd->ctypeid = id;
  return cd;
}

LJ_FUNC GCcdata *lj_cdata_newref(CTState *cts, const void *pp, CTypeID id);
LJ_FUNC GCcdata *lj_cdata_newv(lua_State *L, CTypeID id, CTSize sz,
                               CTSize align);

LJ_FUNC void LJ_FASTCALL lj_cdata_free(global_State *g, GCcdata *cd);
LJ_FUNC void lj_cdata_setfin(lua_State *L, GCcdata *cd, GCobj *obj,
                             uint32_t it);

LJ_FUNC CType *lj_cdata_index(CTState *cts, GCcdata *cd, cTValue *key,
                              uint8_t **pp, CTInfo *qual);
LJ_FUNC int lj_cdata_get(CTState *cts, CType *s, TValue *o, uint8_t *sp);
LJ_FUNC void lj_cdata_set(CTState *cts, CType *d, uint8_t *dp, TValue *o,
                          CTInfo qual);

#endif

#endif
```

One Level Up Top Level

src/lj_cdata.c - luajit-2.0-src

Functions defined

- cdata_getconst
- lj_cdata_free
- lj_cdata_get
- lj_cdata_index
- lj_cdata_newref
- lj_cdata_newv
- lj_cdata_set
- lj_cdata_setfin

Source code

```
1  /*
2  ** C data management.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #include "lj_obj.h"
7
8  #if LJ_HASFFI
9
10  #include "lj_gc.h"
11  #include "lj_err.h"
12  #include "lj_tab.h"
13  #include "lj_ctype.h"
14  #include "lj_cconv.h"
15  #include "lj_cdata.h"
16
17  /* -- C data allocation --------------------------------------------------- */
18
19  /* Allocate a new C data object holding a reference to another object. */
20  GCcdata *lj_cdata_newref(CTState *cts, const void *p, CTypeID id)
21  {
22    CTypeID refid = lj_ctype_intern(cts, CTINFO_REF(id), CTSIZE_PTR);
23    GCcdata *cd = lj_cdata_new(cts, refid, CTSIZE_PTR);
24    *(const void **)cdataptr(cd) = p;
25    return cd;
26  }
27
28  /* Allocate variable-sized or specially aligned C data object. */
29  GCcdata *lj_cdata_newv(lua_State *L, CTypeID id, CTSize sz, CTSize align)
30  {
31    global_State *g;
32    MSize extra = sizeof(GCcdataVar) + sizeof(GCcdata) +
33                  (align > CT_MEMALIGN ? (1u<<align) - (1u<<CT_MEMALIGN) : 0);
34    char *p = lj_mem_newt(L, extra + sz, char);
35    uintptr_t adata = (uintptr_t)p + sizeof(GCcdataVar) + sizeof(GCcdata);
36    uintptr_t almask = (1u << align) - 1u;
37    GCcdata *cd = (GCcdata *)(((adata + almask) & ~almask) - sizeof(GCcdata));
38    lua_assert((char *)cd - p < 65536);
39    cdatav(cd)->offset = (uint16_t)((char *)cd - p);
40    cdatav(cd)->extra = extra;
41    cdatav(cd)->len = sz;
42    g = G(L);
43    setgcrefr(cd->nextgc, g->gc.root);
44    setgcref(g->gc.root, obj2gco(cd));
45    newwhite(g, obj2gco(cd));
46    cd->marked |= 0x80;
47    cd->gct = ~LJ_TCDATA;
```

```
  cd->ctypeid = id;
  return cd;
}

/* Free a C data object. */
void LJ_FASTCALL lj_cdata_free(global_State *g, GCcdata *cd)
{
  if (LJ_UNLIKELY(cd->marked & LJ_GC_CDATA_FIN)) {
    GCobj *root;
    makewhite(g, obj2gco(cd));
    markfinalized(obj2gco(cd));
    if ((root = gcref(g->gc.mmudata)) != NULL) {
      setgcrefr(cd->nextgc, root->gch.nextgc);
      setgcref(root->gch.nextgc, obj2gco(cd));
      setgcref(g->gc.mmudata, obj2gco(cd));
    } else {
      setgcref(cd->nextgc, obj2gco(cd));
      setgcref(g->gc.mmudata, obj2gco(cd));
    }
  } else if (LJ_LIKELY(!cdataisv(cd))) {
    CType *ct = ctype_raw(ctype_ctsG(g), cd->ctypeid);
    CTSize sz = ctype_hassize(ct->info) ? ct->size : CTSIZE_PTR;
    lua_assert(ctype_hassize(ct->info) || ctype_isfunc(ct->info) ||
               ctype_isextern(ct->info));
    lj_mem_free(g, cd, sizeof(GCcdata) + sz);
  } else {
    lj_mem_free(g, memcdatav(cd), sizecdatav(cd));
  }
}

void lj_cdata_setfin(lua_State *L, GCcdata *cd, GCobj *obj, uint32_t it)
{
  GCtab *t = ctype_ctsG(G(L))->finalizer;
  if (gcref(t->metatable)) {
    /* Add cdata to finalizer table, if still enabled. */
    TValue *tv, tmp;
    setcdataV(L, &tmp, cd);
    lj_gc_anybarriert(L, t);
    tv = lj_tab_set(L, t, &tmp);
    setgcV(L, tv, obj, it);
    if (!tvisnil(tv))
      cd->marked |= LJ_GC_CDATA_FIN;
    else
      cd->marked &= ~LJ_GC_CDATA_FIN;
  }
}

/* -- C data indexing ---------------------------------------------------- */

/* Index C data by a TValue. Return CType and pointer. */
CType *lj_cdata_index(CTState *cts, GCcdata *cd, cTValue *key, uint8_t **pp,
                      CTInfo *qual)
{
  uint8_t *p = (uint8_t *)cdataptr(cd);
  CType *ct = ctype_get(cts, cd->ctypeid);
  ptrdiff_t idx;

  /* Resolve reference for cdata object. */
  if (ctype_isref(ct->info)) {
    lua_assert(ct->size == CTSIZE_PTR);
    p = *(uint8_t **)p;
    ct = ctype_child(cts, ct);
  }

collect_attrib:
  /* Skip attributes and collect qualifiers. */
  while (ctype_isattrib(ct->info)) {
    if (ctype_attrib(ct->info) == CTA_QUAL) *qual |= ct->size;
    ct = ctype_child(cts, ct);
  }
  lua_assert(!ctype_isref(ct->info));  /* Interning rejects refs to refs. */

  if (tvisint(key)) {
    idx = (ptrdiff_t)intV(key);
    goto integer_key;
  } else if (tvisnum(key)) {  /* Numeric key. */
```

```
    idx = LJ_64 ? (ptrdiff_t)numV(key) : (ptrdiff_t)lj_num2int(numV(key));
  integer_key:
    if (ctype_ispointer(ct->info)) {
      CTSize sz = lj_ctype_size(cts, ctype_cid(ct->info));  /* Element size. */
      if (sz == CTSIZE_INVALID)
        lj_err_caller(cts->L, LJ_ERR_FFI_INVSIZE);
      if (ctype_isptr(ct->info)) {
        p = (uint8_t *)cdata_getptr(p, ct->size);
      } else if ((ct->info & (CTF_VECTOR|CTF_COMPLEX))) {
        if ((ct->info & CTF_COMPLEX)) idx &= 1;
        *qual |= CTF_CONST;  /* Valarray elements are constant. */
      }
      *pp = p + idx*(int32_t)sz;
      return ct;
    }
  } else if (tviscdata(key)) {  /* Integer cdata key. */
    GCcdata *cdk = cdataV(key);
    CType *ctk = ctype_raw(cts, cdk->ctypeid);
    if (ctype_isenum(ctk->info)) ctk = ctype_child(cts, ctk);
    if (ctype_isinteger(ctk->info)) {
      lj_cconv_ct_ct(cts, ctype_get(cts, CTID_INT_PSZ), ctk,
                     (uint8_t *)&idx, cdataptr(cdk), 0);
      goto integer_key;
    }
  } else if (tvisstr(key)) {  /* String key. */
    GCstr *name = strV(key);
    if (ctype_isstruct(ct->info)) {
      CTSize ofs;
      CType *fct = lj_ctype_getfieldq(cts, ct, name, &ofs, qual);
      if (fct) {
        *pp = p + ofs;
        return fct;
      }
    } else if (ctype_iscomplex(ct->info)) {
      if (name->len == 2) {
        *qual |= CTF_CONST;  /* Complex fields are constant. */
        if (strdata(name)[0] == 'r' && strdata(name)[1] == 'e') {
          *pp = p;
          return ct;
        } else if (strdata(name)[0] == 'i' && strdata(name)[1] == 'm') {
          *pp = p + (ct->size >> 1);
          return ct;
        }
      }
    } else if (cd->ctypeid == CTID_CTYPEID) {
      /* Allow indexing a (pointer to) struct constructor to get constants. */
      CType *sct = ctype_raw(cts, *(CTypeID *)p);
      if (ctype_isptr(sct->info))
        sct = ctype_rawchild(cts, sct);
      if (ctype_isstruct(sct->info)) {
        CTSize ofs;
        CType *fct = lj_ctype_getfield(cts, sct, name, &ofs);
        if (fct && ctype_isconstval(fct->info))
          return fct;
      }
      ct = sct;  /* Allow resolving metamethods for constructors, too. */
    }
  }
  if (ctype_isptr(ct->info)) {  /* Automatically perform '->'. */
    if (ctype_isstruct(ctype_rawchild(cts, ct)->info)) {
      p = (uint8_t *)cdata_getptr(p, ct->size);
      ct = ctype_child(cts, ct);
      goto collect_attrib;
    }
  }
  *qual |= 1;  /* Lookup failed. */
  return ct;  /* But return the resolved raw type. */
}

/* -- C data getters ------------------------------------------------------ */

/* Get constant value and convert to TValue. */
static void cdata_getconst(CTState *cts, TValue *o, CType *ct)
{
  CType *ctt = ctype_child(cts, ct);
  lua_assert(ctype_isinteger(ctt->info) && ctt->size <= 4);
```

```
  /* Constants are already zero-extended/sign-extended to 32 bits. */
  if ((ctt->info & CTF_UNSIGNED) && (int32_t)ct->size < 0)
    setnumV(o, (lua_Number)(uint32_t)ct->size);
  else
    setintV(o, (int32_t)ct->size);
}

/* Get C data value and convert to TValue. */
int lj_cdata_get(CTState *cts, CType *s, TValue *o, uint8_t *sp)
{
  CTypeID sid;

  if (ctype_isconstval(s->info)) {
    cdata_getconst(cts, o, s);
    return 0;  /* No GC step needed. */
  } else if (ctype_isbitfield(s->info)) {
    return lj_cconv_tv_bf(cts, s, o, sp);
  }

  /* Get child type of pointer/array/field. */
  lua_assert(ctype_ispointer(s->info) || ctype_isfield(s->info));
  sid = ctype_cid(s->info);
  s = ctype_get(cts, sid);

  /* Resolve reference for field. */
  if (ctype_isref(s->info)) {
    lua_assert(s->size == CTSIZE_PTR);
    sp = *(uint8_t **)sp;
    sid = ctype_cid(s->info);
    s = ctype_get(cts, sid);
  }

  /* Skip attributes. */
  while (ctype_isattrib(s->info))
    s = ctype_child(cts, s);

  return lj_cconv_tv_ct(cts, s, sid, o, sp);
}

/* -- C data setters ------------------------------------------------------ */

/* Convert TValue and set C data value. */
void lj_cdata_set(CTState *cts, CType *d, uint8_t *dp, TValue *o, CTInfo qual)
{
  if (ctype_isconstval(d->info)) {
    goto err_const;
  } else if (ctype_isbitfield(d->info)) {
    if (((d->info|qual) & CTF_CONST)) goto err_const;
    lj_cconv_bf_tv(cts, d, dp, o);
    return;
  }

  /* Get child type of pointer/array/field. */
  lua_assert(ctype_ispointer(d->info) || ctype_isfield(d->info));
  d = ctype_child(cts, d);

  /* Resolve reference for field. */
  if (ctype_isref(d->info)) {
    lua_assert(d->size == CTSIZE_PTR);
    dp = *(uint8_t **)dp;
    d = ctype_child(cts, d);
  }

  /* Skip attributes and collect qualifiers. */
  for (;;) {
    if (ctype_isattrib(d->info)) {
      if (ctype_attrib(d->info) == CTA_QUAL) qual |= d->size;
    } else {
      break;
    }
    d = ctype_child(cts, d);
  }

  lua_assert(ctype_hassize(d->info) && !ctype_isvoid(d->info));

  if (((d->info|qual) & CTF_CONST)) {
```

```
276   err_const:
277     lj_err_caller(cts->L, LJ_ERR_FFI_WRCONST);
278   }
279
280   lj_cconv_ct_tv(cts, d, dp, o, 0);
281 }
282
283 #endif
```

One Level Up Top Level

src/lj_cconv.c - luajit-2.0-src

Functions defined

Source code

```
1  /*
2  ** C type conversions.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #include "lj_obj.h"
7
8  #if LJ_HASFFI
9
10  #include "lj_err.h"
11  #include "lj_tab.h"
12  #include "lj_ctype.h"
13  #include "lj_cdata.h"
14  #include "lj_cconv.h"
15  #include "lj_ccallback.h"
16
17  /* -- Conversion errors ---------------------------------------------------- */
18
19  /* Bad conversion. */
20  LJ_NORET static void cconv_err_conv(CTState *cts, CType *d, CType *s,
21                                      CTInfo flags)
22  {
23    const char *dst = strdata(lj_ctype_repr(cts->L, ctype_typeid(cts, d), NULL));
24    const char *src;
```

```
  if ((flags & CCF_FROMTV))
    src = lj_obj_typename[1+(ctype_isnum(s->info) ? LUA_TNUMBER :
                             ctype_isarray(s->info) ? LUA_TSTRING : LUA_TNIL)];
  else
    src = strdata(lj_ctype_repr(cts->L, ctype_typeid(cts, s), NULL));
  if (CCF_GETARG(flags))
    lj_err_argv(cts->L, CCF_GETARG(flags), LJ_ERR_FFI_BADCONV, src, dst);
  else
    lj_err_callerv(cts->L, LJ_ERR_FFI_BADCONV, src, dst);
}

/* Bad conversion from TValue. */
LJ_NORET static void cconv_err_convtv(CTState *cts, CType *d, TValue *o,
                                      CTInfo flags)
{
  const char *dst = strdata(lj_ctype_repr(cts->L, ctype_typeid(cts, d), NULL));
  const char *src = lj_typename(o);
  if (CCF_GETARG(flags))
    lj_err_argv(cts->L, CCF_GETARG(flags), LJ_ERR_FFI_BADCONV, src, dst);
  else
    lj_err_callerv(cts->L, LJ_ERR_FFI_BADCONV, src, dst);
}

/* Initializer overflow. */
LJ_NORET static void cconv_err_initov(CTState *cts, CType *d)
{
  const char *dst = strdata(lj_ctype_repr(cts->L, ctype_typeid(cts, d), NULL));
  lj_err_callerv(cts->L, LJ_ERR_FFI_INITOV, dst);
}

/* -- C type compatibility checks ----------------------------------------- */

/* Get raw type and qualifiers for a child type. Resolves enums, too. */
static CType *cconv_childqual(CTState *cts, CType *ct, CTInfo *qual)
{
  ct = ctype_child(cts, ct);
  for (;;) {
    if (ctype_isattrib(ct->info)) {
      if (ctype_attrib(ct->info) == CTA_QUAL) *qual |= ct->size;
    } else if (!ctype_isenum(ct->info)) {
      break;
    }
    ct = ctype_child(cts, ct);
  }
  *qual |= (ct->info & CTF_QUAL);
  return ct;
}

/* Check for compatible types when converting to a pointer.
** Note: these checks are more relaxed than what C99 mandates.
*/
int lj_cconv_compatptr(CTState *cts, CType *d, CType *s, CTInfo flags)
{
  if (!((flags & CCF_CAST) || d == s)) {
    CTInfo dqual = 0, squal = 0;
    d = cconv_childqual(cts, d, &dqual);
    if (!ctype_isstruct(s->info))
      s = cconv_childqual(cts, s, &squal);
    if ((flags & CCF_SAME)) {
      if (dqual != squal)
        return 0;  /* Different qualifiers. */
    } else if (!(flags & CCF_IGNQUAL)) {
      if ((dqual & squal) != squal)
        return 0;  /* Discarded qualifiers. */
      if (ctype_isvoid(d->info) || ctype_isvoid(s->info))
        return 1;  /* Converting to/from void * is always ok. */
    }
    if (ctype_type(d->info) != ctype_type(s->info) ||
        d->size != s->size)
      return 0;  /* Different type or different size. */
    if (ctype_isnum(d->info)) {
      if (((d->info ^ s->info) & (CTF_BOOL|CTF_FP)))
        return 0;  /* Different numeric types. */
    } else if (ctype_ispointer(d->info)) {
      /* Check child types for compatibility. */
       return lj_cconv_compatptr(cts, d, s, flags|CCF_SAME);
```

```
    } else if (ctype_isstruct(d->info)) {
      if (d != s)
        return 0;  /* Must be exact same type for struct/union. */
    } else if (ctype_isfunc(d->info)) {
      /* NYI: structural equality of functions. */
    }
  }
  return 1;  /* Types are compatible. */
}

/* -- C type to C type conversion ----------------------------------------- */

/* Convert C type to C type. Caveat: expects to get the raw CType!
**
** Note: This is only used by the interpreter and not optimized at all.
** The JIT compiler will do a much better job specializing for each case.
*/
void lj_cconv_ct_ct(CTState *cts, CType *d, CType *s,
                    uint8_t *dp, uint8_t *sp, CTInfo flags)
{
  CTSize dsize = d->size, ssize = s->size;
  CTInfo dinfo = d->info, sinfo = s->info;
  void *tmpptr;

  lua_assert(!ctype_isenum(dinfo) && !ctype_isenum(sinfo));
  lua_assert(!ctype_isattrib(dinfo) && !ctype_isattrib(sinfo));

  if (ctype_type(dinfo) > CT_MAYCONVERT || ctype_type(sinfo) > CT_MAYCONVERT)
    goto err_conv;

  /* Some basic sanity checks. */
  lua_assert(!ctype_isnum(dinfo) || dsize > 0);
  lua_assert(!ctype_isnum(sinfo) || ssize > 0);
  lua_assert(!ctype_isbool(dinfo) || dsize == 1 || dsize == 4);
  lua_assert(!ctype_isbool(sinfo) || ssize == 1 || ssize == 4);
  lua_assert(!ctype_isinteger(dinfo) || (1u<<lj_fls(dsize)) == dsize);
  lua_assert(!ctype_isinteger(sinfo) || (1u<<lj_fls(ssize)) == ssize);

  switch (cconv_idx2(dinfo, sinfo)) {
  /* Destination is a bool. */
  case CCX(B, B):
    /* Source operand is already normalized. */
    if (dsize == 1) *dp = *sp; else *(int *)dp = *sp;
    break;
  case CCX(B, I): {
    MSize i;
    uint8_t b = 0;
    for (i = 0; i < ssize; i++) b |= sp[i];
    b = (b != 0);
    if (dsize == 1) *dp = b; else *(int *)dp = b;
    break;
    }
  case CCX(B, F): {
    uint8_t b;
    if (ssize == sizeof(double)) b = (*(double *)sp != 0);
    else if (ssize == sizeof(float)) b = (*(float *)sp != 0);
    else goto err_conv;  /* NYI: long double. */
    if (dsize == 1) *dp = b; else *(int *)dp = b;
    break;
    }

  /* Destination is an integer. */
  case CCX(I, B):
  case CCX(I, I):
  conv_I_I:
    if (dsize > ssize) {  /* Zero-extend or sign-extend LSB. */
#if LJ_LE
      uint8_t fill = (!(sinfo & CTF_UNSIGNED) && (sp[ssize-1]&0x80)) ? 0xff : 0;
      memcpy(dp, sp, ssize);
      memset(dp + ssize, fill, dsize-ssize);
#else
      uint8_t fill = (!(sinfo & CTF_UNSIGNED) && (sp[0]&0x80)) ? 0xff : 0;
      memset(dp, fill, dsize-ssize);
      memcpy(dp + (dsize-ssize), sp, ssize);
#endif
    } else {  /* Copy LSB. */
```

```
#if LJ_LE
      memcpy(dp, sp, dsize);
#else
      memcpy(dp, sp + (ssize-dsize), dsize);
#endif
    }
    break;
  case CCX(I, F): {
    double n;  /* Always convert via double. */
  conv_I_F:
    /* Convert source to double. */
    if (ssize == sizeof(double)) n = *(double *)sp;
    else if (ssize == sizeof(float)) n = (double)*(float *)sp;
    else goto err_conv;  /* NYI: long double. */
    /* Then convert double to integer. */
    /* The conversion must exactly match the semantics of JIT-compiled code! */
    if (dsize < 4 || (dsize == 4 && !(dinfo & CTF_UNSIGNED))) {
      int32_t i = (int32_t)n;
      if (dsize == 4) *(int32_t *)dp = i;
      else if (dsize == 2) *(int16_t *)dp = (int16_t)i;
      else *(int8_t *)dp = (int8_t)i;
    } else if (dsize == 4) {
      *(uint32_t *)dp = (uint32_t)n;
    } else if (dsize == 8) {
      if (!(dinfo & CTF_UNSIGNED))
        *(int64_t *)dp = (int64_t)n;
      else
        *(uint64_t *)dp = lj_num2u64(n);
    } else {
      goto err_conv;  /* NYI: conversion to >64 bit integers. */
    }
    break;
    }
  case CCX(I, C):
    s = ctype_child(cts, s);
    sinfo = s->info;
    ssize = s->size;
    goto conv_I_F;  /* Just convert re. */
  case CCX(I, P):
    if (!(flags & CCF_CAST)) goto err_conv;
    sinfo = CTINFO(CT_NUM, CTF_UNSIGNED);
    goto conv_I_I;
  case CCX(I, A):
    if (!(flags & CCF_CAST)) goto err_conv;
    sinfo = CTINFO(CT_NUM, CTF_UNSIGNED);
    ssize = CTSIZE_PTR;
    tmpptr = sp;
    sp = (uint8_t *)&tmpptr;
    goto conv_I_I;

  /* Destination is a floating-point number. */
  case CCX(F, B):
  case CCX(F, I): {
    double n;  /* Always convert via double. */
  conv_F_I:
    /* First convert source to double. */
    /* The conversion must exactly match the semantics of JIT-compiled code! */
    if (ssize < 4 || (ssize == 4 && !(sinfo & CTF_UNSIGNED))) {
      int32_t i;
      if (ssize == 4) {
        i = *(int32_t *)sp;
      } else if (!(sinfo & CTF_UNSIGNED)) {
        if (ssize == 2) i = *(int16_t *)sp;
        else i = *(int8_t *)sp;
      } else {
        if (ssize == 2) i = *(uint16_t *)sp;
        else i = *(uint8_t *)sp;
      }
      n = (double)i;
    } else if (ssize == 4) {
      n = (double)*(uint32_t *)sp;
    } else if (ssize == 8) {
      if (!(sinfo & CTF_UNSIGNED)) n = (double)*(int64_t *)sp;
      else n = (double)*(uint64_t *)sp;
    } else {
      goto err_conv;  /* NYI: conversion from >64 bit integers. */
```

```
    }
    /* Convert double to destination. */
    if (dsize == sizeof(double)) *(double *)dp = n;
    else if (dsize == sizeof(float)) *(float *)dp = (float)n;
    else goto err_conv;  /* NYI: long double. */
    break;
    }
  case CCX(F, F): {
    double n;  /* Always convert via double. */
  conv_F_F:
    if (ssize == dsize) goto copyval;
    /* Convert source to double. */
    if (ssize == sizeof(double)) n = *(double *)sp;
    else if (ssize == sizeof(float)) n = (double)*(float *)sp;
    else goto err_conv;  /* NYI: long double. */
    /* Convert double to destination. */
    if (dsize == sizeof(double)) *(double *)dp = n;
    else if (dsize == sizeof(float)) *(float *)dp = (float)n;
    else goto err_conv;  /* NYI: long double. */
    break;
    }
  case CCX(F, C):
    s = ctype_child(cts, s);
    sinfo = s->info;
    ssize = s->size;
    goto conv_F_F;  /* Ignore im, and convert from re. */

  /* Destination is a complex number. */
  case CCX(C, I):
    d = ctype_child(cts, d);
    dinfo = d->info;
    dsize = d->size;
    memset(dp + dsize, 0, dsize);  /* Clear im. */
    goto conv_F_I;  /* Convert to re. */
  case CCX(C, F):
    d = ctype_child(cts, d);
    dinfo = d->info;
    dsize = d->size;
    memset(dp + dsize, 0, dsize);  /* Clear im. */
    goto conv_F_F;  /* Convert to re. */

  case CCX(C, C):
    if (dsize != ssize) {  /* Different types: convert re/im separately. */
      CType *dc = ctype_child(cts, d);
      CType *sc = ctype_child(cts, s);
      lj_cconv_ct_ct(cts, dc, sc, dp, sp, flags);
      lj_cconv_ct_ct(cts, dc, sc, dp + dc->size, sp + sc->size, flags);
      return;
    }
    goto copyval;  /* Otherwise this is easy. */

  /* Destination is a vector. */
  case CCX(V, I):
  case CCX(V, F):
  case CCX(V, C): {
    CType *dc = ctype_child(cts, d);
    CTSize esize;
    /* First convert the scalar to the first element. */
    lj_cconv_ct_ct(cts, dc, s, dp, sp, flags);
    /* Then replicate it to the other elements (splat). */
    for (sp = dp, esize = dc->size; dsize > esize; dsize -= esize) {
      dp += esize;
      memcpy(dp, sp, esize);
    }
    break;
    }

  case CCX(V, V):
    /* Copy same-sized vectors, even for different lengths/element-types. */
    if (dsize != ssize) goto err_conv;
    goto copyval;

  /* Destination is a pointer. */
  case CCX(P, I):
    if (!(flags & CCF_CAST)) goto err_conv;
    dinfo = CTINFO(CT_NUM, CTF_UNSIGNED);
```

```
    goto conv_I_I;

  case CCX(P, F):
    if (!(flags & CCF_CAST) || !(flags & CCF_FROMTV)) goto err_conv;
    /* The signed conversion is cheaper. x64 really has 47 bit pointers. */
    dinfo = CTINFO(CT_NUM, (LJ_64 && dsize == 8) ? 0 : CTF_UNSIGNED);
    goto conv_I_F;

  case CCX(P, P):
    if (!lj_cconv_compatptr(cts, d, s, flags)) goto err_conv;
    cdata_setptr(dp, dsize, cdata_getptr(sp, ssize));
    break;

  case CCX(P, A):
  case CCX(P, S):
    if (!lj_cconv_compatptr(cts, d, s, flags)) goto err_conv;
    cdata_setptr(dp, dsize, sp);
    break;

  /* Destination is an array. */
  case CCX(A, A):
    if ((flags & CCF_CAST) || (d->info & CTF_VLA) || dsize != ssize ||
        d->size == CTSIZE_INVALID || !lj_cconv_compatptr(cts, d, s, flags))
      goto err_conv;
    goto copyval;

  /* Destination is a struct/union. */
  case CCX(S, S):
    if ((flags & CCF_CAST) || (d->info & CTF_VLA) || d != s)
      goto err_conv;  /* Must be exact same type. */
copyval:  /* Copy value. */
    lua_assert(dsize == ssize);
    memcpy(dp, sp, dsize);
    break;

  default:
  err_conv:
    cconv_err_conv(cts, d, s, flags);
  }
}

/* -- C type to TValue conversion ----------------------------------------- */

/* Convert C type to TValue. Caveat: expects to get the raw CType! */
int lj_cconv_tv_ct(CTState *cts, CType *s, CTypeID sid,
                   TValue *o, uint8_t *sp)
{
  CTInfo sinfo = s->info;
  if (ctype_isnum(sinfo)) {
    if (!ctype_isbool(sinfo)) {
      if (ctype_isinteger(sinfo) && s->size > 4) goto copyval;
      if (LJ_DUALNUM && ctype_isinteger(sinfo)) {
        int32_t i;
        lj_cconv_ct_ct(cts, ctype_get(cts, CTID_INT32), s,
                       (uint8_t *)&i, sp, 0);
        if ((sinfo & CTF_UNSIGNED) && i < 0)
          setnumV(o, (lua_Number)(uint32_t)i);
        else
          setintV(o, i);
      } else {
        lj_cconv_ct_ct(cts, ctype_get(cts, CTID_DOUBLE), s,
                       (uint8_t *)&o->n, sp, 0);
        /* Numbers are NOT canonicalized here! Beware of uninitialized data. */
        lua_assert(tvisnum(o));
      }
    } else {
      uint32_t b = s->size == 1 ? (*sp != 0) : (*(int *)sp != 0);
      setboolV(o, b);
      setboolV(&cts->g->tmptv2, b);  /* Remember for trace recorder. */
    }
    return 0;
  } else if (ctype_isrefarray(sinfo) || ctype_isstruct(sinfo)) {
    /* Create reference. */
    setcdataV(cts->L, o, lj_cdata_newref(cts, sp, sid));
    return 1;  /* Need GC step. */
  } else {
```

```
    GCcdata *cd;
    CTSize sz;
  copyval:  /* Copy value. */
    sz = s->size;
    lua_assert(sz != CTSIZE_INVALID);
    /* Attributes are stripped, qualifiers are kept (but mostly ignored). */
    cd = lj_cdata_new(cts, ctype_typeid(cts, s), sz);
    setcdataV(cts->L, o, cd);
    memcpy(cdataptr(cd), sp, sz);
    return 1;  /* Need GC step. */
  }
}

/* Convert bitfield to TValue. */
int lj_cconv_tv_bf(CTState *cts, CType *s, TValue *o, uint8_t *sp)
{
  CTInfo info = s->info;
  CTSize pos, bsz;
  uint32_t val;
  lua_assert(ctype_isbitfield(info));
  /* NYI: packed bitfields may cause misaligned reads. */
  switch (ctype_bitcsz(info)) {
  case 4: val = *(uint32_t *)sp; break;
  case 2: val = *(uint16_t *)sp; break;
  case 1: val = *(uint8_t *)sp; break;
  default: lua_assert(0); val = 0; break;
  }
  /* Check if a packed bitfield crosses a container boundary. */
  pos = ctype_bitpos(info);
  bsz = ctype_bitbsz(info);
  lua_assert(pos < 8*ctype_bitcsz(info));
  lua_assert(bsz > 0 && bsz <= 8*ctype_bitcsz(info));
  if (pos + bsz > 8*ctype_bitcsz(info))
    lj_err_caller(cts->L, LJ_ERR_FFI_NYIPACKBIT);
  if (!(info & CTF_BOOL)) {
    CTSize shift = 32 - bsz;
    if (!(info & CTF_UNSIGNED)) {
      setintV(o, (int32_t)(val << (shift-pos)) >> shift);
    } else {
      val = (val << (shift-pos)) >> shift;
      if (!LJ_DUALNUM || (int32_t)val < 0)
        setnumV(o, (lua_Number)(uint32_t)val);
      else
        setintV(o, (int32_t)val);
    }
  } else {
    lua_assert(bsz == 1);
    setboolV(o, (val >> pos) & 1);
  }
  return 0;  /* No GC step needed. */
}

/* -- TValue to C type conversion ----------------------------------------- */

/* Convert table to array. */
static void cconv_array_tab(CTState *cts, CType *d,
                            uint8_t *dp, GCtab *t, CTInfo flags)
{
  int32_t i;
  CType *dc = ctype_rawchild(cts, d);  /* Array element type. */
  CTSize size = d->size, esize = dc->size, ofs = 0;
  for (i = 0; ; i++) {
    TValue *tv = (TValue *)lj_tab_getint(t, i);
    if (!tv || tvisnil(tv)) {
      if (i == 0) continue;  /* Try again for 1-based tables. */
      break;  /* Stop at first nil. */
    }
    if (ofs >= size)
      cconv_err_initov(cts, d);
    lj_cconv_ct_tv(cts, dc, dp + ofs, tv, flags);
    ofs += esize;
  }
  if (size != CTSIZE_INVALID) {  /* Only fill up arrays with known size. */
    if (ofs == esize) {  /* Replicate a single element. */
      for (; ofs < size; ofs += esize) memcpy(dp + ofs, dp, esize);
    } else {  /* Otherwise fill the remainder with zero. */
```

```
      memset(dp + ofs, 0, size - ofs);
    }
  }
}

/* Convert table to sub-struct/union. */
static void cconv_substruct_tab(CTState *cts, CType *d, uint8_t *dp,
                                GCtab *t, int32_t *ip, CTInfo flags)
{
  CTypeID id = d->sib;
  while (id) {
    CType *df = ctype_get(cts, id);
    id = df->sib;
    if (ctype_isfield(df->info) || ctype_isbitfield(df->info)) {
      TValue *tv;
      int32_t i = *ip, iz = i;
      if (!gcref(df->name)) continue;  /* Ignore unnamed fields. */
      if (i >= 0) {
      retry:
        tv = (TValue *)lj_tab_getint(t, i);
        if (!tv || tvisnil(tv)) {
          if (i == 0) { i = 1; goto retry; }  /* 1-based tables. */
          if (iz == 0) { *ip = i = -1; goto tryname; }  /* Init named fields. */
          break;  /* Stop at first nil. */
        }
        *ip = i + 1;
      } else {
      tryname:
        tv = (TValue *)lj_tab_getstr(t, gco2str(gcref(df->name)));
        if (!tv || tvisnil(tv)) continue;
      }
      if (ctype_isfield(df->info))
        lj_cconv_ct_tv(cts, ctype_rawchild(cts, df), dp+df->size, tv, flags);
      else
        lj_cconv_bf_tv(cts, df, dp+df->size, tv);
      if ((d->info & CTF_UNION)) break;
    } else if (ctype_isxattrib(df->info, CTA_SUBTYPE)) {
      cconv_substruct_tab(cts, ctype_rawchild(cts, df),
                          dp+df->size, t, ip, flags);
    }  /* Ignore all other entries in the chain. */
  }
}

/* Convert table to struct/union. */
static void cconv_struct_tab(CTState *cts, CType *d,
                             uint8_t *dp, GCtab *t, CTInfo flags)
{
  int32_t i = 0;
  memset(dp, 0, d->size);  /* Much simpler to clear the struct first. */
  cconv_substruct_tab(cts, d, dp, t, &i, flags);
}

/* Convert TValue to C type. Caveat: expects to get the raw CType! */
void lj_cconv_ct_tv(CTState *cts, CType *d,
                    uint8_t *dp, TValue *o, CTInfo flags)
{
  CTypeID sid = CTID_P_VOID;
  CType *s;
  void *tmpptr;
  uint8_t tmpbool, *sp = (uint8_t *)&tmpptr;
  if (LJ_LIKELY(tvisint(o))) {
    sp = (uint8_t *)&o->i;
    sid = CTID_INT32;
    flags |= CCF_FROMTV;
  } else if (LJ_LIKELY(tvisnum(o))) {
    sp = (uint8_t *)&o->n;
    sid = CTID_DOUBLE;
    flags |= CCF_FROMTV;
  } else if (tviscdata(o)) {
    sp = cdataptr(cdataV(o));
    sid = cdataV(o)->ctypeid;
    s = ctype_get(cts, sid);
    if (ctype_isref(s->info)) {  /* Resolve reference for value. */
      lua_assert(s->size == CTSIZE_PTR);
      sp = *(void **)sp;
      sid = ctype_cid(s->info);
```

```
    }
    s = ctype_raw(cts, sid);
    if (ctype_isfunc(s->info)) {
      sid = lj_ctype_intern(cts, CTINFO(CT_PTR, CTALIGN_PTR|sid), CTSIZE_PTR);
    } else {
      if (ctype_isenum(s->info)) s = ctype_child(cts, s);
      goto doconv;
    }
  } else if (tvisstr(o)) {
    GCstr *str = strV(o);
    if (ctype_isenum(d->info)) {  /* Match string against enum constant. */
      CTSize ofs;
      CType *cct = lj_ctype_getfield(cts, d, str, &ofs);
      if (!cct || !ctype_isconstval(cct->info))
        goto err_conv;
      lua_assert(d->size == 4);
      sp = (uint8_t *)&cct->size;
      sid = ctype_cid(cct->info);
    } else if (ctype_isrefarray(d->info)) {  /* Copy string to array. */
      CType *dc = ctype_rawchild(cts, d);
      CTSize sz = str->len+1;
      if (!ctype_isinteger(dc->info) || dc->size != 1)
        goto err_conv;
      if (d->size != 0 && d->size < sz)
        sz = d->size;
      memcpy(dp, strdata(str), sz);
      return;
    } else {  /* Otherwise pass it as a const char[]. */
      sp = (uint8_t *)strdata(str);
      sid = CTID_A_CCHAR;
      flags |= CCF_FROMTV;
    }
  } else if (tvistab(o)) {
    if (ctype_isarray(d->info)) {
      cconv_array_tab(cts, d, dp, tabV(o), flags);
      return;
    } else if (ctype_isstruct(d->info)) {
      cconv_struct_tab(cts, d, dp, tabV(o), flags);
      return;
    } else {
      goto err_conv;
    }
  } else if (tvisbool(o)) {
    tmpbool = boolV(o);
    sp = &tmpbool;
    sid = CTID_BOOL;
  } else if (tvisnil(o)) {
    tmpptr = (void *)0;
    flags |= CCF_FROMTV;
  } else if (tvisudata(o)) {
    GCudata *ud = udataV(o);
    tmpptr = uddata(ud);
    if (ud->udtype == UDTYPE_IO_FILE)
      tmpptr = *(void **)tmpptr;
  } else if (tvislightud(o)) {
    tmpptr = lightudV(o);
  } else if (tvisfunc(o)) {
    void *p = lj_ccallback_new(cts, d, funcV(o));
    if (p) {
      *(void **)dp = p;
      return;
    }
    goto err_conv;
  } else {
  err_conv:
    cconv_err_convtv(cts, d, o, flags);
  }
  s = ctype_get(cts, sid);
doconv:
  if (ctype_isenum(d->info)) d = ctype_child(cts, d);
  lj_cconv_ct_ct(cts, d, s, dp, sp, flags);
}

/* Convert TValue to bitfield. */
void lj_cconv_bf_tv(CTState *cts, CType *d, uint8_t *dp, TValue *o)
{
```

```
  CTInfo info = d->info;
  CTSize pos, bsz;
  uint32_t val, mask;
  lua_assert(ctype_isbitfield(info));
  if ((info & CTF_BOOL)) {
    uint8_t tmpbool;
    lua_assert(ctype_bitbsz(info) == 1);
    lj_cconv_ct_tv(cts, ctype_get(cts, CTID_BOOL), &tmpbool, o, 0);
    val = tmpbool;
  } else {
    CTypeID did = (info & CTF_UNSIGNED) ? CTID_UINT32 : CTID_INT32;
    lj_cconv_ct_tv(cts, ctype_get(cts, did), (uint8_t *)&val, o, 0);
  }
  pos = ctype_bitpos(info);
  bsz = ctype_bitbsz(info);
  lua_assert(pos < 8*ctype_bitcsz(info));
  lua_assert(bsz > 0 && bsz <= 8*ctype_bitcsz(info));
  /* Check if a packed bitfield crosses a container boundary. */
  if (pos + bsz > 8*ctype_bitcsz(info))
    lj_err_caller(cts->L, LJ_ERR_FFI_NYIPACKBIT);
  mask = ((1u << bsz) - 1u) << pos;
  val = (val << pos) & mask;
  /* NYI: packed bitfields may cause misaligned reads/writes. */
  switch (ctype_bitcsz(info)) {
  case 4: *(uint32_t *)dp = (*(uint32_t *)dp & ~mask) | (uint32_t)val; break;
  case 2: *(uint16_t *)dp = (*(uint16_t *)dp & ~mask) | (uint16_t)val; break;
  case 1: *(uint8_t *)dp = (*(uint8_t *)dp & ~mask) | (uint8_t)val; break;
  default: lua_assert(0); break;
  }
}

/* -- Initialize C type with TValues -------------------------------------- */

/* Initialize an array with TValues. */
static void cconv_array_init(CTState *cts, CType *d, CTSize sz, uint8_t *dp,
                             TValue *o, MSize len)
{
  CType *dc = ctype_rawchild(cts, d);  /* Array element type. */
  CTSize ofs, esize = dc->size;
  MSize i;
  if (len*esize > sz)
    cconv_err_initov(cts, d);
  for (i = 0, ofs = 0; i < len; i++, ofs += esize)
    lj_cconv_ct_tv(cts, dc, dp + ofs, o + i, 0);
  if (ofs == esize) {  /* Replicate a single element. */
    for (; ofs < sz; ofs += esize) memcpy(dp + ofs, dp, esize);
  } else {  /* Otherwise fill the remainder with zero. */
    memset(dp + ofs, 0, sz - ofs);
  }
}

/* Initialize a sub-struct/union with TValues. */
static void cconv_substruct_init(CTState *cts, CType *d, uint8_t *dp,
                                 TValue *o, MSize len, MSize *ip)
{
  CTypeID id = d->sib;
  while (id) {
    CType *df = ctype_get(cts, id);
    id = df->sib;
    if (ctype_isfield(df->info) || ctype_isbitfield(df->info)) {
      MSize i = *ip;
      if (!gcref(df->name)) continue;  /* Ignore unnamed fields. */
      if (i >= len) break;
      *ip = i + 1;
      if (ctype_isfield(df->info))
        lj_cconv_ct_tv(cts, ctype_rawchild(cts, df), dp+df->size, o + i, 0);
      else
        lj_cconv_bf_tv(cts, df, dp+df->size, o + i);
      if ((d->info & CTF_UNION)) break;
    } else if (ctype_isxattrib(df->info, CTA_SUBTYPE)) {
      cconv_substruct_init(cts, ctype_rawchild(cts, df),
                           dp+df->size, o, len, ip);
      if ((d->info & CTF_UNION)) break;
    }  /* Ignore all other entries in the chain. */
  }
}
```

```
709
710 /* Initialize a struct/union with TValues. */
711 static void cconv_struct_init(CTState *cts, CType *d, CTSize sz, uint8_t *dp,
712                               TValue *o, MSize len)
713 {
714   MSize i = 0;
715   memset(dp, 0, sz);  /* Much simpler to clear the struct first. */
716   cconv_substruct_init(cts, d, dp, o, len, &i);
717   if (i < len)
718     cconv_err_initov(cts, d);
719 }
720
721 /* Check whether to use a multi-value initializer.
722 ** This is true if an aggregate is to be initialized with a value.
723 ** Valarrays are treated as values here so ct_tv handles (V|C, I|F).
724 */
725 int lj_cconv_multi_init(CTState *cts, CType *d, TValue *o)
726 {
727   if (!(ctype_isrefarray(d->info) || ctype_isstruct(d->info)))
728     return 0;  /* Destination is not an aggregate. */
729   if (tvistab(o) || (tvisstr(o) && !ctype_isstruct(d->info)))
730     return 0;  /* Initializer is not a value. */
731   if (tviscdata(o) && lj_ctype_rawref(cts, cdataV(o)->ctypeid) == d)
732     return 0;  /* Source and destination are identical aggregates. */
733   return 1;  /* Otherwise the initializer is a value. */
734 }
735
736 /* Initialize C type with TValues. Caveat: expects to get the raw CType! */
737 void lj_cconv_ct_init(CTState *cts, CType *d, CTSize sz,
738                       uint8_t *dp, TValue *o, MSize len)
739 {
740   if (len == 0)
741     memset(dp, 0, sz);
742   else if (len == 1 && !lj_cconv_multi_init(cts, d, o))
743     lj_cconv_ct_tv(cts, d, dp, o, 0);
744   else if (ctype_isarray(d->info))  /* Also handles valarray init with len>1. */
745     cconv_array_init(cts, d, sz, dp, o, len);
746   else if (ctype_isstruct(d->info))
747     cconv_struct_init(cts, d, sz, dp, o, len);
748   else
749     cconv_err_initov(cts, d);
750 }
751
752 #endif
```

One Level Up Top Level

src/lj_cconv.h - luajit-2.0-src

Functions defined

- cconv_idx

Macros defined

- CCF_ARG
- CCF_ARG_SHIFT
- CCF_CAST
- CCF_FROMTV
- CCF_GETARG
- CCF_IGNQUAL
- CCF_SAME
- CCX
- _LJ_CCONV_H
- cconv_idx2

Source code

```
1  /*
2  ** C type conversions.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_CCONV_H
7  #define _LJ_CCONV_H
8
9  #include "lj_obj.h"
10  #include "lj_ctype.h"
11
12  #if LJ_HASFFI
13
14  /* Compressed C type index. ORDER CCX. */
15  enum {
16    CCX_B,        /* Bool. */
17    CCX_I,        /* Integer. */
18    CCX_F,        /* Floating-point number. */
19    CCX_C,        /* Complex. */
20    CCX_V,        /* Vector. */
21    CCX_P,        /* Pointer. */
22    CCX_A,        /* Refarray. */
23    CCX_S               /* Struct/union. */
24  };
25
26  /* Convert C type info to compressed C type index. ORDER CT. ORDER CCX. */
27  static LJ_AINLINE uint32_t cconv_idx(CTInfo info)
28  {
29    uint32_t idx = ((info >> 26) & 15u);  /* Dispatch bits. */
30    lua_assert(ctype_type(info) <= CT_MAYCONVERT);
31  #if LJ_64
32    idx = ((uint32_t)(U64x(f436fff5,fff7f021) >> 4*idx) & 15u);
33  #else
34    idx = (((idx < 8 ? 0xfff7f021u : 0xf436fff5) >> 4*(idx & 7u)) & 15u);
35  #endif
36    lua_assert(idx < 8);
```

```
  return idx;
}

#define cconv_idx2(dinfo, sinfo) \
  ((cconv_idx((dinfo)) << 3) + cconv_idx((sinfo)))

#define CCX(dst, src)               ((CCX_##dst << 3) + CCX_##src)

/* Conversion flags. */
#define CCF_CAST        0x00000001u
#define CCF_FROMTV        0x00000002u
#define CCF_SAME        0x00000004u
#define CCF_IGNQUAL        0x00000008u

#define CCF_ARG_SHIFT        8
#define CCF_ARG(n)        ((n) << CCF_ARG_SHIFT)
#define CCF_GETARG(f)        ((f) >> CCF_ARG_SHIFT)

LJ_FUNC int lj_cconv_compatptr(CTState *cts, CType *d, CType *s, CTInfo flags);
LJ_FUNC void lj_cconv_ct_ct(CTState *cts, CType *d, CType *s,
                            uint8_t *dp, uint8_t *sp, CTInfo flags);
LJ_FUNC int lj_cconv_tv_ct(CTState *cts, CType *s, CTypeID sid,
                           TValue *o, uint8_t *sp);
LJ_FUNC int lj_cconv_tv_bf(CTState *cts, CType *s, TValue *o, uint8_t *sp);
LJ_FUNC void lj_cconv_ct_tv(CTState *cts, CType *d,
                            uint8_t *dp, TValue *o, CTInfo flags);
LJ_FUNC void lj_cconv_bf_tv(CTState *cts, CType *d, uint8_t *dp, TValue *o);
LJ_FUNC int lj_cconv_multi_init(CTState *cts, CType *d, TValue *o);
LJ_FUNC void lj_cconv_ct_init(CTState *cts, CType *d, CTSize sz,
                              uint8_t *dp, TValue *o, MSize len);

#endif

#endif
```

One Level Up Top Level

src/lj_udata.c - luajit-2.0-src

Functions defined

- lj_udata_free
- lj_udata_new

Macros defined

- LUA_CORE
- lj_udata_c

Source code

```
1  /*
2  ** Userdata handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lj_udata_c
7  #define LUA_CORE
8
9  #include "lj_obj.h"
10  #include "lj_gc.h"
11  #include "lj_udata.h"
12
13  GCudata *lj_udata_new(lua_State *L, MSize sz, GCtab *env)
14  {
15    GCudata *ud = lj_mem_newt(L, sizeof(GCudata) + sz, GCudata);
16    global_State *g = G(L);
17    newwhite(g, ud);  /* Not finalized. */
18    ud->gct = ~LJ_TUDATA;
19    ud->udtype = UDTYPE_USERDATA;
20    ud->len = sz;
21    /* NOBARRIER: The GCudata is new (marked white). */
22    setgcrefnull(ud->metatable);
23    setgcref(ud->env, obj2gco(env));
24    /* Chain to userdata list (after main thread). */
25    setgcrefr(ud->nextgc, mainthread(g)->nextgc);
26    setgcref(mainthread(g)->nextgc, obj2gco(ud));
27    return ud;
28  }
29
30  void LJ_FASTCALL lj_udata_free(global_State *g, GCudata *ud)
31  {
32    lj_mem_free(g, ud, sizeudata(ud));
33  }
34
```

src/lj_debug.c - luajit-2.0-src

Functions defined

- debug_frameline
- debug_framepc
- debug_localname
- debug_putchunkname
- debug_varname
- lj_debug_addloc
- lj_debug_dumpstack
- lj_debug_frame
- lj_debug_funcname
- lj_debug_getinfo
- lj_debug_line
- lj_debug_pushloc
- lj_debug_shortname
- lj_debug_slotname
- lj_debug_uvname
- lj_debug_uvnamev
- luaL_traceback
- lua_getinfo
- lua_getlocal
- lua_getstack
- lua_setlocal

Macros defined

- LUA_CORE
- NO_BCPOS
- TRACEBACK_LEVELS1
- TRACEBACK_LEVELS2
- VARNAMESTR
- VARNAMESTR
- lj_debug_c

Source code

```
/*
** Debugging and introspection.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_debug_c
#define LUA_CORE

#include "lj_obj.h"
#include "lj_err.h"
#include "lj_debug.h"
#include "lj_buf.h"
#include "lj_tab.h"
#include "lj_state.h"
#include "lj_frame.h"
#include "lj_bc.h"
#include "lj_strfmt.h"
#if LJ_HASJIT
#include "lj_jit.h"
#endif

/* -- Frames -------------------------------------------------------------- */

/* Get frame corresponding to a level. */
cTValue *lj_debug_frame(lua_State *L, int level, int *size)
{
  cTValue *frame, *nextframe, *bot = tvref(L->stack)+LJ_FR2;
  /* Traverse frames backwards. */
  for (nextframe = frame = L->base-1; frame > bot; ) {
    if (frame_gc(frame) == obj2gco(L))
      level++;  /* Skip dummy frames. See lj_err_optype_call(). */
    if (level-- == 0) {
      *size = (int)(nextframe - frame);
      return frame;  /* Level found. */
    }
    nextframe = frame;
    if (frame_islua(frame)) {
      frame = frame_prevl(frame);
    } else {
      if (frame_isvarg(frame))
        level++;  /* Skip vararg pseudo-frame. */
      frame = frame_prevd(frame);
    }
  }
  *size = level;
  return NULL;  /* Level not found. */
}

/* Invalid bytecode position. */
#define NO_BCPOS        (~(BCPos)0)

/* Return bytecode position for function/frame or NO_BCPOS. */
static BCPos debug_framepc(lua_State *L, GCfunc *fn, cTValue *nextframe)
{
  const BCIns *ins;
  GCproto *pt;
  BCPos pos;
  lua_assert(fn->c.gct == ~LJ_TFUNC || fn->c.gct == ~LJ_TTHREAD);
  if (!isluafunc(fn)) {  /* Cannot derive a PC for non-Lua functions. */
    return NO_BCPOS;
  } else if (nextframe == NULL) {  /* Lua function on top. */
    void *cf = cframe_raw(L->cframe);
    if (cf == NULL || (char *)cframe_pc(cf) == (char *)cframe_L(cf))
      return NO_BCPOS;
    ins = cframe_pc(cf);  /* Only happens during error/hook handling. */
  } else {
    if (frame_islua(nextframe)) {
      ins = frame_pc(nextframe);
    } else if (frame_iscont(nextframe)) {
      ins = frame_contpc(nextframe);
    } else {
      /* Lua function below errfunc/gc/hook: find cframe to get the PC. */
      void *cf = cframe_raw(L->cframe);
```

```
      TValue *f = L->base-1;
      for (;;) {
        if (cf == NULL)
          return NO_BCPOS;
        while (cframe_nres(cf) < 0) {
          if (f >= restorestack(L, -cframe_nres(cf)))
            break;
          cf = cframe_raw(cframe_prev(cf));
          if (cf == NULL)
            return NO_BCPOS;
        }
        if (f < nextframe)
          break;
        if (frame_islua(f)) {
          f = frame_prevl(f);
        } else {
          if (frame_isc(f) || (frame_iscont(f) && frame_iscont_fficb(f)))
            cf = cframe_raw(cframe_prev(cf));
          f = frame_prevd(f);
        }
      }
      ins = cframe_pc(cf);
    }
  }
  pt = funcproto(fn);
  pos = proto_bcpos(pt, ins) - 1;
#if LJ_HASJIT
  if (pos > pt->sizebc) {  /* Undo the effects of lj_trace_exit for JLOOP. */
    GCtrace *T = (GCtrace *)((char *)(ins-1) - offsetof(GCtrace, startins));
    lua_assert(bc_isret(bc_op(ins[-1])));
    pos = proto_bcpos(pt, mref(T->startpc, const BCIns));
  }
#endif
  return pos;
}

/* -- Line numbers ---------------------------------------------------------- */

/* Get line number for a bytecode position. */
BCLine LJ_FASTCALL lj_debug_line(GCproto *pt, BCPos pc)
{
  const void *lineinfo = proto_lineinfo(pt);
  if (pc <= pt->sizebc && lineinfo) {
    BCLine first = pt->firstline;
    if (pc == pt->sizebc) return first + pt->numline;
    if (pc-- == 0) return first;
    if (pt->numline < 256)
      return first + (BCLine)((const uint8_t *)lineinfo)[pc];
    else if (pt->numline < 65536)
      return first + (BCLine)((const uint16_t *)lineinfo)[pc];
    else
      return first + (BCLine)((const uint32_t *)lineinfo)[pc];
  }
  return 0;
}

/* Get line number for function/frame. */
static BCLine debug_frameline(lua_State *L, GCfunc *fn, cTValue *nextframe)
{
  BCPos pc = debug_framepc(L, fn, nextframe);
  if (pc != NO_BCPOS) {
    GCproto *pt = funcproto(fn);
    lua_assert(pc <= pt->sizebc);
    return lj_debug_line(pt, pc);
  }
  return -1;
}

/* -- Variable names -------------------------------------------------------- */

/* Get name of a local variable from slot number and PC. */
static const char *debug_varname(const GCproto *pt, BCPos pc, BCReg slot)
{
  const char *p = (const char *)proto_varinfo(pt);
  if (p) {
    BCPos lastpc = 0;
```

```
    for (;;) {
      const char *name = p;
      uint32_t vn = *(const uint8_t *)p;
      BCPos startpc, endpc;
      if (vn < VARNAME__MAX) {
        if (vn == VARNAME_END) break;  /* End of varinfo. */
      } else {
        do { p++; } while (*(const uint8_t *)p);  /* Skip over variable name. */
      }
      p++;
      lastpc = startpc = lastpc + lj_buf_ruleb128(&p);
      if (startpc > pc) break;
      endpc = startpc + lj_buf_ruleb128(&p);
      if (pc < endpc && slot-- == 0) {
        if (vn < VARNAME__MAX) {
#define VARNAMESTR(name, str)        str "\0"
          name = VARNAMEDEF(VARNAMESTR);
#undef VARNAMESTR
          if (--vn) while (*name++ || --vn) ;
        }
        return name;
      }
    }
  }
  return NULL;
}

/* Get name of local variable from 1-based slot number and function/frame. */
static TValue *debug_localname(lua_State *L, const lua_Debug *ar,
                               const char **name, BCReg slot1)
{
  uint32_t offset = (uint32_t)ar->i_ci & 0xffff;
  uint32_t size = (uint32_t)ar->i_ci >> 16;
  TValue *frame = tvref(L->stack) + offset;
  TValue *nextframe = size ? frame + size : NULL;
  GCfunc *fn = frame_func(frame);
  BCPos pc = debug_framepc(L, fn, nextframe);
  if (!nextframe) nextframe = L->top+LJ_FR2;
  if ((int)slot1 < 0) {  /* Negative slot number is for varargs. */
    if (pc != NO_BCPOS) {
      GCproto *pt = funcproto(fn);
      if ((pt->flags & PROTO_VARARG)) {
        slot1 = pt->numparams + (BCReg)(-(int)slot1);
        if (frame_isvarg(frame)) {  /* Vararg frame has been set up? (pc!=0) */
          nextframe = frame;
          frame = frame_prevd(frame);
        }
        if (frame + slot1+LJ_FR2 < nextframe) {
          *name = "(*vararg)";
          return frame+slot1;
        }
      }
    }
    return NULL;
  }
  if (pc != NO_BCPOS &&
      (*name = debug_varname(funcproto(fn), pc, slot1-1)) != NULL)
    ;
  else if (slot1 > 0 && frame + slot1+LJ_FR2 < nextframe)
    *name = "(*temporary)";
  return frame+slot1;
}

/* Get name of upvalue. */
const char *lj_debug_uvname(GCproto *pt, uint32_t idx)
{
  const uint8_t *p = proto_uvinfo(pt);
  lua_assert(idx < pt->sizeuv);
  if (!p) return "";
  if (idx) while (*p++ || --idx) ;
  return (const char *)p;
}

/* Get name and value of upvalue. */
const char *lj_debug_uvnamev(cTValue *o, uint32_t idx, TValue **tvp)
{
```

```
  if (tvisfunc(o)) {
    GCfunc *fn = funcV(o);
    if (isluafunc(fn)) {
      GCproto *pt = funcproto(fn);
      if (idx < pt->sizeuv) {
        *tvp = uvval(&gcref(fn->l.uvptr[idx])->uv);
        return lj_debug_uvname(pt, idx);
      }
    } else {
      if (idx < fn->c.nupvalues) {
        *tvp = &fn->c.upvalue[idx];
        return "";
      }
    }
  }
  return NULL;
}

/* Deduce name of an object from slot number and PC. */
const char *lj_debug_slotname(GCproto *pt, const BCIns *ip, BCReg slot,
                              const char **name)
{
  const char *lname;
restart:
  lname = debug_varname(pt, proto_bcpos(pt, ip), slot);
  if (lname != NULL) { *name = lname; return "local"; }
  while (--ip > proto_bc(pt)) {
    BCIns ins = *ip;
    BCOp op = bc_op(ins);
    BCReg ra = bc_a(ins);
    if (bcmode_a(op) == BCMbase) {
      if (slot >= ra && (op != BC_KNIL || slot <= bc_d(ins)))
        return NULL;
    } else if (bcmode_a(op) == BCMdst && ra == slot) {
      switch (bc_op(ins)) {
      case BC_MOV:
        if (ra == slot) { slot = bc_d(ins); goto restart; }
        break;
      case BC_GGET:
        *name = strdata(gco2str(proto_kgc(pt, ~(ptrdiff_t)bc_d(ins))));
        return "global";
      case BC_TGETS:
        *name = strdata(gco2str(proto_kgc(pt, ~(ptrdiff_t)bc_c(ins))));
        if (ip > proto_bc(pt)) {
          BCIns insp = ip[-1];
          if (bc_op(insp) == BC_MOV && bc_a(insp) == ra+1+LJ_FR2 &&
              bc_d(insp) == bc_b(ins))
            return "method";
        }
        return "field";
      case BC_UGET:
        *name = lj_debug_uvname(pt, bc_d(ins));
        return "upvalue";
      default:
        return NULL;
      }
    }
  }
  return NULL;
}

/* Deduce function name from caller of a frame. */
const char *lj_debug_funcname(lua_State *L, cTValue *frame, const char **name)
{
  cTValue *pframe;
  GCfunc *fn;
  BCPos pc;
  if (frame <= tvref(L->stack)+LJ_FR2)
    return NULL;
  if (frame_isvarg(frame))
    frame = frame_prevd(frame);
  pframe = frame_prev(frame);
  fn = frame_func(pframe);
  pc = debug_framepc(L, fn, frame);
  if (pc != NO_BCPOS) {
    GCproto *pt = funcproto(fn);
```

```
    const BCIns *ip = &proto_bc(pt)[check_exp(pc < pt->sizebc, pc)];
    MMS mm = bcmode_mm(bc_op(*ip));
    if (mm == MM_call) {
      BCReg slot = bc_a(*ip);
      if (bc_op(*ip) == BC_ITERC) slot -= 3;
      return lj_debug_slotname(pt, ip, slot, name);
    } else if (mm != MM__MAX) {
      *name = strdata(mmname_str(G(L), mm));
      return "metamethod";
    }
  }
  return NULL;
}

/* -- Source code locations ---------------------------------------------- */

/* Generate shortened source name. */
void lj_debug_shortname(char *out, GCstr *str, BCLine line)
{
  const char *src = strdata(str);
  if (*src == '=') {
    strncpy(out, src+1, LUA_IDSIZE);  /* Remove first char. */
    out[LUA_IDSIZE-1] = '\0';  /* Ensures null termination. */
  } else if (*src == '@') {  /* Output "source", or "...source". */
    size_t len = str->len-1;
    src++;  /* Skip the `@' */
    if (len >= LUA_IDSIZE) {
      src += len-(LUA_IDSIZE-4);  /* Get last part of file name. */
      *out++ = '.'; *out++ = '.'; *out++ = '.';
    }
    strcpy(out, src);
  } else {  /* Output [string "string"] or [builtin:name]. */
    size_t len;  /* Length, up to first control char. */
    for (len = 0; len < LUA_IDSIZE-12; len++)
      if (((const unsigned char *)src)[len] < ' ') break;
    strcpy(out, line == ~(BCLine)0 ? "[builtin:" : "[string \""); out += 9;
    if (src[len] != '\0') {  /* Must truncate? */
      if (len > LUA_IDSIZE-15) len = LUA_IDSIZE-15;
      strncpy(out, src, len); out += len;
      strcpy(out, "..."); out += 3;
    } else {
      strcpy(out, src); out += len;
    }
    strcpy(out, line == ~(BCLine)0 ? "]" : "\"]");
  }
}

/* Add current location of a frame to error message. */
void lj_debug_addloc(lua_State *L, const char *msg,
                     cTValue *frame, cTValue *nextframe)
{
  if (frame) {
    GCfunc *fn = frame_func(frame);
    if (isluafunc(fn)) {
      BCLine line = debug_frameline(L, fn, nextframe);
      if (line >= 0) {
        GCproto *pt = funcproto(fn);
        char buf[LUA_IDSIZE];
        lj_debug_shortname(buf, proto_chunkname(pt), pt->firstline);
        lj_strfmt_pushf(L, "%s:%d: %s", buf, line, msg);
        return;
      }
    }
  }
  lj_strfmt_pushf(L, "%s", msg);
}

/* Push location string for a bytecode position to Lua stack. */
void lj_debug_pushloc(lua_State *L, GCproto *pt, BCPos pc)
{
  GCstr *name = proto_chunkname(pt);
  const char *s = strdata(name);
  MSize i, len = name->len;
  BCLine line = lj_debug_line(pt, pc);
  if (pt->firstline == ~(BCLine)0) {
    lj_strfmt_pushf(L, "builtin:%s", s);
```

```
  } else if (*s == '@') {
    s++; len--;
    for (i = len; i > 0; i--)
      if (s[i] == '/' || s[i] == '\\') {
        s += i+1;
        break;
      }
    lj_strfmt_pushf(L, "%s:%d", s, line);
  } else if (len > 40) {
    lj_strfmt_pushf(L, "%p:%d", pt, line);
  } else if (*s == '=') {
    lj_strfmt_pushf(L, "%s:%d", s+1, line);
  } else {
    lj_strfmt_pushf(L, "\"%s\":%d", s, line);
  }
}

/* -- Public debug API ---------------------------------------------------- */

/* lua_getupvalue() and lua_setupvalue() are in lj_api.c. */

LUA_API const char *lua_getlocal(lua_State *L, const lua_Debug *ar, int n)
{
  const char *name = NULL;
  if (ar) {
    TValue *o = debug_localname(L, ar, &name, (BCReg)n);
    if (name) {
      copyTV(L, L->top, o);
      incr_top(L);
    }
  } else if (tvisfunc(L->top-1) && isluafunc(funcV(L->top-1))) {
    name = debug_varname(funcproto(funcV(L->top-1)), 0, (BCReg)n-1);
  }
  return name;
}

LUA_API const char *lua_setlocal(lua_State *L, const lua_Debug *ar, int n)
{
  const char *name = NULL;
  TValue *o = debug_localname(L, ar, &name, (BCReg)n);
  if (name)
    copyTV(L, o, L->top-1);
  L->top--;
  return name;
}

int lj_debug_getinfo(lua_State *L, const char *what, lj_Debug *ar, int ext)
{
  int opt_f = 0, opt_L = 0;
  TValue *frame = NULL;
  TValue *nextframe = NULL;
  GCfunc *fn;
  if (*what == '>') {
    TValue *func = L->top - 1;
    api_check(L, tvisfunc(func));
    fn = funcV(func);
    L->top--;
    what++;
  } else {
    uint32_t offset = (uint32_t)ar->i_ci & 0xffff;
    uint32_t size = (uint32_t)ar->i_ci >> 16;
    lua_assert(offset != 0);
    frame = tvref(L->stack) + offset;
    if (size) nextframe = frame + size;
    lua_assert(frame <= tvref(L->maxstack) &&
               (!nextframe || nextframe <= tvref(L->maxstack)));
    fn = frame_func(frame);
    lua_assert(fn->c.gct == ~LJ_TFUNC);
  }
  for (; *what; what++) {
    if (*what == 'S') {
      if (isluafunc(fn)) {
        GCproto *pt = funcproto(fn);
        BCLine firstline = pt->firstline;
        GCstr *name = proto_chunkname(pt);
        ar->source = strdata(name);
```

```
        lj_debug_shortname(ar->short_src, name, pt->firstline);
        ar->linedefined = (int)firstline;
        ar->lastlinedefined = (int)(firstline + pt->numline);
        ar->what = (firstline || !pt->numline) ? "Lua" : "main";
      } else {
        ar->source = "=[C]";
        ar->short_src[0] = '[';
        ar->short_src[1] = 'C';
        ar->short_src[2] = ']';
        ar->short_src[3] = '\0';
        ar->linedefined = -1;
        ar->lastlinedefined = -1;
        ar->what = "C";
      }
    } else if (*what == 'l') {
      ar->currentline = frame ? lj_debug_frameline(L, fn, nextframe) : -1;
    } else if (*what == 'u') {
      ar->nups = fn->c.nupvalues;
      if (ext) {
        if (isluafunc(fn)) {
          GCproto *pt = funcproto(fn);
          ar->nparams = pt->numparams;
          ar->isvararg = !!(pt->flags & PROTO_VARARG);
        } else {
          ar->nparams = 0;
          ar->isvararg = 1;
        }
      }
    } else if (*what == 'n') {
      ar->namewhat = frame ? lj_debug_funcname(L, frame, &ar->name) : NULL;
      if (ar->namewhat == NULL) {
        ar->namewhat = "";
        ar->name = NULL;
      }
    } else if (*what == 'f') {
      opt_f = 1;
    } else if (*what == 'L') {
      opt_L = 1;
    } else {
      return 0;  /* Bad option. */
    }
  }
  if (opt_f) {
    setfuncV(L, L->top, fn);
    incr_top(L);
  }
  if (opt_L) {
    if (isluafunc(fn)) {
      GCtab *t = lj_tab_new(L, 0, 0);
      GCproto *pt = funcproto(fn);
      const void *lineinfo = proto_lineinfo(pt);
      if (lineinfo) {
        BCLine first = pt->firstline;
        int sz = pt->numline < 256 ? 1 : pt->numline < 65536 ? 2 : 4;
        MSize i, szl = pt->sizebc-1;
        for (i = 0; i < szl; i++) {
          BCLine line = first +
            (sz == 1 ? (BCLine)((const uint8_t *)lineinfo)[i] :
             sz == 2 ? (BCLine)((const uint16_t *)lineinfo)[i] :
             (BCLine)((const uint32_t *)lineinfo)[i]);
          setboolV(lj_tab_setint(L, t, line), 1);
        }
      }
      settabV(L, L->top, t);
    } else {
      setnilV(L->top);
    }
    incr_top(L);
  }
  return 1;  /* Ok. */
}

LUA_API int lua_getinfo(lua_State *L, const char *what, lua_Debug *ar)
{
  return lj_debug_getinfo(L, what, (lj_Debug *)ar, 0);
}
```

```c

LUA_API int lua_getstack(lua_State *L, int level, lua_Debug *ar)
{
  int size;
  cTValue *frame = lj_debug_frame(L, level, &size);
  if (frame) {
    ar->i_ci = (size << 16) + (int)(frame - tvref(L->stack));
    return 1;
  } else {
    ar->i_ci = level - size;
    return 0;
  }
}

#if LJ_HASPROFILE
/* Put the chunkname into a buffer. */
static int debug_putchunkname(SBuf *sb, GCproto *pt, int pathstrip)
{
  GCstr *name = proto_chunkname(pt);
  const char *p = strdata(name);
  if (pt->firstline == ~(BCLine)0) {
    lj_buf_putmem(sb, "[builtin:", 9);
    lj_buf_putstr(sb, name);
    lj_buf_putb(sb, ']');
    return 0;
  }
  if (*p == '=' || *p == '@') {
    MSize len = name->len-1;
    p++;
    if (pathstrip) {
      int i;
      for (i = len-1; i >= 0; i--)
        if (p[i] == '/' || p[i] == '\\') {
          len -= i+1;
          p = p+i+1;
          break;
        }
    }
    lj_buf_putmem(sb, p, len);
  } else {
    lj_buf_putmem(sb, "[string]", 8);
  }
  return 1;
}

/* Put a compact stack dump into a buffer. */
void lj_debug_dumpstack(lua_State *L, SBuf *sb, const char *fmt, int depth)
{
  int level = 0, dir = 1, pathstrip = 1;
  MSize lastlen = 0;
  if (depth < 0) { level = ~depth; depth = dir = -1; }  /* Reverse frames. */
  while (level != depth) {  /* Loop through all frame. */
    int size;
    cTValue *frame = lj_debug_frame(L, level, &size);
    if (frame) {
      cTValue *nextframe = size ? frame+size : NULL;
      GCfunc *fn = frame_func(frame);
      const uint8_t *p = (const uint8_t *)fmt;
      int c;
      while ((c = *p++)) {
        switch (c) {
        case 'p':  /* Preserve full path. */
          pathstrip = 0;
          break;
        case 'F': case 'f': {  /* Dump function name. */
          const char *name;
          const char *what = lj_debug_funcname(L, frame, &name);
          if (what) {
            if (c == 'F' && isluafunc(fn)) {  /* Dump module:name for 'F'. */
              GCproto *pt = funcproto(fn);
              if (pt->firstline != ~(BCLine)0) {  /* Not a bytecode builtin. */
                debug_putchunkname(sb, pt, pathstrip);
                lj_buf_putb(sb, ':');
              }
            }
            lj_buf_putmem(sb, name, (MSize)strlen(name));
```

```
            break;
          }  /* else: can't derive a name, dump module:line. */
          }
          /* fallthrough */
        case 'l':  /* Dump module:line. */
          if (isluafunc(fn)) {
            GCproto *pt = funcproto(fn);
            if (debug_putchunkname(sb, pt, pathstrip)) {
              /* Regular Lua function. */
              BCLine line = c == 'l' ? debug_frameline(L, fn, nextframe) :
                                       pt->firstline;
              lj_buf_putb(sb, ':');
              lj_strfmt_putint(sb, line >= 0 ? line : pt->firstline);
            }
          } else if (isffunc(fn)) {  /* Dump numbered builtins. */
            lj_buf_putmem(sb, "[builtin#", 9);
            lj_strfmt_putint(sb, fn->c.ffid);
            lj_buf_putb(sb, ']');
          } else {  /* Dump C function address. */
            lj_buf_putb(sb, '@');
            lj_strfmt_putptr(sb, fn->c.f);
          }
          break;
        case 'Z':  /* Zap trailing separator. */
          lastlen = sbuflen(sb);
          break;
        default:
          lj_buf_putb(sb, c);
          break;
        }
      }
    } else if (dir == 1) {
      break;
    } else {
      level -= size;  /* Reverse frame order: quickly skip missing level. */
    }
    level += dir;
  }
  if (lastlen)
    setsbufP(sb, sbufB(sb) + lastlen);  /* Zap trailing separator. */
}
#endif

/* Number of frames for the leading and trailing part of a traceback. */
#define TRACEBACK_LEVELS1        12
#define TRACEBACK_LEVELS2        10

LUALIB_API void luaL_traceback (lua_State *L, lua_State *L1, const char *msg,
                                int level)
{
  int top = (int)(L->top - L->base);
  int lim = TRACEBACK_LEVELS1;
  lua_Debug ar;
  if (msg) lua_pushfstring(L, "%s\n", msg);
  lua_pushliteral(L, "stack traceback:");
  while (lua_getstack(L1, level++, &ar)) {
    GCfunc *fn;
    if (level > lim) {
      if (!lua_getstack(L1, level + TRACEBACK_LEVELS2, &ar)) {
        level--;
      } else {
        lua_pushliteral(L, "\n\t...");
        lua_getstack(L1, -10, &ar);
        level = ar.i_ci - TRACEBACK_LEVELS2;
      }
      lim = 2147483647;
      continue;
    }
    lua_getinfo(L1, "Snlf", &ar);
    fn = funcV(L1->top-1); L1->top--;
    if (isffunc(fn) && !*ar.namewhat)
      lua_pushfstring(L, "\n\t[builtin#%d]:", fn->c.ffid);
    else
      lua_pushfstring(L, "\n\t%s:", ar.short_src);
    if (ar.currentline > 0)
      lua_pushfstring(L, "%d:", ar.currentline);
```

```
682     if (*ar.namewhat) {
683       lua_pushfstring(L, " in function " LUA_QS, ar.name);
684     } else {
685       if (*ar.what == 'm') {
686         lua_pushliteral(L, " in main chunk");
687       } else if (*ar.what == 'C') {
688         lua_pushfstring(L, " at %p", fn->c.f);
689       } else {
690         lua_pushfstring(L, " in function <%s:%d>",
691                         ar.short_src, ar.linedefined);
692       }
693     }
694     if ((int)(L->top - L->base) - top >= 15)
695       lua_concat(L, (int)(L->top - L->base) - top);
696   }
697   lua_concat(L, (int)(L->top - L->base) - top);
698 }
699
```

One Level Up Top Level

src/lj_debug.h - luajit-2.0-src

Data types defined

- lj_Debug
- lj_Debug

Macros defined

- VARNAMEDEF
- VARNAMEENUM
- VARNAMEENUM
- _LJ_DEBUG_H

Source code

```
1  /*
2  ** Debugging and introspection.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_DEBUG_H
7  #define _LJ_DEBUG_H
8
9  #include "lj_obj.h"
10
11 typedef struct lj_Debug {
12   /* Common fields. Must be in the same order as in lua.h. */
13   int event;
14   const char *name;
15   const char *namewhat;
16   const char *what;
17   const char *source;
18   int currentline;
19   int nups;
20   int linedefined;
21   int lastlinedefined;
22   char short_src[LUA_IDSIZE];
23   int i_ci;
24   /* Extended fields. Only valid if lj_debug_getinfo() is called with ext = 1.*/
25   int nparams;
26   int isvararg;
27 } lj_Debug;
28
29 LJ_FUNC cTValue *lj_debug_frame(lua_State *L, int level, int *size);
30 LJ_FUNC BCLine LJ_FASTCALL lj_debug_line(GCproto *pt, BCPos pc);
31 LJ_FUNC const char *lj_debug_uvname(GCproto *pt, uint32_t idx);
32 LJ_FUNC const char *lj_debug_uvnamev(cTValue *o, uint32_t idx, TValue **tvp);
33 LJ_FUNC const char *lj_debug_slotname(GCproto *pt, const BCIns *pc,
34                                       BCReg slot, const char **name);
35 LJ_FUNC const char *lj_debug_funcname(lua_State *L, cTValue *frame,
36                                       const char **name);
37 LJ_FUNC void lj_debug_shortname(char *out, GCstr *str, BCLine line);
38 LJ_FUNC void lj_debug_addloc(lua_State *L, const char *msg,
39                              cTValue *frame, cTValue *nextframe);
40 LJ_FUNC void lj_debug_pushloc(lua_State *L, GCproto *pt, BCPos pc);
41 LJ_FUNC int lj_debug_getinfo(lua_State *L, const char *what, lj_Debug *ar,
42                              int ext);
43 #if LJ_HASPROFILE
44 LJ_FUNC void lj_debug_dumpstack(lua_State *L, SBuf *sb, const char *fmt,
45                                 int depth);
46 #endif
47
48 /* Fixed internal variable names. */
```

```
#define VARNAMEDEF(_) \
  _(FOR_IDX, "(for index)") \
  _(FOR_STOP, "(for limit)") \
  _(FOR_STEP, "(for step)") \
  _(FOR_GEN, "(for generator)") \
  _(FOR_STATE, "(for state)") \
  _(FOR_CTL, "(for control)")

enum {
  VARNAME_END,
#define VARNAMEENUM(name, str)        VARNAME_##name,
  VARNAMEDEF(VARNAMEENUM)
#undef VARNAMEENUM
  VARNAME__MAX
};

#endif
```

One Level Up Top Level

src/lib_debug.c - luajit-2.0-src

Global variables defined

- KEY_HOOK

Functions defined

- LJLIB_CF(debug_getregistry)
- LJLIB_CF(debug_getmetatable) LJLIB_REC(.)
- LJLIB_CF(debug_setmetatable)
- LJLIB_CF(debug_getfenv)
- LJLIB_CF(debug_setfenv)
- LJLIB_CF(debug_getinfo)
- LJLIB_CF(debug_getlocal)
- LJLIB_CF(debug_setlocal)
- LJLIB_CF(debug_getupvalue)
- LJLIB_CF(debug_setupvalue)
- LJLIB_CF(debug_upvalueid)
- LJLIB_CF(debug_upvaluejoin)
- LJLIB_CF(debug_getuservalue)
- LJLIB_CF(debug_setuservalue)
- LJLIB_CF(debug_sethook)
- LJLIB_CF(debug_gethook)
- LJLIB_CF(debug_debug)
- LJLIB_CF(debug_traceback)
- debug_getupvalue
- getthread
- hookf
- luaopen_debug
- makemask
- settabsb
- settabsi
- settabss
- treatstackoption

- unmakemask

Macros defined

- LEVELS1
- LEVELS2
- LJLIB_MODULE_debug
- LUA_LIB
- lib_debug_c

Source code

```
/*
** Debug library.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Major portions taken verbatim or adapted from the Lua interpreter.
** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
*/

#define lib_debug_c
 #define LUA_LIB

 #include "lua.h"
 #include "lauxlib.h"
 #include "lualib.h"

 #include "lj_obj.h"
 #include "lj_gc.h"
 #include "lj_err.h"
 #include "lj_debug.h"
 #include "lj_lib.h"

 /* ------------------------------------------------------------------------ */

 #define LJLIB_MODULE_debug

 LJLIB_CF(debug_getregistry)
 {
   copyTV(L, L->top++, registry(L));
   return 1;
 }

 LJLIB_CF(debug_getmetatable)        LJLIB_REC(.)
 {
   lj_lib_checkany(L, 1);
   if (!lua_getmetatable(L, 1)) {
     setnilV(L->top-1);
   }
   return 1;
 }

 LJLIB_CF(debug_setmetatable)
 {
   lj_lib_checktabornil(L, 2);
   L->top = L->base+2;
   lua_setmetatable(L, 1);
 #if !LJ_52
   setboolV(L->top-1, 1);
 #endif
   return 1;
 }

 LJLIB_CF(debug_getfenv)
 {
   lj_lib_checkany(L, 1);
   lua_getfenv(L, 1);
```

```
  return 1;
}

LJLIB_CF(debug_setfenv)
{
  lj_lib_checktab(L, 2);
  L->top = L->base+2;
  if (!lua_setfenv(L, 1))
    lj_err_caller(L, LJ_ERR_SETFENV);
  return 1;
}

/* ------------------------------------------------------------------------ */

static void settabss(lua_State *L, const char *i, const char *v)
{
  lua_pushstring(L, v);
  lua_setfield(L, -2, i);
}

static void settabsi(lua_State *L, const char *i, int v)
{
  lua_pushinteger(L, v);
  lua_setfield(L, -2, i);
}

static void settabsb(lua_State *L, const char *i, int v)
{
  lua_pushboolean(L, v);
  lua_setfield(L, -2, i);
}

static lua_State *getthread(lua_State *L, int *arg)
{
  if (L->base < L->top && tvisthread(L->base)) {
    *arg = 1;
    return threadV(L->base);
  } else {
    *arg = 0;
    return L;
  }
}

static void treatstackoption(lua_State *L, lua_State *L1, const char *fname)
 {
   if (L == L1) {
     lua_pushvalue(L, -2);
     lua_remove(L, -3);
   }
   else
     lua_xmove(L1, L, 1);
   lua_setfield(L, -2, fname);
 }

 LJLIB_CF(debug_getinfo)
 {
   lj_Debug ar;
   int arg, opt_f = 0, opt_L = 0;
   lua_State *L1 = getthread(L, &arg);
   const char *options = luaL_optstring(L, arg+2, "flnSu");
   if (lua_isnumber(L, arg+1)) {
     if (!lua_getstack(L1, (int)lua_tointeger(L, arg+1), (lua_Debug *)&ar)) {
       setnilV(L->top-1);
       return 1;
     }
   } else if (L->base+arg < L->top && tvisfunc(L->base+arg)) {
     options = lua_pushfstring(L, ">%s", options);
     setfuncV(L1, L1->top++, funcV(L->base+arg));
   } else {
     lj_err_arg(L, arg+1, LJ_ERR_NOFUNCL);
   }
   if (!lj_debug_getinfo(L1, options, &ar, 1))
     lj_err_arg(L, arg+2, LJ_ERR_INVOPT);
   lua_createtable(L, 0, 16);  /* Create result table. */
   for (; *options; options++) {
     switch (*options) {
```

```
    case 'S':
      settabss(L, "source", ar.source);
      settabss(L, "short_src", ar.short_src);
      settabsi(L, "linedefined", ar.linedefined);
      settabsi(L, "lastlinedefined", ar.lastlinedefined);
      settabss(L, "what", ar.what);
      break;
    case 'l':
      settabsi(L, "currentline", ar.currentline);
      break;
    case 'u':
      settabsi(L, "nups", ar.nups);
      settabsi(L, "nparams", ar.nparams);
      settabsb(L, "isvararg", ar.isvararg);
      break;
    case 'n':
      settabss(L, "name", ar.name);
      settabss(L, "namewhat", ar.namewhat);
      break;
    case 'f': opt_f = 1; break;
    case 'L': opt_L = 1; break;
    default: break;
    }
  }
  if (opt_L) treatstackoption(L, L1, "activelines");
  if (opt_f) treatstackoption(L, L1, "func");
  return 1;  /* Return result table. */
}

LJLIB_CF(debug_getlocal)
{
  int arg;
  lua_State *L1 = getthread(L, &arg);
  lua_Debug ar;
  const char *name;
  int slot = lj_lib_checkint(L, arg+2);
  if (tvisfunc(L->base+arg)) {
    L->top = L->base+arg+1;
    lua_pushstring(L, lua_getlocal(L, NULL, slot));
    return 1;
  }
  if (!lua_getstack(L1, lj_lib_checkint(L, arg+1), &ar))
    lj_err_arg(L, arg+1, LJ_ERR_LVLRNG);
  name = lua_getlocal(L1, &ar, slot);
  if (name) {
    lua_xmove(L1, L, 1);
    lua_pushstring(L, name);
    lua_pushvalue(L, -2);
    return 2;
  } else {
    setnilV(L->top-1);
    return 1;
  }
}

LJLIB_CF(debug_setlocal)
{
  int arg;
  lua_State *L1 = getthread(L, &arg);
  lua_Debug ar;
  TValue *tv;
  if (!lua_getstack(L1, lj_lib_checkint(L, arg+1), &ar))
    lj_err_arg(L, arg+1, LJ_ERR_LVLRNG);
  tv = lj_lib_checkany(L, arg+3);
  copyTV(L1, L1->top++, tv);
  lua_pushstring(L, lua_setlocal(L1, &ar, lj_lib_checkint(L, arg+2)));
  return 1;
}

static int debug_getupvalue(lua_State *L, int get)
{
  int32_t n = lj_lib_checkint(L, 2);
  const char *name;
  lj_lib_checkfunc(L, 1);
  name = get ? lua_getupvalue(L, 1, n) : lua_setupvalue(L, 1, n);
  if (name) {
```

```
    lua_pushstring(L, name);
    if (!get) return 1;
    copyTV(L, L->top, L->top-2);
    L->top++;
    return 2;
  }
  return 0;
}

LJLIB_CF(debug_getupvalue)
{
  return debug_getupvalue(L, 1);
}

LJLIB_CF(debug_setupvalue)
{
  lj_lib_checkany(L, 3);
  return debug_getupvalue(L, 0);
}

LJLIB_CF(debug_upvalueid)
{
  GCfunc *fn = lj_lib_checkfunc(L, 1);
  int32_t n = lj_lib_checkint(L, 2) - 1;
  if ((uint32_t)n >= fn->l.nupvalues)
    lj_err_arg(L, 2, LJ_ERR_IDXRNG);
  setlightudV(L->top-1, isluafunc(fn) ? (void *)gcref(fn->l.uvptr[n]) :
                                        (void *)&fn->c.upvalue[n]);
  return 1;
}

LJLIB_CF(debug_upvaluejoin)
{
  GCfunc *fn[2];
  GCRef *p[2];
  int i;
  for (i = 0; i < 2; i++) {
    int32_t n;
    fn[i] = lj_lib_checkfunc(L, 2*i+1);
    if (!isluafunc(fn[i]))
      lj_err_arg(L, 2*i+1, LJ_ERR_NOLFUNC);
    n = lj_lib_checkint(L, 2*i+2) - 1;
    if ((uint32_t)n >= fn[i]->l.nupvalues)
      lj_err_arg(L, 2*i+2, LJ_ERR_IDXRNG);
    p[i] = &fn[i]->l.uvptr[n];
  }
  setgcrefr(*p[0], *p[1]);
  lj_gc_objbarrier(L, fn[0], gcref(*p[1]));
  return 0;
}

#if LJ_52
LJLIB_CF(debug_getuservalue)
{
  TValue *o = L->base;
  if (o < L->top && tvisudata(o))
    settabV(L, o, tabref(udataV(o)->env));
  else
    setnilV(o);
  L->top = o+1;
  return 1;
}

LJLIB_CF(debug_setuservalue)
{
  TValue *o = L->base;
  if (!(o < L->top && tvisudata(o)))
    lj_err_argt(L, 1, LUA_TUSERDATA);
  if (!(o+1 < L->top && tvistab(o+1)))
    lj_err_argt(L, 2, LUA_TTABLE);
  L->top = o+2;
  lua_setfenv(L, 1);
  return 1;
}
#endif

```

```c
/* ------------------------------------------------------------------------ */

static const char KEY_HOOK = 'h';

static void hookf(lua_State *L, lua_Debug *ar)
{
  static const char *const hooknames[] =
    {"call", "return", "line", "count", "tail return"};
  lua_pushlightuserdata(L, (void *)&KEY_HOOK);
  lua_rawget(L, LUA_REGISTRYINDEX);
  if (lua_isfunction(L, -1)) {
    lua_pushstring(L, hooknames[(int)ar->event]);
    if (ar->currentline >= 0)
      lua_pushinteger(L, ar->currentline);
    else lua_pushnil(L);
    lua_call(L, 2, 0);
  }
}

static int makemask(const char *smask, int count)
{
  int mask = 0;
  if (strchr(smask, 'c')) mask |= LUA_MASKCALL;
  if (strchr(smask, 'r')) mask |= LUA_MASKRET;
  if (strchr(smask, 'l')) mask |= LUA_MASKLINE;
  if (count > 0) mask |= LUA_MASKCOUNT;
  return mask;
}

static char *unmakemask(int mask, char *smask)
{
  int i = 0;
  if (mask & LUA_MASKCALL) smask[i++] = 'c';
  if (mask & LUA_MASKRET) smask[i++] = 'r';
  if (mask & LUA_MASKLINE) smask[i++] = 'l';
  smask[i] = '\0';
  return smask;
}

LJLIB_CF(debug_sethook)
{
  int arg, mask, count;
  lua_Hook func;
  (void)getthread(L, &arg);
  if (lua_isnoneornil(L, arg+1)) {
    lua_settop(L, arg+1);
    func = NULL; mask = 0; count = 0;  /* turn off hooks */
  } else {
    const char *smask = luaL_checkstring(L, arg+2);
    luaL_checktype(L, arg+1, LUA_TFUNCTION);
    count = luaL_optint(L, arg+3, 0);
    func = hookf; mask = makemask(smask, count);
  }
  lua_pushlightuserdata(L, (void *)&KEY_HOOK);
  lua_pushvalue(L, arg+1);
  lua_rawset(L, LUA_REGISTRYINDEX);
  lua_sethook(L, func, mask, count);
  return 0;
}

LJLIB_CF(debug_gethook)
{
  char buff[5];
  int mask = lua_gethookmask(L);
  lua_Hook hook = lua_gethook(L);
  if (hook != NULL && hook != hookf) {  /* external hook? */
    lua_pushliteral(L, "external hook");
  } else {
    lua_pushlightuserdata(L, (void *)&KEY_HOOK);
    lua_rawget(L, LUA_REGISTRYINDEX);   /* get hook */
  }
  lua_pushstring(L, unmakemask(mask, buff));
  lua_pushinteger(L, lua_gethookcount(L));
  return 3;
}
```

```
/* ------------------------------------------------------------------------ */

LJLIB_CF(debug_debug)
{
  for (;;) {
    char buffer[250];
    fputs("lua_debug> ", stderr);
    if (fgets(buffer, sizeof(buffer), stdin) == 0 ||
        strcmp(buffer, "cont\n") == 0)
      return 0;
    if (luaL_loadbuffer(L, buffer, strlen(buffer), "=(debug command)") ||
        lua_pcall(L, 0, 0, 0)) {
      fputs(lua_tostring(L, -1), stderr);
      fputs("\n", stderr);
    }
    lua_settop(L, 0);  /* remove eventual returns */
  }
}

/* ------------------------------------------------------------------------ */

#define LEVELS1        12        /* size of the first part of the stack */
#define LEVELS2        10        /* size of the second part of the stack */

LJLIB_CF(debug_traceback)
{
  int arg;
  lua_State *L1 = getthread(L, &arg);
  const char *msg = lua_tostring(L, arg+1);
  if (msg == NULL && L->top > L->base+arg)
    L->top = L->base+arg+1;
  else
    luaL_traceback(L, L1, msg, lj_lib_optint(L, arg+2, (L == L1)));
  return 1;
}

/* ------------------------------------------------------------------------ */

#include "lj_libdef.h"

LUALIB_API int luaopen_debug(lua_State *L)
{
  LJ_LIB_REG(L, LUA_DBLIBNAME, debug);
  return 1;
}

```

One Level Up Top Level

src/lib_bit.c - luajit-2.0-src

Functions defined

- LJLIB_ASM(bit_tobit) LJLIB_REC(bit_tobit)
- LJLIB_ASM(bit_bnot) LJLIB_REC(bit_unary IR_BNOT)
- LJLIB_ASM(bit_bswap) LJLIB_REC(bit_unary IR_BSWAP)
- LJLIB_ASM(bit_lshift) LJLIB_REC(bit_shift IR_BSHL)
- LJLIB_ASM_(bit_rshift) LJLIB_REC(bit_shift IR_BSHR)
- LJLIB_ASM_(bit_bor) LJLIB_REC(bit_nary IR_BOR)
- bit_checkbit
- bit_result64
- luaopen_bit

Macros defined

- LJLIB_MODULE_bit
- LUA_LIB
- lib_bit_c

Source code

```
1  /*
2  ** Bit manipulation library.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lib_bit_c
7  #define LUA_LIB
8
9  #include "lua.h"
10  #include "lauxlib.h"
11  #include "lualib.h"
12
13  #include "lj_obj.h"
14  #include "lj_err.h"
15  #include "lj_buf.h"
16  #include "lj_strscan.h"
17  #include "lj_strfmt.h"
18  #if LJ_HASFFI
19  #include "lj_ctype.h"
20  #include "lj_cdata.h"
21  #include "lj_cconv.h"
22  #include "lj_carith.h"
23  #endif
24  #include "lj_ff.h"
25  #include "lj_lib.h"
26
27  /* ------------------------------------------------------------------------ */
28
29  #define LJLIB_MODULE_bit
30
31  #if LJ_HASFFI
32  static int bit_result64(lua_State *L, CTypeID id, uint64_t x)
33  {
34    GCcdata *cd = lj_cdata_new_(L, id, 8);
```

```
  *(uint64_t *)cdataptr(cd) = x;
  setcdataV(L, L->base-1-LJ_FR2, cd);
  return FFH_RES(1);
}
#else
static int32_t bit_checkbit(lua_State *L, int narg)
{
  TValue *o = L->base + narg-1;
  if (!(o < L->top && lj_strscan_numberobj(o)))
    lj_err_argt(L, narg, LUA_TNUMBER);
  if (LJ_LIKELY(tvisint(o))) {
    return intV(o);
  } else {
    int32_t i = lj_num2bit(numV(o));
    if (LJ_DUALNUM) setintV(o, i);
    return i;
  }
}
#endif

LJLIB_ASM(bit_tobit)                LJLIB_REC(bit_tobit)
{
#if LJ_HASFFI
  CTypeID id = 0;
  setintV(L->base-1-LJ_FR2, (int32_t)lj_carith_check64(L, 1, &id));
  return FFH_RES(1);
#else
  lj_lib_checknumber(L, 1);
  return FFH_RETRY;
#endif
}

LJLIB_ASM(bit_bnot)                LJLIB_REC(bit_unary IR_BNOT)
{
#if LJ_HASFFI
  CTypeID id = 0;
  uint64_t x = lj_carith_check64(L, 1, &id);
  return id ? bit_result64(L, id, ~x) : FFH_RETRY;
#else
  lj_lib_checknumber(L, 1);
  return FFH_RETRY;
#endif
}

LJLIB_ASM(bit_bswap)                LJLIB_REC(bit_unary IR_BSWAP)
{
#if LJ_HASFFI
  CTypeID id = 0;
  uint64_t x = lj_carith_check64(L, 1, &id);
  return id ? bit_result64(L, id, lj_bswap64(x)) : FFH_RETRY;
#else
  lj_lib_checknumber(L, 1);
  return FFH_RETRY;
#endif
}

LJLIB_ASM(bit_lshift)                LJLIB_REC(bit_shift IR_BSHL)
{
#if LJ_HASFFI
  CTypeID id = 0, id2 = 0;
  uint64_t x = lj_carith_check64(L, 1, &id);
  int32_t sh = (int32_t)lj_carith_check64(L, 2, &id2);
  if (id) {
    x = lj_carith_shift64(x, sh, curr_func(L)->c.ffid - (int)FF_bit_lshift);
    return bit_result64(L, id, x);
   }
   if (id2) setintV(L->base+1, sh);
   return FFH_RETRY;
 #else
   lj_lib_checknumber(L, 1);
   bit_checkbit(L, 2);
   return FFH_RETRY;
 #endif
 }
 LJLIB_ASM_(bit_rshift)                LJLIB_REC(bit_shift IR_BSHR)
 LJLIB_ASM_(bit_arshift)                LJLIB_REC(bit_shift IR_BSAR)
```

```
LJLIB_ASM_(bit_rol)                LJLIB_REC(bit_shift IR_BROL)
LJLIB_ASM_(bit_ror)                LJLIB_REC(bit_shift IR_BROR)

LJLIB_ASM(bit_band)                LJLIB_REC(bit_nary IR_BAND)
{
#if LJ_HASFFI
  CTypeID id = 0;
  TValue *o = L->base, *top = L->top;
  int i = 0;
  do { lj_carith_check64(L, ++i, &id); } while (++o < top);
  if (id) {
    CTState *cts = ctype_cts(L);
    CType *ct = ctype_get(cts, id);
    int op = curr_func(L)->c.ffid - (int)FF_bit_bor;
    uint64_t x, y = op >= 0 ? 0 : ~(uint64_t)0;
    o = L->base;
    do {
      lj_cconv_ct_tv(cts, ct, (uint8_t *)&x, o, 0);
      if (op < 0) y &= x; else if (op == 0) y |= x; else y ^= x;
    } while (++o < top);
    return bit_result64(L, id, y);
  }
  return FFH_RETRY;
#else
  int i = 0;
  do { lj_lib_checknumber(L, ++i); } while (L->base+i < L->top);
  return FFH_RETRY;
#endif
}
LJLIB_ASM_(bit_bor)                LJLIB_REC(bit_nary IR_BOR)
LJLIB_ASM_(bit_bxor)               LJLIB_REC(bit_nary IR_BXOR)

/* ------------------------------------------------------------------------ */

LJLIB_CF(bit_tohex)                LJLIB_REC(.)
{
#if LJ_HASFFI
  CTypeID id = 0, id2 = 0;
  uint64_t b = lj_carith_check64(L, 1, &id);
  int32_t n = L->base+1>=L->top ? (id ? 16 : 8) :
                                  (int32_t)lj_carith_check64(L, 2, &id2);
#else
  uint32_t b = (uint32_t)bit_checkbit(L, 1);
  int32_t n = L->base+1>=L->top ? 8 : bit_checkbit(L, 2);
#endif
  SBuf *sb = lj_buf_tmp_(L);
  SFormat sf = (STRFMT_UINT|STRFMT_T_HEX);
  if (n < 0) { n = -n; sf |= STRFMT_F_UPPER; }
  sf |= ((SFormat)((n+1)&255) << STRFMT_SH_PREC);
#if LJ_HASFFI
  if (n < 16) b &= ((uint64_t)1 << 4*n)-1;
#else
  if (n < 8) b &= (1u << 4*n)-1;
#endif
  sb = lj_strfmt_putfxint(sb, sf, b);
  setstrV(L, L->top-1, lj_buf_str(L, sb));
  lj_gc_check(L);
  return 1;
}

/* ------------------------------------------------------------------------ */

#include "lj_libdef.h"

LUALIB_API int luaopen_bit(lua_State *L)
{
  LJ_LIB_REG(L, LUA_BITLIBNAME, bit);
  return 1;
}
```

One Level Up Top Level

src/lj_carith.c - luajit-2.0-src

Data types defined

- CDArith
- CDArith

Functions defined

- B64DEF
- B64DEF
- B64DEF
- B64DEF
- B64DEF
- carith_checkarg
- carith_int64
- carith_ptr
- lj_carith_check64
- lj_carith_divi64
- lj_carith_divu64
- lj_carith_meta
- lj_carith_modi64
- lj_carith_modu64
- lj_carith_mul64
- lj_carith_op
- lj_carith_powi64
- lj_carith_powu64
- lj_carith_shift64

Macros defined

- B64DEF
- B64DEF
- B64DEF

Source code

```
1 /*
2 ** C data arithmetic.
3 ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
```

```c
*/

#include "lj_obj.h"

#if LJ_HASFFI

#include "lj_gc.h"
#include "lj_err.h"
#include "lj_tab.h"
#include "lj_meta.h"
#include "lj_ir.h"
#include "lj_ctype.h"
#include "lj_cconv.h"
#include "lj_cdata.h"
#include "lj_carith.h"
#include "lj_strscan.h"

/* -- C data arithmetic --------------------------------------------------- */

/* Binary operands of an operator converted to ctypes. */
typedef struct CDArith {
  uint8_t *p[2];
  CType *ct[2];
} CDArith;

/* Check arguments for arithmetic metamethods. */
static int carith_checkarg(lua_State *L, CTState *cts, CDArith *ca)
{
  TValue *o = L->base;
  int ok = 1;
  MSize i;
  if (o+1 >= L->top)
    lj_err_argt(L, 1, LUA_TCDATA);
  for (i = 0; i < 2; i++, o++) {
    if (tviscdata(o)) {
      GCcdata *cd = cdataV(o);
      CTypeID id = (CTypeID)cd->ctypeid;
      CType *ct = ctype_raw(cts, id);
      uint8_t *p = (uint8_t *)cdataptr(cd);
      if (ctype_isptr(ct->info)) {
        p = (uint8_t *)cdata_getptr(p, ct->size);
        if (ctype_isref(ct->info)) ct = ctype_rawchild(cts, ct);
      } else if (ctype_isfunc(ct->info)) {
        p = (uint8_t *)*(void **)p;
        ct = ctype_get(cts,
          lj_ctype_intern(cts, CTINFO(CT_PTR, CTALIGN_PTR|id), CTSIZE_PTR));
      }
      if (ctype_isenum(ct->info)) ct = ctype_child(cts, ct);
      ca->ct[i] = ct;
      ca->p[i] = p;
    } else if (tvisint(o)) {
      ca->ct[i] = ctype_get(cts, CTID_INT32);
      ca->p[i] = (uint8_t *)&o->i;
    } else if (tvisnum(o)) {
      ca->ct[i] = ctype_get(cts, CTID_DOUBLE);
      ca->p[i] = (uint8_t *)&o->n;
    } else if (tvisnil(o)) {
      ca->ct[i] = ctype_get(cts, CTID_P_VOID);
      ca->p[i] = (uint8_t *)0;
    } else if (tvisstr(o)) {
      TValue *o2 = i == 0 ? o+1 : o-1;
      CType *ct = ctype_raw(cts, cdataV(o2)->ctypeid);
      ca->ct[i] = NULL;
      ca->p[i] = (uint8_t *)strVdata(o);
      ok = 0;
      if (ctype_isenum(ct->info)) {
        CTSize ofs;
        CType *cct = lj_ctype_getfield(cts, ct, strV(o), &ofs);
        if (cct && ctype_isconstval(cct->info)) {
          ca->ct[i] = ctype_child(cts, cct);
          ca->p[i] = (uint8_t *)&cct->size;  /* Assumes ct does not grow. */
          ok = 1;
        } else {
          ca->ct[1-i] = ct;  /* Use enum to improve error message. */
          ca->p[1-i] = NULL;
          break;
```

```
        }
      }
    } else {
      ca->ct[i] = NULL;
      ca->p[i] = (void *)(intptr_t)1;  /* To make it unequal. */
      ok = 0;
    }
  }
  return ok;
}

/* Pointer arithmetic. */
static int carith_ptr(lua_State *L, CTState *cts, CDArith *ca, MMS mm)
{
  CType *ctp = ca->ct[0];
  uint8_t *pp = ca->p[0];
  ptrdiff_t idx;
  CTSize sz;
  CTypeID id;
  GCcdata *cd;
  if (ctype_isptr(ctp->info) || ctype_isrefarray(ctp->info)) {
    if ((mm == MM_sub || mm == MM_eq || mm == MM_lt || mm == MM_le) &&
        (ctype_isptr(ca->ct[1]->info) || ctype_isrefarray(ca->ct[1]->info))) {
      uint8_t *pp2 = ca->p[1];
      if (mm == MM_eq) {  /* Pointer equality. Incompatible pointers are ok. */
        setboolV(L->top-1, (pp == pp2));
        return 1;
      }
      if (!lj_cconv_compatptr(cts, ctp, ca->ct[1], CCF_IGNQUAL))
        return 0;
      if (mm == MM_sub) {  /* Pointer difference. */
        intptr_t diff;
        sz = lj_ctype_size(cts, ctype_cid(ctp->info));  /* Element size. */
        if (sz == 0 || sz == CTSIZE_INVALID)
          return 0;
        diff = ((intptr_t)pp - (intptr_t)pp2) / (int32_t)sz;
        /* All valid pointer differences on x64 are in (-2^47, +2^47),
        ** which fits into a double without loss of precision.
        */
        setintptrV(L->top-1, (int32_t)diff);
        return 1;
      } else if (mm == MM_lt) {  /* Pointer comparison (unsigned). */
        setboolV(L->top-1, ((uintptr_t)pp < (uintptr_t)pp2));
        return 1;
      } else {
        lua_assert(mm == MM_le);
        setboolV(L->top-1, ((uintptr_t)pp <= (uintptr_t)pp2));
        return 1;
      }
    }
    if (!((mm == MM_add || mm == MM_sub) && ctype_isnum(ca->ct[1]->info)))
      return 0;
    lj_cconv_ct_ct(cts, ctype_get(cts, CTID_INT_PSZ), ca->ct[1],
                   (uint8_t *)&idx, ca->p[1], 0);
    if (mm == MM_sub) idx = -idx;
  } else if (mm == MM_add && ctype_isnum(ctp->info) &&
      (ctype_isptr(ca->ct[1]->info) || ctype_isrefarray(ca->ct[1]->info))) {
    /* Swap pointer and index. */
    ctp = ca->ct[1]; pp = ca->p[1];
    lj_cconv_ct_ct(cts, ctype_get(cts, CTID_INT_PSZ), ca->ct[0],
                   (uint8_t *)&idx, ca->p[0], 0);
  } else {
    return 0;
  }
  sz = lj_ctype_size(cts, ctype_cid(ctp->info));  /* Element size. */
  if (sz == CTSIZE_INVALID)
    return 0;
  pp += idx*(int32_t)sz;  /* Compute pointer + index. */
  id = lj_ctype_intern(cts, CTINFO(CT_PTR, CTALIGN_PTR|ctype_cid(ctp->info)),
                       CTSIZE_PTR);
  cd = lj_cdata_new(cts, id, CTSIZE_PTR);
  *(uint8_t **)cdataptr(cd) = pp;
  setcdataV(L, L->top-1, cd);
  lj_gc_check(L);
  return 1;
}
```

```
156
157 /* 64 bit integer arithmetic. */
158 static int carith_int64(lua_State *L, CTState *cts, CDArith *ca, MMS mm)
159 {
160   if (ctype_isnum(ca->ct[0]->info) && ca->ct[0]->size <= 8 &&
161       ctype_isnum(ca->ct[1]->info) && ca->ct[1]->size <= 8) {
162     CTypeID id = (((ca->ct[0]->info & CTF_UNSIGNED) && ca->ct[0]->size == 8) ||
163                   ((ca->ct[1]->info & CTF_UNSIGNED) && ca->ct[1]->size == 8)) ?
164                  CTID_UINT64 : CTID_INT64;
165     CType *ct = ctype_get(cts, id);
166     GCcdata *cd;
167     uint64_t u0, u1, *up;
168     lj_cconv_ct_ct(cts, ct, ca->ct[0], (uint8_t *)&u0, ca->p[0], 0);
169     if (mm != MM_unm)
170       lj_cconv_ct_ct(cts, ct, ca->ct[1], (uint8_t *)&u1, ca->p[1], 0);
171     switch (mm) {
172     case MM_eq:
173       setboolV(L->top-1, (u0 == u1));
174       return 1;
175     case MM_lt:
176       setboolV(L->top-1,
177                id == CTID_INT64 ? ((int64_t)u0 < (int64_t)u1) : (u0 < u1));
178       return 1;
179     case MM_le:
180       setboolV(L->top-1,
181                id == CTID_INT64 ? ((int64_t)u0 <= (int64_t)u1) : (u0 <= u1));
182       return 1;
183     default: break;
184     }
185     cd = lj_cdata_new(cts, id, 8);
186     up = (uint64_t *)cdataptr(cd);
187     setcdataV(L, L->top-1, cd);
188     switch (mm) {
189     case MM_add: *up = u0 + u1; break;
190     case MM_sub: *up = u0 - u1; break;
191     case MM_mul: *up = u0 * u1; break;
192     case MM_div:
193       if (id == CTID_INT64)
194         *up = (uint64_t)lj_carith_divi64((int64_t)u0, (int64_t)u1);
195       else
196         *up = lj_carith_divu64(u0, u1);
197       break;
198     case MM_mod:
199       if (id == CTID_INT64)
200         *up = (uint64_t)lj_carith_modi64((int64_t)u0, (int64_t)u1);
201       else
202         *up = lj_carith_modu64(u0, u1);
203       break;
204     case MM_pow:
205       if (id == CTID_INT64)
206         *up = (uint64_t)lj_carith_powi64((int64_t)u0, (int64_t)u1);
207       else
208         *up = lj_carith_powu64(u0, u1);
209       break;
210     case MM_unm: *up = (uint64_t)-(int64_t)u0; break;
211     default: lua_assert(0); break;
212     }
213     lj_gc_check(L);
214     return 1;
215   }
216   return 0;
217 }
218
219 /* Handle ctype arithmetic metamethods. */
220 static int lj_carith_meta(lua_State *L, CTState *cts, CDArith *ca, MMS mm)
221 {
222   cTValue *tv = NULL;
223   if (tviscdata(L->base)) {
224     CTypeID id = cdataV(L->base)->ctypeid;
225     CType *ct = ctype_raw(cts, id);
226     if (ctype_isptr(ct->info)) id = ctype_cid(ct->info);
227     tv = lj_ctype_meta(cts, id, mm);
228   }
229   if (!tv && L->base+1 < L->top && tviscdata(L->base+1)) {
230     CTypeID id = cdataV(L->base+1)->ctypeid;
231     CType *ct = ctype_raw(cts, id);
```

```
    if (ctype_isptr(ct->info)) id = ctype_cid(ct->info);
    tv = lj_ctype_meta(cts, id, mm);
  }
  if (!tv) {
    const char *repr[2];
    int i, isenum = -1, isstr = -1;
    if (mm == MM_eq) {  /* Equality checks never raise an error. */
      int eq = ca->p[0] == ca->p[1];
      setboolV(L->top-1, eq);
      setboolV(&G(L)->tmptv2, eq);  /* Remember for trace recorder. */
      return 1;
    }
    for (i = 0; i < 2; i++) {
      if (ca->ct[i] && tviscdata(L->base+i)) {
        if (ctype_isenum(ca->ct[i]->info)) isenum = i;
        repr[i] = strdata(lj_ctype_repr(L, ctype_typeid(cts, ca->ct[i]), NULL));
      } else {
        if (tvisstr(&L->base[i])) isstr = i;
        repr[i] = lj_typename(&L->base[i]);
      }
    }
    if ((isenum ^ isstr) == 1)
      lj_err_callerv(L, LJ_ERR_FFI_BADCONV, repr[isstr], repr[isenum]);
    lj_err_callerv(L, mm == MM_len ? LJ_ERR_FFI_BADLEN :
                      mm == MM_concat ? LJ_ERR_FFI_BADCONCAT :
                      mm < MM_add ? LJ_ERR_FFI_BADCOMP : LJ_ERR_FFI_BADARITH,
                   repr[0], repr[1]);
  }
  return lj_meta_tailcall(L, tv);
}

/* Arithmetic operators for cdata. */
int lj_carith_op(lua_State *L, MMS mm)
{
  CTState *cts = ctype_cts(L);
  CDArith ca;
  if (carith_checkarg(L, cts, &ca)) {
    if (carith_int64(L, cts, &ca, mm) || carith_ptr(L, cts, &ca, mm)) {
      copyTV(L, &G(L)->tmptv2, L->top-1);  /* Remember for trace recorder. */
      return 1;
    }
  }
  return lj_carith_meta(L, cts, &ca, mm);
}

/* -- 64 bit bit operations helpers ---------------------------------------- */

#if LJ_64
#define B64DEF(name) \
  static LJ_AINLINE uint64_t lj_carith_##name(uint64_t x, int32_t sh)
#else
/* Not inlined on 32 bit archs, since some of these are quite lengthy. */
#define B64DEF(name) \
  uint64_t LJ_NOINLINE lj_carith_##name(uint64_t x, int32_t sh)
#endif

B64DEF(shl64) { return x << (sh&63); }
B64DEF(shr64) { return x >> (sh&63); }
B64DEF(sar64) { return (uint64_t)((int64_t)x >> (sh&63)); }
B64DEF(rol64) { return lj_rol(x, (sh&63)); }
B64DEF(ror64) { return lj_ror(x, (sh&63)); }

#undef B64DEF

uint64_t lj_carith_shift64(uint64_t x, int32_t sh, int op)
{
  switch (op) {
  case IR_BSHL-IR_BSHL: x = lj_carith_shl64(x, sh); break;
  case IR_BSHR-IR_BSHL: x = lj_carith_shr64(x, sh); break;
  case IR_BSAR-IR_BSHL: x = lj_carith_sar64(x, sh); break;
  case IR_BROL-IR_BSHL: x = lj_carith_rol64(x, sh); break;
  case IR_BROR-IR_BSHL: x = lj_carith_ror64(x, sh); break;
  default: lua_assert(0); break;
  }
  return x;
}
```

```
308
309 /* Equivalent to lj_lib_checkbit(), but handles cdata. */
310 uint64_t lj_carith_check64(lua_State *L, int narg, CTypeID *id)
311 {
312   TValue *o = L->base + narg-1;
313   if (o >= L->top) {
314   err:
315     lj_err_argt(L, narg, LUA_TNUMBER);
316   } else if (LJ_LIKELY(tvisnumber(o))) {
317     /* Handled below. */
318   } else if (tviscdata(o)) {
319     CTState *cts = ctype_cts(L);
320     uint8_t *sp = (uint8_t *)cdataptr(cdataV(o));
321     CTypeID sid = cdataV(o)->ctypeid;
322     CType *s = ctype_get(cts, sid);
323     uint64_t x;
324     if (ctype_isref(s->info)) {
325       sp = *(void **)sp;
326       sid = ctype_cid(s->info);
327     }
328     s = ctype_raw(cts, sid);
329     if (ctype_isenum(s->info)) s = ctype_child(cts, s);
330     if ((s->info & (CTMASK_NUM|CTF_BOOL|CTF_FP|CTF_UNSIGNED)) ==
331         CTINFO(CT_NUM, CTF_UNSIGNED) && s->size == 8)
332       *id = CTID_UINT64;  /* Use uint64_t, since it has the highest rank. */
333     else if (!*id)
334       *id = CTID_INT64;  /* Use int64_t, unless already set. */
335     lj_cconv_ct_ct(cts, ctype_get(cts, *id), s,
336                    (uint8_t *)&x, sp, CCF_ARG(narg));
337     return x;
338   } else if (!(tvisstr(o) && lj_strscan_number(strV(o), o))) {
339     goto err;
340   }
341   if (LJ_LIKELY(tvisint(o))) {
342     return (uint32_t)intV(o);
343   } else {
344     int32_t i = lj_num2bit(numV(o));
345     if (LJ_DUALNUM) setintV(o, i);
346     return (uint32_t)i;
347   }
348 }
349
350
351 /* -- 64 bit integer arithmetic helpers ---------------------------------- */
352
353 #if LJ_32 && LJ_HASJIT
354 /* Signed/unsigned 64 bit multiplication. */
355 int64_t lj_carith_mul64(int64_t a, int64_t b)
356 {
357   return a * b;
358 }
359 #endif
360
361 /* Unsigned 64 bit division. */
362 uint64_t lj_carith_divu64(uint64_t a, uint64_t b)
363 {
364   if (b == 0) return U64x(80000000,00000000);
365   return a / b;
366 }
367
368 /* Signed 64 bit division. */
369 int64_t lj_carith_divi64(int64_t a, int64_t b)
370 {
371   if (b == 0 || (a == (int64_t)U64x(80000000,00000000) && b == -1))
372     return U64x(80000000,00000000);
373   return a / b;
374 }
375
376 /* Unsigned 64 bit modulo. */
377 uint64_t lj_carith_modu64(uint64_t a, uint64_t b)
378 {
379   if (b == 0) return U64x(80000000,00000000);
380   return a % b;
381 }
382
383 /* Signed 64 bit modulo. */
```

```
int64_t lj_carith_modi64(int64_t a, int64_t b)
{
  if (b == 0) return U64x(80000000,00000000);
  if (a == (int64_t)U64x(80000000,00000000) && b == -1) return 0;
  return a % b;
}

/* Unsigned 64 bit x^k. */
uint64_t lj_carith_powu64(uint64_t x, uint64_t k)
{
  uint64_t y;
  if (k == 0)
    return 1;
  for (; (k & 1) == 0; k >>= 1) x *= x;
  y = x;
  if ((k >>= 1) != 0) {
    for (;;) {
      x *= x;
      if (k == 1) break;
      if (k & 1) y *= x;
      k >>= 1;
    }
    y *= x;
  }
  return y;
}

/* Signed 64 bit x^k. */
int64_t lj_carith_powi64(int64_t x, int64_t k)
{
  if (k == 0)
    return 1;
  if (k < 0) {
    if (x == 0)
      return U64x(7fffffff,ffffffff);
    else if (x == 1)
      return 1;
    else if (x == -1)
      return (k & 1) ? -1 : 1;
    else
      return 0;
  }
  return (int64_t)lj_carith_powu64((uint64_t)x, (uint64_t)k);
}

#endif
```

One Level Up Top Level

src/lib_jit.c - luajit-2.0-src

Global variables defined

- KEY_PROFILE_FUNC
- KEY_PROFILE_THREAD
- jit_param_default
- jit_trlinkname

Functions defined

- LJLIB_CF(jit_on)
- LJLIB_CF(jit_off)
- LJLIB_CF(jit_flush)
- LJLIB_CF(jit_status)
- LJLIB_CF(jit_attach)
- LJLIB_CF(jit_util_funcinfo)
- LJLIB_CF(jit_util_funcbc)
- LJLIB_CF(jit_util_funck)
- LJLIB_CF(jit_util_funcuvname)
- LJLIB_CF(jit_util_traceinfo)
- LJLIB_CF(jit_util_traceir)
- LJLIB_CF(jit_util_tracek)
- LJLIB_CF(jit_util_tracesnap)
- LJLIB_CF(jit_util_tracemc)
- LJLIB_CF(jit_util_traceexitstub)
- LJLIB_CF(jit_util_ircalladdr)
- LJLIB_CF(jit_opt_start)
- LJLIB_CF(jit_profile_start)
- LJLIB_CF(jit_profile_stop)
- LJLIB_CF(jit_profile_dumpstack)
- check_Lproto
- flagbits_to_strings
- jit_checktrace
- jit_cpudetect

- jit_init
- jit_profile_callback
- jitopt_flag
- jitopt_level
- jitopt_param
- luaopen_jit
- luaopen_jit_profile
- luaopen_jit_util
- setintfield
- setjitmode

Macros defined

- JIT_F_SSE2
- JIT_PARAMINIT
- JIT_PARAMINIT
- LJLIB_MODULE_jit
- LJLIB_MODULE_jit_opt
- LJLIB_MODULE_jit_profile
- LJLIB_MODULE_jit_util
- LUA_LIB
- lib_jit_c

Source code

```
/*
** JIT library.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lib_jit_c
#define LUA_LIB

#include "lua.h"
 #include "lauxlib.h"
 #include "lualib.h"

 #include "lj_obj.h"
 #include "lj_gc.h"
 #include "lj_err.h"
 #include "lj_debug.h"
 #include "lj_str.h"
 #include "lj_tab.h"
 #include "lj_state.h"
 #include "lj_bc.h"
 #if LJ_HASFFI
 #include "lj_ctype.h"
 #endif
 #if LJ_HASJIT
 #include "lj_ir.h"
 #include "lj_jit.h"
```

```c
#include "lj_ircall.h"
#include "lj_iropt.h"
#include "lj_target.h"
#endif
#include "lj_trace.h"
#include "lj_dispatch.h"
#include "lj_vm.h"
#include "lj_vmevent.h"
#include "lj_lib.h"

#include "luajit.h"

/* -- jit.* functions ----------------------------------------------------- */

#define LJLIB_MODULE_jit

static int setjitmode(lua_State *L, int mode)
{
  int idx = 0;
  if (L->base == L->top || tvisnil(L->base)) {  /* jit.on/off/flush([nil]) */
    mode |= LUAJIT_MODE_ENGINE;
  } else {
    /* jit.on/off/flush(func|proto, nil|true|false) */
    if (tvisfunc(L->base) || tvisproto(L->base))
      idx = 1;
    else if (!tvistrue(L->base))  /* jit.on/off/flush(true, nil|true|false) */
      goto err;
    if (L->base+1 < L->top && tvisbool(L->base+1))
      mode |= boolV(L->base+1) ? LUAJIT_MODE_ALLFUNC : LUAJIT_MODE_ALLSUBFUNC;
    else
      mode |= LUAJIT_MODE_FUNC;
  }
  if (luaJIT_setmode(L, idx, mode) != 1) {
    if ((mode & LUAJIT_MODE_MASK) == LUAJIT_MODE_ENGINE)
      lj_err_caller(L, LJ_ERR_NOJIT);
  err:
    lj_err_argt(L, 1, LUA_TFUNCTION);
  }
  return 0;
}

LJLIB_CF(jit_on)
{
  return setjitmode(L, LUAJIT_MODE_ON);
}

LJLIB_CF(jit_off)
{
  return setjitmode(L, LUAJIT_MODE_OFF);
}

LJLIB_CF(jit_flush)
{
#if LJ_HASJIT
  if (L->base < L->top && tvisnumber(L->base)) {
    int traceno = lj_lib_checkint(L, 1);
    luaJIT_setmode(L, traceno, LUAJIT_MODE_FLUSH|LUAJIT_MODE_TRACE);
    return 0;
  }
#endif
  return setjitmode(L, LUAJIT_MODE_FLUSH);
}

#if LJ_HASJIT
/* Push a string for every flag bit that is set. */
static void flagbits_to_strings(lua_State *L, uint32_t flags, uint32_t base,
                                const char *str)
{
  for (; *str; base <<= 1, str += 1+*str)
    if (flags & base)
      setstrV(L, L->top++, lj_str_new(L, str+1, *(uint8_t *)str));
}
#endif

 LJLIB_CF(jit_status)
 {
```

```c
#if LJ_HASJIT
  jit_State *J = L2J(L);
  L->top = L->base;
  setboolV(L->top++, (J->flags & JIT_F_ON) ? 1 : 0);
  flagbits_to_strings(L, J->flags, JIT_F_CPU_FIRST, JIT_F_CPUSTRING);
  flagbits_to_strings(L, J->flags, JIT_F_OPT_FIRST, JIT_F_OPTSTRING);
  return (int)(L->top - L->base);
#else
  setboolV(L->top++, 0);
  return 1;
#endif
}

LJLIB_CF(jit_attach)
{
#ifdef LUAJIT_DISABLE_VMEVENT
  luaL_error(L, "vmevent API disabled");
#else
  GCfunc *fn = lj_lib_checkfunc(L, 1);
  GCstr *s = lj_lib_optstr(L, 2);
  luaL_findtable(L, LUA_REGISTRYINDEX, LJ_VMEVENTS_REGKEY, LJ_VMEVENTS_HSIZE);
  if (s) {  /* Attach to given event. */
    const uint8_t *p = (const uint8_t *)strdata(s);
    uint32_t h = s->len;
    while (*p) h = h ^ (lj_rol(h, 6) + *p++);
    lua_pushvalue(L, 1);
    lua_rawseti(L, -2, VMEVENT_HASHIDX(h));
    G(L)->vmevmask = VMEVENT_NOCACHE;  /* Invalidate cache. */
  } else {  /* Detach if no event given. */
    setnilV(L->top++);
    while (lua_next(L, -2)) {
      L->top--;
      if (tvisfunc(L->top) && funcV(L->top) == fn) {
        setnilV(lj_tab_set(L, tabV(L->top-2), L->top-1));
      }
    }
  }
#endif
  return 0;
}

LJLIB_PUSH(top-5) LJLIB_SET(os)
LJLIB_PUSH(top-4) LJLIB_SET(arch)
LJLIB_PUSH(top-3) LJLIB_SET(version_num)
LJLIB_PUSH(top-2) LJLIB_SET(version)

#include "lj_libdef.h"

/* -- jit.util.* functions ------------------------------------------------ */

#define LJLIB_MODULE_jit_util

/* -- Reflection API for Lua functions ---------------------------------- */

/* Return prototype of first argument (Lua function or prototype object) */
static GCproto *check_Lproto(lua_State *L, int nolua)
{
  TValue *o = L->base;
  if (L->top > o) {
    if (tvisproto(o)) {
      return protoV(o);
    } else if (tvisfunc(o)) {
      if (isluafunc(funcV(o)))
        return funcproto(funcV(o));
      else if (nolua)
        return NULL;
    }
  }
  lj_err_argt(L, 1, LUA_TFUNCTION);
  return NULL;  /* unreachable */
}

static void setintfield(lua_State *L, GCtab *t, const char *name, int32_t val)
{
  setintV(lj_tab_setstr(L, t, lj_str_newz(L, name)), val);
}
```

```c

/* local info = jit.util.funcinfo(func [,pc]) */
LJLIB_CF(jit_util_funcinfo)
{
  GCproto *pt = check_Lproto(L, 1);
  if (pt) {
    BCPos pc = (BCPos)lj_lib_optint(L, 2, 0);
    GCtab *t;
    lua_createtable(L, 0, 16);  /* Increment hash size if fields are added. */
    t = tabV(L->top-1);
    setintfield(L, t, "linedefined", pt->firstline);
    setintfield(L, t, "lastlinedefined", pt->firstline + pt->numline);
    setintfield(L, t, "stackslots", pt->framesize);
    setintfield(L, t, "params", pt->numparams);
    setintfield(L, t, "bytecodes", (int32_t)pt->sizebc);
    setintfield(L, t, "gcconsts", (int32_t)pt->sizekgc);
    setintfield(L, t, "nconsts", (int32_t)pt->sizekn);
    setintfield(L, t, "upvalues", (int32_t)pt->sizeuv);
    if (pc < pt->sizebc)
      setintfield(L, t, "currentline", lj_debug_line(pt, pc));
    lua_pushboolean(L, (pt->flags & PROTO_VARARG));
    lua_setfield(L, -2, "isvararg");
    lua_pushboolean(L, (pt->flags & PROTO_CHILD));
    lua_setfield(L, -2, "children");
    setstrV(L, L->top++, proto_chunkname(pt));
    lua_setfield(L, -2, "source");
    lj_debug_pushloc(L, pt, pc);
    lua_setfield(L, -2, "loc");
  } else {
    GCfunc *fn = funcV(L->base);
    GCtab *t;
    lua_createtable(L, 0, 4);  /* Increment hash size if fields are added. */
    t = tabV(L->top-1);
    if (!iscfunc(fn))
      setintfield(L, t, "ffid", fn->c.ffid);
    setintptrV(lj_tab_setstr(L, t, lj_str_newlit(L, "addr")),
               (intptr_t)(void *)fn->c.f);
    setintfield(L, t, "upvalues", fn->c.nupvalues);
  }
  return 1;
}

/* local ins, m = jit.util.funcbc(func, pc) */
LJLIB_CF(jit_util_funcbc)
{
  GCproto *pt = check_Lproto(L, 0);
  BCPos pc = (BCPos)lj_lib_checkint(L, 2);
  if (pc < pt->sizebc) {
    BCIns ins = proto_bc(pt)[pc];
    BCOp op = bc_op(ins);
    lua_assert(op < BC__MAX);
    setintV(L->top, ins);
    setintV(L->top+1, lj_bc_mode[op]);
    L->top += 2;
    return 2;
  }
  return 0;
}

/* local k = jit.util.funck(func, idx) */
LJLIB_CF(jit_util_funck)
{
  GCproto *pt = check_Lproto(L, 0);
  ptrdiff_t idx = (ptrdiff_t)lj_lib_checkint(L, 2);
  if (idx >= 0) {
    if (idx < (ptrdiff_t)pt->sizekn) {
      copyTV(L, L->top-1, proto_knumtv(pt, idx));
      return 1;
    }
  } else {
    if (~idx < (ptrdiff_t)pt->sizekgc) {
      GCobj *gc = proto_kgc(pt, idx);
      setgcV(L, L->top-1, gc, ~gc->gch.gct);
      return 1;
    }
  }
```

```
  return 0;
}

/* local name = jit.util.funcuvname(func, idx) */
LJLIB_CF(jit_util_funcuvname)
{
  GCproto *pt = check_Lproto(L, 0);
  uint32_t idx = (uint32_t)lj_lib_checkint(L, 2);
  if (idx < pt->sizeuv) {
    setstrV(L, L->top-1, lj_str_newz(L, lj_debug_uvname(pt, idx)));
    return 1;
  }
  return 0;
}

/* -- Reflection API for traces ---------------------------------------- */

#if LJ_HASJIT

/* Check trace argument. Must not throw for non-existent trace numbers. */
static GCtrace *jit_checktrace(lua_State *L)
{
  TraceNo tr = (TraceNo)lj_lib_checkint(L, 1);
  jit_State *J = L2J(L);
  if (tr > 0 && tr < J->sizetrace)
    return traceref(J, tr);
  return NULL;
}

/* Names of link types. ORDER LJ_TRLINK */
static const char *const jit_trlinkname[] = {
  "none", "root", "loop", "tail-recursion", "up-recursion", "down-recursion",
  "interpreter", "return", "stitch"
};

/* local info = jit.util.traceinfo(tr) */
LJLIB_CF(jit_util_traceinfo)
{
  GCtrace *T = jit_checktrace(L);
  if (T) {
    GCtab *t;
    lua_createtable(L, 0, 8);  /* Increment hash size if fields are added. */
    t = tabV(L->top-1);
    setintfield(L, t, "nins", (int32_t)T->nins - REF_BIAS - 1);
    setintfield(L, t, "nk", REF_BIAS - (int32_t)T->nk);
    setintfield(L, t, "link", T->link);
    setintfield(L, t, "nexit", T->nsnap);
    setstrV(L, L->top++, lj_str_newz(L, jit_trlinkname[T->linktype]));
    lua_setfield(L, -2, "linktype");
    /* There are many more fields. Add them only when needed. */
    return 1;
  }
  return 0;
}

/* local m, ot, op1, op2, prev = jit.util.traceir(tr, idx) */
LJLIB_CF(jit_util_traceir)
{
  GCtrace *T = jit_checktrace(L);
  IRRef ref = (IRRef)lj_lib_checkint(L, 2) + REF_BIAS;
  if (T && ref >= REF_BIAS && ref < T->nins) {
    IRIns *ir = &T->ir[ref];
    int32_t m = lj_ir_mode[ir->o];
    setintV(L->top-2, m);
    setintV(L->top-1, ir->ot);
    setintV(L->top++, (int32_t)ir->op1 - (irm_op1(m)==IRMref ? REF_BIAS : 0));
    setintV(L->top++, (int32_t)ir->op2 - (irm_op2(m)==IRMref ? REF_BIAS : 0));
    setintV(L->top++, ir->prev);
    return 5;
  }
  return 0;
}

/* local k, t [, slot] = jit.util.tracek(tr, idx) */
LJLIB_CF(jit_util_tracek)
{
```

```
  GCtrace *T = jit_checktrace(L);
  IRRef ref = (IRRef)lj_lib_checkint(L, 2) + REF_BIAS;
  if (T && ref >= T->nk && ref < REF_BIAS) {
    IRIns *ir = &T->ir[ref];
    int32_t slot = -1;
    if (ir->o == IR_KSLOT) {
      slot = ir->op2;
      ir = &T->ir[ir->op1];
    }
#if LJ_HASFFI
    if (ir->o == IR_KINT64 && !ctype_ctsG(G(L))) {
      ptrdiff_t oldtop = savestack(L, L->top);
      luaopen_ffi(L);  /* Load FFI library on-demand. */
      L->top = restorestack(L, oldtop);
    }
#endif
    lj_ir_kvalue(L, L->top-2, ir);
    setintV(L->top-1, (int32_t)irt_type(ir->t));
    if (slot == -1)
      return 2;
    setintV(L->top++, slot);
    return 3;
  }
  return 0;
}

/* local snap = jit.util.tracesnap(tr, sn) */
LJLIB_CF(jit_util_tracesnap)
{
  GCtrace *T = jit_checktrace(L);
  SnapNo sn = (SnapNo)lj_lib_checkint(L, 2);
  if (T && sn < T->nsnap) {
    SnapShot *snap = &T->snap[sn];
    SnapEntry *map = &T->snapmap[snap->mapofs];
    MSize n, nent = snap->nent;
    GCtab *t;
    lua_createtable(L, nent+2, 0);
    t = tabV(L->top-1);
    setintV(lj_tab_setint(L, t, 0), (int32_t)snap->ref - REF_BIAS);
    setintV(lj_tab_setint(L, t, 1), (int32_t)snap->nslots);
    for (n = 0; n < nent; n++)
      setintV(lj_tab_setint(L, t, (int32_t)(n+2)), (int32_t)map[n]);
    setintV(lj_tab_setint(L, t, (int32_t)(nent+2)), (int32_t)SNAP(255, 0, 0));
    return 1;
  }
  return 0;
}

/* local mcode, addr, loop = jit.util.tracemc(tr) */
LJLIB_CF(jit_util_tracemc)
{
  GCtrace *T = jit_checktrace(L);
  if (T && T->mcode != NULL) {
    setstrV(L, L->top-1, lj_str_new(L, (const char *)T->mcode, T->szmcode));
    setintptrV(L->top++, (intptr_t)(void *)T->mcode);
    setintV(L->top++, T->mcloop);
    return 3;
  }
  return 0;
}

/* local addr = jit.util.traceexitstub([tr,] exitno) */
LJLIB_CF(jit_util_traceexitstub)
{
#ifdef EXITSTUBS_PER_GROUP
  ExitNo exitno = (ExitNo)lj_lib_checkint(L, 1);
  jit_State *J = L2J(L);
  if (exitno < EXITSTUBS_PER_GROUP*LJ_MAX_EXITSTUBGR) {
    setintptrV(L->top-1, (intptr_t)(void *)exitstub_addr(J, exitno));
    return 1;
  }
#else
  if (L->top > L->base+1) {  /* Don't throw for one-argument variant. */
    GCtrace *T = jit_checktrace(L);
    ExitNo exitno = (ExitNo)lj_lib_checkint(L, 2);
    ExitNo maxexit = T->root ? T->nsnap+1 : T->nsnap;
```

```
    if (T && T->mcode != NULL && exitno < maxexit) {
      setintptrV(L->top-1, (intptr_t)(void *)exitstub_trace_addr(T, exitno));
      return 1;
    }
  }
#endif
  return 0;
}

/* local addr = jit.util.ircalladdr(idx) */
LJLIB_CF(jit_util_ircalladdr)
{
  uint32_t idx = (uint32_t)lj_lib_checkint(L, 1);
  if (idx < IRCALL__MAX) {
    setintptrV(L->top-1, (intptr_t)(void *)lj_ir_callinfo[idx].func);
    return 1;
  }
  return 0;
}

#endif

#include "lj_libdef.h"

static int luaopen_jit_util(lua_State *L)
{
  LJ_LIB_REG(L, NULL, jit_util);
  return 1;
}

/* -- jit.opt module ------------------------------------------------------ */

#if LJ_HASJIT

#define LJLIB_MODULE_jit_opt

/* Parse optimization level. */
static int jitopt_level(jit_State *J, const char *str)
{
  if (str[0] >= '0' && str[0] <= '9' && str[1] == '\0') {
    uint32_t flags;
    if (str[0] == '0') flags = JIT_F_OPT_0;
    else if (str[0] == '1') flags = JIT_F_OPT_1;
    else if (str[0] == '2') flags = JIT_F_OPT_2;
    else flags = JIT_F_OPT_3;
    J->flags = (J->flags & ~JIT_F_OPT_MASK) | flags;
    return 1;  /* Ok. */
  }
  return 0;  /* No match. */
}

/* Parse optimization flag. */
static int jitopt_flag(jit_State *J, const char *str)
{
  const char *lst = JIT_F_OPTSTRING;
  uint32_t opt;
  int set = 1;
  if (str[0] == '+') {
    str++;
  } else if (str[0] == '-') {
    str++;
    set = 0;
  } else if (str[0] == 'n' && str[1] == 'o') {
    str += str[2] == '-' ? 3 : 2;
    set = 0;
  }
  for (opt = JIT_F_OPT_FIRST; ; opt <<= 1) {
    size_t len = *(const uint8_t *)lst;
    if (len == 0)
      break;
    if (strncmp(str, lst+1, len) == 0 && str[len] == '\0') {
      if (set) J->flags |= opt; else J->flags &= ~opt;
      return 1;  /* Ok. */
    }
    lst += 1+len;
  }
```

```
  return 0;  /* No match. */
}

/* Parse optimization parameter. */
static int jitopt_param(jit_State *J, const char *str)
{
  const char *lst = JIT_P_STRING;
  int i;
  for (i = 0; i < JIT_P__MAX; i++) {
    size_t len = *(const uint8_t *)lst;
    lua_assert(len != 0);
    if (strncmp(str, lst+1, len) == 0 && str[len] == '=') {
      int32_t n = 0;
      const char *p = &str[len+1];
      while (*p >= '0' && *p <= '9')
        n = n*10 + (*p++ - '0');
      if (*p) return 0;  /* Malformed number. */
      J->param[i] = n;
      if (i == JIT_P_hotloop)
        lj_dispatch_init_hotcount(J2G(J));
      return 1;  /* Ok. */
    }
    lst += 1+len;
  }
  return 0;  /* No match. */
}

/* jit.opt.start(flags...) */
LJLIB_CF(jit_opt_start)
{
  jit_State *J = L2J(L);
  int nargs = (int)(L->top - L->base);
  if (nargs == 0) {
    J->flags = (J->flags & ~JIT_F_OPT_MASK) | JIT_F_OPT_DEFAULT;
  } else {
    int i;
    for (i = 1; i <= nargs; i++) {
      const char *str = strdata(lj_lib_checkstr(L, i));
      if (!jitopt_level(J, str) &&
          !jitopt_flag(J, str) &&
          !jitopt_param(J, str))
        lj_err_callerv(L, LJ_ERR_JITOPT, str);
    }
  }
  return 0;
}

#include "lj_libdef.h"

#endif

/* -- jit.profile module ---------------------------------------------- */

#if LJ_HASPROFILE

#define LJLIB_MODULE_jit_profile

/* Not loaded by default, use: local profile = require("jit.profile") */

static const char KEY_PROFILE_THREAD = 't';
static const char KEY_PROFILE_FUNC = 'f';

static void jit_profile_callback(lua_State *L2, lua_State *L, int samples,
                                 int vmstate)
{
  TValue key;
  cTValue *tv;
  setlightudV(&key, (void *)&KEY_PROFILE_FUNC);
  tv = lj_tab_get(L, tabV(registry(L)), &key);
  if (tvisfunc(tv)) {
    char vmst = (char)vmstate;
    int status;
    setfuncV(L2, L2->top++, funcV(tv));
    setthreadV(L2, L2->top++, L);
    setintV(L2->top++, samples);
    setstrV(L2, L2->top++, lj_str_new(L2, &vmst, 1));
```

```
    status = lua_pcall(L2, 3, 0, 0);  /* callback(thread, samples, vmstate) */
    if (status) {
      if (G(L2)->panic) G(L2)->panic(L2);
      exit(EXIT_FAILURE);
    }
    lj_trace_abort(G(L2));
  }
}

/* profile.start(mode, cb) */
LJLIB_CF(jit_profile_start)
{
  GCtab *registry = tabV(registry(L));
  GCstr *mode = lj_lib_optstr(L, 1);
  GCfunc *func = lj_lib_checkfunc(L, 2);
  lua_State *L2 = lua_newthread(L);  /* Thread that runs profiler callback. */
  TValue key;
  /* Anchor thread and function in registry. */
  setlightudV(&key, (void *)&KEY_PROFILE_THREAD);
  setthreadV(L, lj_tab_set(L, registry, &key), L2);
  setlightudV(&key, (void *)&KEY_PROFILE_FUNC);
  setfuncV(L, lj_tab_set(L, registry, &key), func);
  lj_gc_anybarriert(L, registry);
  luaJIT_profile_start(L, mode ? strdata(mode) : "",
                       (luaJIT_profile_callback)jit_profile_callback, L2);
  return 0;
}

/* profile.stop() */
LJLIB_CF(jit_profile_stop)
{
  GCtab *registry;
  TValue key;
  luaJIT_profile_stop(L);
  registry = tabV(registry(L));
  setlightudV(&key, (void *)&KEY_PROFILE_THREAD);
  setnilV(lj_tab_set(L, registry, &key));
  setlightudV(&key, (void *)&KEY_PROFILE_FUNC);
  setnilV(lj_tab_set(L, registry, &key));
  lj_gc_anybarriert(L, registry);
  return 0;
}

/* dump = profile.dumpstack([thread,] fmt, depth) */
LJLIB_CF(jit_profile_dumpstack)
{
  lua_State *L2 = L;
  int arg = 0;
  size_t len;
  int depth;
  GCstr *fmt;
  const char *p;
  if (L->top > L->base && tvisthread(L->base)) {
    L2 = threadV(L->base);
    arg = 1;
  }
  fmt = lj_lib_checkstr(L, arg+1);
  depth = lj_lib_checkint(L, arg+2);
  p = luaJIT_profile_dumpstack(L2, strdata(fmt), depth, &len);
  lua_pushlstring(L, p, len);
  return 1;
}

#include "lj_libdef.h"

static int luaopen_jit_profile(lua_State *L)
{
  LJ_LIB_REG(L, NULL, jit_profile);
  return 1;
}

#endif

/* -- JIT compiler initialization ----------------------------------------- */

#if LJ_HASJIT
```

```
/* Default values for JIT parameters. */
static const int32_t jit_param_default[JIT_P__MAX+1] = {
#define JIT_PARAMINIT(len, name, value)        (value),
JIT_PARAMDEF(JIT_PARAMINIT)
#undef JIT_PARAMINIT
  0
};
#endif

#if LJ_TARGET_ARM && LJ_TARGET_LINUX
#include <sys/utsname.h>
#endif

/* Arch-dependent CPU detection. */
static uint32_t jit_cpudetect(lua_State *L)
{
  uint32_t flags = 0;
#if LJ_TARGET_X86ORX64
  uint32_t vendor[4];
  uint32_t features[4];
  if (lj_vm_cpuid(0, vendor) && lj_vm_cpuid(1, features)) {
#if !LJ_HASJIT
#define JIT_F_SSE2        2
#endif
    flags |= ((features[3] >> 26)&1) * JIT_F_SSE2;
#if LJ_HASJIT
    flags |= ((features[2] >> 0)&1) * JIT_F_SSE3;
    flags |= ((features[2] >> 19)&1) * JIT_F_SSE4_1;
    if (vendor[2] == 0x6c65746e) {  /* Intel. */
      if ((features[0] & 0x0fff0ff0) == 0x000106c0)  /* Atom. */
        flags |= JIT_F_LEA_AGU;
    } else if (vendor[2] == 0x444d4163) {  /* AMD. */
      uint32_t fam = (features[0] & 0x0ff00f00);
      if (fam >= 0x00000f00)  /* K8, K10. */
        flags |= JIT_F_PREFER_IMUL;
    }
#endif
  }
  /* Check for required instruction set support on x86 (unnecessary on x64). */
#if LJ_TARGET_X86
  if (!(flags & JIT_F_SSE2))
    luaL_error(L, "CPU with SSE2 required");
#endif
#elif LJ_TARGET_ARM
#if LJ_HASJIT
  int ver = LJ_ARCH_VERSION;  /* Compile-time ARM CPU detection. */
#if LJ_TARGET_LINUX
  if (ver < 70) {  /* Runtime ARM CPU detection. */
    struct utsname ut;
    uname(&ut);
    if (strncmp(ut.machine, "armv", 4) == 0) {
      if (ut.machine[4] >= '7')
        ver = 70;
      else if (ut.machine[4] == '6')
        ver = 60;
    }
  }
#endif
  flags |= ver >= 70 ? JIT_F_ARMV7 :
           ver >= 61 ? JIT_F_ARMV6T2_ :
           ver >= 60 ? JIT_F_ARMV6_ : 0;
  flags |= LJ_ARCH_HASFPU == 0 ? 0 : ver >= 70 ? JIT_F_VFPV3 : JIT_F_VFPV2;
#endif
#elif LJ_TARGET_ARM64
  /* No optional CPU features to detect (for now). */
#elif LJ_TARGET_PPC
#if LJ_HASJIT
#if LJ_ARCH_SQRT
  flags |= JIT_F_SQRT;
#endif
#if LJ_ARCH_ROUND
  flags |= JIT_F_ROUND;
#endif
#endif
#elif LJ_TARGET_MIPS
#if LJ_HASJIT
```

```
  /* Compile-time MIPS CPU detection. */
#if LJ_ARCH_VERSION >= 20
  flags |= JIT_F_MIPS32R2;
#endif
  /* Runtime MIPS CPU detection. */
#if defined(__GNUC__)
  if (!(flags & JIT_F_MIPS32R2)) {
    int x;
    /* On MIPS32R1 rotr is treated as srl. rotr r2,r2,1 -> srl r2,r2,1. */
    __asm__("li $2, 1\n\t.long 0x00221042\n\tmove %0, $2" : "=r"(x) : : "$2");
    if (x) flags |= JIT_F_MIPS32R2;  /* Either 0x80000000 (R2) or 0 (R1). */
  }
#endif
#endif
#else
#error "Missing CPU detection for this architecture"
#endif
  UNUSED(L);
  return flags;
}

/* Initialize JIT compiler. */
static void jit_init(lua_State *L)
{
  uint32_t flags = jit_cpudetect(L);
#if LJ_HASJIT
  jit_State *J = L2J(L);
  J->flags = flags | JIT_F_ON | JIT_F_OPT_DEFAULT;
  memcpy(J->param, jit_param_default, sizeof(J->param));
  lj_dispatch_update(G(L));
#else
  UNUSED(flags);
#endif
}

LUALIB_API int luaopen_jit(lua_State *L)
{
  jit_init(L);
  lua_pushliteral(L, LJ_OS_NAME);
  lua_pushliteral(L, LJ_ARCH_NAME);
  lua_pushinteger(L, LUAJIT_VERSION_NUM);
  lua_pushliteral(L, LUAJIT_VERSION);
  LJ_LIB_REG(L, LUA_JITLIBNAME, jit);
#if LJ_HASPROFILE
  lj_lib_prereg(L, LUA_JITLIBNAME ".profile", luaopen_jit_profile,
                tabref(L->env));
#endif
#ifndef LUAJIT_DISABLE_JITUTIL
  lj_lib_prereg(L, LUA_JITLIBNAME ".util", luaopen_jit_util, tabref(L->env));
#endif
#if LJ_HASJIT
  LJ_LIB_REG(L, "jit.opt", jit_opt);
#endif
  L->top -= 2;
  return 1;
}
```

One Level Up Top Level

src/luajit.h - luajit-2.0-src

Data types defined

- luaJIT_profile_callback

Macros defined

- LUAJIT_COPYRIGHT
- LUAJIT_MODE_FLUSH
- LUAJIT_MODE_MASK
- LUAJIT_MODE_OFF
- LUAJIT_MODE_ON
- LUAJIT_URL
- LUAJIT_VERSION
- LUAJIT_VERSION_NUM
- LUAJIT_VERSION_SYM
- _LUAJIT_H

Source code

```
/*
** LuaJIT -- a Just-In-Time Compiler for Lua. http://luajit.org/
**
** Copyright (C) 2005-2015 Mike Pall. All rights reserved.
**
** Permission is hereby granted, free of charge, to any person obtaining
** a copy of this software and associated documentation files (the
** "Software"), to deal in the Software without restriction, including
** without limitation the rights to use, copy, modify, merge, publish,
 ** distribute, sublicense, and/or sell copies of the Software, and to
 ** permit persons to whom the Software is furnished to do so, subject to
 ** the following conditions:
 **
 ** The above copyright notice and this permission notice shall be
 ** included in all copies or substantial portions of the Software.
 **
 ** THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 ** EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 ** MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 ** IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
 ** CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 ** TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
 ** SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
 **
 ** [ MIT license: http://www.opensource.org/licenses/mit-license.php ]
 */

#ifndef _LUAJIT_H
#define _LUAJIT_H

#include "lua.h"

#define LUAJIT_VERSION              "LuaJIT 2.1.0-alpha"
#define LUAJIT_VERSION_NUM        20100  /* Version 2.1.0 = 02.01.00. */
#define LUAJIT_VERSION_SYM        luaJIT_version_2_1_0_alpha
#define LUAJIT_COPYRIGHT        "Copyright (C) 2005-2015 Mike Pall"
```

```
#define LUAJIT_URL                "http://luajit.org/"

/* Modes for luaJIT_setmode. */
#define LUAJIT_MODE_MASK        0x00ff

enum {
  LUAJIT_MODE_ENGINE,                /* Set mode for whole JIT engine. */
  LUAJIT_MODE_DEBUG,                /* Set debug mode (idx = level). */

  LUAJIT_MODE_FUNC,                /* Change mode for a function. */
  LUAJIT_MODE_ALLFUNC,                /* Recurse into subroutine protos. */
  LUAJIT_MODE_ALLSUBFUNC,        /* Change only the subroutines. */

  LUAJIT_MODE_TRACE,                /* Flush a compiled trace. */

  LUAJIT_MODE_WRAPCFUNC = 0x10,        /* Set wrapper mode for C function calls. */

  LUAJIT_MODE_MAX
};

/* Flags or'ed in to the mode. */
#define LUAJIT_MODE_OFF                0x0000        /* Turn feature off. */
#define LUAJIT_MODE_ON                0x0100        /* Turn feature on. */
#define LUAJIT_MODE_FLUSH        0x0200        /* Flush JIT-compiled code. */

/* LuaJIT public C API. */

/* Control the JIT engine. */
LUA_API int luaJIT_setmode(lua_State *L, int idx, int mode);

/* Low-overhead profiling API. */
typedef void (*luaJIT_profile_callback)(void *data, lua_State *L,
                                        int samples, int vmstate);
LUA_API void luaJIT_profile_start(lua_State *L, const char *mode,
                                  luaJIT_profile_callback cb, void *data);
LUA_API void luaJIT_profile_stop(lua_State *L);
LUA_API const char *luaJIT_profile_dumpstack(lua_State *L, const char *fmt,
                                             int depth, size_t *len);

/* Enforce (dynamic) linker error for version mismatches. Call from main. */
LUA_API void LUAJIT_VERSION_SYM(void);

#endif
```

One Level Up Top Level

src/lj_profile.c - luajit-2.0-src

Global variables defined

- profile_state

Data types defined

- ProfileState
- ProfileState
- WMM_TPFUNC

Functions defined

- lj_profile_hook_enter
- lj_profile_hook_leave
- lj_profile_interpreter
- luaJIT_profile_dumpstack
- luaJIT_profile_start
- luaJIT_profile_stop
- profile_signal
- profile_thread
- profile_thread
- profile_timer_start
- profile_timer_start
- profile_timer_start
- profile_timer_stop
- profile_timer_stop
- profile_timer_stop
- profile_trigger

Macros defined

- LJ_PROFILE_INTERVAL_DEFAULT
- LUA_CORE
- WIN32_LEAN_AND_MEAN
- lj_profile_c
- profile_lock

- profile_lock
- profile_lock
- profile_unlock
- profile_unlock
- profile_unlock

Source code

```
1  /*
2  ** Low-overhead profiling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lj_profile_c
7  #define LUA_CORE
8
9  #include "lj_obj.h"
10
11 #if LJ_HASPROFILE
12
13 #include "lj_buf.h"
14 #include "lj_frame.h"
15 #include "lj_debug.h"
16 #include "lj_dispatch.h"
17 #if LJ_HASJIT
18 #include "lj_jit.h"
19 #include "lj_trace.h"
20 #endif
21 #include "lj_profile.h"
22
23 #include "luajit.h"
24
25 #if LJ_PROFILE_SIGPROF
26
27 #include <sys/time.h>
28 #include <signal.h>
29 #define profile_lock(ps)        UNUSED(ps)
30 #define profile_unlock(ps)        UNUSED(ps)
31
32 #elif LJ_PROFILE_PTHREAD
33
34 #include <pthread.h>
35 #include <time.h>
36 #if LJ_TARGET_PS3
37 #include <sys/timer.h>
38 #endif
39 #define profile_lock(ps)        pthread_mutex_lock(&ps->lock)
40 #define profile_unlock(ps)        pthread_mutex_unlock(&ps->lock)
41
42 #elif LJ_PROFILE_WTHREAD
43
44 #define WIN32_LEAN_AND_MEAN
45 #if LJ_TARGET_XBOX360
46 #include <xtl.h>
47 #include <xbox.h>
48 #else
49 #include <windows.h>
50 #endif
51 typedef unsigned int (WINAPI *WMM_TPFUNC)(unsigned int);
52 #define profile_lock(ps)        EnterCriticalSection(&ps->lock)
53 #define profile_unlock(ps)        LeaveCriticalSection(&ps->lock)
54
55 #endif
56
57 /* Profiler state. */
58 typedef struct ProfileState {
59   global_State *g;                /* VM state that started the profiler. */
60   luaJIT_profile_callback cb;        /* Profiler callback. */
61   void *data;                        /* Profiler callback data. */
```

```
  SBuf sb;                        /* String buffer for stack dumps. */
  int interval;                        /* Sample interval in milliseconds. */
  int samples;                        /* Number of samples for next callback. */
  int vmstate;                        /* VM state when profile timer triggered. */
#if LJ_PROFILE_SIGPROF
  struct sigaction oldsa;        /* Previous SIGPROF state. */
#elif LJ_PROFILE_PTHREAD
  pthread_mutex_t lock;                /* g->hookmask update lock. */
  pthread_t thread;                /* Timer thread. */
  int abort;                        /* Abort timer thread. */
#elif LJ_PROFILE_WTHREAD
#if LJ_TARGET_WINDOWS
  HINSTANCE wmm;                /* WinMM library handle. */
  WMM_TPFUNC wmm_tbp;                /* WinMM timeBeginPeriod function. */
  WMM_TPFUNC wmm_tep;                /* WinMM timeEndPeriod function. */
#endif
  CRITICAL_SECTION lock;        /* g->hookmask update lock. */
  HANDLE thread;                /* Timer thread. */
  int abort;                        /* Abort timer thread. */
#endif
} ProfileState;

/* Sadly, we have to use a static profiler state.
**
** The SIGPROF variant needs a static pointer to the global state, anyway.
** And it would be hard to extend for multiple threads. You can still use
** multiple VMs in multiple threads, but only profile one at a time.
*/
static ProfileState profile_state;

/* Default sample interval in milliseconds. */
#define LJ_PROFILE_INTERVAL_DEFAULT        10

/* -- Profiler/hook interaction ------------------------------------------- */

#if !LJ_PROFILE_SIGPROF
void LJ_FASTCALL lj_profile_hook_enter(global_State *g)
{
  ProfileState *ps = &profile_state;
  if (ps->g) {
    profile_lock(ps);
    hook_enter(g);
    profile_unlock(ps);
  } else {
    hook_enter(g);
  }
}

void LJ_FASTCALL lj_profile_hook_leave(global_State *g)
{
  ProfileState *ps = &profile_state;
  if (ps->g) {
    profile_lock(ps);
    hook_leave(g);
    profile_unlock(ps);
  } else {
    hook_leave(g);
  }
}
#endif

/* -- Profile callbacks --------------------------------------------------- */

/* Callback from profile hook (HOOK_PROFILE already cleared). */
void LJ_FASTCALL lj_profile_interpreter(lua_State *L)
{
  ProfileState *ps = &profile_state;
  global_State *g = G(L);
  uint8_t mask;
  profile_lock(ps);
  mask = (g->hookmask & ~HOOK_PROFILE);
  if (!(mask & HOOK_VMEVENT)) {
    int samples = ps->samples;
    ps->samples = 0;
    g->hookmask = HOOK_VMEVENT;
    lj_dispatch_update(g);
```

```
    profile_unlock(ps);
    ps->cb(ps->data, L, samples, ps->vmstate);  /* Invoke user callback. */
    profile_lock(ps);
    mask |= (g->hookmask & HOOK_PROFILE);
  }
  g->hookmask = mask;
  lj_dispatch_update(g);
  profile_unlock(ps);
}

/* Trigger profile hook. Asynchronous call from OS-specific profile timer. */
static void profile_trigger(ProfileState *ps)
{
  global_State *g = ps->g;
  uint8_t mask;
  profile_lock(ps);
  ps->samples++;  /* Always increment number of samples. */
  mask = g->hookmask;
  if (!(mask & (HOOK_PROFILE|HOOK_VMEVENT))) {  /* Set profile hook. */
    int st = g->vmstate;
    ps->vmstate = st >= 0 ? 'N' :
                  st == ~LJ_VMST_INTERP ? 'I' :
                  st == ~LJ_VMST_C ? 'C' :
                  st == ~LJ_VMST_GC ? 'G' : 'J';
    g->hookmask = (mask | HOOK_PROFILE);
    lj_dispatch_update(g);
  }
  profile_unlock(ps);
}

/* -- OS-specific profile timer handling ------------------------------- */

#if LJ_PROFILE_SIGPROF

/* SIGPROF handler. */
static void profile_signal(int sig)
{
  UNUSED(sig);
  profile_trigger(&profile_state);
}

/* Start profiling timer. */
static void profile_timer_start(ProfileState *ps)
{
  int interval = ps->interval;
  struct itimerval tm;
  struct sigaction sa;
  tm.it_value.tv_sec = tm.it_interval.tv_sec = interval / 1000;
  tm.it_value.tv_usec = tm.it_interval.tv_usec = (interval % 1000) * 1000;
  setitimer(ITIMER_PROF, &tm, NULL);
  sa.sa_flags = SA_RESTART;
  sa.sa_handler = profile_signal;
  sigemptyset(&sa.sa_mask);
  sigaction(SIGPROF, &sa, &ps->oldsa);
}

/* Stop profiling timer. */
static void profile_timer_stop(ProfileState *ps)
{
  struct itimerval tm;
  tm.it_value.tv_sec = tm.it_interval.tv_sec = 0;
  tm.it_value.tv_usec = tm.it_interval.tv_usec = 0;
  setitimer(ITIMER_PROF, &tm, NULL);
  sigaction(SIGPROF, &ps->oldsa, NULL);
}

#elif LJ_PROFILE_PTHREAD

/* POSIX timer thread. */
static void *profile_thread(ProfileState *ps)
{
  int interval = ps->interval;
#if !LJ_TARGET_PS3
  struct timespec ts;
  ts.tv_sec = interval / 1000;
  ts.tv_nsec = (interval % 1000) * 1000000;
```

```
#endif
  while (1) {
#if LJ_TARGET_PS3
    sys_timer_usleep(interval * 1000);
#else
    nanosleep(&ts, NULL);
#endif
    if (ps->abort) break;
    profile_trigger(ps);
  }
  return NULL;
}

/* Start profiling timer thread. */
static void profile_timer_start(ProfileState *ps)
{
  pthread_mutex_init(&ps->lock, 0);
  ps->abort = 0;
  pthread_create(&ps->thread, NULL, (void *(*)(void *))profile_thread, ps);
}

/* Stop profiling timer thread. */
static void profile_timer_stop(ProfileState *ps)
{
  ps->abort = 1;
  pthread_join(ps->thread, NULL);
  pthread_mutex_destroy(&ps->lock);
}

#elif LJ_PROFILE_WTHREAD

/* Windows timer thread. */
static DWORD WINAPI profile_thread(void *psx)
{
  ProfileState *ps = (ProfileState *)psx;
  int interval = ps->interval;
#if LJ_TARGET_WINDOWS
  ps->wmm_tbp(interval);
#endif
  while (1) {
    Sleep(interval);
    if (ps->abort) break;
    profile_trigger(ps);
  }
#if LJ_TARGET_WINDOWS
  ps->wmm_tep(interval);
#endif
  return 0;
}

/* Start profiling timer thread. */
static void profile_timer_start(ProfileState *ps)
{
#if LJ_TARGET_WINDOWS
  if (!ps->wmm) {  /* Load WinMM library on-demand. */
    ps->wmm = LoadLibraryA("winmm.dll");
    if (ps->wmm) {
      ps->wmm_tbp = (WMM_TPFUNC)GetProcAddress(ps->wmm, "timeBeginPeriod");
      ps->wmm_tep = (WMM_TPFUNC)GetProcAddress(ps->wmm, "timeEndPeriod");
      if (!ps->wmm_tbp || !ps->wmm_tep) {
        ps->wmm = NULL;
        return;
      }
    }
  }
#endif
  InitializeCriticalSection(&ps->lock);
  ps->abort = 0;
  ps->thread = CreateThread(NULL, 0, profile_thread, ps, 0, NULL);
}

/* Stop profiling timer thread. */
static void profile_timer_stop(ProfileState *ps)
{
  ps->abort = 1;
  WaitForSingleObject(ps->thread, INFINITE);
```

```
  DeleteCriticalSection(&ps->lock);
}

#endif

/* -- Public profiling API ------------------------------------------- */

/* Start profiling. */
LUA_API void luaJIT_profile_start(lua_State *L, const char *mode,
                                  luaJIT_profile_callback cb, void *data)
{
  ProfileState *ps = &profile_state;
  int interval = LJ_PROFILE_INTERVAL_DEFAULT;
  while (*mode) {
    int m = *mode++;
    switch (m) {
    case 'i':
      interval = 0;
      while (*mode >= '0' && *mode <= '9')
        interval = interval * 10 + (*mode++ - '0');
      if (interval <= 0) interval = 1;
      break;
#if LJ_HASJIT
    case 'l': case 'f':
      L2J(L)->prof_mode = m;
      lj_trace_flushall(L);
      break;
#endif
    default:  /* Ignore unknown mode chars. */
      break;
    }
  }
  if (ps->g) {
    luaJIT_profile_stop(L);
    if (ps->g) return;  /* Profiler in use by another VM. */
  }
  ps->g = G(L);
  ps->interval = interval;
  ps->cb = cb;
  ps->data = data;
  ps->samples = 0;
  lj_buf_init(L, &ps->sb);
  profile_timer_start(ps);
}

/* Stop profiling. */
LUA_API void luaJIT_profile_stop(lua_State *L)
{
  ProfileState *ps = &profile_state;
  global_State *g = ps->g;
  if (G(L) == g) {  /* Only stop profiler if started by this VM. */
    profile_timer_stop(ps);
    g->hookmask &= ~HOOK_PROFILE;
    lj_dispatch_update(g);
#if LJ_HASJIT
    G2J(g)->prof_mode = 0;
    lj_trace_flushall(L);
#endif
    lj_buf_free(g, &ps->sb);
    setmref(ps->sb.b, NULL);
    setmref(ps->sb.e, NULL);
    ps->g = NULL;
  }
}

/* Return a compact stack dump. */
LUA_API const char *luaJIT_profile_dumpstack(lua_State *L, const char *fmt,
                                             int depth, size_t *len)
{
  ProfileState *ps = &profile_state;
  SBuf *sb = &ps->sb;
  setsbufL(sb, L);
  lj_buf_reset(sb);
  lj_debug_dumpstack(L, sb, fmt, depth);
  *len = (size_t)sbuflen(sb);
  return sbufB(sb);
```

```
366 }
367
368 #endif
```

One Level Up Top Level

src/lj_vmevent.h - luajit-2.0-src

Data types defined

- VMEvent

Macros defined

- LJ_VMEVENTS_HSIZE
- LJ_VMEVENTS_REGKEY
- VMEVENT_DEF
- VMEVENT_HASH
- VMEVENT_HASHIDX
- VMEVENT_MASK
- VMEVENT_NOCACHE
- _LJ_VMEVENT_H
- lj_vmevent_send
- lj_vmevent_send
- lj_vmevent_send_
- lj_vmevent_send_

Source code

```
1  /*
2  ** VM event handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_VMEVENT_H
7  #define _LJ_VMEVENT_H
8
9  #include "lj_obj.h"
10
11 /* Registry key for VM event handler table. */
12 #define LJ_VMEVENTS_REGKEY        "_VMEVENTS"
13 #define LJ_VMEVENTS_HSIZE        4
14
15 #define VMEVENT_MASK(ev)        ((uint8_t)1 << ((int)(ev) & 7))
16 #define VMEVENT_HASH(ev)        ((int)(ev) & ~7)
17 #define VMEVENT_HASHIDX(h)        ((int)(h) << 3)
18 #define VMEVENT_NOCACHE                255
19
20 #define VMEVENT_DEF(name, hash) \
21   LJ_VMEVENT_##name##_, \
22   LJ_VMEVENT_##name = ((LJ_VMEVENT_##name##_) & 7)|((hash) << 3)
23
24 /* VM event IDs. */
25 typedef enum {
26   VMEVENT_DEF(BC,        0x00003883),
27   VMEVENT_DEF(TRACE,        0xb2d91467),
28   VMEVENT_DEF(RECORD,        0x9284bf4f),
29   VMEVENT_DEF(TEXIT,        0xb29df2b0),
30   LJ_VMEVENT__MAX
31 } VMEvent;
32
```

```
#ifdef LUAJIT_DISABLE_VMEVENT
#define lj_vmevent_send(L, ev, args)                UNUSED(L)
#define lj_vmevent_send_(L, ev, args, post)         UNUSED(L)
#else
#define lj_vmevent_send(L, ev, args) \
  if (G(L)->vmevmask & VMEVENT_MASK(LJ_VMEVENT_##ev)) { \
    ptrdiff_t argbase = lj_vmevent_prepare(L, LJ_VMEVENT_##ev); \
    if (argbase) { \
      args \
      lj_vmevent_call(L, argbase); \
    } \
  }
#define lj_vmevent_send_(L, ev, args, post) \
  if (G(L)->vmevmask & VMEVENT_MASK(LJ_VMEVENT_##ev)) { \
    ptrdiff_t argbase = lj_vmevent_prepare(L, LJ_VMEVENT_##ev); \
    if (argbase) { \
      args \
      lj_vmevent_call(L, argbase); \
      post \
    } \
  }

LJ_FUNC ptrdiff_t lj_vmevent_prepare(lua_State *L, VMEvent ev);
LJ_FUNC void lj_vmevent_call(lua_State *L, ptrdiff_t argbase);
#endif

#endif
```

One Level Up Top Level

src/lj_vmevent.c - luajit-2.0-src

Functions defined

- lj_vmevent_call
- lj_vmevent_prepare

Macros defined

- LUA_CORE
- lj_vmevent_c

Source code

```
1  /*
2  ** VM event handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #include <stdio.h>
7
8  #define lj_vmevent_c
9  #define LUA_CORE
10
11 #include "lj_obj.h"
12 #include "lj_str.h"
13 #include "lj_tab.h"
14 #include "lj_state.h"
15 #include "lj_dispatch.h"
16 #include "lj_vm.h"
17 #include "lj_vmevent.h"
18
19 ptrdiff_t lj_vmevent_prepare(lua_State *L, VMEvent ev)
20 {
21   global_State *g = G(L);
22   GCstr *s = lj_str_newlit(L, LJ_VMEVENTS_REGKEY);
23   cTValue *tv = lj_tab_getstr(tabV(registry(L)), s);
24   if (tvistab(tv)) {
25     int hash = VMEVENT_HASH(ev);
26     tv = lj_tab_getint(tabV(tv), hash);
27     if (tv && tvisfunc(tv)) {
28       lj_state_checkstack(L, LUA_MINSTACK);
29       setfuncV(L, L->top++, funcV(tv));
30       if (LJ_FR2) setnilV(L->top++);
31       return savestack(L, L->top);
32     }
33   }
34   g->vmevmask &= ~VMEVENT_MASK(ev);  /* No handler: cache this fact. */
35   return 0;
36 }
37
38 void lj_vmevent_call(lua_State *L, ptrdiff_t argbase)
39 {
40   global_State *g = G(L);
41   uint8_t oldmask = g->vmevmask;
42   uint8_t oldh = hook_save(g);
43   int status;
44   g->vmevmask = 0;  /* Disable all events. */
45   hook_vmevent(g);
46   status = lj_vm_pcall(L, restorestack(L, argbase), 0+1, 0);
47   if (LJ_UNLIKELY(status)) {
48     /* Really shouldn't use stderr here, but where else to complain? */
49     L->top--;
50     fputs("VM handler failed: ", stderr);
51     fputs(tvisstr(L->top) ? strVdata(L->top) : "?", stderr);
52     fputc('\n', stderr);
```

```
  }
  hook_restore(g, oldh);
  if (g->vmevmask != VMEVENT_NOCACHE)
    g->vmevmask = oldmask;  /* Restore event mask, but not if not modified. */
}

```

One Level Up Top Level

src/lib_ffi.c - luajit-2.0-src

Functions defined

- LJLIB_CF(ffi_meta___index) LJLIB_REC(cdata_index 0)
- LJLIB_CF(ffi_meta___newindex) LJLIB_REC(cdata_index 1)
- LJLIB_CF(ffi_meta___eq) LJLIB_REC(cdata_arith MM_eq)
- LJLIB_CF(ffi_meta___len) LJLIB_REC(cdata_arith MM_len)
- LJLIB_CF(ffi_meta___lt) LJLIB_REC(cdata_arith MM_lt)
- LJLIB_CF(ffi_meta___le) LJLIB_REC(cdata_arith MM_le)
- LJLIB_CF(ffi_meta___concat) LJLIB_REC(cdata_arith MM_concat)
- LJLIB_CF(ffi_meta___call) LJLIB_REC(cdata_call)
- LJLIB_CF(ffi_meta___add) LJLIB_REC(cdata_arith MM_add)
- LJLIB_CF(ffi_meta___sub) LJLIB_REC(cdata_arith MM_sub)
- LJLIB_CF(ffi_meta___mul) LJLIB_REC(cdata_arith MM_mul)
- LJLIB_CF(ffi_meta___div) LJLIB_REC(cdata_arith MM_div)
- LJLIB_CF(ffi_meta___mod) LJLIB_REC(cdata_arith MM_mod)
- LJLIB_CF(ffi_meta___pow) LJLIB_REC(cdata_arith MM_pow)
- LJLIB_CF(ffi_meta___unm) LJLIB_REC(cdata_arith MM_unm)
- LJLIB_CF(ffi_meta___tostring)
- LJLIB_CF(ffi_meta___pairs)
- LJLIB_CF(ffi_meta___ipairs)
- LJLIB_CF(ffi_clib___index) LJLIB_REC(clib_index 1)
- LJLIB_CF(ffi_clib___newindex) LJLIB_REC(clib_index 0)
- LJLIB_CF(ffi_clib___gc)
- LJLIB_CF(ffi_callback_free)
- LJLIB_CF(ffi_callback_set)
- LJLIB_CF(ffi_new) LJLIB_REC(.)
- LJLIB_CF(ffi_cast) LJLIB_REC(ffi_new)
- LJLIB_CF(ffi_typeof) LJLIB_REC(.)
- LJLIB_CF(ffi_typeinfo)
- LJLIB_CF(ffi_istype) LJLIB_REC(.)
- LJLIB_CF(ffi_sizeof) LJLIB_REC(ffi_xof FF_ffi_sizeof)
- LJLIB_CF(ffi_alignof) LJLIB_REC(ffi_xof FF_ffi_alignof)

- LJLIB_CF(ffi_offsetof) LJLIB_REC(ffi_xof FF_ffi_offsetof)
- LJLIB_CF(ffi_errno) LJLIB_REC(.)
- LJLIB_CF(ffi_string) LJLIB_REC(.)
- LJLIB_CF(ffi_copy) LJLIB_REC(.)
- LJLIB_CF(ffi_fill) LJLIB_REC(.)
- LJLIB_CF(ffi_abi) LJLIB_REC(.)
- LJLIB_CF(ffi_metatype)
- LJLIB_CF(ffi_gc) LJLIB_REC(.)
- LJLIB_CF(ffi_load)
- LJLIB_PUSH(top-1) LJLIB_SET(__index)
- ffi_arith
- ffi_callback_set
- ffi_checkcdata
- ffi_checkctype
- ffi_checkint
- ffi_checkptr
- ffi_clib_index
- ffi_finalizer
- ffi_index_meta
- ffi_pairs
- ffi_register_module
- luaopen_ffi

Macros defined

- H_
- H_
- LJLIB_MODULE_ffi
- LJLIB_MODULE_ffi_callback
- LJLIB_MODULE_ffi_clib
- LJLIB_MODULE_ffi_meta
- LUA_LIB
- lib_ffi_c

Source code

```
1 /*
```

```c
/*
** FFI library.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lib_ffi_c
#define LUA_LIB

#include <errno.h>

#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

#include "lj_obj.h"

#if LJ_HASFFI

#include "lj_gc.h"
#include "lj_err.h"
#include "lj_str.h"
#include "lj_tab.h"
#include "lj_meta.h"
#include "lj_ctype.h"
#include "lj_cparse.h"
#include "lj_cdata.h"
#include "lj_cconv.h"
#include "lj_carith.h"
#include "lj_ccall.h"
#include "lj_ccallback.h"
#include "lj_clib.h"
#include "lj_strfmt.h"
#include "lj_ff.h"
#include "lj_lib.h"

/* -- C type checks ------------------------------------------------------- */

/* Check first argument for a C type and returns its ID. */
static CTypeID ffi_checkctype(lua_State *L, CTState *cts, TValue *param)
{
  TValue *o = L->base;
  if (!(o < L->top)) {
  err_argtype:
    lj_err_argtype(L, 1, "C type");
  }
  if (tvisstr(o)) {  /* Parse an abstract C type declaration. */
    GCstr *s = strV(o);
    CPState cp;
    int errcode;
    cp.L = L;
    cp.cts = cts;
    cp.srcname = strdata(s);
    cp.p = strdata(s);
    cp.param = param;
    cp.mode = CPARSE_MODE_ABSTRACT|CPARSE_MODE_NOIMPLICIT;
    errcode = lj_cparse(&cp);
    if (errcode) lj_err_throw(L, errcode);  /* Propagate errors. */
    return cp.val.id;
  } else {
    GCcdata *cd;
    if (!tviscdata(o)) goto err_argtype;
    if (param && param < L->top) lj_err_arg(L, 1, LJ_ERR_FFI_NUMPARAM);
    cd = cdataV(o);
    return cd->ctypeid == CTID_CTYPEID ? *(CTypeID *)cdataptr(cd) : cd->ctypeid;
  }
}

/* Check argument for C data and return it. */
static GCcdata *ffi_checkcdata(lua_State *L, int narg)
{
  TValue *o = L->base + narg-1;
  if (!(o < L->top && tviscdata(o)))
    lj_err_argt(L, narg, LUA_TCDATA);
  return cdataV(o);
}

/* Convert argument to C pointer. */
```

```
78  static void *ffi_checkptr(lua_State *L, int narg, CTypeID id)
79  {
80    CTState *cts = ctype_cts(L);
81    TValue *o = L->base + narg-1;
82    void *p;
83    if (o >= L->top)
84      lj_err_arg(L, narg, LJ_ERR_NOVAL);
85    lj_cconv_ct_tv(cts, ctype_get(cts, id), (uint8_t *)&p, o, CCF_ARG(narg));
86    return p;
87  }
88
89  /* Convert argument to int32_t. */
90  static int32_t ffi_checkint(lua_State *L, int narg)
91  {
92    CTState *cts = ctype_cts(L);
93    TValue *o = L->base + narg-1;
94    int32_t i;
95    if (o >= L->top)
96      lj_err_arg(L, narg, LJ_ERR_NOVAL);
97    lj_cconv_ct_tv(cts, ctype_get(cts, CTID_INT32), (uint8_t *)&i, o,
98                   CCF_ARG(narg));
99    return i;
100 }
101
102 /* -- C type metamethods ------------------------------------------------ */
103
104 #define LJLIB_MODULE_ffi_meta
105
106 /* Handle ctype __index/__newindex metamethods. */
107 static int ffi_index_meta(lua_State *L, CTState *cts, CType *ct, MMS mm)
108 {
109   CTypeID id = ctype_typeid(cts, ct);
110   CTValue *tv = lj_ctype_meta(cts, id, mm);
111   TValue *base = L->base;
112   if (!tv) {
113     const char *s;
114   err_index:
115     s = strdata(lj_ctype_repr(L, id, NULL));
116     if (tvisstr(L->base+1)) {
117       lj_err_callerv(L, LJ_ERR_FFI_BADMEMBER, s, strVdata(L->base+1));
118     } else {
119       const char *key = tviscdata(L->base+1) ?
120         strdata(lj_ctype_repr(L, cdataV(L->base+1)->ctypeid, NULL)) :
121         lj_typename(L->base+1);
122       lj_err_callerv(L, LJ_ERR_FFI_BADIDXW, s, key);
123     }
124   }
125   if (!tvisfunc(tv)) {
126     if (mm == MM_index) {
127       CTValue *o = lj_meta_tget(L, tv, base+1);
128       if (o) {
129         if (tvisnil(o)) goto err_index;
130         copyTV(L, L->top-1, o);
131         return 1;
132       }
133     } else {
134       TValue *o = lj_meta_tset(L, tv, base+1);
135       if (o) {
136         copyTV(L, o, base+2);
137         return 0;
138       }
139     }
140     copyTV(L, base, L->top);
141     tv = L->top-1-LJ_FR2;
142   }
143   return lj_meta_tailcall(L, tv);
144 }
145
146 LJLIB_CF(ffi_meta___index)        LJLIB_REC(cdata_index 0)
147 {
148   CTState *cts = ctype_cts(L);
149   CTInfo qual = 0;
150   CType *ct;
151   uint8_t *p;
152   TValue *o = L->base;
153   if (!(o+1 < L->top && tviscdata(o)))  /* Also checks for presence of key. */
```

```
    lj_err_argt(L, 1, LUA_TCDATA);
  ct = lj_cdata_index(cts, cdataV(o), o+1, &p, &qual);
  if ((qual & 1))
    return ffi_index_meta(L, cts, ct, MM_index);
  if (lj_cdata_get(cts, ct, L->top-1, p))
    lj_gc_check(L);
  return 1;
}

LJLIB_CF(ffi_meta___newindex)        LJLIB_REC(cdata_index 1)
{
  CTState *cts = ctype_cts(L);
  CTInfo qual = 0;
  CType *ct;
  uint8_t *p;
  TValue *o = L->base;
  if (!(o+2 < L->top && tviscdata(o)))  /* Also checks for key and value. */
    lj_err_argt(L, 1, LUA_TCDATA);
  ct = lj_cdata_index(cts, cdataV(o), o+1, &p, &qual);
  if ((qual & 1)) {
    if ((qual & CTF_CONST))
      lj_err_caller(L, LJ_ERR_FFI_WRCONST);
    return ffi_index_meta(L, cts, ct, MM_newindex);
  }
  lj_cdata_set(cts, ct, p, o+2, qual);
  return 0;
}

/* Common handler for cdata arithmetic. */
static int ffi_arith(lua_State *L)
{
  MMS mm = (MMS)(curr_func(L)->c.ffid - (int)FF_ffi_meta___eq + (int)MM_eq);
  return lj_carith_op(L, mm);
}

/* The following functions must be in contiguous ORDER MM. */
LJLIB_CF(ffi_meta___eq)              LJLIB_REC(cdata_arith MM_eq)
{
  return ffi_arith(L);
}

LJLIB_CF(ffi_meta___len)        LJLIB_REC(cdata_arith MM_len)
{
  return ffi_arith(L);
}

LJLIB_CF(ffi_meta___lt)              LJLIB_REC(cdata_arith MM_lt)
{
  return ffi_arith(L);
}

LJLIB_CF(ffi_meta___le)              LJLIB_REC(cdata_arith MM_le)
{
  return ffi_arith(L);
}

LJLIB_CF(ffi_meta___concat)        LJLIB_REC(cdata_arith MM_concat)
{
  return ffi_arith(L);
}

/* Forward declaration. */
static int lj_cf_ffi_new(lua_State *L);

LJLIB_CF(ffi_meta___call)        LJLIB_REC(cdata_call)
{
  CTState *cts = ctype_cts(L);
  GCcdata *cd = ffi_checkcdata(L, 1);
  CTypeID id = cd->ctypeid;
  CType *ct;
  cTValue *tv;
  MMS mm = MM_call;
  if (cd->ctypeid == CTID_CTYPEID) {
    id = *(CTypeID *)cdataptr(cd);
    mm = MM_new;
  } else {
```

```
    int ret = lj_ccall_func(L, cd);
    if (ret >= 0)
      return ret;
  }
  /* Handle ctype __call/__new metamethod. */
  ct = ctype_raw(cts, id);
  if (ctype_isptr(ct->info)) id = ctype_cid(ct->info);
  tv = lj_ctype_meta(cts, id, mm);
  if (tv)
    return lj_meta_tailcall(L, tv);
  else if (mm == MM_call)
    lj_err_callerv(L, LJ_ERR_FFI_BADCALL, strdata(lj_ctype_repr(L, id, NULL)));
  return lj_cf_ffi_new(L);
}

LJLIB_CF(ffi_meta___add)        LJLIB_REC(cdata_arith MM_add)
{
  return ffi_arith(L);
}

LJLIB_CF(ffi_meta___sub)        LJLIB_REC(cdata_arith MM_sub)
{
  return ffi_arith(L);
}

LJLIB_CF(ffi_meta___mul)        LJLIB_REC(cdata_arith MM_mul)
{
  return ffi_arith(L);
}

LJLIB_CF(ffi_meta___div)        LJLIB_REC(cdata_arith MM_div)
{
  return ffi_arith(L);
}

LJLIB_CF(ffi_meta___mod)        LJLIB_REC(cdata_arith MM_mod)
{
  return ffi_arith(L);
}

LJLIB_CF(ffi_meta___pow)        LJLIB_REC(cdata_arith MM_pow)
{
  return ffi_arith(L);
}

LJLIB_CF(ffi_meta___unm)        LJLIB_REC(cdata_arith MM_unm)
{
  return ffi_arith(L);
}
/* End of contiguous ORDER MM. */

LJLIB_CF(ffi_meta___tostring)
{
  GCcdata *cd = ffi_checkcdata(L, 1);
  const char *msg = "cdata<%s>: %p";
  CTypeID id = cd->ctypeid;
  void *p = cdataptr(cd);
  if (id == CTID_CTYPEID) {
    msg = "ctype<%s>";
    id = *(CTypeID *)p;
  } else {
    CTState *cts = ctype_cts(L);
    CType *ct = ctype_raw(cts, id);
    if (ctype_isref(ct->info)) {
      p = *(void **)p;
      ct = ctype_rawchild(cts, ct);
    }
    if (ctype_iscomplex(ct->info)) {
      setstrV(L, L->top-1, lj_ctype_repr_complex(L, cdataptr(cd), ct->size));
      goto checkgc;
    } else if (ct->size == 8 && ctype_isinteger(ct->info)) {
      setstrV(L, L->top-1, lj_ctype_repr_int64(L, *(uint64_t *)cdataptr(cd),
                                               (ct->info & CTF_UNSIGNED)));
      goto checkgc;
    } else if (ctype_isfunc(ct->info)) {
      p = *(void **)p;
```

```
    } else if (ctype_isenum(ct->info)) {
      msg = "cdata<%s>: %d";
      p = (void *)(uintptr_t)*(uint32_t **)p;
    } else {
      if (ctype_isptr(ct->info)) {
        p = cdata_getptr(p, ct->size);
        ct = ctype_rawchild(cts, ct);
      }
      if (ctype_isstruct(ct->info) || ctype_isvector(ct->info)) {
        /* Handle ctype __tostring metamethod. */
        cTValue *tv = lj_ctype_meta(cts, ctype_typeid(cts, ct), MM_tostring);
        if (tv)
          return lj_meta_tailcall(L, tv);
      }
    }
  }
  lj_strfmt_pushf(L, msg, strdata(lj_ctype_repr(L, id, NULL)), p);
checkgc:
  lj_gc_check(L);
  return 1;
}

static int ffi_pairs(lua_State *L, MMS mm)
{
  CTState *cts = ctype_cts(L);
  CTypeID id = ffi_checkcdata(L, 1)->ctypeid;
  CType *ct = ctype_raw(cts, id);
  cTValue *tv;
  if (ctype_isptr(ct->info)) id = ctype_cid(ct->info);
  tv = lj_ctype_meta(cts, id, mm);
  if (!tv)
    lj_err_callerv(L, LJ_ERR_FFI_BADMM, strdata(lj_ctype_repr(L, id, NULL)),
                   strdata(mmname_str(G(L), mm)));
  return lj_meta_tailcall(L, tv);
}

LJLIB_CF(ffi_meta___pairs)
{
  return ffi_pairs(L, MM_pairs);
}

LJLIB_CF(ffi_meta___ipairs)
{
  return ffi_pairs(L, MM_ipairs);
}

LJLIB_PUSH("ffi") LJLIB_SET(__metatable)

#include "lj_libdef.h"

/* -- C library metamethods ----------------------------------------------- */

#define LJLIB_MODULE_ffi_clib

/* Index C library by a name. */
static TValue *ffi_clib_index(lua_State *L)
{
  TValue *o = L->base;
  CLibrary *cl;
  if (!(o < L->top && tvisudata(o) && udataV(o)->udtype == UDTYPE_FFI_CLIB))
    lj_err_argt(L, 1, LUA_TUSERDATA);
  cl = (CLibrary *)uddata(udataV(o));
  if (!(o+1 < L->top && tvisstr(o+1)))
    lj_err_argt(L, 2, LUA_TSTRING);
  return lj_clib_index(L, cl, strV(o+1));
}

LJLIB_CF(ffi_clib___index)        LJLIB_REC(clib_index 1)
{
  TValue *tv = ffi_clib_index(L);
  if (tviscdata(tv)) {
    CTState *cts = ctype_cts(L);
    GCcdata *cd = cdataV(tv);
    CType *s = ctype_get(cts, cd->ctypeid);
    if (ctype_isextern(s->info)) {
      CTypeID sid = ctype_cid(s->info);
```

```
      void *sp = *(void **)cdataptr(cd);
      CType *ct = ctype_raw(cts, sid);
      if (lj_cconv_tv_ct(cts, ct, sid, L->top-1, sp))
        lj_gc_check(L);
      return 1;
    }
  }
  copyTV(L, L->top-1, tv);
  return 1;
}

LJLIB_CF(ffi_clib___newindex)        LJLIB_REC(clib_index 0)
{
  TValue *tv = ffi_clib_index(L);
  TValue *o = L->base+2;
  if (o < L->top && tviscdata(tv)) {
    CTState *cts = ctype_cts(L);
    GCcdata *cd = cdataV(tv);
    CType *d = ctype_get(cts, cd->ctypeid);
    if (ctype_isextern(d->info)) {
      CTInfo qual = 0;
      for (;;) {  /* Skip attributes and collect qualifiers. */
        d = ctype_child(cts, d);
        if (!ctype_isattrib(d->info)) break;
        if (ctype_attrib(d->info) == CTA_QUAL) qual |= d->size;
      }
      if (!((d->info|qual) & CTF_CONST)) {
        lj_cconv_ct_tv(cts, d, *(void **)cdataptr(cd), o, 0);
        return 0;
      }
    }
  }
  lj_err_caller(L, LJ_ERR_FFI_WRCONST);
  return 0;  /* unreachable */
}

LJLIB_CF(ffi_clib___gc)
{
  TValue *o = L->base;
  if (o < L->top && tvisudata(o) && udataV(o)->udtype == UDTYPE_FFI_CLIB)
    lj_clib_unload((CLibrary *)uddata(udataV(o)));
  return 0;
}

#include "lj_libdef.h"

/* -- Callback function metamethods --------------------------------------- */

#define LJLIB_MODULE_ffi_callback

static int ffi_callback_set(lua_State *L, GCfunc *fn)
{
  GCcdata *cd = ffi_checkcdata(L, 1);
  CTState *cts = ctype_cts(L);
  CType *ct = ctype_raw(cts, cd->ctypeid);
  if (ctype_isptr(ct->info) && (LJ_32 || ct->size == 8)) {
    MSize slot = lj_ccallback_ptr2slot(cts, *(void **)cdataptr(cd));
    if (slot < cts->cb.sizeid && cts->cb.cbid[slot] != 0) {
      GCtab *t = cts->miscmap;
      TValue *tv = lj_tab_setint(L, t, (int32_t)slot);
      if (fn) {
        setfuncV(L, tv, fn);
        lj_gc_anybarriert(L, t);
      } else {
        setnilV(tv);
        cts->cb.cbid[slot] = 0;
        cts->cb.topid = slot < cts->cb.topid ? slot : cts->cb.topid;
      }
      return 0;
    }
  }
  lj_err_caller(L, LJ_ERR_FFI_BADCBACK);
  return 0;
}

LJLIB_CF(ffi_callback_free)
```

```
{
  return ffi_callback_set(L, NULL);
}

LJLIB_CF(ffi_callback_set)
{
  GCfunc *fn = lj_lib_checkfunc(L, 2);
  return ffi_callback_set(L, fn);
}

LJLIB_PUSH(top-1) LJLIB_SET(__index)

#include "lj_libdef.h"

/* -- FFI library functions ---------------------------------------------- */

#define LJLIB_MODULE_ffi

LJLIB_CF(ffi_cdef)
{
  GCstr *s = lj_lib_checkstr(L, 1);
  CPState cp;
  int errcode;
  cp.L = L;
  cp.cts = ctype_cts(L);
  cp.srcname = strdata(s);
  cp.p = strdata(s);
  cp.param = L->base+1;
  cp.mode = CPARSE_MODE_MULTI|CPARSE_MODE_DIRECT;
  errcode = lj_cparse(&cp);
  if (errcode) lj_err_throw(L, errcode);  /* Propagate errors. */
  lj_gc_check(L);
  return 0;
}

LJLIB_CF(ffi_new)        LJLIB_REC(.)
{
  CTState *cts = ctype_cts(L);
  CTypeID id = ffi_checkctype(L, cts, NULL);
  CType *ct = ctype_raw(cts, id);
  CTSize sz;
  CTInfo info = lj_ctype_info(cts, id, &sz);
  TValue *o = L->base+1;
  GCcdata *cd;
  if ((info & CTF_VLA)) {
    o++;
    sz = lj_ctype_vlsize(cts, ct, (CTSize)ffi_checkint(L, 2));
  }
  if (sz == CTSIZE_INVALID)
    lj_err_arg(L, 1, LJ_ERR_FFI_INVSIZE);
  if (!(info & CTF_VLA) && ctype_align(info) <= CT_MEMALIGN)
    cd = lj_cdata_new(cts, id, sz);
  else
    cd = lj_cdata_newv(L, id, sz, ctype_align(info));
  setcdataV(L, o-1, cd);  /* Anchor the uninitialized cdata. */
  lj_cconv_ct_init(cts, ct, sz, cdataptr(cd),
                   o, (MSize)(L->top - o));  /* Initialize cdata. */
  if (ctype_isstruct(ct->info)) {
    /* Handle ctype __gc metamethod. Use the fast lookup here. */
    cTValue *tv = lj_tab_getinth(cts->miscmap, -(int32_t)id);
    if (tv && tvistab(tv) && (tv = lj_meta_fast(L, tabV(tv), MM_gc))) {
      GCtab *t = cts->finalizer;
      if (gcref(t->metatable)) {
        /* Add to finalizer table, if still enabled. */
        copyTV(L, lj_tab_set(L, t, o-1), tv);
        lj_gc_anybarriert(L, t);
        cd->marked |= LJ_GC_CDATA_FIN;
      }
    }
  }
  L->top = o;  /* Only return the cdata itself. */
  lj_gc_check(L);
  return 1;
}

LJLIB_CF(ffi_cast)        LJLIB_REC(ffi_new)
```

```
{
  CTState *cts = ctype_cts(L);
  CTypeID id = ffi_checkctype(L, cts, NULL);
  CType *d = ctype_raw(cts, id);
  TValue *o = lj_lib_checkany(L, 2);
  L->top = o+1;  /* Make sure this is the last item on the stack. */
  if (!(ctype_isnum(d->info) || ctype_isptr(d->info) || ctype_isenum(d->info)))
    lj_err_arg(L, 1, LJ_ERR_FFI_INVTYPE);
  if (!(tviscdata(o) && cdataV(o)->ctypeid == id)) {
    GCcdata *cd = lj_cdata_new(cts, id, d->size);
    lj_cconv_ct_tv(cts, d, cdataptr(cd), o, CCF_CAST);
    setcdataV(L, o, cd);
    lj_gc_check(L);
  }
  return 1;
}

LJLIB_CF(ffi_typeof)        LJLIB_REC(.)
{
  CTState *cts = ctype_cts(L);
  CTypeID id = ffi_checkctype(L, cts, L->base+1);
  GCcdata *cd = lj_cdata_new(cts, CTID_CTYPEID, 4);
  *(CTypeID *)cdataptr(cd) = id;
  setcdataV(L, L->top-1, cd);
  lj_gc_check(L);
  return 1;
}

/* Internal and unsupported API. */
LJLIB_CF(ffi_typeinfo)
{
  CTState *cts = ctype_cts(L);
  CTypeID id = (CTypeID)ffi_checkint(L, 1);
  if (id > 0 && id < cts->top) {
    CType *ct = ctype_get(cts, id);
    GCtab *t;
    lua_createtable(L, 0, 4);  /* Increment hash size if fields are added. */
    t = tabV(L->top-1);
    setintV(lj_tab_setstr(L, t, lj_str_newlit(L, "info")), (int32_t)ct->info);
    if (ct->size != CTSIZE_INVALID)
      setintV(lj_tab_setstr(L, t, lj_str_newlit(L, "size")), (int32_t)ct->size);
    if (ct->sib)
      setintV(lj_tab_setstr(L, t, lj_str_newlit(L, "sib")), (int32_t)ct->sib);
    if (gcref(ct->name)) {
      GCstr *s = gco2str(gcref(ct->name));
      setstrV(L, lj_tab_setstr(L, t, lj_str_newlit(L, "name")), s);
    }
    lj_gc_check(L);
    return 1;
  }
  return 0;
}

LJLIB_CF(ffi_istype)        LJLIB_REC(.)
{
  CTState *cts = ctype_cts(L);
  CTypeID id1 = ffi_checkctype(L, cts, NULL);
  TValue *o = lj_lib_checkany(L, 2);
  int b = 0;
  if (tviscdata(o)) {
    GCcdata *cd = cdataV(o);
    CTypeID id2 = cd->ctypeid == CTID_CTYPEID ? *(CTypeID *)cdataptr(cd) :
                                                cd->ctypeid;
    CType *ct1 = lj_ctype_rawref(cts, id1);
    CType *ct2 = lj_ctype_rawref(cts, id2);
    if (ct1 == ct2) {
      b = 1;
    } else if (ctype_type(ct1->info) == ctype_type(ct2->info) &&
               ct1->size == ct2->size) {
      if (ctype_ispointer(ct1->info))
        b = lj_cconv_compatptr(cts, ct1, ct2, CCF_IGNQUAL);
      else if (ctype_isnum(ct1->info) || ctype_isvoid(ct1->info))
        b = (((ct1->info ^ ct2->info) & ~(CTF_QUAL|CTF_LONG)) == 0);
    } else if (ctype_isstruct(ct1->info) && ctype_isptr(ct2->info) &&
               ct1 == ctype_rawchild(cts, ct2)) {
      b = 1;
```

```
    }
  }
  setboolV(L->top-1, b);
  setboolV(&G(L)->tmptv2, b);  /* Remember for trace recorder. */
  return 1;
}

LJLIB_CF(ffi_sizeof)        LJLIB_REC(ffi_xof FF_ffi_sizeof)
{
  CTState *cts = ctype_cts(L);
  CTypeID id = ffi_checkctype(L, cts, NULL);
  CTSize sz;
  if (LJ_UNLIKELY(tviscdata(L->base) && cdataisv(cdataV(L->base)))) {
    sz = cdatavlen(cdataV(L->base));
  } else {
    CType *ct = lj_ctype_rawref(cts, id);
    if (ctype_isvltype(ct->info))
      sz = lj_ctype_vlsize(cts, ct, (CTSize)ffi_checkint(L, 2));
    else
      sz = ctype_hassize(ct->info) ? ct->size : CTSIZE_INVALID;
    if (LJ_UNLIKELY(sz == CTSIZE_INVALID)) {
      setnilV(L->top-1);
      return 1;
    }
  }
  setintV(L->top-1, (int32_t)sz);
  return 1;
}

LJLIB_CF(ffi_alignof)        LJLIB_REC(ffi_xof FF_ffi_alignof)
{
  CTState *cts = ctype_cts(L);
  CTypeID id = ffi_checkctype(L, cts, NULL);
  CTSize sz = 0;
  CTInfo info = lj_ctype_info(cts, id, &sz);
  setintV(L->top-1, 1 << ctype_align(info));
  return 1;
}

LJLIB_CF(ffi_offsetof)        LJLIB_REC(ffi_xof FF_ffi_offsetof)
{
  CTState *cts = ctype_cts(L);
  CTypeID id = ffi_checkctype(L, cts, NULL);
  GCstr *name = lj_lib_checkstr(L, 2);
  CType *ct = lj_ctype_rawref(cts, id);
  CTSize ofs;
  if (ctype_isstruct(ct->info) && ct->size != CTSIZE_INVALID) {
    CType *fct = lj_ctype_getfield(cts, ct, name, &ofs);
    if (fct) {
      setintV(L->top-1, ofs);
      if (ctype_isfield(fct->info)) {
        return 1;
      } else if (ctype_isbitfield(fct->info)) {
        setintV(L->top++, ctype_bitpos(fct->info));
        setintV(L->top++, ctype_bitbsz(fct->info));
        return 3;
      }
    }
  }
  return 0;
}

LJLIB_CF(ffi_errno)        LJLIB_REC(.)
{
  int err = errno;
  if (L->top > L->base)
    errno = ffi_checkint(L, 1);
  setintV(L->top++, err);
  return 1;
}

LJLIB_CF(ffi_string)        LJLIB_REC(.)
{
  CTState *cts = ctype_cts(L);
  TValue *o = lj_lib_checkany(L, 1);
  const char *p;
```

```
  size_t len;
  if (o+1 < L->top && !tvisnil(o+1)) {
    len = (size_t)ffi_checkint(L, 2);
    lj_cconv_ct_tv(cts, ctype_get(cts, CTID_P_CVOID), (uint8_t *)&p, o,
                   CCF_ARG(1));
  } else {
    lj_cconv_ct_tv(cts, ctype_get(cts, CTID_P_CCHAR), (uint8_t *)&p, o,
                   CCF_ARG(1));
    len = strlen(p);
  }
  L->top = o+1;  /* Make sure this is the last item on the stack. */
  setstrV(L, o, lj_str_new(L, p, len));
  lj_gc_check(L);
  return 1;
}

LJLIB_CF(ffi_copy)        LJLIB_REC(.)
{
  void *dp = ffi_checkptr(L, 1, CTID_P_VOID);
  void *sp = ffi_checkptr(L, 2, CTID_P_CVOID);
  TValue *o = L->base+1;
  CTSize len;
  if (tvisstr(o) && o+1 >= L->top)
    len = strV(o)->len+1;  /* Copy Lua string including trailing '\0'. */
  else
    len = (CTSize)ffi_checkint(L, 3);
  memcpy(dp, sp, len);
  return 0;
}

LJLIB_CF(ffi_fill)        LJLIB_REC(.)
{
  void *dp = ffi_checkptr(L, 1, CTID_P_VOID);
  CTSize len = (CTSize)ffi_checkint(L, 2);
  int32_t fill = 0;
  if (L->base+2 < L->top && !tvisnil(L->base+2)) fill = ffi_checkint(L, 3);
  memset(dp, fill, len);
  return 0;
}

#define H_(le, be)        LJ_ENDIAN_SELECT(0x##le, 0x##be)

/* Test ABI string. */
LJLIB_CF(ffi_abi)        LJLIB_REC(.)
{
  GCstr *s = lj_lib_checkstr(L, 1);
  int b = 0;
  switch (s->hash) {
#if LJ_64
  case H_(849858eb,ad35fd06): b = 1; break;  /* 64bit */
#else
  case H_(662d3c79,d0e22477): b = 1; break;  /* 32bit */
#endif
#if LJ_ARCH_HASFPU
  case H_(e33ee463,e33ee463): b = 1; break;  /* fpu */
#endif
#if LJ_ABI_SOFTFP
  case H_(61211a23,c2e8c81c): b = 1; break;  /* softfp */
#else
  case H_(539417a8,8ce0812f): b = 1; break;  /* hardfp */
#endif
#if LJ_ABI_EABI
  case H_(2182df8f,f2ed1152): b = 1; break;  /* eabi */
#endif
#if LJ_ABI_WIN
  case H_(4ab624a8,4ab624a8): b = 1; break;  /* win */
#endif
  case H_(3af93066,1f001464): b = 1; break;  /* le/be */
#if LJ_GC64
  case H_(9e89d2c9,13c83c92): b = 1; break;  /* gc64 */
#endif
  default:
    break;
  }
  setboolV(L->top-1, b);
  setboolV(&G(L)->tmptv2, b);  /* Remember for trace recorder. */
```

```
  return 1;
}

#undef H_

LJLIB_PUSH(top-8) LJLIB_SET(!)  /* Store reference to miscmap table. */

LJLIB_CF(ffi_metatype)
{
  CTState *cts = ctype_cts(L);
  CTypeID id = ffi_checkctype(L, cts, NULL);
  GCtab *mt = lj_lib_checktab(L, 2);
  GCtab *t = cts->miscmap;
  CType *ct = ctype_get(cts, id);  /* Only allow raw types. */
  TValue *tv;
  GCcdata *cd;
  if (!(ctype_isstruct(ct->info) || ctype_iscomplex(ct->info) ||
        ctype_isvector(ct->info)))
    lj_err_arg(L, 1, LJ_ERR_FFI_INVTYPE);
  tv = lj_tab_setinth(L, t, -(int32_t)id);
  if (!tvisnil(tv))
    lj_err_caller(L, LJ_ERR_PROTMT);
  settabV(L, tv, mt);
  lj_gc_anybarriert(L, t);
  cd = lj_cdata_new(cts, CTID_CTYPEID, 4);
  *(CTypeID *)cdataptr(cd) = id;
  setcdataV(L, L->top-1, cd);
  lj_gc_check(L);
  return 1;
}

LJLIB_PUSH(top-7) LJLIB_SET(!)  /* Store reference to finalizer table. */

LJLIB_CF(ffi_gc)        LJLIB_REC(.)
{
  GCcdata *cd = ffi_checkcdata(L, 1);
  TValue *fin = lj_lib_checkany(L, 2);
  CTState *cts = ctype_cts(L);
  CType *ct = ctype_raw(cts, cd->ctypeid);
  if (!(ctype_isptr(ct->info) || ctype_isstruct(ct->info) ||
        ctype_isrefarray(ct->info)))
    lj_err_arg(L, 1, LJ_ERR_FFI_INVTYPE);
  lj_cdata_setfin(L, cd, gcval(fin), itype(fin));
  L->top = L->base+1;  /* Pass through the cdata object. */
  return 1;
}

LJLIB_PUSH(top-5) LJLIB_SET(!)  /* Store clib metatable in func environment. */

LJLIB_CF(ffi_load)
{
  GCstr *name = lj_lib_checkstr(L, 1);
  int global = (L->base+1 < L->top && tvistruecond(L->base+1));
  lj_clib_load(L, tabref(curr_func(L)->c.env), name, global);
  return 1;
}

LJLIB_PUSH(top-4) LJLIB_SET(C)
LJLIB_PUSH(top-3) LJLIB_SET(os)
LJLIB_PUSH(top-2) LJLIB_SET(arch)

#include "lj_libdef.h"

/* ------------------------------------------------------------------------ */

/* Create special weak-keyed finalizer table. */
static GCtab *ffi_finalizer(lua_State *L)
{
  /* NOBARRIER: The table is new (marked white). */
  GCtab *t = lj_tab_new(L, 0, 1);
  settabV(L, L->top++, t);
  setgcref(t->metatable, obj2gco(t));
  setstrV(L, lj_tab_setstr(L, t, lj_str_newlit(L, "__mode")),
          lj_str_newlit(L, "K"));
  t->nomm = (uint8_t)(~(1u<<MM_mode));
  return t;
```

```
}

/* Register FFI module as loaded. */
static void ffi_register_module(lua_State *L)
{
  cTValue *tmp = lj_tab_getstr(tabV(registry(L)), lj_str_newlit(L, "_LOADED"));
  if (tmp && tvistab(tmp)) {
    GCtab *t = tabV(tmp);
    copyTV(L, lj_tab_setstr(L, t, lj_str_newlit(L, LUA_FFILIBNAME)), L->top-1);
    lj_gc_anybarriert(L, t);
  }
}

LUALIB_API int luaopen_ffi(lua_State *L)
{
  CTState *cts = lj_ctype_init(L);
  settabV(L, L->top++, (cts->miscmap = lj_tab_new(L, 0, 1)));
  cts->finalizer = ffi_finalizer(L);
  LJ_LIB_REG(L, NULL, ffi_meta);
  /* NOBARRIER: basemt is a GC root. */
  setgcref(basemt_it(G(L), LJ_TCDATA), obj2gco(tabV(L->top-1)));
  LJ_LIB_REG(L, NULL, ffi_clib);
  LJ_LIB_REG(L, NULL, ffi_callback);
  /* NOBARRIER: the key is new and lj_tab_newkey() handles the barrier. */
  settabV(L, lj_tab_setstr(L, cts->miscmap, &cts->g->strempty), tabV(L->top-1));
  L->top--;
  lj_clib_default(L, tabV(L->top-1));  /* Create ffi.C default namespace. */
  lua_pushliteral(L, LJ_OS_NAME);
  lua_pushliteral(L, LJ_ARCH_NAME);
  LJ_LIB_REG(L, NULL, ffi);  /* Note: no global "ffi" created! */
  ffi_register_module(L);
  return 1;
}

#endif
```

One Level Up Top Level

src/lj_ccall.c - luajit-2.0-src

Functions defined

- ccall_classify_ct
- ccall_classify_struct
- ccall_classify_struct
- ccall_classify_struct
- ccall_classify_struct
- ccall_get_results
- ccall_set_args
- ccall_struct_arg
- ccall_struct_reg
- ccall_struct_ret
- lj_ccall_ctid_vararg
- lj_ccall_func

Macros defined

- CCALL_HANDLE_COMPLEXARG
- CCALL_HANDLE_COMPLEXARG
- CCALL_HANDLE_COMPLEXARG
- CCALL_HANDLE_COMPLEXARG
- CCALL_HANDLE_COMPLEXARG
- CCALL_HANDLE_COMPLEXARG
- CCALL_HANDLE_COMPLEXARG
- CCALL_HANDLE_COMPLEXARG
- CCALL_HANDLE_COMPLEXRET
- CCALL_HANDLE_COMPLEXRET
- CCALL_HANDLE_COMPLEXRET
- CCALL_HANDLE_COMPLEXRET
- CCALL_HANDLE_COMPLEXRET
- CCALL_HANDLE_COMPLEXRET
- CCALL_HANDLE_COMPLEXRET
- CCALL_HANDLE_COMPLEXRET

- CCALL_HANDLE_COMPLEXRET
- CCALL_HANDLE_COMPLEXRET2
- CCALL_HANDLE_COMPLEXRET2
- CCALL_HANDLE_COMPLEXRET2
- CCALL_HANDLE_COMPLEXRET2
- CCALL_HANDLE_COMPLEXRET2
- CCALL_HANDLE_COMPLEXRET2
- CCALL_HANDLE_COMPLEXRET2
- CCALL_HANDLE_COMPLEXRET2
- CCALL_HANDLE_REGARG
- CCALL_HANDLE_REGARG
- CCALL_HANDLE_REGARG
- CCALL_HANDLE_REGARG
- CCALL_HANDLE_REGARG
- CCALL_HANDLE_REGARG
- CCALL_HANDLE_REGARG
- CCALL_HANDLE_REGARG_FP1
- CCALL_HANDLE_REGARG_FP1
- CCALL_HANDLE_REGARG_FP2
- CCALL_HANDLE_REGARG_FP2
- CCALL_HANDLE_RET
- CCALL_HANDLE_RET
- CCALL_HANDLE_RET
- CCALL_HANDLE_STRUCTARG
- CCALL_HANDLE_STRUCTARG
- CCALL_HANDLE_STRUCTARG
- CCALL_HANDLE_STRUCTARG
- CCALL_HANDLE_STRUCTARG
- CCALL_HANDLE_STRUCTARG
- CCALL_HANDLE_STRUCTARG
- CCALL_HANDLE_STRUCTARG
- CCALL_HANDLE_STRUCTRET
- CCALL_HANDLE_STRUCTRET

- CCALL_HANDLE_STRUCTRET
- CCALL_HANDLE_STRUCTRET
- CCALL_HANDLE_STRUCTRET
- CCALL_HANDLE_STRUCTRET
- CCALL_HANDLE_STRUCTRET
- CCALL_HANDLE_STRUCTRET
- CCALL_HANDLE_STRUCTRET
- CCALL_HANDLE_STRUCTRET
- CCALL_HANDLE_STRUCTRET2
- CCALL_HANDLE_STRUCTRET2
- CCALL_HANDLE_STRUCTRET2
- CCALL_HANDLE_STRUCTRET2
- CCALL_HANDLE_STRUCTRET2
- CCALL_RCL_INT
- CCALL_RCL_MEM
- CCALL_RCL_SSE

Source code

```
/*
** FFI C call handling.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#include "lj_obj.h"

#if LJ_HASFFI

 #include "lj_gc.h"
 #include "lj_err.h"
 #include "lj_tab.h"
 #include "lj_ctype.h"
 #include "lj_cconv.h"
 #include "lj_cdata.h"
 #include "lj_ccall.h"
 #include "lj_trace.h"

 /* Target-specific handling of register arguments. */
 #if LJ_TARGET_X86
 /* -- x86 calling conventions ---------------------------------------------- */

 #if LJ_ABI_WIN

 #define CCALL_HANDLE_STRUCTRET \
   /* Return structs bigger than 8 by reference (on stack only). */ \
   cc->retref = (sz > 8); \
   if (cc->retref) cc->stack[nsp++] = (GPRArg)dp;

 #define CCALL_HANDLE_COMPLEXRET CCALL_HANDLE_STRUCTRET

 #else

 #if LJ_TARGET_OSX

 #define CCALL_HANDLE_STRUCTRET \
```

```
  /* Return structs of size 1, 2, 4 or 8 in registers. */ \
  cc->retref = !(sz == 1 || sz == 2 || sz == 4 || sz == 8); \
  if (cc->retref) { \
    if (ngpr < maxgpr) \
      cc->gpr[ngpr++] = (GPRArg)dp; \
    else \
      cc->stack[nsp++] = (GPRArg)dp; \
  } else {  /* Struct with single FP field ends up in FPR. */ \
    cc->resx87 = ccall_classify_struct(cts, ctr); \
  }

#define CCALL_HANDLE_STRUCTRET2 \
  if (cc->resx87) sp = (uint8_t *)&cc->fpr[0]; \
  memcpy(dp, sp, ctr->size);

#else

#define CCALL_HANDLE_STRUCTRET \
  cc->retref = 1;  /* Return all structs by reference (in reg or on stack). */ \
  if (ngpr < maxgpr) \
    cc->gpr[ngpr++] = (GPRArg)dp; \
  else \
    cc->stack[nsp++] = (GPRArg)dp;

#endif

#define CCALL_HANDLE_COMPLEXRET \
  /* Return complex float in GPRs and complex double by reference. */ \
  cc->retref = (sz > 8); \
  if (cc->retref) { \
    if (ngpr < maxgpr) \
      cc->gpr[ngpr++] = (GPRArg)dp; \
    else \
      cc->stack[nsp++] = (GPRArg)dp; \
  }

#endif

#define CCALL_HANDLE_COMPLEXRET2 \
  if (!cc->retref) \
    *(int64_t *)dp = *(int64_t *)sp;  /* Copy complex float from GPRs. */

#define CCALL_HANDLE_STRUCTARG \
  ngpr = maxgpr;  /* Pass all structs by value on the stack. */

#define CCALL_HANDLE_COMPLEXARG \
  isfp = 1;  /* Pass complex by value on stack. */

#define CCALL_HANDLE_REGARG \
  if (!isfp) {  /* Only non-FP values may be passed in registers. */ \
    if (n > 1) {  /* Anything > 32 bit is passed on the stack. */ \
      if (!LJ_ABI_WIN) ngpr = maxgpr;  /* Prevent reordering. */ \
    } else if (ngpr + 1 <= maxgpr) { \
      dp = &cc->gpr[ngpr]; \
      ngpr += n; \
      goto done; \
    } \
  }

#elif LJ_TARGET_X64 && LJ_ABI_WIN
/* -- Windows/x64 calling conventions ------------------------------------ */

#define CCALL_HANDLE_STRUCTRET \
   /* Return structs of size 1, 2, 4 or 8 in a GPR. */ \
   cc->retref = !(sz == 1 || sz == 2 || sz == 4 || sz == 8); \
   if (cc->retref) cc->gpr[ngpr++] = (GPRArg)dp;

 #define CCALL_HANDLE_COMPLEXRET CCALL_HANDLE_STRUCTRET

 #define CCALL_HANDLE_COMPLEXRET2 \
   if (!cc->retref) \
     *(int64_t *)dp = *(int64_t *)sp;  /* Copy complex float from GPRs. */

 #define CCALL_HANDLE_STRUCTARG \
   /* Pass structs of size 1, 2, 4 or 8 in a GPR by value. */ \
   if (!(sz == 1 || sz == 2 || sz == 4 || sz == 8)) { \
```

```
    rp = cdataptr(lj_cdata_new(cts, did, sz)); \
    sz = CTSIZE_PTR;  /* Pass all other structs by reference. */ \
  }

#define CCALL_HANDLE_COMPLEXARG \
  /* Pass complex float in a GPR and complex double by reference. */ \
  if (sz != 2*sizeof(float)) { \
    rp = cdataptr(lj_cdata_new(cts, did, sz)); \
    sz = CTSIZE_PTR; \
  }

/* Windows/x64 argument registers are strictly positional (use ngpr). */
#define CCALL_HANDLE_REGARG \
  if (isfp) { \
    if (ngpr < maxgpr) { dp = &cc->fpr[ngpr++]; nfpr = ngpr; goto done; } \
  } else { \
    if (ngpr < maxgpr) { dp = &cc->gpr[ngpr++]; goto done; } \
  }

#elif LJ_TARGET_X64
/* -- POSIX/x64 calling conventions ---------------------------------------- */

#define CCALL_HANDLE_STRUCTRET \
  int rcl[2]; rcl[0] = rcl[1] = 0; \
  if (ccall_classify_struct(cts, ctr, rcl, 0)) { \
    cc->retref = 1;  /* Return struct by reference. */ \
    cc->gpr[ngpr++] = (GPRArg)dp; \
  } else { \
    cc->retref = 0;  /* Return small structs in registers. */ \
  }

#define CCALL_HANDLE_STRUCTRET2 \
  int rcl[2]; rcl[0] = rcl[1] = 0; \
  ccall_classify_struct(cts, ctr, rcl, 0); \
  ccall_struct_ret(cc, rcl, dp, ctr->size);

#define CCALL_HANDLE_COMPLEXRET \
  /* Complex values are returned in one or two FPRs. */ \
  cc->retref = 0;

#define CCALL_HANDLE_COMPLEXRET2 \
  if (ctr->size == 2*sizeof(float)) {  /* Copy complex float from FPR. */ \
    *(int64_t *)dp = cc->fpr[0].l[0]; \
  } else {  /* Copy non-contiguous complex double from FPRs. */ \
    ((int64_t *)dp)[0] = cc->fpr[0].l[0]; \
    ((int64_t *)dp)[1] = cc->fpr[1].l[0]; \
  }

#define CCALL_HANDLE_STRUCTARG \
  int rcl[2]; rcl[0] = rcl[1] = 0; \
  if (!ccall_classify_struct(cts, d, rcl, 0)) { \
    cc->nsp = nsp; cc->ngpr = ngpr; cc->nfpr = nfpr; \
    if (ccall_struct_arg(cc, cts, d, rcl, o, narg)) goto err_nyi; \
    nsp = cc->nsp; ngpr = cc->ngpr; nfpr = cc->nfpr; \
    continue; \
  }  /* Pass all other structs by value on stack. */

#define CCALL_HANDLE_COMPLEXARG \
  isfp = 2;  /* Pass complex in FPRs or on stack. Needs postprocessing. */

#define CCALL_HANDLE_REGARG \
  if (isfp) {  /* Try to pass argument in FPRs. */ \
    int n2 = ctype_isvector(d->info) ? 1 : n; \
    if (nfpr + n2 <= CCALL_NARG_FPR) { \
      dp = &cc->fpr[nfpr]; \
      nfpr += n2; \
      goto done; \
    } \
  } else {  /* Try to pass argument in GPRs. */ \
    /* Note that reordering is explicitly allowed in the x64 ABI. */ \
    if (n <= 2 && ngpr + n <= maxgpr) { \
      dp = &cc->gpr[ngpr]; \
      ngpr += n; \
      goto done; \
    } \
  }
```

```
#elif LJ_TARGET_ARM
/* -- ARM calling conventions --------------------------------------------- */

#if LJ_ABI_SOFTFP

#define CCALL_HANDLE_STRUCTRET \
  /* Return structs of size <= 4 in a GPR. */ \
  cc->retref = !(sz <= 4); \
  if (cc->retref) cc->gpr[ngpr++] = (GPRArg)dp;

#define CCALL_HANDLE_COMPLEXRET \
  cc->retref = 1;  /* Return all complex values by reference. */ \
  cc->gpr[ngpr++] = (GPRArg)dp;

#define CCALL_HANDLE_COMPLEXRET2 \
  UNUSED(dp); /* Nothing to do. */

#define CCALL_HANDLE_STRUCTARG \
  /* Pass all structs by value in registers and/or on the stack. */

#define CCALL_HANDLE_COMPLEXARG \
  /* Pass complex by value in 2 or 4 GPRs. */

#define CCALL_HANDLE_REGARG_FP1
#define CCALL_HANDLE_REGARG_FP2

#else

#define CCALL_HANDLE_STRUCTRET \
  cc->retref = !ccall_classify_struct(cts, ctr, ct); \
  if (cc->retref) cc->gpr[ngpr++] = (GPRArg)dp;

#define CCALL_HANDLE_STRUCTRET2 \
  if (ccall_classify_struct(cts, ctr, ct) > 1) sp = (uint8_t *)&cc->fpr[0]; \
  memcpy(dp, sp, ctr->size);

#define CCALL_HANDLE_COMPLEXRET \
  if (!(ct->info & CTF_VARARG)) cc->retref = 0;  /* Return complex in FPRs. */

#define CCALL_HANDLE_COMPLEXRET2 \
  if (!(ct->info & CTF_VARARG)) memcpy(dp, &cc->fpr[0], ctr->size);

#define CCALL_HANDLE_STRUCTARG \
  isfp = (ccall_classify_struct(cts, d, ct) > 1);
  /* Pass all structs by value in registers and/or on the stack. */

#define CCALL_HANDLE_COMPLEXARG \
  isfp = 1;  /* Pass complex by value in FPRs or on stack. */

#define CCALL_HANDLE_REGARG_FP1 \
  if (isfp && !(ct->info & CTF_VARARG)) { \
    if ((d->info & CTF_ALIGN) > CTALIGN_PTR) { \
      if (nfpr + (n >> 1) <= CCALL_NARG_FPR) { \
        dp = &cc->fpr[nfpr]; \
        nfpr += (n >> 1); \
        goto done; \
      } \
    } else { \
      if (sz > 1 && fprodd != nfpr) fprodd = 0; \
      if (fprodd) { \
        if (2*nfpr+n <= 2*CCALL_NARG_FPR+1) { \
          dp = (void *)&cc->fpr[fprodd-1].f[1]; \
          nfpr += (n >> 1); \
          if ((n & 1)) fprodd = 0; else fprodd = nfpr-1; \
          goto done; \
        } \
      } else { \
        if (2*nfpr+n <= 2*CCALL_NARG_FPR) { \
          dp = (void *)&cc->fpr[nfpr]; \
          nfpr += (n >> 1); \
          if ((n & 1)) fprodd = ++nfpr; else fprodd = 0; \
          goto done; \
        } \
      } \
    } \
```

```
    fprodd = 0;  /* No reordering after the first FP value is on stack. */ \
  } else {

#define CCALL_HANDLE_REGARG_FP2        }

#endif

#define CCALL_HANDLE_REGARG \
  CCALL_HANDLE_REGARG_FP1 \
  if ((d->info & CTF_ALIGN) > CTALIGN_PTR) { \
    if (ngpr < maxgpr) \
      ngpr = (ngpr + 1u) & ~1u;  /* Align to regpair. */ \
  } \
  if (ngpr < maxgpr) { \
    dp = &cc->gpr[ngpr]; \
    if (ngpr + n > maxgpr) { \
      nsp += ngpr + n - maxgpr;  /* Assumes contiguous gpr/stack fields. */ \
      if (nsp > CCALL_MAXSTACK) goto err_nyi;  /* Too many arguments. */ \
      ngpr = maxgpr; \
    } else { \
      ngpr += n; \
    } \
    goto done; \
  } CCALL_HANDLE_REGARG_FP2

#define CCALL_HANDLE_RET \
  if ((ct->info & CTF_VARARG)) sp = (uint8_t *)&cc->gpr[0];

#elif LJ_TARGET_ARM64
/* -- ARM64 calling conventions ------------------------------------------- */

#define CCALL_HANDLE_STRUCTRET \
  cc->retref = !ccall_classify_struct(cts, ctr); \
  if (cc->retref) cc->retp = dp;

#define CCALL_HANDLE_STRUCTRET2 \
  unsigned int cl = ccall_classify_struct(cts, ctr); \
  if ((cl & 4)) { /* Combine float HFA from separate registers. */ \
    CTSize i = (cl >> 8) - 1; \
    do { ((uint32_t *)dp)[i] = cc->fpr[i].u32; } while (i--); \
  } else { \
    if (cl > 1) sp = (uint8_t *)&cc->fpr[0]; \
    memcpy(dp, sp, ctr->size); \
  }

#define CCALL_HANDLE_COMPLEXRET \
  /* Complex values are returned in one or two FPRs. */ \
  cc->retref = 0;

#define CCALL_HANDLE_COMPLEXRET2 \
  if (ctr->size == 2*sizeof(float)) {  /* Copy complex float from FPRs. */ \
    ((float *)dp)[0] = cc->fpr[0].f; \
    ((float *)dp)[1] = cc->fpr[1].f; \
  } else {  /* Copy complex double from FPRs. */ \
    ((double *)dp)[0] = cc->fpr[0].d; \
    ((double *)dp)[1] = cc->fpr[1].d; \
  }

#define CCALL_HANDLE_STRUCTARG \
  unsigned int cl = ccall_classify_struct(cts, d); \
  if (cl == 0) {  /* Pass struct by reference. */ \
    rp = cdataptr(lj_cdata_new(cts, did, sz)); \
    sz = CTSIZE_PTR; \
  } else if (cl > 1) {  /* Pass struct in FPRs or on stack. */ \
    isfp = (cl & 4) ? 2 : 1; \
  }  /* else: Pass struct in GPRs or on stack. */

#define CCALL_HANDLE_COMPLEXARG \
  /* Pass complex by value in separate (!) FPRs or on stack. */ \
  isfp = ctr->size == 2*sizeof(float) ? 2 : 1;

#define CCALL_HANDLE_REGARG \
  if (LJ_TARGET_IOS && isva) { \
    /* IOS: All variadic arguments are on the stack. */ \
  } else if (isfp) {  /* Try to pass argument in FPRs. */ \
    int n2 = ctype_isvector(d->info) ? 1 : n*isfp; \
```

```
    if (nfpr + n2 <= CCALL_NARG_FPR) { \
      dp = &cc->fpr[nfpr]; \
      nfpr += n2; \
      goto done; \
    } else { \
      nfpr = CCALL_NARG_FPR;  /* Prevent reordering. */ \
      if (LJ_TARGET_IOS && d->size < 8) goto err_nyi; \
    } \
  } else {  /* Try to pass argument in GPRs. */ \
    if (!LJ_TARGET_IOS && (d->info & CTF_ALIGN) > CTALIGN_PTR) \
      ngpr = (ngpr + 1u) & ~1u;  /* Align to regpair. */ \
    if (ngpr + n <= maxgpr) { \
      dp = &cc->gpr[ngpr]; \
      ngpr += n; \
      goto done; \
    } else { \
      ngpr = maxgpr;  /* Prevent reordering. */ \
      if (LJ_TARGET_IOS && d->size < 8) goto err_nyi; \
    } \
  }

#elif LJ_TARGET_PPC
/* -- PPC calling conventions --------------------------------------------- */

#define CCALL_HANDLE_STRUCTRET \
  cc->retref = 1;  /* Return all structs by reference. */ \
  cc->gpr[ngpr++] = (GPRArg)dp;

#define CCALL_HANDLE_COMPLEXRET \
  /* Complex values are returned in 2 or 4 GPRs. */ \
  cc->retref = 0;

#define CCALL_HANDLE_COMPLEXRET2 \
  memcpy(dp, sp, ctr->size);  /* Copy complex from GPRs. */

#define CCALL_HANDLE_STRUCTARG \
  rp = cdataptr(lj_cdata_new(cts, did, sz)); \
  sz = CTSIZE_PTR;  /* Pass all structs by reference. */

#define CCALL_HANDLE_COMPLEXARG \
  /* Pass complex by value in 2 or 4 GPRs. */

#define CCALL_HANDLE_REGARG \
  if (isfp) {  /* Try to pass argument in FPRs. */ \
    if (nfpr + 1 <= CCALL_NARG_FPR) { \
      dp = &cc->fpr[nfpr]; \
      nfpr += 1; \
      d = ctype_get(cts, CTID_DOUBLE);  /* FPRs always hold doubles. */ \
      goto done; \
    } \
  } else {  /* Try to pass argument in GPRs. */ \
    if (n > 1) { \
      lua_assert(n == 2 || n == 4);  /* int64_t or complex (float). */ \
      if (ctype_isinteger(d->info)) \
        ngpr = (ngpr + 1u) & ~1u;  /* Align int64_t to regpair. */ \
      else if (ngpr + n > maxgpr) \
        ngpr = maxgpr;  /* Prevent reordering. */ \
    } \
    if (ngpr + n <= maxgpr) { \
      dp = &cc->gpr[ngpr]; \
      ngpr += n; \
      goto done; \
    } \
  }

#define CCALL_HANDLE_RET \
  if (ctype_isfp(ctr->info) && ctr->size == sizeof(float)) \
    ctr = ctype_get(cts, CTID_DOUBLE);  /* FPRs always hold doubles. */

#elif LJ_TARGET_MIPS
/* -- MIPS calling conventions -------------------------------------------- */

#define CCALL_HANDLE_STRUCTRET \
  cc->retref = 1;  /* Return all structs by reference. */ \
  cc->gpr[ngpr++] = (GPRArg)dp;
```

```
#define CCALL_HANDLE_COMPLEXRET \
  /* Complex values are returned in 1 or 2 FPRs. */ \
  cc->retref = 0;

#define CCALL_HANDLE_COMPLEXRET2 \
  if (ctr->size == 2*sizeof(float)) {  /* Copy complex float from FPRs. */ \
    ((float *)dp)[0] = cc->fpr[0].f; \
    ((float *)dp)[1] = cc->fpr[1].f; \
  } else {  /* Copy complex double from FPRs. */ \
    ((double *)dp)[0] = cc->fpr[0].d; \
    ((double *)dp)[1] = cc->fpr[1].d; \
  }

#define CCALL_HANDLE_STRUCTARG \
  /* Pass all structs by value in registers and/or on the stack. */

#define CCALL_HANDLE_COMPLEXARG \
  /* Pass complex by value in 2 or 4 GPRs. */

#define CCALL_HANDLE_REGARG \
  if (isfp && nfpr < CCALL_NARG_FPR && !(ct->info & CTF_VARARG)) { \
    /* Try to pass argument in FPRs. */ \
    dp = n == 1 ? (void *)&cc->fpr[nfpr].f : (void *)&cc->fpr[nfpr].d; \
    nfpr++; ngpr += n; \
    goto done; \
  } else {  /* Try to pass argument in GPRs. */ \
    nfpr = CCALL_NARG_FPR; \
    if ((d->info & CTF_ALIGN) > CTALIGN_PTR) \
      ngpr = (ngpr + 1u) & ~1u;  /* Align to regpair. */ \
    if (ngpr < maxgpr) { \
      dp = &cc->gpr[ngpr]; \
      if (ngpr + n > maxgpr) { \
        nsp += ngpr + n - maxgpr;  /* Assumes contiguous gpr/stack fields. */ \
        if (nsp > CCALL_MAXSTACK) goto err_nyi;  /* Too many arguments. */ \
        ngpr = maxgpr; \
      } else { \
        ngpr += n; \
      } \
      goto done; \
    } \
  }

#define CCALL_HANDLE_RET \
  if (ctype_isfp(ctr->info) && ctr->size == sizeof(float)) \
    sp = (uint8_t *)&cc->fpr[0].f;

#else
#error "Missing calling convention definitions for this architecture"
#endif

#ifndef CCALL_HANDLE_STRUCTRET2
#define CCALL_HANDLE_STRUCTRET2 \
  memcpy(dp, sp, ctr->size);  /* Copy struct return value from GPRs. */
#endif

/* -- x86 OSX ABI struct classification ----------------------------------- */

#if LJ_TARGET_X86 && LJ_TARGET_OSX

/* Check for struct with single FP field. */
static int ccall_classify_struct(CTState *cts, CType *ct)
{
  CTSize sz = ct->size;
  if (!(sz == sizeof(float) || sz == sizeof(double))) return 0;
  if ((ct->info & CTF_UNION)) return 0;
  while (ct->sib) {
    ct = ctype_get(cts, ct->sib);
    if (ctype_isfield(ct->info)) {
      CType *sct = ctype_rawchild(cts, ct);
      if (ctype_isfp(sct->info)) {
        if (sct->size == sz)
          return (sz >> 2);  /* Return 1 for float or 2 for double. */
      } else if (ctype_isstruct(sct->info)) {
        if (sct->size)
          return ccall_classify_struct(cts, sct);
      } else {
```

```
        break;
      }
    } else if (ctype_isbitfield(ct->info)) {
      break;
    } else if (ctype_isxattrib(ct->info, CTA_SUBTYPE)) {
      CType *sct = ctype_rawchild(cts, ct);
      if (sct->size)
        return ccall_classify_struct(cts, sct);
    }
  }
  return 0;
}

#endif

/* -- x64 struct classification ------------------------------------------- */

#if LJ_TARGET_X64 && !LJ_ABI_WIN

/* Register classes for x64 struct classification. */
#define CCALL_RCL_INT		1
#define CCALL_RCL_SSE		2
#define CCALL_RCL_MEM		4
/* NYI: classify vectors. */

static int ccall_classify_struct(CTState *cts, CType *ct, int *rcl, CTSize ofs);

/* Classify a C type. */
static void ccall_classify_ct(CTState *cts, CType *ct, int *rcl, CTSize ofs)
{
  if (ctype_isarray(ct->info)) {
    CType *cct = ctype_rawchild(cts, ct);
    CTSize eofs, esz = cct->size, asz = ct->size;
    for (eofs = 0; eofs < asz; eofs += esz)
      ccall_classify_ct(cts, cct, rcl, ofs+eofs);
  } else if (ctype_isstruct(ct->info)) {
    ccall_classify_struct(cts, ct, rcl, ofs);
  } else {
    int cl = ctype_isfp(ct->info) ? CCALL_RCL_SSE : CCALL_RCL_INT;
    lua_assert(ctype_hassize(ct->info));
    if ((ofs & (ct->size-1))) cl = CCALL_RCL_MEM;  /* Unaligned. */
    rcl[(ofs >= 8)] |= cl;
  }
}

/* Recursively classify a struct based on its fields. */
static int ccall_classify_struct(CTState *cts, CType *ct, int *rcl, CTSize ofs)
{
  if (ct->size > 16) return CCALL_RCL_MEM;  /* Too big, gets memory class. */
  while (ct->sib) {
    CTSize fofs;
    ct = ctype_get(cts, ct->sib);
    fofs = ofs+ct->size;
    if (ctype_isfield(ct->info))
      ccall_classify_ct(cts, ctype_rawchild(cts, ct), rcl, fofs);
    else if (ctype_isbitfield(ct->info))
      rcl[(fofs >= 8)] |= CCALL_RCL_INT;  /* NYI: unaligned bitfields? */
    else if (ctype_isxattrib(ct->info, CTA_SUBTYPE))
      ccall_classify_struct(cts, ctype_rawchild(cts, ct), rcl, fofs);
  }
  return ((rcl[0]|rcl[1]) & CCALL_RCL_MEM);  /* Memory class? */
}

/* Try to split up a small struct into registers. */
static int ccall_struct_reg(CCallState *cc, GPRArg *dp, int *rcl)
{
  MSize ngpr = cc->ngpr, nfpr = cc->nfpr;
  uint32_t i;
  for (i = 0; i < 2; i++) {
    lua_assert(!(rcl[i] & CCALL_RCL_MEM));
    if ((rcl[i] & CCALL_RCL_INT)) {  /* Integer class takes precedence. */
      if (ngpr >= CCALL_NARG_GPR) return 1;  /* Register overflow. */
      cc->gpr[ngpr++] = dp[i];
    } else if ((rcl[i] & CCALL_RCL_SSE)) {
      if (nfpr >= CCALL_NARG_FPR) return 1;  /* Register overflow. */
      cc->fpr[nfpr++].l[0] = dp[i];
```

```
    }
  }
  cc->ngpr = ngpr; cc->nfpr = nfpr;
  return 0;  /* Ok. */
}

/* Pass a small struct argument. */
static int ccall_struct_arg(CCallState *cc, CTState *cts, CType *d, int *rcl,
                            TValue *o, int narg)
{
  GPRArg dp[2];
  dp[0] = dp[1] = 0;
  /* Convert to temp. struct. */
  lj_cconv_ct_tv(cts, d, (uint8_t *)dp, o, CCF_ARG(narg));
  if (ccall_struct_reg(cc, dp, rcl)) {  /* Register overflow? Pass on stack. */
    MSize nsp = cc->nsp, n = rcl[1] ? 2 : 1;
    if (nsp + n > CCALL_MAXSTACK) return 1;  /* Too many arguments. */
    cc->nsp = nsp + n;
    memcpy(&cc->stack[nsp], dp, n*CTSIZE_PTR);
  }
  return 0;  /* Ok. */
}

/* Combine returned small struct. */
static void ccall_struct_ret(CCallState *cc, int *rcl, uint8_t *dp, CTSize sz)
{
  GPRArg sp[2];
  MSize ngpr = 0, nfpr = 0;
  uint32_t i;
  for (i = 0; i < 2; i++) {
    if ((rcl[i] & CCALL_RCL_INT)) {  /* Integer class takes precedence. */
      sp[i] = cc->gpr[ngpr++];
    } else if ((rcl[i] & CCALL_RCL_SSE)) {
      sp[i] = cc->fpr[nfpr++].l[0];
    }
  }
  memcpy(dp, sp, sz);
}
#endif

/* -- ARM hard-float ABI struct classification ---------------------------- */

#if LJ_TARGET_ARM && !LJ_ABI_SOFTFP

/* Classify a struct based on its fields. */
static unsigned int ccall_classify_struct(CTState *cts, CType *ct, CType *ctf)
{
  CTSize sz = ct->size;
  unsigned int r = 0, n = 0, isu = (ct->info & CTF_UNION);
  if ((ctf->info & CTF_VARARG)) goto noth;
  while (ct->sib) {
    CType *sct;
    ct = ctype_get(cts, ct->sib);
    if (ctype_isfield(ct->info)) {
      sct = ctype_rawchild(cts, ct);
      if (ctype_isfp(sct->info)) {
        r |= sct->size;
        if (!isu) n++; else if (n == 0) n = 1;
      } else if (ctype_iscomplex(sct->info)) {
        r |= (sct->size >> 1);
        if (!isu) n += 2; else if (n < 2) n = 2;
      } else if (ctype_isstruct(sct->info)) {
        goto substruct;
      } else {
        goto noth;
      }
    } else if (ctype_isbitfield(ct->info)) {
      goto noth;
    } else if (ctype_isxattrib(ct->info, CTA_SUBTYPE)) {
      sct = ctype_rawchild(cts, ct);
    substruct:
      if (sct->size > 0) {
        unsigned int s = ccall_classify_struct(cts, sct, ctf);
        if (s <= 1) goto noth;
        r |= (s & 255);
        if (!isu) n += (s >> 8); else if (n < (s >>8)) n = (s >> 8);
```

```
      }
    }
  }
  if ((r == 4 || r == 8) && n <= 4)
    return r + (n << 8);
noth:  /* Not a homogeneous float/double aggregate. */
  return (sz <= 4);  /* Return structs of size <= 4 in a GPR. */
}

#endif

/* -- ARM64 ABI struct classification ------------------------------------ */

#if LJ_TARGET_ARM64

/* Classify a struct based on its fields. */
static unsigned int ccall_classify_struct(CTState *cts, CType *ct)
{
  CTSize sz = ct->size;
  unsigned int r = 0, n = 0, isu = (ct->info & CTF_UNION);
  while (ct->sib) {
    CType *sct;
    ct = ctype_get(cts, ct->sib);
    if (ctype_isfield(ct->info)) {
      sct = ctype_rawchild(cts, ct);
      if (ctype_isfp(sct->info)) {
        r |= sct->size;
        if (!isu) n++; else if (n == 0) n = 1;
      } else if (ctype_iscomplex(sct->info)) {
        r |= (sct->size >> 1);
        if (!isu) n += 2; else if (n < 2) n = 2;
      } else if (ctype_isstruct(sct->info)) {
        goto substruct;
      } else {
        goto noth;
      }
    } else if (ctype_isbitfield(ct->info)) {
      goto noth;
    } else if (ctype_isxattrib(ct->info, CTA_SUBTYPE)) {
      sct = ctype_rawchild(cts, ct);
    substruct:
      if (sct->size > 0) {
        unsigned int s = ccall_classify_struct(cts, sct);
        if (s <= 1) goto noth;
        r |= (s & 255);
        if (!isu) n += (s >> 8); else if (n < (s >>8)) n = (s >> 8);
      }
    }
  }
  if ((r == 4 || r == 8) && n <= 4)
    return r + (n << 8);
noth:  /* Not a homogeneous float/double aggregate. */
  return (sz <= 16);  /* Return structs of size <= 16 in GPRs. */
}

#endif

/* -- Common C call handling --------------------------------------------- */

/* Infer the destination CTypeID for a vararg argument. */
CTypeID lj_ccall_ctid_vararg(CTState *cts, cTValue *o)
{
  if (tvisnumber(o)) {
    return CTID_DOUBLE;
  } else if (tviscdata(o)) {
    CTypeID id = cdataV(o)->ctypeid;
    CType *s = ctype_get(cts, id);
    if (ctype_isrefarray(s->info)) {
      return lj_ctype_intern(cts,
               CTINFO(CT_PTR, CTALIGN_PTR|ctype_cid(s->info)), CTSIZE_PTR);
    } else if (ctype_isstruct(s->info) || ctype_isfunc(s->info)) {
      /* NYI: how to pass a struct by value in a vararg argument? */
      return lj_ctype_intern(cts, CTINFO(CT_PTR, CTALIGN_PTR|id), CTSIZE_PTR);
    } else if (ctype_isfp(s->info) && s->size == sizeof(float)) {
      return CTID_DOUBLE;
    } else {
```

```
      return id;
    }
  } else if (tvisstr(o)) {
    return CTID_P_CCHAR;
  } else if (tvisbool(o)) {
    return CTID_BOOL;
  } else {
    return CTID_P_VOID;
  }
}

/* Setup arguments for C call. */
static int ccall_set_args(lua_State *L, CTState *cts, CType *ct,
                          CCallState *cc)
{
  int gcsteps = 0;
  TValue *o, *top = L->top;
  CTypeID fid;
  CType *ctr;
  MSize maxgpr, ngpr = 0, nsp = 0, narg;
#if CCALL_NARG_FPR
  MSize nfpr = 0;
#if LJ_TARGET_ARM
  MSize fprodd = 0;
#endif
#endif

  /* Clear unused regs to get some determinism in case of misdeclaration. */
  memset(cc->gpr, 0, sizeof(cc->gpr));
#if CCALL_NUM_FPR
  memset(cc->fpr, 0, sizeof(cc->fpr));
#endif

#if LJ_TARGET_X86
  /* x86 has several different calling conventions. */
  cc->resx87 = 0;
  switch (ctype_cconv(ct->info)) {
  case CTCC_FASTCALL: maxgpr = 2; break;
  case CTCC_THISCALL: maxgpr = 1; break;
  default: maxgpr = 0; break;
  }
#else
  maxgpr = CCALL_NARG_GPR;
#endif

  /* Perform required setup for some result types. */
  ctr = ctype_rawchild(cts, ct);
  if (ctype_isvector(ctr->info)) {
    if (!(CCALL_VECTOR_REG && (ctr->size == 8 || ctr->size == 16)))
      goto err_nyi;
  } else if (ctype_iscomplex(ctr->info) || ctype_isstruct(ctr->info)) {
    /* Preallocate cdata object and anchor it after arguments. */
    CTSize sz = ctr->size;
    GCcdata *cd = lj_cdata_new(cts, ctype_cid(ct->info), sz);
    void *dp = cdataptr(cd);
    setcdataV(L, L->top++, cd);
    if (ctype_isstruct(ctr->info)) {
      CCALL_HANDLE_STRUCTRET
    } else {
      CCALL_HANDLE_COMPLEXRET
    }
#if LJ_TARGET_X86
  } else if (ctype_isfp(ctr->info)) {
    cc->resx87 = ctr->size == sizeof(float) ? 1 : 2;
#endif
  }

  /* Skip initial attributes. */
  fid = ct->sib;
  while (fid) {
    CType *ctf = ctype_get(cts, fid);
    if (!ctype_isattrib(ctf->info)) break;
    fid = ctf->sib;
  }

  /* Walk through all passed arguments. */
```

```
  for (o = L->base+1, narg = 1; o < top; o++, narg++) {
    CTypeID did;
    CType *d;
    CTSize sz;
    MSize n, isfp = 0, isva = 0;
    void *dp, *rp = NULL;

    if (fid) {  /* Get argument type from field. */
      CType *ctf = ctype_get(cts, fid);
      fid = ctf->sib;
      lua_assert(ctype_isfield(ctf->info));
      did = ctype_cid(ctf->info);
    } else {
      if (!(ct->info & CTF_VARARG))
        lj_err_caller(L, LJ_ERR_FFI_NUMARG);  /* Too many arguments. */
      did = lj_ccall_ctid_vararg(cts, o);  /* Infer vararg type. */
      isva = 1;
    }
    d = ctype_raw(cts, did);
    sz = d->size;

    /* Find out how (by value/ref) and where (GPR/FPR) to pass an argument. */
    if (ctype_isnum(d->info)) {
      if (sz > 8) goto err_nyi;
      if ((d->info & CTF_FP))
        isfp = 1;
    } else if (ctype_isvector(d->info)) {
      if (CCALL_VECTOR_REG && (sz == 8 || sz == 16))
        isfp = 1;
      else
        goto err_nyi;
    } else if (ctype_isstruct(d->info)) {
      CCALL_HANDLE_STRUCTARG
    } else if (ctype_iscomplex(d->info)) {
      CCALL_HANDLE_COMPLEXARG
    } else {
      sz = CTSIZE_PTR;
    }
    sz = (sz + CTSIZE_PTR-1) & ~(CTSIZE_PTR-1);
    n = sz / CTSIZE_PTR;  /* Number of GPRs or stack slots needed. */

    CCALL_HANDLE_REGARG  /* Handle register arguments. */

    /* Otherwise pass argument on stack. */
    if (CCALL_ALIGN_STACKARG && !rp && (d->info & CTF_ALIGN) > CTALIGN_PTR) {
      MSize align = (1u << ctype_align(d->info-CTALIGN_PTR)) -1;
      nsp = (nsp + align) & ~align;  /* Align argument on stack. */
    }
    if (nsp + n > CCALL_MAXSTACK) {  /* Too many arguments. */
    err_nyi:
      lj_err_caller(L, LJ_ERR_FFI_NYICALL);
    }
    dp = &cc->stack[nsp];
    nsp += n;
    isva = 0;

  done:
    if (rp) {  /* Pass by reference. */
      gcsteps++;
      *(void **)dp = rp;
      dp = rp;
    }
    lj_cconv_ct_tv(cts, d, (uint8_t *)dp, o, CCF_ARG(narg));
    /* Extend passed integers to 32 bits at least. */
    if (ctype_isinteger_or_bool(d->info) && d->size < 4) {
      if (d->info & CTF_UNSIGNED)
        *(uint32_t *)dp = d->size == 1 ? (uint32_t)*(uint8_t *)dp :
                                         (uint32_t)*(uint16_t *)dp;
      else
        *(int32_t *)dp = d->size == 1 ? (int32_t)*(int8_t *)dp :
                                        (int32_t)*(int16_t *)dp;
    }
#if LJ_TARGET_X64 && LJ_ABI_WIN
    if (isva) {  /* Windows/x64 mirrors varargs in both register sets. */
      if (nfpr == ngpr)
        cc->gpr[ngpr-1] = cc->fpr[ngpr-1].l[0];
```

```
      else
        cc->fpr[ngpr-1].l[0] = cc->gpr[ngpr-1];
    }
#else
    UNUSED(isva);
#endif
#if LJ_TARGET_X64 && !LJ_ABI_WIN
    if (isfp == 2 && n == 2 && (uint8_t *)dp == (uint8_t *)&cc->fpr[nfpr-2]) {
      cc->fpr[nfpr-1].d[0] = cc->fpr[nfpr-2].d[1];  /* Split complex double. */
      cc->fpr[nfpr-2].d[1] = 0;
    }
#elif LJ_TARGET_ARM64
    if (isfp == 2 && (uint8_t *)dp < (uint8_t *)cc->stack) {
      /* Split float HFA or complex float into separate registers. */
      CTSize i = (sz >> 2) - 1;
      do { ((uint64_t *)dp)[i] = ((uint32_t *)dp)[i]; } while (i--);
    }
#else
    UNUSED(isfp);
#endif
  }
  if (fid) lj_err_caller(L, LJ_ERR_FFI_NUMARG);  /* Too few arguments. */

#if LJ_TARGET_X64 || LJ_TARGET_PPC
  cc->nfpr = nfpr;  /* Required for vararg functions. */
#endif
  cc->nsp = nsp;
  cc->spadj = (CCALL_SPS_FREE + CCALL_SPS_EXTRA)*CTSIZE_PTR;
  if (nsp > CCALL_SPS_FREE)
    cc->spadj += (((nsp-CCALL_SPS_FREE)*CTSIZE_PTR + 15u) & ~15u);
  return gcsteps;
}

/* Get results from C call. */
static int ccall_get_results(lua_State *L, CTState *cts, CType *ct,
                             CCallState *cc, int *ret)
{
  CType *ctr = ctype_rawchild(cts, ct);
  uint8_t *sp = (uint8_t *)&cc->gpr[0];
  if (ctype_isvoid(ctr->info)) {
    *ret = 0;  /* Zero results. */
    return 0;  /* No additional GC step. */
  }
  *ret = 1;  /* One result. */
  if (ctype_isstruct(ctr->info)) {
    /* Return cdata object which is already on top of stack. */
    if (!cc->retref) {
      void *dp = cdataptr(cdataV(L->top-1));  /* Use preallocated object. */
      CCALL_HANDLE_STRUCTRET2
    }
    return 1;  /* One GC step. */
  }
  if (ctype_iscomplex(ctr->info)) {
    /* Return cdata object which is already on top of stack. */
    void *dp = cdataptr(cdataV(L->top-1));  /* Use preallocated object. */
    CCALL_HANDLE_COMPLEXRET2
    return 1;  /* One GC step. */
  }
  if (LJ_BE && ctype_isinteger_or_bool(ctr->info) && ctr->size < CTSIZE_PTR)
    sp += (CTSIZE_PTR - ctr->size);
#if CCALL_NUM_FPR
  if (ctype_isfp(ctr->info) || ctype_isvector(ctr->info))
    sp = (uint8_t *)&cc->fpr[0];
#endif
#ifdef CCALL_HANDLE_RET
  CCALL_HANDLE_RET
#endif
  /* No reference types end up here, so there's no need for the CTypeID. */
  lua_assert(!(ctype_isrefarray(ctr->info) || ctype_isstruct(ctr->info)));
  return lj_cconv_tv_ct(cts, ctr, 0, L->top-1, sp);
}

/* Call C function. */
int lj_ccall_func(lua_State *L, GCcdata *cd)
{
  CTState *cts = ctype_cts(L);
```

```
  CType *ct = ctype_raw(cts, cd->ctypeid);
  CTSize sz = CTSIZE_PTR;
  if (ctype_isptr(ct->info)) {
    sz = ct->size;
    ct = ctype_rawchild(cts, ct);
  }
  if (ctype_isfunc(ct->info)) {
    CCallState cc;
    int gcsteps, ret;
    cc.func = (void (*)(void))cdata_getptr(cdataptr(cd), sz);
    gcsteps = ccall_set_args(L, cts, ct, &cc);
    ct = (CType *)((intptr_t)ct-(intptr_t)cts->tab);
    cts->cb.slot = ~0u;
    lj_vm_ffi_call(&cc);
    if (cts->cb.slot != ~0u) {  /* Blacklist function that called a callback. */
      TValue tv;
      setlightudV(&tv, (void *)cc.func);
      setboolV(lj_tab_set(L, cts->miscmap, &tv), 1);
    }
    ct = (CType *)((intptr_t)ct+(intptr_t)cts->tab);  /* May be reallocated. */
    gcsteps += ccall_get_results(L, cts, ct, &cc, &ret);
#if LJ_TARGET_X86 && LJ_ABI_WIN
    /* Automatically detect __stdcall and fix up C function declaration. */
    if (cc.spadj && ctype_cconv(ct->info) == CTCC_CDECL) {
      CTF_INSERT(ct->info, CCONV, CTCC_STDCALL);
      lj_trace_abort(G(L));
    }
#endif
    while (gcsteps-- > 0)
      lj_gc_check(L);
    return ret;
  }
  return -1;  /* Not a function. */
}

#endif
```

One Level Up Top Level

src/lj_ccall.h - luajit-2.0-src

Global variables defined

- FPRArg

Data types defined

- CCallState
- CCallState
- FPRArg
- FPRArg
- FPRArg
- FPRArg
- FPRArg
- FPRArg
- FPRArg
- FPRArg
- GPRArg
- GPRArg
- GPRArg
- GPRArg
- GPRArg

Macros defined

- CCALL_ALIGN_CALLSTATE
- CCALL_ALIGN_CALLSTATE
- CCALL_ALIGN_STACKARG
- CCALL_ALIGN_STACKARG
- CCALL_MAXSTACK
- CCALL_NARG_FPR
- CCALL_NARG_FPR
- CCALL_NARG_FPR
- CCALL_NARG_FPR
- CCALL_NARG_FPR
- CCALL_NARG_FPR

- CCALL_NARG_FPR
- CCALL_NARG_FPR
- CCALL_NARG_GPR
- CCALL_NARG_GPR
- CCALL_NARG_GPR
- CCALL_NARG_GPR
- CCALL_NARG_GPR
- CCALL_NARG_GPR
- CCALL_NARG_GPR
- CCALL_NRET_FPR
- CCALL_NRET_FPR
- CCALL_NRET_FPR
- CCALL_NRET_FPR
- CCALL_NRET_FPR
- CCALL_NRET_FPR
- CCALL_NRET_FPR
- CCALL_NRET_FPR
- CCALL_NRET_GPR
- CCALL_NRET_GPR
- CCALL_NRET_GPR
- CCALL_NRET_GPR
- CCALL_NRET_GPR
- CCALL_NRET_GPR
- CCALL_NRET_GPR
- CCALL_NUM_FPR
- CCALL_NUM_GPR
- CCALL_SPS_EXTRA
- CCALL_SPS_EXTRA
- CCALL_SPS_EXTRA
- CCALL_SPS_EXTRA
- CCALL_SPS_FREE
- CCALL_SPS_FREE
- CCALL_SPS_FREE

- CCALL_SPS_FREE
- CCALL_SPS_FREE
- CCALL_VECTOR_REG
- CCALL_VECTOR_REG
- _LJ_CCALL_H

Source code

```
/*
** FFI C call handling.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_CCALL_H
#define _LJ_CCALL_H

#include "lj_obj.h"
#include "lj_ctype.h"

#if LJ_HASFFI

/* -- C calling conventions ---------------------------------------------- */

#if LJ_TARGET_X86ORX64

#if LJ_TARGET_X86
#define CCALL_NARG_GPR		2	/* For fastcall arguments. */
#define CCALL_NARG_FPR		0
#define CCALL_NRET_GPR		2
#define CCALL_NRET_FPR		1	/* For FP results on x87 stack. */
#define CCALL_ALIGN_STACKARG	0	/* Don't align argument on stack. */
#elif LJ_ABI_WIN
#define CCALL_NARG_GPR		4
#define CCALL_NARG_FPR		4
#define CCALL_NRET_GPR		1
#define CCALL_NRET_FPR		1
#define CCALL_SPS_EXTRA		4
#else
#define CCALL_NARG_GPR		6
#define CCALL_NARG_FPR		8
#define CCALL_NRET_GPR		2
#define CCALL_NRET_FPR		2
#define CCALL_VECTOR_REG	1	/* Pass vectors in registers. */
#endif

#define CCALL_SPS_FREE		1
#define CCALL_ALIGN_CALLSTATE	16

typedef LJ_ALIGN(16) union FPRArg {
  double d[2];
  float f[4];
  uint8_t b[16];
  uint16_t s[8];
  int i[4];
  int64_t l[2];
} FPRArg;

typedef intptr_t GPRArg;

#elif LJ_TARGET_ARM

#define CCALL_NARG_GPR		4
#define CCALL_NRET_GPR		2	/* For softfp double. */
#if LJ_ABI_SOFTFP
#define CCALL_NARG_FPR		0
#define CCALL_NRET_FPR		0
#else
#define CCALL_NARG_FPR		8
#define CCALL_NRET_FPR		4
```

```
#endif
#define CCALL_SPS_FREE               0

typedef intptr_t GPRArg;
typedef union FPRArg {
  double d;
  float f[2];
} FPRArg;

#elif LJ_TARGET_ARM64

#define CCALL_NARG_GPR               8
#define CCALL_NRET_GPR               2
#define CCALL_NARG_FPR               8
#define CCALL_NRET_FPR               4
#define CCALL_SPS_FREE               0

typedef intptr_t GPRArg;
typedef union FPRArg {
  double d;
  float f;
  uint32_t u32;
} FPRArg;

#elif LJ_TARGET_PPC

#define CCALL_NARG_GPR               8
#define CCALL_NARG_FPR               8
#define CCALL_NRET_GPR               4        /* For complex double. */
#define CCALL_NRET_FPR               1
#define CCALL_SPS_EXTRA              4
#define CCALL_SPS_FREE               0

typedef intptr_t GPRArg;
typedef double FPRArg;

#elif LJ_TARGET_MIPS

 #define CCALL_NARG_GPR              4
 #define CCALL_NARG_FPR              2
 #define CCALL_NRET_GPR              2
 #define CCALL_NRET_FPR              2
 #define CCALL_SPS_EXTRA             7
 #define CCALL_SPS_FREE              1

 typedef intptr_t GPRArg;
 typedef union FPRArg {
   double d;
   struct { LJ_ENDIAN_LOHI(float f; , float g;) };
 } FPRArg;

 #else
 #error "Missing calling convention definitions for this architecture"
 #endif

 #ifndef CCALL_SPS_EXTRA
 #define CCALL_SPS_EXTRA              0
 #endif
 #ifndef CCALL_VECTOR_REG
 #define CCALL_VECTOR_REG        0
 #endif
 #ifndef CCALL_ALIGN_STACKARG
 #define CCALL_ALIGN_STACKARG        1
 #endif
 #ifndef CCALL_ALIGN_CALLSTATE
 #define CCALL_ALIGN_CALLSTATE       8
 #endif

 #define CCALL_NUM_GPR \
   (CCALL_NARG_GPR > CCALL_NRET_GPR ? CCALL_NARG_GPR : CCALL_NRET_GPR)
 #define CCALL_NUM_FPR \
   (CCALL_NARG_FPR > CCALL_NRET_FPR ? CCALL_NARG_FPR : CCALL_NRET_FPR)

 /* Check against constants in lj_ctype.h. */
 LJ_STATIC_ASSERT(CCALL_NUM_GPR <= CCALL_MAX_GPR);
 LJ_STATIC_ASSERT(CCALL_NUM_FPR <= CCALL_MAX_FPR);
```

```
138
139 #define CCALL_MAXSTACK                32
140
141 /* -- C call state ---------------------------------------------------- */
142
143 typedef LJ_ALIGN(CCALL_ALIGN_CALLSTATE) struct CCallState {
144   void (*func)(void);               /* Pointer to called function. */
145   uint32_t spadj;               /* Stack pointer adjustment. */
146   uint8_t nsp;                       /* Number of stack slots. */
147   uint8_t retref;               /* Return value by reference. */
148 #if LJ_TARGET_X64
149   uint8_t ngpr;                       /* Number of arguments in GPRs. */
150   uint8_t nfpr;                       /* Number of arguments in FPRs. */
151 #elif LJ_TARGET_X86
152   uint8_t resx87;               /* Result on x87 stack: 1:float, 2:double. */
153 #elif LJ_TARGET_ARM64
154   void *retp;                      /* Aggregate return pointer in x8. */
155 #elif LJ_TARGET_PPC
156   uint8_t nfpr;                       /* Number of arguments in FPRs. */
157 #endif
158 #if LJ_32
159   int32_t align1;
160 #endif
161 #if CCALL_NUM_FPR
162   FPRArg fpr[CCALL_NUM_FPR];        /* Arguments/results in FPRs. */
163 #endif
164   GPRArg gpr[CCALL_NUM_GPR];        /* Arguments/results in GPRs. */
165   GPRArg stack[CCALL_MAXSTACK];        /* Stack slots. */
166 } CCallState;
167
168 /* -- C call handling ------------------------------------------------- */
169
170 /* Really belongs to lj_vm.h. */
171 LJ_ASMF void LJ_FASTCALL lj_vm_ffi_call(CCallState *cc);
172
173 LJ_FUNC CTypeID lj_ccall_ctid_vararg(CTState *cts, cTValue *o);
174 LJ_FUNC int lj_ccall_func(lua_State *L, GCcdata *cd);
175
176 #endif
177
178 #endif
```

One Level Up Top Level

src/lib_init.c - luajit-2.0-src

Global variables defined

- lj_lib_load
- lj_lib_preload

Functions defined

- luaL_openlibs

Macros defined

- LUA_LIB
- lib_init_c

Source code

```
1  /*
2  ** Library initialization.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  **
5  ** Major parts taken verbatim from the Lua interpreter.
6  ** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
7  */
8
9  #define lib_init_c
10 #define LUA_LIB
11
12 #include "lua.h"
13 #include "lauxlib.h"
14 #include "lualib.h"
15
16 #include "lj_arch.h"
17
18 static const luaL_Reg lj_lib_load[] = {
19   { "",                       luaopen_base },
20   { LUA_LOADLIBNAME,        luaopen_package },
21   { LUA_TABLIBNAME,         luaopen_table },
22   { LUA_IOLIBNAME,          luaopen_io },
23   { LUA_OSLIBNAME,          luaopen_os },
24   { LUA_STRLIBNAME,         luaopen_string },
25   { LUA_MATHLIBNAME,        luaopen_math },
26   { LUA_DBLIBNAME,          luaopen_debug },
27   { LUA_BITLIBNAME,         luaopen_bit },
28   { LUA_JITLIBNAME,         luaopen_jit },
29   { NULL,                 NULL }
30 };
31
32 static const luaL_Reg lj_lib_preload[] = {
33 #if LJ_HASFFI
34   { LUA_FFILIBNAME,         luaopen_ffi },
35 #endif
36   { NULL,                 NULL }
37 };
38
39 LUALIB_API void luaL_openlibs(lua_State *L)
40 {
41   const luaL_Reg *lib;
42   for (lib = lj_lib_load; lib->func; lib++) {
43     lua_pushcfunction(L, lib->func);
44     lua_pushstring(L, lib->name);
45     lua_call(L, 1, 0);
46   }
```

```
  luaL_findtable(L, LUA_REGISTRYINDEX, "_PRELOAD",
                 sizeof(lj_lib_preload)/sizeof(lj_lib_preload[0])-1);
  for (lib = lj_lib_preload; lib->func; lib++) {
    lua_pushcfunction(L, lib->func);
    lua_setfield(L, -2, lib->name);
  }
  lua_pop(L, 1);
}

```

One Level Up Top Level

src/lj_opt_loop.c - luajit-2.0-src

Data types defined

- LoopState
- LoopState

Functions defined

- cploop_opt
- lj_opt_loop
- loop_emit_phi
- loop_subst_snap
- loop_undo
- loop_unroll

Macros defined

- IR
- IR
- LUA_CORE
- emitir
- emitir
- emitir_raw
- emitir_raw
- lj_opt_loop_c

Source code

```
1  /*
2  ** LOOP: Loop Optimizations.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lj_opt_loop_c
7  #define LUA_CORE
8
9  #include "lj_obj.h"
10
11 #if LJ_HASJIT
12
13 #include "lj_err.h"
14 #include "lj_buf.h"
15 #include "lj_ir.h"
16 #include "lj_jit.h"
17 #include "lj_iropt.h"
18 #include "lj_trace.h"
19 #include "lj_snap.h"
20 #include "lj_vm.h"
21
```

```
/* Loop optimization:
**
** Traditional Loop-Invariant Code Motion (LICM) splits the instructions
** of a loop into invariant and variant instructions. The invariant
** instructions are hoisted out of the loop and only the variant
** instructions remain inside the loop body.
**
** Unfortunately LICM is mostly useless for compiling dynamic languages.
** The IR has many guards and most of the subsequent instructions are
** control-dependent on them. The first non-hoistable guard would
** effectively prevent hoisting of all subsequent instructions.
**
** That's why we use a special form of unrolling using copy-substitution,
** combined with redundancy elimination:
**
** The recorded instruction stream is re-emitted to the compiler pipeline
** with substituted operands. The substitution table is filled with the
** refs returned by re-emitting each instruction. This can be done
** on-the-fly, because the IR is in strict SSA form, where every ref is
** defined before its use.
**
** This aproach generates two code sections, separated by the LOOP
** instruction:
**
** 1. The recorded instructions form a kind of pre-roll for the loop. It
** contains a mix of invariant and variant instructions and performs
** exactly one loop iteration (but not necessarily the 1st iteration).
**
** 2. The loop body contains only the variant instructions and performs
** all remaining loop iterations.
**
** On first sight that looks like a waste of space, because the variant
** instructions are present twice. But the key insight is that the
** pre-roll honors the control-dependencies for *both* the pre-roll itself
** *and* the loop body!
**
** It also means one doesn't have to explicitly model control-dependencies
** (which, BTW, wouldn't help LICM much). And it's much easier to
** integrate sparse snapshotting with this approach.
**
** One of the nicest aspects of this approach is that all of the
** optimizations of the compiler pipeline (FOLD, CSE, FWD, etc.) can be
** reused with only minor restrictions (e.g. one should not fold
** instructions across loop-carried dependencies).
**
** But in general all optimizations can be applied which only need to look
** backwards into the generated instruction stream. At any point in time
** during the copy-substitution process this contains both a static loop
** iteration (the pre-roll) and a dynamic one (from the to-be-copied
** instruction up to the end of the partial loop body).
**
** Since control-dependencies are implicitly kept, CSE also applies to all
** kinds of guards. The major advantage is that all invariant guards can
** be hoisted, too.
**
** Load/store forwarding works across loop iterations, too. This is
** important if loop-carried dependencies are kept in upvalues or tables.
** E.g. 'self.idx = self.idx + 1' deep down in some OO-style method may
** become a forwarded loop-recurrence after inlining.
**
** Since the IR is in SSA form, loop-carried dependencies have to be
** modeled with PHI instructions. The potential candidates for PHIs are
** collected on-the-fly during copy-substitution. After eliminating the
** redundant ones, PHI instructions are emitted *below* the loop body.
**
** Note that this departure from traditional SSA form doesn't change the
** semantics of the PHI instructions themselves. But it greatly simplifies
** on-the-fly generation of the IR and the machine code.
*/

/* Some local macros to save typing. Undef'd at the end. */
#define IR(ref)                (&J->cur.ir[(ref)])

/* Pass IR on to next optimization in chain (FOLD). */
#define emitir(ot, a, b)        (lj_ir_set(J, (ot), (a), (b)), lj_opt_fold(J))

```

```
/* Emit raw IR without passing through optimizations. */
#define emitir_raw(ot, a, b)        (lj_ir_set(J, (ot), (a), (b)), lj_ir_emit(J))

/* -- PHI elimination ------------------------------------------------------ */

/* Emit or eliminate collected PHIs. */
static void loop_emit_phi(jit_State *J, IRRef1 *subst, IRRef1 *phi, IRRef nphi,
                          SnapNo onsnap)
{
  int passx = 0;
  IRRef i, j, nslots;
  IRRef invar = J->chain[IR_LOOP];
  /* Pass #1: mark redundant and potentially redundant PHIs. */
  for (i = 0, j = 0; i < nphi; i++) {
    IRRef lref = phi[i];
    IRRef rref = subst[lref];
    if (lref == rref || rref == REF_DROP) {  /* Invariants are redundant. */
      irt_clearphi(IR(lref)->t);
    } else {
      phi[j++] = (IRRef1)lref;
      if (!(IR(rref)->op1 == lref || IR(rref)->op2 == lref)) {
        /* Quick check for simple recurrences failed, need pass2. */
        irt_setmark(IR(lref)->t);
        passx = 1;
      }
    }
  }
  nphi = j;
  /* Pass #2: traverse variant part and clear marks of non-redundant PHIs. */
  if (passx) {
    SnapNo s;
    for (i = J->cur.nins-1; i > invar; i--) {
      IRIns *ir = IR(i);
      if (!irref_isk(ir->op2)) irt_clearmark(IR(ir->op2)->t);
      if (!irref_isk(ir->op1)) {
        irt_clearmark(IR(ir->op1)->t);
        if (ir->op1 < invar &&
            ir->o >= IR_CALLN && ir->o <= IR_CARG) {  /* ORDER IR */
          ir = IR(ir->op1);
          while (ir->o == IR_CARG) {
            if (!irref_isk(ir->op2)) irt_clearmark(IR(ir->op2)->t);
            if (irref_isk(ir->op1)) break;
            ir = IR(ir->op1);
            irt_clearmark(ir->t);
          }
        }
      }
    }
    for (s = J->cur.nsnap-1; s >= onsnap; s--) {
      SnapShot *snap = &J->cur.snap[s];
      SnapEntry *map = &J->cur.snapmap[snap->mapofs];
      MSize n, nent = snap->nent;
      for (n = 0; n < nent; n++) {
        IRRef ref = snap_ref(map[n]);
        if (!irref_isk(ref)) irt_clearmark(IR(ref)->t);
      }
    }
  }
  /* Pass #3: add PHIs for variant slots without a corresponding SLOAD. */
  nslots = J->baseslot+J->maxslot;
  for (i = 1; i < nslots; i++) {
    IRRef ref = tref_ref(J->slot[i]);
    while (!irref_isk(ref) && ref != subst[ref]) {
      IRIns *ir = IR(ref);
      irt_clearmark(ir->t);  /* Unmark potential uses, too. */
      if (irt_isphi(ir->t) || irt_ispri(ir->t))
        break;
      irt_setphi(ir->t);
      if (nphi >= LJ_MAX_PHI)
        lj_trace_err(J, LJ_TRERR_PHIOV);
      phi[nphi++] = (IRRef1)ref;
      ref = subst[ref];
      if (ref > invar)
        break;
    }
  }
```

```
  /* Pass #4: propagate non-redundant PHIs. */
  while (passx) {
    passx = 0;
    for (i = 0; i < nphi; i++) {
      IRRef lref = phi[i];
      IRIns *ir = IR(lref);
      if (!irt_ismarked(ir->t)) {  /* Propagate only from unmarked PHIs. */
        IRIns *irr = IR(subst[lref]);
        if (irt_ismarked(irr->t)) {  /* Right ref points to other PHI? */
          irt_clearmark(irr->t);  /* Mark that PHI as non-redundant. */
          passx = 1;  /* Retry. */
        }
      }
    }
  }
  /* Pass #5: emit PHI instructions or eliminate PHIs. */
  for (i = 0; i < nphi; i++) {
    IRRef lref = phi[i];
    IRIns *ir = IR(lref);
    if (!irt_ismarked(ir->t)) {  /* Emit PHI if not marked. */
      IRRef rref = subst[lref];
      if (rref > invar)
        irt_setphi(IR(rref)->t);
      emitir_raw(IRT(IR_PHI, irt_type(ir->t)), lref, rref);
    } else {  /* Otherwise eliminate PHI. */
      irt_clearmark(ir->t);
      irt_clearphi(ir->t);
    }
  }
}

/* -- Loop unrolling using copy-substitution ------------------------------ */

/* Copy-substitute snapshot. */
static void loop_subst_snap(jit_State *J, SnapShot *osnap,
                            SnapEntry *loopmap, IRRef1 *subst)
{
  SnapEntry *nmap, *omap = &J->cur.snapmap[osnap->mapofs];
  SnapEntry *nextmap = &J->cur.snapmap[snap_nextofs(&J->cur, osnap)];
  MSize nmapofs;
  MSize on, ln, nn, onent = osnap->nent;
  BCReg nslots = osnap->nslots;
  SnapShot *snap = &J->cur.snap[J->cur.nsnap];
  if (irt_isguard(J->guardemit)) {  /* Guard inbetween? */
    nmapofs = J->cur.nsnapmap;
    J->cur.nsnap++;  /* Add new snapshot. */
  } else {  /* Otherwise overwrite previous snapshot. */
    snap--;
    nmapofs = snap->mapofs;
  }
  J->guardemit.irt = 0;
  /* Setup new snapshot. */
  snap->mapofs = (uint16_t)nmapofs;
  snap->ref = (IRRef1)J->cur.nins;
  snap->nslots = nslots;
  snap->topslot = osnap->topslot;
  snap->count = 0;
  nmap = &J->cur.snapmap[nmapofs];
  /* Substitute snapshot slots. */
  on = ln = nn = 0;
  while (on < onent) {
    SnapEntry osn = omap[on], lsn = loopmap[ln];
    if (snap_slot(lsn) < snap_slot(osn)) {  /* Copy slot from loop map. */
      nmap[nn++] = lsn;
      ln++;
    } else {  /* Copy substituted slot from snapshot map. */
      if (snap_slot(lsn) == snap_slot(osn)) ln++;  /* Shadowed loop slot. */
      if (!irref_isk(snap_ref(osn)))
        osn = snap_setref(osn, subst[snap_ref(osn)]);
      nmap[nn++] = osn;
      on++;
    }
  }
  while (snap_slot(loopmap[ln]) < nslots)  /* Copy remaining loop slots. */
    nmap[nn++] = loopmap[ln++];
  snap->nent = (uint8_t)nn;
```

```
  omap += onent;
  nmap += nn;
  while (omap < nextmap)  /* Copy PC + frame links. */
    *nmap++ = *omap++;
  J->cur.nsnapmap = (uint16_t)(nmap - J->cur.snapmap);
}

typedef struct LoopState {
  jit_State *J;
  IRRef1 *subst;
  MSize sizesubst;
} LoopState;

/* Unroll loop. */
static void loop_unroll(LoopState *lps)
{
  jit_State *J = lps->J;
  IRRef1 phi[LJ_MAX_PHI];
  uint32_t nphi = 0;
  IRRef1 *subst;
  SnapNo onsnap;
  SnapShot *osnap, *loopsnap;
  SnapEntry *loopmap, *psentinel;
  IRRef ins, invar;

  /* Allocate substitution table.
  ** Only non-constant refs in [REF_BIAS,invar) are valid indexes.
  */
  invar = J->cur.nins;
  lps->sizesubst = invar - REF_BIAS;
  lps->subst = lj_mem_newvec(J->L, lps->sizesubst, IRRef1);
  subst = lps->subst - REF_BIAS;
  subst[REF_BASE] = REF_BASE;

  /* LOOP separates the pre-roll from the loop body. */
  emitir_raw(IRTG(IR_LOOP, IRT_NIL), 0, 0);

  /* Grow snapshot buffer and map for copy-substituted snapshots.
  ** Need up to twice the number of snapshots minus #0 and loop snapshot.
  ** Need up to twice the number of entries plus fallback substitutions
  ** from the loop snapshot entries for each new snapshot.
  ** Caveat: both calls may reallocate J->cur.snap and J->cur.snapmap!
  */
  onsnap = J->cur.nsnap;
  lj_snap_grow_buf(J, 2*onsnap-2);
  lj_snap_grow_map(J, J->cur.nsnapmap*2+(onsnap-2)*J->cur.snap[onsnap-1].nent);

  /* The loop snapshot is used for fallback substitutions. */
  loopsnap = &J->cur.snap[onsnap-1];
  loopmap = &J->cur.snapmap[loopsnap->mapofs];
  /* The PC of snapshot #0 and the loop snapshot must match. */
  psentinel = &loopmap[loopsnap->nent];
  lua_assert(*psentinel == J->cur.snapmap[J->cur.snap[0].nent]);
  *psentinel = SNAP(255, 0, 0);  /* Replace PC with temporary sentinel. */

  /* Start substitution with snapshot #1 (#0 is empty for root traces). */
  osnap = &J->cur.snap[1];

  /* Copy and substitute all recorded instructions and snapshots. */
  for (ins = REF_FIRST; ins < invar; ins++) {
    IRIns *ir;
    IRRef op1, op2;

    if (ins >= osnap->ref)  /* Instruction belongs to next snapshot? */
      loop_subst_snap(J, osnap++, loopmap, subst);  /* Copy-substitute it. */

    /* Substitute instruction operands. */
    ir = IR(ins);
    op1 = ir->op1;
    if (!irref_isk(op1)) op1 = subst[op1];
    op2 = ir->op2;
    if (!irref_isk(op2)) op2 = subst[op2];
    if (irm_kind(lj_ir_mode[ir->o]) == IRM_N &&
        op1 == ir->op1 && op2 == ir->op2) {  /* Regular invariant ins? */
      subst[ins] = (IRRef1)ins;  /* Shortcut. */
    } else {
```

```
      /* Re-emit substituted instruction to the FOLD/CSE/etc. pipeline. */
      IRType1 t = ir->t;  /* Get this first, since emitir may invalidate ir. */
      IRRef ref = tref_ref(emitir(ir->ot & ~IRT_ISPHI, op1, op2));
      subst[ins] = (IRRef1)ref;
      if (ref != ins) {
        IRIns *irr = IR(ref);
        if (ref < invar) {  /* Loop-carried dependency? */
          /* Potential PHI? */
          if (!irref_isk(ref) && !irt_isphi(irr->t) && !irt_ispri(irr->t)) {
            irt_setphi(irr->t);
            if (nphi >= LJ_MAX_PHI)
              lj_trace_err(J, LJ_TRERR_PHIOV);
            phi[nphi++] = (IRRef1)ref;
          }
          /* Check all loop-carried dependencies for type instability. */
          if (!irt_sametype(t, irr->t)) {
            if (irt_isinteger(t) && irt_isinteger(irr->t))
              continue;
            else if (irt_isnum(t) && irt_isinteger(irr->t))  /* Fix int->num. */
              ref = tref_ref(emitir(IRTN(IR_CONV), ref, IRCONV_NUM_INT));
            else if (irt_isnum(irr->t) && irt_isinteger(t))  /* Fix num->int. */
              ref = tref_ref(emitir(IRTGI(IR_CONV), ref,
                                    IRCONV_INT_NUM|IRCONV_CHECK));
            else
              lj_trace_err(J, LJ_TRERR_TYPEINS);
            subst[ins] = (IRRef1)ref;
            irr = IR(ref);
            goto phiconv;
          }
        } else if (ref != REF_DROP && irr->o == IR_CONV &&
                   ref > invar && irr->op1 < invar) {
          /* May need an extra PHI for a CONV. */
          ref = irr->op1;
          irr = IR(ref);
        phiconv:
          if (ref < invar && !irref_isk(ref) && !irt_isphi(irr->t)) {
            irt_setphi(irr->t);
            if (nphi >= LJ_MAX_PHI)
              lj_trace_err(J, LJ_TRERR_PHIOV);
            phi[nphi++] = (IRRef1)ref;
          }
        }
      }
    }
  }
  if (!irt_isguard(J->guardemit))  /* Drop redundant snapshot. */
    J->cur.nsnapmap = (uint16_t)J->cur.snap[--J->cur.nsnap].mapofs;
  lua_assert(J->cur.nsnapmap <= J->sizesnapmap);
  *psentinel = J->cur.snapmap[J->cur.snap[0].nent];  /* Restore PC. */

  loop_emit_phi(J, subst, phi, nphi, onsnap);
}

/* Undo any partial changes made by the loop optimization. */
static void loop_undo(jit_State *J, IRRef ins, SnapNo nsnap, MSize nsnapmap)
{
  ptrdiff_t i;
  SnapShot *snap = &J->cur.snap[nsnap-1];
  SnapEntry *map = J->cur.snapmap;
  map[snap->mapofs + snap->nent] = map[J->cur.snap[0].nent];  /* Restore PC. */
  J->cur.nsnapmap = (uint16_t)nsnapmap;
  J->cur.nsnap = nsnap;
  J->guardemit.irt = 0;
  lj_ir_rollback(J, ins);
  for (i = 0; i < BPROP_SLOTS; i++) {  /* Remove backprop. cache entries. */
    BPropEntry *bp = &J->bpropcache[i];
    if (bp->val >= ins)
      bp->key = 0;
  }
  for (ins--; ins >= REF_FIRST; ins--) {  /* Remove flags. */
    IRIns *ir = IR(ins);
    irt_clearphi(ir->t);
    irt_clearmark(ir->t);
  }
}

```

```
/* Protected callback for loop optimization. */
static TValue *cploop_opt(lua_State *L, lua_CFunction dummy, void *ud)
{
  UNUSED(L); UNUSED(dummy);
  loop_unroll((LoopState *)ud);
  return NULL;
}

/* Loop optimization. */
int lj_opt_loop(jit_State *J)
{
  IRRef nins = J->cur.nins;
  SnapNo nsnap = J->cur.nsnap;
  MSize nsnapmap = J->cur.nsnapmap;
  LoopState lps;
  int errcode;
  lps.J = J;
  lps.subst = NULL;
  lps.sizesubst = 0;
  errcode = lj_vm_cpcall(J->L, NULL, &lps, cploop_opt);
  lj_mem_freevec(J2G(J), lps.subst, lps.sizesubst, IRRef1);
  if (LJ_UNLIKELY(errcode)) {
    lua_State *L = J->L;
    if (errcode == LUA_ERRRUN && tvisnumber(L->top-1)) {  /* Trace error? */
      int32_t e = numberVint(L->top-1);
      switch ((TraceError)e) {
      case LJ_TRERR_TYPEINS:  /* Type instability. */
      case LJ_TRERR_GFAIL:  /* Guard would always fail. */
        /* Unrolling via recording fixes many cases, e.g. a flipped boolean. */
        if (--J->instunroll < 0)  /* But do not unroll forever. */
          break;
        L->top--;  /* Remove error object. */
        loop_undo(J, nins, nsnap, nsnapmap);
        return 1;  /* Loop optimization failed, continue recording. */
      default:
        break;
      }
    }
    lj_err_throw(L, errcode);  /* Propagate all other errors. */
  }
  return 0;  /* Loop optimization is ok. */
}

#undef IR
#undef emitir
#undef emitir_raw

#endif
```

One Level Up Top Level

src/lj_snap.h - luajit-2.0-src

Functions defined

- lj_snap_grow_buf
- lj_snap_grow_map

Macros defined

- _LJ_SNAP_H

Source code

```
1  /*
2  ** Snapshot handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_SNAP_H
7  #define _LJ_SNAP_H
8
9  #include "lj_obj.h"
10 #include "lj_jit.h"
11
12 #if LJ_HASJIT
13 LJ_FUNC void lj_snap_add(jit_State *J);
14 LJ_FUNC void lj_snap_purge(jit_State *J);
15 LJ_FUNC void lj_snap_shrink(jit_State *J);
16 LJ_FUNC IRIns *lj_snap_regspmap(GCtrace *T, SnapNo snapno, IRIns *ir);
17 LJ_FUNC void lj_snap_replay(jit_State *J, GCtrace *T);
18 LJ_FUNC const BCIns *lj_snap_restore(jit_State *J, void *exptr);
19 LJ_FUNC void lj_snap_grow_buf_(jit_State *J, MSize need);
20 LJ_FUNC void lj_snap_grow_map_(jit_State *J, MSize need);
21
22 static LJ_AINLINE void lj_snap_grow_buf(jit_State *J, MSize need)
23 {
24   if (LJ_UNLIKELY(need > J->sizesnap)) lj_snap_grow_buf_(J, need);
25 }
26
27 static LJ_AINLINE void lj_snap_grow_map(jit_State *J, MSize need)
28 {
29   if (LJ_UNLIKELY(need > J->sizesnapmap)) lj_snap_grow_map_(J, need);
30 }
31
32 #endif
33
34 #endif
```

src/lj_snap.c - luajit-2.0-src

Functions defined

- lj_snap_add
- lj_snap_grow_buf_
- lj_snap_grow_map_
- lj_snap_purge
- lj_snap_regspmap
- lj_snap_replay
- lj_snap_restore
- lj_snap_shrink
- snap_dedup
- snap_pref
- snap_renamefilter
- snap_renameref
- snap_replay_const
- snap_restoredata
- snap_restoreval
- snap_sunk_store
- snap_sunk_store2
- snap_unsink
- snap_usedef
- snapshot_framelinks
- snapshot_slots
- snapshot_stack

Macros defined

- DEF_SLOT
- DEF_SLOT
- IR
- IR
- LUA_CORE
- SNAP_USEDEF_SLOTS

- USE_SLOT
- USE_SLOT
- emitir
- emitir
- emitir_raw
- emitir_raw
- lj_snap_c

Source code

```
/*
** Snapshot handling.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_snap_c
#define LUA_CORE

#include "lj_obj.h"

#if LJ_HASJIT

#include "lj_gc.h"
#include "lj_tab.h"
#include "lj_state.h"
#include "lj_frame.h"
#include "lj_bc.h"
#include "lj_ir.h"
#include "lj_jit.h"
#include "lj_iropt.h"
#include "lj_trace.h"
#include "lj_snap.h"
#include "lj_target.h"
#if LJ_HASFFI
#include "lj_ctype.h"
#include "lj_cdata.h"
#endif

/* Some local macros to save typing. Undef'd at the end. */
#define IR(ref)                (&J->cur.ir[(ref)])

/* Pass IR on to next optimization in chain (FOLD). */
#define emitir(ot, a, b)        (lj_ir_set(J, (ot), (a), (b)), lj_opt_fold(J))

/* Emit raw IR without passing through optimizations. */
#define emitir_raw(ot, a, b)        (lj_ir_set(J, (ot), (a), (b)), lj_ir_emit(J))

/* -- Snapshot buffer allocation ----------------------------------------- */

/* Grow snapshot buffer. */
void lj_snap_grow_buf_(jit_State *J, MSize need)
{
  MSize maxsnap = (MSize)J->param[JIT_P_maxsnap];
  if (need > maxsnap)
    lj_trace_err(J, LJ_TRERR_SNAPOV);
  lj_mem_growvec(J->L, J->snapbuf, J->sizesnap, maxsnap, SnapShot);
  J->cur.snap = J->snapbuf;
}

/* Grow snapshot map buffer. */
void lj_snap_grow_map_(jit_State *J, MSize need)
{
  if (need < 2*J->sizesnapmap)
    need = 2*J->sizesnapmap;
  else if (need < 64)
    need = 64;
```

```
  J->snapmapbuf = (SnapEntry *)lj_mem_realloc(J->L, J->snapmapbuf,
                    J->sizesnapmap*sizeof(SnapEntry), need*sizeof(SnapEntry));
  J->cur.snapmap = J->snapmapbuf;
  J->sizesnapmap = need;
}

/* -- Snapshot generation ------------------------------------------------- */

/* Add all modified slots to the snapshot. */
static MSize snapshot_slots(jit_State *J, SnapEntry *map, BCReg nslots)
{
  IRRef retf = J->chain[IR_RETF];  /* Limits SLOAD restore elimination. */
  BCReg s;
  MSize n = 0;
  for (s = 0; s < nslots; s++) {
    TRef tr = J->slot[s];
    IRRef ref = tref_ref(tr);
    if (ref) {
      SnapEntry sn = SNAP_TR(s, tr);
      IRIns *ir = IR(ref);
      if (!(sn & (SNAP_CONT|SNAP_FRAME)) &&
          ir->o == IR_SLOAD && ir->op1 == s && ref > retf) {
        /* No need to snapshot unmodified non-inherited slots. */
        if (!(ir->op2 & IRSLOAD_INHERIT))
          continue;
        /* No need to restore readonly slots and unmodified non-parent slots. */
        if (!(LJ_DUALNUM && (ir->op2 & IRSLOAD_CONVERT)) &&
            (ir->op2 & (IRSLOAD_READONLY|IRSLOAD_PARENT)) != IRSLOAD_PARENT)
          sn |= SNAP_NORESTORE;
      }
      if (LJ_SOFTFP && irt_isnum(ir->t))
        sn |= SNAP_SOFTFPNUM;
      map[n++] = sn;
    }
  }
  return n;
}

/* Add frame links at the end of the snapshot. */
static BCReg snapshot_framelinks(jit_State *J, SnapEntry *map)
{
  cTValue *frame = J->L->base - 1;
  cTValue *lim = J->L->base - J->baseslot;
  GCfunc *fn = frame_func(frame);
  cTValue *ftop = isluafunc(fn) ? (frame+funcproto(fn)->framesize) : J->L->top;
  MSize f = 0;
  lua_assert(!LJ_FR2);  /* TODO_FR2: store 64 bit PCs. */
  map[f++] = SNAP_MKPC(J->pc);  /* The current PC is always the first entry. */
  while (frame > lim) {  /* Backwards traversal of all frames above base. */
    if (frame_islua(frame)) {
      map[f++] = SNAP_MKPC(frame_pc(frame));
      frame = frame_prevl(frame);
    } else if (frame_iscont(frame)) {
      map[f++] = SNAP_MKFTSZ(frame_ftsz(frame));
      map[f++] = SNAP_MKPC(frame_contpc(frame));
      frame = frame_prevd(frame);
    } else {
      lua_assert(!frame_isc(frame));
      map[f++] = SNAP_MKFTSZ(frame_ftsz(frame));
      frame = frame_prevd(frame);
      continue;
    }
    if (frame + funcproto(frame_func(frame))->framesize > ftop)
      ftop = frame + funcproto(frame_func(frame))->framesize;
  }
  lua_assert(f == (MSize)(1 + J->framedepth));
  return (BCReg)(ftop - lim);
}

/* Take a snapshot of the current stack. */
static void snapshot_stack(jit_State *J, SnapShot *snap, MSize nsnapmap)
{
  BCReg nslots = J->baseslot + J->maxslot;
  MSize nent;
  SnapEntry *p;
  /* Conservative estimate. */
```

```
  lj_snap_grow_map(J, nsnapmap + nslots + (MSize)J->framedepth+1);
  p = &J->cur.snapmap[nsnapmap];
  nent = snapshot_slots(J, p, nslots);
  snap->topslot = (uint8_t)snapshot_framelinks(J, p + nent);
  snap->mapofs = (uint16_t)nsnapmap;
  snap->ref = (IRRef1)J->cur.nins;
  snap->nent = (uint8_t)nent;
  snap->nslots = (uint8_t)nslots;
  snap->count = 0;
  J->cur.nsnapmap = (uint16_t)(nsnapmap + nent + 1 + J->framedepth);
}

/* Add or merge a snapshot. */
void lj_snap_add(jit_State *J)
{
  MSize nsnap = J->cur.nsnap;
  MSize nsnapmap = J->cur.nsnapmap;
  /* Merge if no ins. inbetween or if requested and no guard inbetween. */
  if (J->mergesnap ? !irt_isguard(J->guardemit) :
      (nsnap > 0 && J->cur.snap[nsnap-1].ref == J->cur.nins)) {
    if (nsnap == 1) {  /* But preserve snap #0 PC. */
      emitir_raw(IRT(IR_NOP, IRT_NIL), 0, 0);
      goto nomerge;
    }
    nsnapmap = J->cur.snap[--nsnap].mapofs;
  } else {
  nomerge:
    lj_snap_grow_buf(J, nsnap+1);
    J->cur.nsnap = (uint16_t)(nsnap+1);
  }
  J->mergesnap = 0;
  J->guardemit.irt = 0;
  snapshot_stack(J, &J->cur.snap[nsnap], nsnapmap);
}

/* -- Snapshot modification ----------------------------------------------- */

#define SNAP_USEDEF_SLOTS        (LJ_MAX_JSLOTS+LJ_STACK_EXTRA)

/* Find unused slots with reaching-definitions bytecode data-flow analysis. */
static BCReg snap_usedef(jit_State *J, uint8_t *udf,
                         const BCIns *pc, BCReg maxslot)
{
  BCReg s;
  GCobj *o;

  if (maxslot == 0) return 0;
#ifdef LUAJIT_USE_VALGRIND
  /* Avoid errors for harmless reads beyond maxslot. */
  memset(udf, 1, SNAP_USEDEF_SLOTS);
#else
  memset(udf, 1, maxslot);
#endif

  /* Treat open upvalues as used. */
  o = gcref(J->L->openupval);
  while (o) {
    if (uvval(gco2uv(o)) < J->L->base) break;
    udf[uvval(gco2uv(o)) - J->L->base] = 0;
    o = gcref(o->gch.nextgc);
  }

#define USE_SLOT(s)               udf[(s)] &= ~1
#define DEF_SLOT(s)               udf[(s)] *= 3

  /* Scan through following bytecode and check for uses/defs. */
  lua_assert(pc >= proto_bc(J->pt) && pc < proto_bc(J->pt) + J->pt->sizebc);
  for (;;) {
    BCIns ins = *pc++;
    BCOp op = bc_op(ins);
    switch (bcmode_b(op)) {
    case BCMvar: USE_SLOT(bc_b(ins)); break;
    default: break;
    }
    switch (bcmode_c(op)) {
    case BCMvar: USE_SLOT(bc_c(ins)); break;
```

```
    case BCMrbase:
      lua_assert(op == BC_CAT);
      for (s = bc_b(ins); s <= bc_c(ins); s++) USE_SLOT(s);
      for (; s < maxslot; s++) DEF_SLOT(s);
      break;
    case BCMjump:
    handle_jump: {
      BCReg minslot = bc_a(ins);
      if (op >= BC_FORI && op <= BC_JFORL) minslot += FORL_EXT;
      else if (op >= BC_ITERL && op <= BC_JITERL) minslot += bc_b(pc[-2])-1;
      else if (op == BC_UCLO) { pc += bc_j(ins); break; }
      for (s = minslot; s < maxslot; s++) DEF_SLOT(s);
      return minslot < maxslot ? minslot : maxslot;
      }
    case BCMlit:
      if (op == BC_JFORL || op == BC_JITERL || op == BC_JLOOP) {
        goto handle_jump;
      } else if (bc_isret(op)) {
        BCReg top = op == BC_RETM ? maxslot : (bc_a(ins) + bc_d(ins)-1);
        for (s = 0; s < bc_a(ins); s++) DEF_SLOT(s);
        for (; s < top; s++) USE_SLOT(s);
        for (; s < maxslot; s++) DEF_SLOT(s);
        return 0;
      }
      break;
    case BCMfunc: return maxslot;  /* NYI: will abort, anyway. */
    default: break;
    }
    switch (bcmode_a(op)) {
    case BCMvar: USE_SLOT(bc_a(ins)); break;
    case BCMdst:
       if (!(op == BC_ISTC || op == BC_ISFC)) DEF_SLOT(bc_a(ins));
       break;
    case BCMbase:
      if (op >= BC_CALLM && op <= BC_VARG) {
        BCReg top = (op == BC_CALLM || op == BC_CALLMT || bc_c(ins) == 0) ?
                    maxslot : (bc_a(ins) + bc_c(ins)+LJ_FR2);
        if (LJ_FR2) DEF_SLOT(bc_a(ins)+1);
        s = bc_a(ins) - ((op == BC_ITERC || op == BC_ITERN) ? 3 : 0);
        for (; s < top; s++) USE_SLOT(s);
        for (; s < maxslot; s++) DEF_SLOT(s);
        if (op == BC_CALLT || op == BC_CALLMT) {
          for (s = 0; s < bc_a(ins); s++) DEF_SLOT(s);
          return 0;
        }
      } else if (op == BC_KNIL) {
        for (s = bc_a(ins); s <= bc_d(ins); s++) DEF_SLOT(s);
      } else if (op == BC_TSETM) {
        for (s = bc_a(ins)-1; s < maxslot; s++) USE_SLOT(s);
      }
      break;
    default: break;
    }
    lua_assert(pc >= proto_bc(J->pt) && pc < proto_bc(J->pt) + J->pt->sizebc);
  }

#undef USE_SLOT
#undef DEF_SLOT

  return 0;  /* unreachable */
}

/* Purge dead slots before the next snapshot. */
void lj_snap_purge(jit_State *J)
{
  uint8_t udf[SNAP_USEDEF_SLOTS];
  BCReg maxslot = J->maxslot;
  BCReg s = snap_usedef(J, udf, J->pc, maxslot);
  for (; s < maxslot; s++)
    if (udf[s] != 0)
      J->base[s] = 0;  /* Purge dead slots. */
}

/* Shrink last snapshot. */
void lj_snap_shrink(jit_State *J)
{
```

```
  SnapShot *snap = &J->cur.snap[J->cur.nsnap-1];
  SnapEntry *map = &J->cur.snapmap[snap->mapofs];
  MSize n, m, nlim, nent = snap->nent;
  uint8_t udf[SNAP_USEDEF_SLOTS];
  BCReg maxslot = J->maxslot;
  BCReg minslot = snap_usedef(J, udf, snap_pc(map[nent]), maxslot);
  BCReg baseslot = J->baseslot;
  maxslot += baseslot;
  minslot += baseslot;
  snap->nslots = (uint8_t)maxslot;
  for (n = m = 0; n < nent; n++) {  /* Remove unused slots from snapshot. */
    BCReg s = snap_slot(map[n]);
    if (s < minslot || (s < maxslot && udf[s-baseslot] == 0))
      map[m++] = map[n];  /* Only copy used slots. */
  }
  snap->nent = (uint8_t)m;
  nlim = J->cur.nsnapmap - snap->mapofs - 1;
  while (n <= nlim) map[m++] = map[n++];  /* Move PC + frame links down. */
  J->cur.nsnapmap = (uint16_t)(snap->mapofs + m);  /* Free up space in map. */
}

/* -- Snapshot access ----------------------------------------------------- */

/* Initialize a Bloom Filter with all renamed refs.
** There are very few renames (often none), so the filter has
** very few bits set. This makes it suitable for negative filtering.
*/
static BloomFilter snap_renamefilter(GCtrace *T, SnapNo lim)
{
  BloomFilter rfilt = 0;
  IRIns *ir;
  for (ir = &T->ir[T->nins-1]; ir->o == IR_RENAME; ir--)
    if (ir->op2 <= lim)
      bloomset(rfilt, ir->op1);
  return rfilt;
}

/* Process matching renames to find the original RegSP. */
static RegSP snap_renameref(GCtrace *T, SnapNo lim, IRRef ref, RegSP rs)
{
  IRIns *ir;
  for (ir = &T->ir[T->nins-1]; ir->o == IR_RENAME; ir--)
    if (ir->op1 == ref && ir->op2 <= lim)
      rs = ir->prev;
  return rs;
}

/* Copy RegSP from parent snapshot to the parent links of the IR. */
IRIns *lj_snap_regspmap(GCtrace *T, SnapNo snapno, IRIns *ir)
{
  SnapShot *snap = &T->snap[snapno];
  SnapEntry *map = &T->snapmap[snap->mapofs];
  BloomFilter rfilt = snap_renamefilter(T, snapno);
  MSize n = 0;
  IRRef ref = 0;
  for ( ; ; ir++) {
    uint32_t rs;
    if (ir->o == IR_SLOAD) {
      if (!(ir->op2 & IRSLOAD_PARENT)) break;
      for ( ; ; n++) {
        lua_assert(n < snap->nent);
        if (snap_slot(map[n]) == ir->op1) {
          ref = snap_ref(map[n++]);
          break;
        }
      }
    } else if (LJ_SOFTFP && ir->o == IR_HIOP) {
      ref++;
    } else if (ir->o == IR_PVAL) {
      ref = ir->op1 + REF_BIAS;
    } else {
      break;
    }
    rs = T->ir[ref].prev;
    if (bloomtest(rfilt, ref))
      rs = snap_renameref(T, snapno, ref, rs);
```

```
    ir->prev = (uint16_t)rs;
    lua_assert(regsp_used(rs));
  }
  return ir;
}

/* -- Snapshot replay ----------------------------------------------------- */

/* Replay constant from parent trace. */
static TRef snap_replay_const(jit_State *J, IRIns *ir)
{
  /* Only have to deal with constants that can occur in stack slots. */
  switch ((IROp)ir->o) {
  case IR_KPRI: return TREF_PRI(irt_type(ir->t));
  case IR_KINT: return lj_ir_kint(J, ir->i);
  case IR_KGC: return lj_ir_kgc(J, ir_kgc(ir), irt_t(ir->t));
  case IR_KNUM: return lj_ir_k64(J, IR_KNUM, ir_knum(ir));
  case IR_KINT64: return lj_ir_k64(J, IR_KINT64, ir_kint64(ir));
  case IR_KPTR: return lj_ir_kptr(J, ir_kptr(ir));  /* Continuation. */
  default: lua_assert(0); return TREF_NIL; break;
  }
}

/* De-duplicate parent reference. */
static TRef snap_dedup(jit_State *J, SnapEntry *map, MSize nmax, IRRef ref)
{
  MSize j;
  for (j = 0; j < nmax; j++)
    if (snap_ref(map[j]) == ref)
      return J->slot[snap_slot(map[j])] & ~(SNAP_CONT|SNAP_FRAME);
  return 0;
}

/* Emit parent reference with de-duplication. */
static TRef snap_pref(jit_State *J, GCtrace *T, SnapEntry *map, MSize nmax,
                      BloomFilter seen, IRRef ref)
{
  IRIns *ir = &T->ir[ref];
  TRef tr;
  if (irref_isk(ref))
    tr = snap_replay_const(J, ir);
  else if (!regsp_used(ir->prev))
    tr = 0;
  else if (!bloomtest(seen, ref) || (tr = snap_dedup(J, map, nmax, ref)) == 0)
    tr = emitir(IRT(IR_PVAL, irt_type(ir->t)), ref - REF_BIAS, 0);
  return tr;
}

/* Check whether a sunk store corresponds to an allocation. Slow path. */
static int snap_sunk_store2(jit_State *J, IRIns *ira, IRIns *irs)
{
  if (irs->o == IR_ASTORE || irs->o == IR_HSTORE ||
      irs->o == IR_FSTORE || irs->o == IR_XSTORE) {
    IRIns *irk = IR(irs->op1);
    if (irk->o == IR_AREF || irk->o == IR_HREFK)
      irk = IR(irk->op1);
    return (IR(irk->op1) == ira);
  }
  return 0;
}

/* Check whether a sunk store corresponds to an allocation. Fast path. */
static LJ_AINLINE int snap_sunk_store(jit_State *J, IRIns *ira, IRIns *irs)
{
  if (irs->s != 255)
    return (ira + irs->s == irs);  /* Fast check. */
  return snap_sunk_store2(J, ira, irs);
}

/* Replay snapshot state to setup side trace. */
void lj_snap_replay(jit_State *J, GCtrace *T)
{
  SnapShot *snap = &T->snap[J->exitno];
  SnapEntry *map = &T->snapmap[snap->mapofs];
  MSize n, nent = snap->nent;
  BloomFilter seen = 0;
```

```
  int pass23 = 0;
  J->framedepth = 0;
  /* Emit IR for slots inherited from parent snapshot. */
  for (n = 0; n < nent; n++) {
    SnapEntry sn = map[n];
    BCReg s = snap_slot(sn);
    IRRef ref = snap_ref(sn);
    IRIns *ir = &T->ir[ref];
    TRef tr;
    /* The bloom filter avoids O(nent^2) overhead for de-duping slots. */
    if (bloomtest(seen, ref) && (tr = snap_dedup(J, map, n, ref)) != 0)
      goto setslot;
    bloomset(seen, ref);
    if (irref_isk(ref)) {
      tr = snap_replay_const(J, ir);
    } else if (!regsp_used(ir->prev)) {
      pass23 = 1;
      lua_assert(s != 0);
      tr = s;
    } else {
      IRType t = irt_type(ir->t);
      uint32_t mode = IRSLOAD_INHERIT|IRSLOAD_PARENT;
      if (LJ_SOFTFP && (sn & SNAP_SOFTFPNUM)) t = IRT_NUM;
      if (ir->o == IR_SLOAD) mode |= (ir->op2 & IRSLOAD_READONLY);
      tr = emitir_raw(IRT(IR_SLOAD, t), s, mode);
    }
  setslot:
    J->slot[s] = tr | (sn&(SNAP_CONT|SNAP_FRAME));  /* Same as TREF_* flags. */
    J->framedepth += ((sn & (SNAP_CONT|SNAP_FRAME)) && s);
    if ((sn & SNAP_FRAME))
      J->baseslot = s+1;
  }
  if (pass23) {
    IRIns *irlast = &T->ir[snap->ref];
    pass23 = 0;
    /* Emit dependent PVALs. */
    for (n = 0; n < nent; n++) {
      SnapEntry sn = map[n];
      IRRef refp = snap_ref(sn);
      IRIns *ir = &T->ir[refp];
      if (regsp_reg(ir->r) == RID_SUNK) {
        if (J->slot[snap_slot(sn)] != snap_slot(sn)) continue;
        pass23 = 1;
        lua_assert(ir->o == IR_TNEW || ir->o == IR_TDUP ||
                   ir->o == IR_CNEW || ir->o == IR_CNEWI);
        if (ir->op1 >= T->nk) snap_pref(J, T, map, nent, seen, ir->op1);
        if (ir->op2 >= T->nk) snap_pref(J, T, map, nent, seen, ir->op2);
        if (LJ_HASFFI && ir->o == IR_CNEWI) {
          if (LJ_32 && refp+1 < T->nins && (ir+1)->o == IR_HIOP)
            snap_pref(J, T, map, nent, seen, (ir+1)->op2);
        } else {
          IRIns *irs;
          for (irs = ir+1; irs < irlast; irs++)
            if (irs->r == RID_SINK && snap_sunk_store(J, ir, irs)) {
              if (snap_pref(J, T, map, nent, seen, irs->op2) == 0)
                snap_pref(J, T, map, nent, seen, T->ir[irs->op2].op1);
              else if ((LJ_SOFTFP || (LJ_32 && LJ_HASFFI)) &&
                       irs+1 < irlast && (irs+1)->o == IR_HIOP)
                snap_pref(J, T, map, nent, seen, (irs+1)->op2);
            }
        }
      } else if (!irref_isk(refp) && !regsp_used(ir->prev)) {
        lua_assert(ir->o == IR_CONV && ir->op2 == IRCONV_NUM_INT);
        J->slot[snap_slot(sn)] = snap_pref(J, T, map, nent, seen, ir->op1);
      }
    }
    /* Replay sunk instructions. */
    for (n = 0; pass23 && n < nent; n++) {
      SnapEntry sn = map[n];
      IRRef refp = snap_ref(sn);
      IRIns *ir = &T->ir[refp];
      if (regsp_reg(ir->r) == RID_SUNK) {
        TRef op1, op2;
        if (J->slot[snap_slot(sn)] != snap_slot(sn)) {  /* De-dup allocs. */
          J->slot[snap_slot(sn)] = J->slot[J->slot[snap_slot(sn)]];
          continue;
```

```
513         }
514         op1 = ir->op1;
515         if (op1 >= T->nk) op1 = snap_pref(J, T, map, nent, seen, op1);
516         op2 = ir->op2;
517         if (op2 >= T->nk) op2 = snap_pref(J, T, map, nent, seen, op2);
518         if (LJ_HASFFI && ir->o == IR_CNEWI) {
519           if (LJ_32 && refp+1 < T->nins && (ir+1)->o == IR_HIOP) {
520             lj_needsplit(J);  /* Emit joining HIOP. */
521             op2 = emitir_raw(IRT(IR_HIOP, IRT_I64), op2,
522                              snap_pref(J, T, map, nent, seen, (ir+1)->op2));
523           }
524           J->slot[snap_slot(sn)] = emitir(ir->ot, op1, op2);
525         } else {
526           IRIns *irs;
527           TRef tr = emitir(ir->ot, op1, op2);
528           J->slot[snap_slot(sn)] = tr;
529           for (irs = ir+1; irs < irlast; irs++)
530             if (irs->r == RID_SINK && snap_sunk_store(J, ir, irs)) {
531               IRIns *irr = &T->ir[irs->op1];
532               TRef val, key = irr->op2, tmp = tr;
533               if (irr->o != IR_FREF) {
534                 IRIns *irk = &T->ir[key];
535                 if (irr->o == IR_HREFK)
536                   key = lj_ir_kslot(J, snap_replay_const(J, &T->ir[irk->op1]),
537                                     irk->op2);
538                 else
539                   key = snap_replay_const(J, irk);
540                 if (irr->o == IR_HREFK || irr->o == IR_AREF) {
541                   IRIns *irf = &T->ir[irr->op1];
542                   tmp = emitir(irf->ot, tmp, irf->op2);
543                 }
544               }
545               tmp = emitir(irr->ot, tmp, key);
546               val = snap_pref(J, T, map, nent, seen, irs->op2);
547               if (val == 0) {
548                 IRIns *irc = &T->ir[irs->op2];
549                 lua_assert(irc->o == IR_CONV && irc->op2 == IRCONV_NUM_INT);
550                 val = snap_pref(J, T, map, nent, seen, irc->op1);
551                 val = emitir(IRTN(IR_CONV), val, IRCONV_NUM_INT);
552               } else if ((LJ_SOFTFP || (LJ_32 && LJ_HASFFI)) &&
553                          irs+1 < irlast && (irs+1)->o == IR_HIOP) {
554                 IRType t = IRT_I64;
555                 if (LJ_SOFTFP && irt_type((irs+1)->t) == IRT_SOFTFP)
556                   t = IRT_NUM;
557                 lj_needsplit(J);
558                 if (irref_isk(irs->op2) && irref_isk((irs+1)->op2)) {
559                   uint64_t k = (uint32_t)T->ir[irs->op2].i +
560                                ((uint64_t)T->ir[(irs+1)->op2].i << 32);
561                   val = lj_ir_k64(J, t == IRT_I64 ? IR_KINT64 : IR_KNUM,
562                                   lj_ir_k64_find(J, k));
563                 } else {
564                   val = emitir_raw(IRT(IR_HIOP, t), val,
565                           snap_pref(J, T, map, nent, seen, (irs+1)->op2));
566                 }
567                 tmp = emitir(IRT(irs->o, t), tmp, val);
568                 continue;
569               }
570               tmp = emitir(irs->ot, tmp, val);
571             } else if (LJ_HASFFI && irs->o == IR_XBAR && ir->o == IR_CNEW) {
572               emitir(IRT(IR_XBAR, IRT_NIL), 0, 0);
573             }
574         }
575       }
576     }
577   }
578   J->base = J->slot + J->baseslot;
579   J->maxslot = snap->nslots - J->baseslot;
580   lj_snap_add(J);
581   if (pass23)  /* Need explicit GC step _after_ initial snapshot. */
582     emitir_raw(IRTG(IR_GCSTEP, IRT_NIL), 0, 0);
583 }
584
585 /* -- Snapshot restore ----------------------------------------------------- */
586
587 static void snap_unsink(jit_State *J, GCtrace *T, ExitState *ex,
588                         SnapNo snapno, BloomFilter rfilt,
```

```
                      IRIns *ir, TValue *o);

/* Restore a value from the trace exit state. */
static void snap_restoreval(jit_State *J, GCtrace *T, ExitState *ex,
                            SnapNo snapno, BloomFilter rfilt,
                            IRRef ref, TValue *o)
{
  IRIns *ir = &T->ir[ref];
  IRType1 t = ir->t;
  RegSP rs = ir->prev;
  if (irref_isk(ref)) {  /* Restore constant slot. */
    lj_ir_kvalue(J->L, o, ir);
    return;
  }
  if (LJ_UNLIKELY(bloomtest(rfilt, ref)))
    rs = snap_renameref(T, snapno, ref, rs);
  lua_assert(!LJ_GC64);  /* TODO_GC64: handle 64 bit references. */
  if (ra_hasspill(regsp_spill(rs))) {  /* Restore from spill slot. */
    int32_t *sps = &ex->spill[regsp_spill(rs)];
    if (irt_isinteger(t)) {
      setintV(o, *sps);
#if !LJ_SOFTFP
    } else if (irt_isnum(t)) {
      o->u64 = *(uint64_t *)sps;
#endif
    } else if (LJ_64 && irt_islightud(t)) {
      /* 64 bit lightuserdata which may escape already has the tag bits. */
      o->u64 = *(uint64_t *)sps;
    } else {
      lua_assert(!irt_ispri(t));  /* PRI refs never have a spill slot. */
      setgcV(J->L, o, (GCobj *)(uintptr_t)*(GCSize *)sps, irt_toitype(t));
    }
  } else {  /* Restore from register. */
    Reg r = regsp_reg(rs);
    if (ra_noreg(r)) {
      lua_assert(ir->o == IR_CONV && ir->op2 == IRCONV_NUM_INT);
      snap_restoreval(J, T, ex, snapno, rfilt, ir->op1, o);
      if (LJ_DUALNUM) setnumV(o, (lua_Number)intV(o));
      return;
    } else if (irt_isinteger(t)) {
      setintV(o, (int32_t)ex->gpr[r-RID_MIN_GPR]);
#if !LJ_SOFTFP
    } else if (irt_isnum(t)) {
      setnumV(o, ex->fpr[r-RID_MIN_FPR]);
#endif
    } else if (LJ_64 && irt_islightud(t)) {
      /* 64 bit lightuserdata which may escape already has the tag bits. */
      o->u64 = ex->gpr[r-RID_MIN_GPR];
    } else if (irt_ispri(t)) {
      setpriV(o, irt_toitype(t));
    } else {
      setgcV(J->L, o, (GCobj *)ex->gpr[r-RID_MIN_GPR], irt_toitype(t));
    }
  }
}

#if LJ_HASFFI
/* Restore raw data from the trace exit state. */
static void snap_restoredata(GCtrace *T, ExitState *ex,
                             SnapNo snapno, BloomFilter rfilt,
                             IRRef ref, void *dst, CTSize sz)
{
  IRIns *ir = &T->ir[ref];
  RegSP rs = ir->prev;
  int32_t *src;
  uint64_t tmp;
  if (irref_isk(ref)) {
    if (ir->o == IR_KNUM || ir->o == IR_KINT64) {
      src = mref(ir->ptr, int32_t);
    } else if (sz == 8) {
      tmp = (uint64_t)(uint32_t)ir->i;
      src = (int32_t *)&tmp;
    } else {
      src = &ir->i;
    }
  } else {
```

```
    if (LJ_UNLIKELY(bloomtest(rfilt, ref)))
      rs = snap_renameref(T, snapno, ref, rs);
    if (ra_hasspill(regsp_spill(rs))) {
      src = &ex->spill[regsp_spill(rs)];
      if (sz == 8 && !irt_is64(ir->t)) {
        tmp = (uint64_t)(uint32_t)*src;
        src = (int32_t *)&tmp;
      }
    } else {
      Reg r = regsp_reg(rs);
      if (ra_noreg(r)) {
        /* Note: this assumes CNEWI is never used for SOFTFP split numbers. */
        lua_assert(sz == 8 && ir->o == IR_CONV && ir->op2 == IRCONV_NUM_INT);
        snap_restoredata(T, ex, snapno, rfilt, ir->op1, dst, 4);
        *(lua_Number *)dst = (lua_Number)*(int32_t *)dst;
        return;
      }
      src = (int32_t *)&ex->gpr[r-RID_MIN_GPR];
#if !LJ_SOFTFP
      if (r >= RID_MAX_GPR) {
        src = (int32_t *)&ex->fpr[r-RID_MIN_FPR];
#if LJ_TARGET_PPC
        if (sz == 4) {  /* PPC FPRs are always doubles. */
          *(float *)dst = (float)*(double *)src;
          return;
        }
#else
        if (LJ_BE && sz == 4) src++;
#endif
      }
#endif
    }
  }
  lua_assert(sz == 1 || sz == 2 || sz == 4 || sz == 8);
  if (sz == 4) *(int32_t *)dst = *src;
  else if (sz == 8) *(int64_t *)dst = *(int64_t *)src;
  else if (sz == 1) *(int8_t *)dst = (int8_t)*src;
  else *(int16_t *)dst = (int16_t)*src;
}
#endif

/* Unsink allocation from the trace exit state. Unsink sunk stores. */
static void snap_unsink(jit_State *J, GCtrace *T, ExitState *ex,
                        SnapNo snapno, BloomFilter rfilt,
                        IRIns *ir, TValue *o)
{
  lua_assert(ir->o == IR_TNEW || ir->o == IR_TDUP ||
             ir->o == IR_CNEW || ir->o == IR_CNEWI);
#if LJ_HASFFI
  if (ir->o == IR_CNEW || ir->o == IR_CNEWI) {
    CTState *cts = ctype_cts(J->L);
    CTypeID id = (CTypeID)T->ir[ir->op1].i;
    CTSize sz = lj_ctype_size(cts, id);
    GCcdata *cd = lj_cdata_new(cts, id, sz);
    setcdataV(J->L, o, cd);
    if (ir->o == IR_CNEWI) {
      uint8_t *p = (uint8_t *)cdataptr(cd);
      lua_assert(sz == 4 || sz == 8);
      if (LJ_32 && sz == 8 && ir+1 < T->ir + T->nins && (ir+1)->o == IR_HIOP) {
        snap_restoredata(T, ex, snapno, rfilt, (ir+1)->op2, LJ_LE?p+4:p, 4);
        if (LJ_BE) p += 4;
        sz = 4;
      }
      snap_restoredata(T, ex, snapno, rfilt, ir->op2, p, sz);
    } else {
      IRIns *irs, *irlast = &T->ir[T->snap[snapno].ref];
      for (irs = ir+1; irs < irlast; irs++)
        if (irs->r == RID_SINK && snap_sunk_store(J, ir, irs)) {
          IRIns *iro = &T->ir[T->ir[irs->op1].op2];
          uint8_t *p = (uint8_t *)cd;
          CTSize szs;
          lua_assert(irs->o == IR_XSTORE && T->ir[irs->op1].o == IR_ADD);
          lua_assert(iro->o == IR_KINT || iro->o == IR_KINT64);
          if (irt_is64(irs->t)) szs = 8;
          else if (irt_isi8(irs->t) || irt_isu8(irs->t)) szs = 1;
          else if (irt_isi16(irs->t) || irt_isu16(irs->t)) szs = 2;
```

```
          else szs = 4;
          if (LJ_64 && iro->o == IR_KINT64)
            p += (int64_t)ir_k64(iro)->u64;
          else
            p += iro->i;
          lua_assert(p >= (uint8_t *)cdataptr(cd) &&
                     p + szs <= (uint8_t *)cdataptr(cd) + sz);
          if (LJ_32 && irs+1 < T->ir + T->nins && (irs+1)->o == IR_HIOP) {
            lua_assert(szs == 4);
            snap_restoredata(T, ex, snapno, rfilt, (irs+1)->op2, LJ_LE?p+4:p,4);
            if (LJ_BE) p += 4;
          }
          snap_restoredata(T, ex, snapno, rfilt, irs->op2, p, szs);
        }
      }
    } else
#endif
    {
      IRIns *irs, *irlast;
      GCtab *t = ir->o == IR_TNEW ? lj_tab_new(J->L, ir->op1, ir->op2) :
                                    lj_tab_dup(J->L, ir_ktab(&T->ir[ir->op1]));
      settabV(J->L, o, t);
      irlast = &T->ir[T->snap[snapno].ref];
      for (irs = ir+1; irs < irlast; irs++)
        if (irs->r == RID_SINK && snap_sunk_store(J, ir, irs)) {
          IRIns *irk = &T->ir[irs->op1];
          TValue tmp, *val;
          lua_assert(irs->o == IR_ASTORE || irs->o == IR_HSTORE ||
                     irs->o == IR_FSTORE);
          if (irk->o == IR_FREF) {
            lua_assert(irk->op2 == IRFL_TAB_META);
            snap_restoreval(J, T, ex, snapno, rfilt, irs->op2, &tmp);
            /* NOBARRIER: The table is new (marked white). */
            setgcref(t->metatable, obj2gco(tabV(&tmp)));
          } else {
            irk = &T->ir[irk->op2];
            if (irk->o == IR_KSLOT) irk = &T->ir[irk->op1];
            lj_ir_kvalue(J->L, &tmp, irk);
            val = lj_tab_set(J->L, t, &tmp);
            /* NOBARRIER: The table is new (marked white). */
            snap_restoreval(J, T, ex, snapno, rfilt, irs->op2, val);
            if (LJ_SOFTFP && irs+1 < T->ir + T->nins && (irs+1)->o == IR_HIOP) {
              snap_restoreval(J, T, ex, snapno, rfilt, (irs+1)->op2, &tmp);
              val->u32.hi = tmp.u32.lo;
            }
          }
        }
    }
  }

  /* Restore interpreter state from exit state with the help of a snapshot. */
  const BCIns *lj_snap_restore(jit_State *J, void *exptr)
  {
    ExitState *ex = (ExitState *)exptr;
    SnapNo snapno = J->exitno;  /* For now, snapno == exitno. */
    GCtrace *T = traceref(J, J->parent);
    SnapShot *snap = &T->snap[snapno];
    MSize n, nent = snap->nent;
    SnapEntry *map = &T->snapmap[snap->mapofs];
    SnapEntry *flinks = &T->snapmap[snap_nextofs(T, snap)-1];
    ptrdiff_t ftsz0;
    TValue *frame;
    BloomFilter rfilt = snap_renamefilter(T, snapno);
    const BCIns *pc = snap_pc(map[nent]);
    lua_State *L = J->L;

    /* Set interpreter PC to the next PC to get correct error messages. */
    setcframe_pc(cframe_raw(L->cframe), pc+1);

    /* Make sure the stack is big enough for the slots from the snapshot. */
    if (LJ_UNLIKELY(L->base + snap->topslot >= tvref(L->maxstack))) {
      L->top = curr_topL(L);
      lj_state_growstack(L, snap->topslot - curr_proto(L)->framesize);
    }

    /* Fill stack slots with data from the registers and spill slots. */
```

```
817   frame = L->base-1;
818   ftsz0 = frame_ftsz(frame);  /* Preserve link to previous frame in slot #0. */
819   for (n = 0; n < nent; n++) {
820     SnapEntry sn = map[n];
821     if (!(sn & SNAP_NORESTORE)) {
822       TValue *o = &frame[snap_slot(sn)];
823       IRRef ref = snap_ref(sn);
824       IRIns *ir = &T->ir[ref];
825       if (ir->r == RID_SUNK) {
826         MSize j;
827         for (j = 0; j < n; j++)
828           if (snap_ref(map[j]) == ref) {  /* De-duplicate sunk allocations. */
829             copyTV(L, o, &frame[snap_slot(map[j])]);
830             goto dupslot;
831           }
832         snap_unsink(J, T, ex, snapno, rfilt, ir, o);
833       dupslot:
834         continue;
835       }
836       snap_restoreval(J, T, ex, snapno, rfilt, ref, o);
837       if (LJ_SOFTFP && (sn & SNAP_SOFTFPNUM) && tvisint(o)) {
838         TValue tmp;
839         snap_restoreval(J, T, ex, snapno, rfilt, ref+1, &tmp);
840         o->u32.hi = tmp.u32.lo;
841       } else if ((sn & (SNAP_CONT|SNAP_FRAME))) {
842         lua_assert(!LJ_FR2);  /* TODO_FR2: store 64 bit PCs. */
843         /* Overwrite tag with frame link. */
844         setframe_ftsz(o, snap_slot(sn) != 0 ? (int32_t)*flinks-- : ftsz0);
845         L->base = o+1;
846       }
847     }
848   }
849   lua_assert(map + nent == flinks);
850
851   /* Compute current stack top. */
852   switch (bc_op(*pc)) {
853   default:
854     if (bc_op(*pc) < BC_FUNCF) {
855       L->top = curr_topL(L);
856       break;
857     }
858     /* fallthrough */
859   case BC_CALLM: case BC_CALLMT: case BC_RETM: case BC_TSETM:
860     L->top = frame + snap->nslots;
861     break;
862   }
863   return pc;
864 }
865
866 #undef IR
867 #undef emitir_raw
868 #undef emitir
869
870 #endif
```

One Level Up Top Level

src/lj_opt_mem.c - luajit-2.0-src

Data types defined

- AliasRet

Functions defined

- aa_ahref
- aa_cnew
- aa_escape
- aa_findcnew
- aa_fref
- aa_table
- aa_uref
- aa_xref
- fwd_aa_tab_clear
- fwd_ahload
- fwd_aload_reassoc
- lj_opt_dse_ahstore
- lj_opt_dse_fstore
- lj_opt_dse_ustore
- lj_opt_dse_xstore
- lj_opt_fwd_aload
- lj_opt_fwd_fload
- lj_opt_fwd_hload
- lj_opt_fwd_href_nokey
- lj_opt_fwd_hrefk
- lj_opt_fwd_tab_len
- lj_opt_fwd_tptr
- lj_opt_fwd_uload
- lj_opt_fwd_wasnonnil
- lj_opt_fwd_xload
- reassoc_trycse
- reassoc_xref

Macros defined

- IR
- IR
- LUA_CORE
- fins
- fins
- fleft
- fleft
- fright
- fright
- lj_opt_mem_c

Source code

```
/*
** Memory access optimizations.
** AA: Alias Analysis using high-level semantic disambiguation.
** FWD: Load Forwarding (L2L) + Store Forwarding (S2L).
** DSE: Dead-Store Elimination.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_opt_mem_c
 #define LUA_CORE

 #include "lj_obj.h"

 #if LJ_HASJIT

 #include "lj_tab.h"
 #include "lj_ir.h"
 #include "lj_jit.h"
 #include "lj_iropt.h"
 #include "lj_ircall.h"

 /* Some local macros to save typing. Undef'd at the end. */
 #define IR(ref)                 (&J->cur.ir[(ref)])
 #define fins                (&J->fold.ins)
 #define fleft                (&J->fold.left)
 #define fright                (&J->fold.right)

 /*
 ** Caveat #1: return value is not always a IRef -- only use with tref_ref().
 ** Caveat #2: FWD relies on active CSE for xREF operands -- see lj_opt_fold().
 */

 /* Return values from alias analysis. */
 typedef enum {
   ALIAS_NO,        /* The two refs CANNOT alias (exact). */
   ALIAS_MAY,        /* The two refs MAY alias (inexact). */
   ALIAS_MUST        /* The two refs MUST alias (exact). */
 } AliasRet;

 /* -- ALOAD/HLOAD forwarding and ASTORE/HSTORE elimination ---------------- */

 /* Simplified escape analysis: check for intervening stores. */
 static AliasRet aa_escape(jit_State *J, IRIns *ir, IRIns *stop)
 {
   IRRef ref = (IRRef)(ir - J->cur.ir);  /* The ref that might be stored. */
   for (ir++; ir < stop; ir++)
     if (ir->op2 == ref &&
```

```
        (ir->o == IR_ASTORE || ir->o == IR_HSTORE ||
         ir->o == IR_USTORE || ir->o == IR_FSTORE))
      return ALIAS_MAY;  /* Reference was stored and might alias. */
  return ALIAS_NO;  /* Reference was not stored. */
}

/* Alias analysis for two different table references. */
static AliasRet aa_table(jit_State *J, IRRef ta, IRRef tb)
{
  IRIns *taba = IR(ta), *tabb = IR(tb);
  int newa, newb;
  lua_assert(ta != tb);
  lua_assert(irt_istab(taba->t) && irt_istab(tabb->t));
  /* Disambiguate new allocations. */
  newa = (taba->o == IR_TNEW || taba->o == IR_TDUP);
  newb = (tabb->o == IR_TNEW || tabb->o == IR_TDUP);
  if (newa && newb)
    return ALIAS_NO;  /* Two different allocations never alias. */
  if (newb) {  /* At least one allocation? */
    IRIns *tmp = taba; taba = tabb; tabb = tmp;
  } else if (!newa) {
    return ALIAS_MAY;  /* Anything else: we just don't know. */
  }
  return aa_escape(J, taba, tabb);
}

/* Alias analysis for array and hash access using key-based disambiguation. */
static AliasRet aa_ahref(jit_State *J, IRIns *refa, IRIns *refb)
{
  IRRef ka = refa->op2;
  IRRef kb = refb->op2;
  IRIns *keya, *keyb;
  IRRef ta, tb;
  if (refa == refb)
    return ALIAS_MUST;  /* Shortcut for same refs. */
  keya = IR(ka);
  if (keya->o == IR_KSLOT) { ka = keya->op1; keya = IR(ka); }
  keyb = IR(kb);
  if (keyb->o == IR_KSLOT) { kb = keyb->op1; keyb = IR(kb); }
  ta = (refa->o==IR_HREFK || refa->o==IR_AREF) ? IR(refa->op1)->op1 : refa->op1;
  tb = (refb->o==IR_HREFK || refb->o==IR_AREF) ? IR(refb->op1)->op1 : refb->op1;
  if (ka == kb) {
    /* Same key. Check for same table with different ref (NEWREF vs. HREF). */
    if (ta == tb)
      return ALIAS_MUST;  /* Same key, same table. */
    else
      return aa_table(J, ta, tb);  /* Same key, possibly different table. */
  }
  if (irref_isk(ka) && irref_isk(kb))
    return ALIAS_NO;  /* Different constant keys. */
  if (refa->o == IR_AREF) {
    /* Disambiguate array references based on index arithmetic. */
    int32_t ofsa = 0, ofsb = 0;
    IRRef basea = ka, baseb = kb;
    lua_assert(refb->o == IR_AREF);
    /* Gather base and offset from t[base] or t[base+-ofs]. */
    if (keya->o == IR_ADD && irref_isk(keya->op2)) {
      basea = keya->op1;
      ofsa = IR(keya->op2)->i;
      if (basea == kb && ofsa != 0)
        return ALIAS_NO;  /* t[base+-ofs] vs. t[base]. */
    }
    if (keyb->o == IR_ADD && irref_isk(keyb->op2)) {
      baseb = keyb->op1;
      ofsb = IR(keyb->op2)->i;
      if (ka == baseb && ofsb != 0)
        return ALIAS_NO;  /* t[base] vs. t[base+-ofs]. */
    }
    if (basea == baseb && ofsa != ofsb)
      return ALIAS_NO;  /* t[base+-o1] vs. t[base+-o2] and o1 != o2. */
  } else {
    /* Disambiguate hash references based on the type of their keys. */
    lua_assert((refa->o==IR_HREF || refa->o==IR_HREFK || refa->o==IR_NEWREF) &&
               (refb->o==IR_HREF || refb->o==IR_HREFK || refb->o==IR_NEWREF));
    if (!irt_sametype(keya->t, keyb->t))
      return ALIAS_NO;  /* Different key types. */
```

```
  }
  if (ta == tb)
    return ALIAS_MAY;  /* Same table, cannot disambiguate keys. */
  else
    return aa_table(J, ta, tb);  /* Try to disambiguate tables. */
}

/* Array and hash load forwarding. */
static TRef fwd_ahload(jit_State *J, IRRef xref)
{
  IRIns *xr = IR(xref);
  IRRef lim = xref;  /* Search limit. */
  IRRef ref;

  /* Search for conflicting stores. */
  ref = J->chain[fins->o+IRDELTA_L2S];
  while (ref > xref) {
    IRIns *store = IR(ref);
    switch (aa_ahref(J, xr, IR(store->op1))) {
    case ALIAS_NO:   break;  /* Continue searching. */
    case ALIAS_MAY:  lim = ref; goto cselim;  /* Limit search for load. */
    case ALIAS_MUST: return store->op2;  /* Store forwarding. */
    }
    ref = store->prev;
  }

  /* No conflicting store (yet): const-fold loads from allocations. */
  {
    IRIns *ir = (xr->o == IR_HREFK || xr->o == IR_AREF) ? IR(xr->op1) : xr;
    IRRef tab = ir->op1;
    ir = IR(tab);
    if (ir->o == IR_TNEW || (ir->o == IR_TDUP && irref_isk(xr->op2))) {
      /* A NEWREF with a number key may end up pointing to the array part.
      ** But it's referenced from HSTORE and not found in the ASTORE chain.
      ** For now simply consider this a conflict without forwarding anything.
      */
      if (xr->o == IR_AREF) {
        IRRef ref2 = J->chain[IR_NEWREF];
        while (ref2 > tab) {
          IRIns *newref = IR(ref2);
          if (irt_isnum(IR(newref->op2)->t))
            goto cselim;
          ref2 = newref->prev;
        }
      }
      /* NEWREF inhibits CSE for HREF, and dependent FLOADs from HREFK/AREF.
      ** But the above search for conflicting stores was limited by xref.
      ** So continue searching, limited by the TNEW/TDUP. Store forwarding
      ** is ok, too. A conflict does NOT limit the search for a matching load.
      */
      while (ref > tab) {
        IRIns *store = IR(ref);
        switch (aa_ahref(J, xr, IR(store->op1))) {
        case ALIAS_NO:   break;  /* Continue searching. */
        case ALIAS_MAY:  goto cselim;  /* Conflicting store. */
        case ALIAS_MUST: return store->op2;  /* Store forwarding. */
        }
        ref = store->prev;
      }
      lua_assert(ir->o != IR_TNEW || irt_isnil(fins->t));
      if (irt_ispri(fins->t)) {
        return TREF_PRI(irt_type(fins->t));
      } else if (irt_isnum(fins->t) || (LJ_DUALNUM && irt_isint(fins->t)) ||
                 irt_isstr(fins->t)) {
        TValue keyv;
        cTValue *tv;
        IRIns *key = IR(xr->op2);
        if (key->o == IR_KSLOT) key = IR(key->op1);
        lj_ir_kvalue(J->L, &keyv, key);
        tv = lj_tab_get(J->L, ir_ktab(IR(ir->op1)), &keyv);
        lua_assert(itype2irt(tv) == irt_type(fins->t));
        if (irt_isnum(fins->t))
          return lj_ir_knum_u64(J, tv->u64);
        else if (LJ_DUALNUM && irt_isint(fins->t))
          return lj_ir_kint(J, intV(tv));
        else
```

```
          return lj_ir_kstr(J, strV(tv));
        }
        /* Othwerwise: don't intern as a constant. */
      }
    }

cselim:
    /* Try to find a matching load. Below the conflicting store, if any. */
    ref = J->chain[fins->o];
    while (ref > lim) {
      IRIns *load = IR(ref);
      if (load->op1 == xref)
        return ref;  /* Load forwarding. */
      ref = load->prev;
    }
    return 0;  /* Conflict or no match. */
}

/* Reassociate ALOAD across PHIs to handle t[i-1] forwarding case. */
static TRef fwd_aload_reassoc(jit_State *J)
{
    IRIns *irx = IR(fins->op1);
    IRIns *key = IR(irx->op2);
    if (key->o == IR_ADD && irref_isk(key->op2)) {
      IRIns *add2 = IR(key->op1);
      if (add2->o == IR_ADD && irref_isk(add2->op2) &&
          IR(key->op2)->i == -IR(add2->op2)->i) {
        IRRef ref = J->chain[IR_AREF];
        IRRef lim = add2->op1;
        if (irx->op1 > lim) lim = irx->op1;
        while (ref > lim) {
          IRIns *ir = IR(ref);
          if (ir->op1 == irx->op1 && ir->op2 == add2->op1)
            return fwd_ahload(J, ref);
          ref = ir->prev;
        }
      }
    }
    return 0;
}

/* ALOAD forwarding. */
TRef LJ_FASTCALL lj_opt_fwd_aload(jit_State *J)
{
    IRRef ref;
    if ((ref = fwd_ahload(J, fins->op1)) ||
        (ref = fwd_aload_reassoc(J)))
      return ref;
    return EMITFOLD;
}

/* HLOAD forwarding. */
TRef LJ_FASTCALL lj_opt_fwd_hload(jit_State *J)
{
    IRRef ref = fwd_ahload(J, fins->op1);
    if (ref)
      return ref;
    return EMITFOLD;
}

/* HREFK forwarding. */
TRef LJ_FASTCALL lj_opt_fwd_hrefk(jit_State *J)
{
    IRRef tab = fleft->op1;
    IRRef ref = J->chain[IR_NEWREF];
    while (ref > tab) {
      IRIns *newref = IR(ref);
      if (tab == newref->op1) {
        if (fright->op1 == newref->op2)
          return ref;  /* Forward from NEWREF. */
        else
          goto docse;
      } else if (aa_table(J, tab, newref->op1) != ALIAS_NO) {
        goto docse;
      }
      ref = newref->prev;
```

```
  }
  /* No conflicting NEWREF: key location unchanged for HREFK of TDUP. */
  if (IR(tab)->o == IR_TDUP)
    fins->t.irt &= ~IRT_GUARD;  /* Drop HREFK guard. */
docse:
  return CSEFOLD;
}

/* Check whether HREF of TNEW/TDUP can be folded to niltv. */
int LJ_FASTCALL lj_opt_fwd_href_nokey(jit_State *J)
{
  IRRef lim = fins->op1;  /* Search limit. */
  IRRef ref;

  /* The key for an ASTORE may end up in the hash part after a NEWREF. */
  if (irt_isnum(fright->t) && J->chain[IR_NEWREF] > lim) {
    ref = J->chain[IR_ASTORE];
    while (ref > lim) {
      if (ref < J->chain[IR_NEWREF])
        return 0;  /* Conflict. */
      ref = IR(ref)->prev;
    }
  }

  /* Search for conflicting stores. */
  ref = J->chain[IR_HSTORE];
  while (ref > lim) {
    IRIns *store = IR(ref);
    if (aa_ahref(J, fins, IR(store->op1)) != ALIAS_NO)
      return 0;  /* Conflict. */
    ref = store->prev;
  }

  return 1;  /* No conflict. Can fold to niltv. */
}

/* Check whether there's no aliasing table.clear. */
static int fwd_aa_tab_clear(jit_State *J, IRRef lim, IRRef ta)
{
  IRRef ref = J->chain[IR_CALLS];
  while (ref > lim) {
    IRIns *calls = IR(ref);
    if (calls->op2 == IRCALL_lj_tab_clear &&
        (ta == calls->op1 || aa_table(J, ta, calls->op1) != ALIAS_NO))
      return 0;  /* Conflict. */
    ref = calls->prev;
  }
  return 1;  /* No conflict. Can safely FOLD/CSE. */
}

/* Check whether there's no aliasing NEWREF/table.clear for the left operand. */
int LJ_FASTCALL lj_opt_fwd_tptr(jit_State *J, IRRef lim)
{
  IRRef ta = fins->op1;
  IRRef ref = J->chain[IR_NEWREF];
  while (ref > lim) {
    IRIns *newref = IR(ref);
    if (ta == newref->op1 || aa_table(J, ta, newref->op1) != ALIAS_NO)
      return 0;  /* Conflict. */
    ref = newref->prev;
  }
  return fwd_aa_tab_clear(J, lim, ta);
}

/* ASTORE/HSTORE elimination. */
TRef LJ_FASTCALL lj_opt_dse_ahstore(jit_State *J)
{
  IRRef xref = fins->op1;  /* xREF reference. */
  IRRef val = fins->op2;  /* Stored value reference. */
  IRIns *xr = IR(xref);
  IRRef1 *refp = &J->chain[fins->o];
  IRRef ref = *refp;
  while (ref > xref) {  /* Search for redundant or conflicting stores. */
    IRIns *store = IR(ref);
    switch (aa_ahref(J, xr, IR(store->op1))) {
    case ALIAS_NO:
```

```
      break;  /* Continue searching. */
    case ALIAS_MAY:        /* Store to MAYBE the same location. */
      if (store->op2 != val)  /* Conflict if the value is different. */
        goto doemit;
      break;  /* Otherwise continue searching. */
    case ALIAS_MUST:        /* Store to the same location. */
      if (store->op2 == val)  /* Same value: drop the new store. */
        return DROPFOLD;
      /* Different value: try to eliminate the redundant store. */
      if (ref > J->chain[IR_LOOP]) {  /* Quick check to avoid crossing LOOP. */
        IRIns *ir;
        /* Check for any intervening guards (includes conflicting loads). */
        for (ir = IR(J->cur.nins-1); ir > store; ir--)
          if (irt_isguard(ir->t) || ir->o == IR_CALLL)
            goto doemit;  /* No elimination possible. */
        /* Remove redundant store from chain and replace with NOP. */
        *refp = store->prev;
        store->o = IR_NOP;
        store->t.irt = IRT_NIL;
        store->op1 = store->op2 = 0;
        store->prev = 0;
        /* Now emit the new store instead. */
      }
      goto doemit;
    }
    ref = *(refp = &store->prev);
  }
doemit:
  return EMITFOLD;  /* Otherwise we have a conflict or simply no match. */
}

/* -- ULOAD forwarding ----------------------------------------------------- */

/* The current alias analysis for upvalues is very simplistic. It only
** disambiguates between the unique upvalues of the same function.
** This is good enough for now, since most upvalues are read-only.
**
** A more precise analysis would be feasible with the help of the parser:
** generate a unique key for every upvalue, even across all prototypes.
** Lacking a realistic use-case, it's unclear whether this is beneficial.
*/
static AliasRet aa_uref(IRIns *refa, IRIns *refb)
{
  if (refa->o != refb->o)
    return ALIAS_NO;  /* Different UREFx type. */
  if (refa->op1 == refb->op1) {  /* Same function. */
    if (refa->op2 == refb->op2)
      return ALIAS_MUST;  /* Same function, same upvalue idx. */
    else
      return ALIAS_NO;  /* Same function, different upvalue idx. */
  } else {  /* Different functions, check disambiguation hash values. */
    if (((refa->op2 ^ refb->op2) & 0xff))
      return ALIAS_NO;  /* Upvalues with different hash values cannot alias. */
    else
      return ALIAS_MAY;  /* No conclusion can be drawn for same hash value. */
  }
}

/* ULOAD forwarding. */
TRef LJ_FASTCALL lj_opt_fwd_uload(jit_State *J)
{
  IRRef uref = fins->op1;
  IRRef lim = REF_BASE;  /* Search limit. */
  IRIns *xr = IR(uref);
  IRRef ref;

  /* Search for conflicting stores. */
  ref = J->chain[IR_USTORE];
  while (ref > lim) {
    IRIns *store = IR(ref);
    switch (aa_uref(xr, IR(store->op1))) {
    case ALIAS_NO:   break;  /* Continue searching. */
    case ALIAS_MAY:  lim = ref; goto cselim;  /* Limit search for load. */
    case ALIAS_MUST: return store->op2;  /* Store forwarding. */
    }
    ref = store->prev;
```

```
  }

cselim:
  /* Try to find a matching load. Below the conflicting store, if any. */

  ref = J->chain[IR_ULOAD];
  while (ref > lim) {
    IRIns *ir = IR(ref);
    if (ir->op1 == uref ||
        (IR(ir->op1)->op12 == IR(uref)->op12 && IR(ir->op1)->o == IR(uref)->o))
      return ref;  /* Match for identical or equal UREFx (non-CSEable UREFO). */
    ref = ir->prev;
  }
  return lj_ir_emit(J);
}

/* USTORE elimination. */
TRef LJ_FASTCALL lj_opt_dse_ustore(jit_State *J)
{
  IRRef xref = fins->op1;  /* xREF reference. */
  IRRef val = fins->op2;  /* Stored value reference. */
  IRIns *xr = IR(xref);
  IRRef1 *refp = &J->chain[IR_USTORE];
  IRRef ref = *refp;
  while (ref > xref) {  /* Search for redundant or conflicting stores. */
    IRIns *store = IR(ref);
    switch (aa_uref(xr, IR(store->op1))) {
    case ALIAS_NO:
      break;  /* Continue searching. */
    case ALIAS_MAY:        /* Store to MAYBE the same location. */
      if (store->op2 != val)  /* Conflict if the value is different. */
        goto doemit;
      break;  /* Otherwise continue searching. */
    case ALIAS_MUST:        /* Store to the same location. */
      if (store->op2 == val)  /* Same value: drop the new store. */
        return DROPFOLD;
      /* Different value: try to eliminate the redundant store. */
      if (ref > J->chain[IR_LOOP]) {  /* Quick check to avoid crossing LOOP. */
        IRIns *ir;
        /* Check for any intervening guards (includes conflicting loads). */
        for (ir = IR(J->cur.nins-1); ir > store; ir--)
          if (irt_isguard(ir->t))
            goto doemit;  /* No elimination possible. */
        /* Remove redundant store from chain and replace with NOP. */
        *refp = store->prev;
        store->o = IR_NOP;
        store->t.irt = IRT_NIL;
        store->op1 = store->op2 = 0;
        store->prev = 0;
        if (ref+1 < J->cur.nins &&
            store[1].o == IR_OBAR && store[1].op1 == xref) {
          IRRef1 *bp = &J->chain[IR_OBAR];
          IRIns *obar;
          for (obar = IR(*bp); *bp > ref+1; obar = IR(*bp))
            bp = &obar->prev;
          /* Remove OBAR, too. */
          *bp = obar->prev;
          obar->o = IR_NOP;
          obar->t.irt = IRT_NIL;
          obar->op1 = obar->op2 = 0;
          obar->prev = 0;
        }
        /* Now emit the new store instead. */
      }
      goto doemit;
    }
    ref = *(refp = &store->prev);
  }
doemit:
  return EMITFOLD;  /* Otherwise we have a conflict or simply no match. */
}

/* -- FLOAD forwarding and FSTORE elimination ---------------------------- */

/* Alias analysis for field access.
** Field loads are cheap and field stores are rare.
```

```
** Simple disambiguation based on field types is good enough.
*/
static AliasRet aa_fref(jit_State *J, IRIns *refa, IRIns *refb)
{
  if (refa->op2 != refb->op2)
    return ALIAS_NO;  /* Different fields. */
  if (refa->op1 == refb->op1)
    return ALIAS_MUST;  /* Same field, same object. */
  else if (refa->op2 >= IRFL_TAB_META && refa->op2 <= IRFL_TAB_NOMM)
    return aa_table(J, refa->op1, refb->op1);  /* Disambiguate tables. */
  else
    return ALIAS_MAY;  /* Same field, possibly different object. */
}

/* Only the loads for mutable fields end up here (see FOLD). */
TRef LJ_FASTCALL lj_opt_fwd_fload(jit_State *J)
{
  IRRef oref = fins->op1;  /* Object reference. */
  IRRef fid = fins->op2;  /* Field ID. */
  IRRef lim = oref;  /* Search limit. */
  IRRef ref;

  /* Search for conflicting stores. */
  ref = J->chain[IR_FSTORE];
  while (ref > oref) {
    IRIns *store = IR(ref);
    switch (aa_fref(J, fins, IR(store->op1))) {
    case ALIAS_NO:   break;  /* Continue searching. */
    case ALIAS_MAY:  lim = ref; goto cselim;  /* Limit search for load. */
    case ALIAS_MUST: return store->op2;  /* Store forwarding. */
    }
    ref = store->prev;
  }

  /* No conflicting store: const-fold field loads from allocations. */
  if (fid == IRFL_TAB_META) {
    IRIns *ir = IR(oref);
    if (ir->o == IR_TNEW || ir->o == IR_TDUP)
      return lj_ir_knull(J, IRT_TAB);
  }

cselim:
  /* Try to find a matching load. Below the conflicting store, if any. */
  return lj_opt_cselim(J, lim);
}

/* FSTORE elimination. */
TRef LJ_FASTCALL lj_opt_dse_fstore(jit_State *J)
{
  IRRef fref = fins->op1;  /* FREF reference. */
  IRRef val = fins->op2;  /* Stored value reference. */
  IRIns *xr = IR(fref);
  IRRef1 *refp = &J->chain[IR_FSTORE];
  IRRef ref = *refp;
  while (ref > fref) {  /* Search for redundant or conflicting stores. */
    IRIns *store = IR(ref);
    switch (aa_fref(J, xr, IR(store->op1))) {
    case ALIAS_NO:
      break;  /* Continue searching. */
    case ALIAS_MAY:
      if (store->op2 != val)  /* Conflict if the value is different. */
        goto doemit;
      break;  /* Otherwise continue searching. */
    case ALIAS_MUST:
      if (store->op2 == val)  /* Same value: drop the new store. */
        return DROPFOLD;
      /* Different value: try to eliminate the redundant store. */
      if (ref > J->chain[IR_LOOP]) {  /* Quick check to avoid crossing LOOP. */
        IRIns *ir;
        /* Check for any intervening guards or conflicting loads. */
        for (ir = IR(J->cur.nins-1); ir > store; ir--)
          if (irt_isguard(ir->t) || (ir->o == IR_FLOAD && ir->op2 == xr->op2))
            goto doemit;  /* No elimination possible. */
        /* Remove redundant store from chain and replace with NOP. */
        *refp = store->prev;
        store->o = IR_NOP;
```

```
        store->t.irt = IRT_NIL;
        store->op1 = store->op2 = 0;
        store->prev = 0;
        /* Now emit the new store instead. */
      }
      goto doemit;
    }
    ref = *(refp = &store->prev);
  }
doemit:
  return EMITFOLD;  /* Otherwise we have a conflict or simply no match. */
}

/* -- XLOAD forwarding and XSTORE elimination ---------------------------- */

/* Find cdata allocation for a reference (if any). */
static IRIns *aa_findcnew(jit_State *J, IRIns *ir)
{
  while (ir->o == IR_ADD) {
    if (!irref_isk(ir->op1)) {
      IRIns *ir1 = aa_findcnew(J, IR(ir->op1));  /* Left-recursion. */
      if (ir1) return ir1;
    }
    if (irref_isk(ir->op2)) return NULL;
    ir = IR(ir->op2);  /* Flatten right-recursion. */
  }
  return ir->o == IR_CNEW ? ir : NULL;
}

/* Alias analysis for two cdata allocations. */
static AliasRet aa_cnew(jit_State *J, IRIns *refa, IRIns *refb)
{
  IRIns *cnewa = aa_findcnew(J, refa);
  IRIns *cnewb = aa_findcnew(J, refb);
  if (cnewa == cnewb)
    return ALIAS_MAY;  /* Same allocation or neither is an allocation. */
  if (cnewa && cnewb)
    return ALIAS_NO;  /* Two different allocations never alias. */
  if (cnewb) { cnewa = cnewb; refb = refa; }
  return aa_escape(J, cnewa, refb);
}

/* Alias analysis for XLOAD/XSTORE. */
static AliasRet aa_xref(jit_State *J, IRIns *refa, IRIns *xa, IRIns *xb)
{
  ptrdiff_t ofsa = 0, ofsb = 0;
  IRIns *refb = IR(xb->op1);
  IRIns *basea = refa, *baseb = refb;
  if (refa == refb && irt_sametype(xa->t, xb->t))
    return ALIAS_MUST;  /* Shortcut for same refs with identical type. */
  /* Offset-based disambiguation. */
  if (refa->o == IR_ADD && irref_isk(refa->op2)) {
    IRIns *irk = IR(refa->op2);
    basea = IR(refa->op1);
    ofsa = (LJ_64 && irk->o == IR_KINT64) ? (ptrdiff_t)ir_k64(irk)->u64 :
                                            (ptrdiff_t)irk->i;
  }
  if (refb->o == IR_ADD && irref_isk(refb->op2)) {
    IRIns *irk = IR(refb->op2);
    baseb = IR(refb->op1);
    ofsb = (LJ_64 && irk->o == IR_KINT64) ? (ptrdiff_t)ir_k64(irk)->u64 :
                                            (ptrdiff_t)irk->i;
  }
  /* Treat constified pointers like base vs. base+offset. */
  if (basea->o == IR_KPTR && baseb->o == IR_KPTR) {
    ofsb += (char *)ir_kptr(baseb) - (char *)ir_kptr(basea);
    baseb = basea;
  }
  /* This implements (very) strict aliasing rules.
  ** Different types do NOT alias, except for differences in signedness.
  ** Type punning through unions is allowed (but forces a reload).
  */
  if (basea == baseb) {
    ptrdiff_t sza = irt_size(xa->t), szb = irt_size(xb->t);
    if (ofsa == ofsb) {
      if (sza == szb && irt_isfp(xa->t) == irt_isfp(xb->t))
```

```
656         return ALIAS_MUST;  /* Same-sized, same-kind. May need to convert. */
657       } else if (ofsa + sza <= ofsb || ofsb + szb <= ofsa) {
658         return ALIAS_NO;  /* Non-overlapping base+-o1 vs. base+-o2. */
659       }
660       /* NYI: extract, extend or reinterpret bits (int <-> fp). */
661       return ALIAS_MAY;  /* Overlapping or type punning: force reload. */
662     }
663     if (!irt_sametype(xa->t, xb->t) &&
664         !(irt_typerange(xa->t, IRT_I8, IRT_U64) &&
665           ((xa->t.irt - IRT_I8) ^ (xb->t.irt - IRT_I8)) == 1))
666       return ALIAS_NO;
667     /* NYI: structural disambiguation. */
668     return aa_cnew(J, basea, baseb);  /* Try to disambiguate allocations. */
669   }
670
671   /* Return CSEd reference or 0. Caveat: swaps lower ref to the right! */
672   static IRRef reassoc_trycse(jit_State *J, IROp op, IRRef op1, IRRef op2)
673   {
674     IRRef ref = J->chain[op];
675     IRRef lim = op1;
676     if (op2 > lim) { lim = op2; op2 = op1; op1 = lim; }
677     while (ref > lim) {
678       IRIns *ir = IR(ref);
679       if (ir->op1 == op1 && ir->op2 == op2)
680         return ref;
681       ref = ir->prev;
682     }
683     return 0;
684   }
685
686   /* Reassociate index references. */
687   static IRRef reassoc_xref(jit_State *J, IRIns *ir)
688   {
689     ptrdiff_t ofs = 0;
690     if (ir->o == IR_ADD && irref_isk(ir->op2)) {  /* Get constant offset. */
691       IRIns *irk = IR(ir->op2);
692       ofs = (LJ_64 && irk->o == IR_KINT64) ? (ptrdiff_t)ir_k64(irk)->u64 :
693                                              (ptrdiff_t)irk->i;
694       ir = IR(ir->op1);
695     }
696     if (ir->o == IR_ADD) {  /* Add of base + index. */
697       /* Index ref > base ref for loop-carried dependences. Only check op1. */
698       IRIns *ir2, *ir1 = IR(ir->op1);
699       int32_t shift = 0;
700       IRRef idxref;
701       /* Determine index shifts. Don't bother with IR_MUL here. */
702       if (ir1->o == IR_BSHL && irref_isk(ir1->op2))
703         shift = IR(ir1->op2)->i;
704       else if (ir1->o == IR_ADD && ir1->op1 == ir1->op2)
705         shift = 1;
706       else
707         ir1 = ir;
708       ir2 = IR(ir1->op1);
709       /* A non-reassociated add. Must be a loop-carried dependence. */
710       if (ir2->o == IR_ADD && irt_isint(ir2->t) && irref_isk(ir2->op2))
711         ofs += (ptrdiff_t)IR(ir2->op2)->i << shift;
712       else
713         return 0;
714       idxref = ir2->op1;
715       /* Try to CSE the reassociated chain. Give up if not found. */
716       if (ir1 != ir &&
717           !(idxref = reassoc_trycse(J, ir1->o, idxref,
718                                     ir1->o == IR_BSHL ? ir1->op2 : idxref)))
719         return 0;
720       if (!(idxref = reassoc_trycse(J, IR_ADD, idxref, ir->op2)))
721         return 0;
722       if (ofs != 0) {
723         IRRef refk = tref_ref(lj_ir_kintp(J, ofs));
724         if (!(idxref = reassoc_trycse(J, IR_ADD, idxref, refk)))
725           return 0;
726       }
727       return idxref;  /* Success, found a reassociated index reference. Phew. */
728     }
729     return 0;  /* Failure. */
730   }
731
```

```
/* XLOAD forwarding. */
TRef LJ_FASTCALL lj_opt_fwd_xload(jit_State *J)
{
  IRRef xref = fins->op1;
  IRIns *xr = IR(xref);
  IRRef lim = xref;  /* Search limit. */
  IRRef ref;

  if ((fins->op2 & IRXLOAD_READONLY))
    goto cselim;
  if ((fins->op2 & IRXLOAD_VOLATILE))
    goto doemit;

  /* Search for conflicting stores. */
  ref = J->chain[IR_XSTORE];
retry:
  if (J->chain[IR_CALLXS] > lim) lim = J->chain[IR_CALLXS];
  if (J->chain[IR_XBAR] > lim) lim = J->chain[IR_XBAR];
  while (ref > lim) {
    IRIns *store = IR(ref);
    switch (aa_xref(J, xr, fins, store)) {
    case ALIAS_NO:   break;  /* Continue searching. */
    case ALIAS_MAY:  lim = ref; goto cselim;  /* Limit search for load. */
    case ALIAS_MUST:
      /* Emit conversion if the loaded type doesn't match the forwarded type. */
      if (!irt_sametype(fins->t, IR(store->op2)->t)) {
        IRType dt = irt_type(fins->t), st = irt_type(IR(store->op2)->t);
        if (dt == IRT_I8 || dt == IRT_I16) {  /* Trunc + sign-extend. */
          st = dt | IRCONV_SEXT;
          dt = IRT_INT;
        } else if (dt == IRT_U8 || dt == IRT_U16) {  /* Trunc + zero-extend. */
          st = dt;
          dt = IRT_INT;
        }
        fins->ot = IRT(IR_CONV, dt);
        fins->op1 = store->op2;
        fins->op2 = (dt<<5)|st;
        return RETRYFOLD;
      }
      return store->op2;  /* Store forwarding. */
    }
    ref = store->prev;
  }

cselim:
  /* Try to find a matching load. Below the conflicting store, if any. */
  ref = J->chain[IR_XLOAD];
  while (ref > lim) {
    /* CSE for XLOAD depends on the type, but not on the IRXLOAD_* flags. */
    if (IR(ref)->op1 == xref && irt_sametype(IR(ref)->t, fins->t))
      return ref;
    ref = IR(ref)->prev;
  }

  /* Reassociate XLOAD across PHIs to handle a[i-1] forwarding case. */
  if (!(fins->op2 & IRXLOAD_READONLY) && J->chain[IR_LOOP] &&
      xref == fins->op1 && (xref = reassoc_xref(J, xr)) != 0) {
    ref = J->chain[IR_XSTORE];
    while (ref > lim)  /* Skip stores that have already been checked. */
      ref = IR(ref)->prev;
    lim = xref;
    xr = IR(xref);
    goto retry;  /* Retry with the reassociated reference. */
  }
doemit:
  return EMITFOLD;
}

/* XSTORE elimination. */
TRef LJ_FASTCALL lj_opt_dse_xstore(jit_State *J)
{
  IRRef xref = fins->op1;
  IRIns *xr = IR(xref);
  IRRef lim = xref;  /* Search limit. */
  IRRef val = fins->op2;  /* Stored value reference. */
  IRRef1 *refp = &J->chain[IR_XSTORE];
```

```
  IRRef ref = *refp;
  if (J->chain[IR_CALLXS] > lim) lim = J->chain[IR_CALLXS];
  if (J->chain[IR_XBAR] > lim) lim = J->chain[IR_XBAR];
  if (J->chain[IR_XSNEW] > lim) lim = J->chain[IR_XSNEW];
  while (ref > lim) {  /* Search for redundant or conflicting stores. */
    IRIns *store = IR(ref);
    switch (aa_xref(J, xr, fins, store)) {
    case ALIAS_NO:
      break;  /* Continue searching. */
    case ALIAS_MAY:
      if (store->op2 != val)  /* Conflict if the value is different. */
        goto doemit;
      break;  /* Otherwise continue searching. */
    case ALIAS_MUST:
      if (store->op2 == val)  /* Same value: drop the new store. */
        return DROPFOLD;
      /* Different value: try to eliminate the redundant store. */
      if (ref > J->chain[IR_LOOP]) {  /* Quick check to avoid crossing LOOP. */
        IRIns *ir;
        /* Check for any intervening guards or any XLOADs (no AA performed). */
        for (ir = IR(J->cur.nins-1); ir > store; ir--)
          if (irt_isguard(ir->t) || ir->o == IR_XLOAD)
            goto doemit;  /* No elimination possible. */
        /* Remove redundant store from chain and replace with NOP. */
        *refp = store->prev;
        store->o = IR_NOP;
        store->t.irt = IRT_NIL;
        store->op1 = store->op2 = 0;
        store->prev = 0;
        /* Now emit the new store instead. */
      }
      goto doemit;
    }
    ref = *(refp = &store->prev);
  }
doemit:
  return EMITFOLD;  /* Otherwise we have a conflict or simply no match. */
}

/* -- Forwarding of lj_tab_len ------------------------------------------- */

/* This is rather simplistic right now, but better than nothing. */
TRef LJ_FASTCALL lj_opt_fwd_tab_len(jit_State *J)
{
  IRRef tab = fins->op1;  /* Table reference. */
  IRRef lim = tab;  /* Search limit. */
  IRRef ref;

  /* Any ASTORE is a conflict and limits the search. */
  if (J->chain[IR_ASTORE] > lim) lim = J->chain[IR_ASTORE];

  /* Search for conflicting HSTORE with numeric key. */
  ref = J->chain[IR_HSTORE];
  while (ref > lim) {
    IRIns *store = IR(ref);
    IRIns *href = IR(store->op1);
    IRIns *key = IR(href->op2);
    if (irt_isnum(key->o == IR_KSLOT ? IR(key->op1)->t : key->t)) {
      lim = ref;  /* Conflicting store found, limits search for TLEN. */
      break;
    }
    ref = store->prev;
  }

  /* Search for aliasing table.clear. */
  if (!fwd_aa_tab_clear(J, lim, tab))
    return lj_ir_emit(J);

  /* Try to find a matching load. Below the conflicting store, if any. */
  return lj_opt_cselim(J, lim);
}

/* -- ASTORE/HSTORE previous type analysis ------------------------------- */

/* Check whether the previous value for a table store is non-nil.
** This can be derived either from a previous store or from a previous
```

```
** load (because all loads from tables perform a type check).
**
** The result of the analysis can be used to avoid the metatable check
** and the guard against HREF returning niltv. Both of these are cheap,
** so let's not spend too much effort on the analysis.
**
** A result of 1 is exact: previous value CANNOT be nil.
** A result of 0 is inexact: previous value MAY be nil.
*/
int lj_opt_fwd_wasnonnil(jit_State *J, IROpT loadop, IRRef xref)
{
  /* First check stores. */
  IRRef ref = J->chain[loadop+IRDELTA_L2S];
  while (ref > xref) {
    IRIns *store = IR(ref);
    if (store->op1 == xref) {  /* Same xREF. */
      /* A nil store MAY alias, but a non-nil store MUST alias. */
      return !irt_isnil(store->t);
    } else if (irt_isnil(store->t)) {  /* Must check any nil store. */
      IRRef skref = IR(store->op1)->op2;
      IRRef xkref = IR(xref)->op2;
      /* Same key type MAY alias. Need ALOAD check due to multiple int types. */
      if (loadop == IR_ALOAD || irt_sametype(IR(skref)->t, IR(xkref)->t)) {
        if (skref == xkref || !irref_isk(skref) || !irref_isk(xkref))
          return 0;  /* A nil store with same const key or var key MAY alias. */
        /* Different const keys CANNOT alias. */
      }  /* Different key types CANNOT alias. */
    }  /* Other non-nil stores MAY alias. */
    ref = store->prev;
  }

  /* Check loads since nothing could be derived from stores. */
  ref = J->chain[loadop];
  while (ref > xref) {
    IRIns *load = IR(ref);
    if (load->op1 == xref) {  /* Same xREF. */
      /* A nil load MAY alias, but a non-nil load MUST alias. */
      return !irt_isnil(load->t);
    }  /* Other non-nil loads MAY alias. */
    ref = load->prev;
  }
  return 0;  /* Nothing derived at all, previous value MAY be nil. */
}

/* ------------------------------------------------------------------------ */

#undef IR
#undef fins
#undef fleft
#undef fright

#endif
```

One Level Up Top Level

src/lj_opt_dce.c - luajit-2.0-src

Functions defined

- dce_marksnap
- dce_propagate
- lj_opt_dce

Macros defined

- IR
- IR
- LUA_CORE
- lj_opt_dce_c

Source code

```
1  /*
2  ** DCE: Dead Code Elimination. Pre-LOOP only -- ASM already performs DCE.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lj_opt_dce_c
7  #define LUA_CORE
8
9  #include "lj_obj.h"
10
11 #if LJ_HASJIT
12
13 #include "lj_ir.h"
14 #include "lj_jit.h"
15 #include "lj_iropt.h"
16
17 /* Some local macros to save typing. Undef'd at the end. */
18 #define IR(ref)                (&J->cur.ir[(ref)])
19
20 /* Scan through all snapshots and mark all referenced instructions. */
21 static void dce_marksnap(jit_State *J)
22 {
23   SnapNo i, nsnap = J->cur.nsnap;
24   for (i = 0; i < nsnap; i++) {
25     SnapShot *snap = &J->cur.snap[i];
26     SnapEntry *map = &J->cur.snapmap[snap->mapofs];
27     MSize n, nent = snap->nent;
28     for (n = 0; n < nent; n++) {
29       IRRef ref = snap_ref(map[n]);
30       if (ref >= REF_FIRST)
31         irt_setmark(IR(ref)->t);
32     }
33   }
34 }
35
36 /* Backwards propagate marks. Replace unused instructions with NOPs. */
37 static void dce_propagate(jit_State *J)
38 {
39   IRRef1 *pchain[IR__MAX];
40   IRRef ins;
41   uint32_t i;
42   for (i = 0; i < IR__MAX; i++) pchain[i] = &J->chain[i];
43   for (ins = J->cur.nins-1; ins >= REF_FIRST; ins--) {
44     IRIns *ir = IR(ins);
45     if (irt_ismarked(ir->t)) {
```

```
      irt_clearmark(ir->t);
      pchain[ir->o] = &ir->prev;
    } else if (!ir_sideeff(ir)) {
      *pchain[ir->o] = ir->prev;  /* Reroute original instruction chain. */
      ir->t.irt = IRT_NIL;
      ir->o = IR_NOP;  /* Replace instruction with NOP. */
      ir->op1 = ir->op2 = 0;
      ir->prev = 0;
      continue;
    }
    if (ir->op1 >= REF_FIRST) irt_setmark(IR(ir->op1)->t);
    if (ir->op2 >= REF_FIRST) irt_setmark(IR(ir->op2)->t);
  }
}

/* Dead Code Elimination.
**
** First backpropagate marks for all used instructions. Then replace
** the unused ones with a NOP. Note that compressing the IR to eliminate
** the NOPs does not pay off.
*/
void lj_opt_dce(jit_State *J)
{
  if ((J->flags & JIT_F_OPT_DCE)) {
    dce_marksnap(J);
    dce_propagate(J);
    memset(J->bpropcache, 0, sizeof(J->bpropcache));  /* Invalidate cache. */
  }
}

#undef IR

#endif
```

One Level Up Top Level

src/lj_opt_sink.c - luajit-2.0-src

Functions defined

- lj_opt_sink
- sink_checkalloc
- sink_checkphi
- sink_mark_ins
- sink_mark_snap
- sink_phidep
- sink_remark_phi
- sink_sweep_ins

Macros defined

- IR
- IR
- LUA_CORE
- lj_opt_sink_c

Source code

```
1  /*
2  ** SINK: Allocation Sinking and Store Sinking.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lj_opt_sink_c
7  #define LUA_CORE
8
9  #include "lj_obj.h"
10
11 #if LJ_HASJIT
12
13 #include "lj_ir.h"
14 #include "lj_jit.h"
15 #include "lj_iropt.h"
16 #include "lj_target.h"
17
18 /* Some local macros to save typing. Undef'd at the end. */
19 #define IR(ref)                (&J->cur.ir[(ref)])
20
21 /* Check whether the store ref points to an eligible allocation. */
22 static IRIns *sink_checkalloc(jit_State *J, IRIns *irs)
23 {
24   IRIns *ir = IR(irs->op1);
25   if (!irref_isk(ir->op2))
26     return NULL;  /* Non-constant key. */
27   if (ir->o == IR_HREFK || ir->o == IR_AREF)
28     ir = IR(ir->op1);
29   else if (!(ir->o == IR_HREF || ir->o == IR_NEWREF ||
30              ir->o == IR_FREF || ir->o == IR_ADD))
31     return NULL;  /* Unhandled reference type (for XSTORE). */
32   ir = IR(ir->op1);
33   if (!(ir->o == IR_TNEW || ir->o == IR_TDUP || ir->o == IR_CNEW))
34     return NULL;  /* Not an allocation. */
```

```
  return ir;  /* Return allocation. */
}

/* Recursively check whether a value depends on a PHI. */
static int sink_phidep(jit_State *J, IRRef ref)
{
  IRIns *ir = IR(ref);
  if (irt_isphi(ir->t)) return 1;
  if (ir->op1 >= REF_FIRST && sink_phidep(J, ir->op1)) return 1;
  if (ir->op2 >= REF_FIRST && sink_phidep(J, ir->op2)) return 1;
  return 0;
}

/* Check whether a value is a sinkable PHI or loop-invariant. */
static int sink_checkphi(jit_State *J, IRIns *ira, IRRef ref)
{
  if (ref >= REF_FIRST) {
    IRIns *ir = IR(ref);
    if (irt_isphi(ir->t) || (ir->o == IR_CONV && ir->op2 == IRCONV_NUM_INT &&
                             irt_isphi(IR(ir->op1)->t))) {
      ira->prev++;
      return 1;  /* Sinkable PHI. */
    }
    /* Otherwise the value must be loop-invariant. */
    return ref < J->loopref && !sink_phidep(J, ref);
  }
  return 1;  /* Constant (non-PHI). */
}

/* Mark non-sinkable allocations using single-pass backward propagation.
**
** Roots for the marking process are:
** - Some PHIs or snapshots (see below).
** - Non-PHI, non-constant values stored to PHI allocations.
** - All guards.
** - Any remaining loads not eliminated by store-to-load forwarding.
** - Stores with non-constant keys.
** - All stored values.
*/
static void sink_mark_ins(jit_State *J)
{
  IRIns *ir, *irlast = IR(J->cur.nins-1);
  for (ir = irlast ; ; ir--) {
    switch (ir->o) {
    case IR_BASE:
      return;  /* Finished. */
    case IR_CALLL:  /* IRCALL_lj_tab_len */
    case IR_ALOAD: case IR_HLOAD: case IR_XLOAD: case IR_TBAR:
      irt_setmark(IR(ir->op1)->t);  /* Mark ref for remaining loads. */
      break;
    case IR_FLOAD:
      if (irt_ismarked(ir->t) || ir->op2 == IRFL_TAB_META)
        irt_setmark(IR(ir->op1)->t);  /* Mark table for remaining loads. */
      break;
    case IR_ASTORE: case IR_HSTORE: case IR_FSTORE: case IR_XSTORE: {
      IRIns *ira = sink_checkalloc(J, ir);
      if (!ira || (irt_isphi(ira->t) && !sink_checkphi(J, ira, ir->op2)))
        irt_setmark(IR(ir->op1)->t);  /* Mark ineligible ref. */
      irt_setmark(IR(ir->op2)->t);  /* Mark stored value. */
      break;
      }
#if LJ_HASFFI
    case IR_CNEWI:
      if (irt_isphi(ir->t) &&
          (!sink_checkphi(J, ir, ir->op2) ||
            (LJ_32 && ir+1 < irlast && (ir+1)->o == IR_HIOP &&
             !sink_checkphi(J, ir, (ir+1)->op2))))
        irt_setmark(ir->t);  /* Mark ineligible allocation. */
      /* fallthrough */
#endif
    case IR_USTORE:
      irt_setmark(IR(ir->op2)->t);  /* Mark stored value. */
      break;
#if LJ_HASFFI
    case IR_CALLXS:
#endif
```

```
    case IR_CALLS:
      irt_setmark(IR(ir->op1)->t);  /* Mark (potentially) stored values. */
      break;
    case IR_PHI: {
      IRIns *irl = IR(ir->op1), *irr = IR(ir->op2);
      irl->prev = irr->prev = 0;  /* Clear PHI value counts. */
      if (irl->o == irr->o &&
          (irl->o == IR_TNEW || irl->o == IR_TDUP ||
           (LJ_HASFFI && (irl->o == IR_CNEW || irl->o == IR_CNEWI))))
        break;
      irt_setmark(irl->t);
      irt_setmark(irr->t);
      break;
      }
    default:
      if (irt_ismarked(ir->t) || irt_isguard(ir->t)) {  /* Propagate mark. */
        if (ir->op1 >= REF_FIRST) irt_setmark(IR(ir->op1)->t);
        if (ir->op2 >= REF_FIRST) irt_setmark(IR(ir->op2)->t);
      }
      break;
    }
  }
}

/* Mark all instructions referenced by a snapshot. */
static void sink_mark_snap(jit_State *J, SnapShot *snap)
{
  SnapEntry *map = &J->cur.snapmap[snap->mapofs];
  MSize n, nent = snap->nent;
  for (n = 0; n < nent; n++) {
    IRRef ref = snap_ref(map[n]);
    if (!irref_isk(ref))
      irt_setmark(IR(ref)->t);
  }
}

/* Iteratively remark PHI refs with differing marks or PHI value counts. */
static void sink_remark_phi(jit_State *J)
{
  IRIns *ir;
  int remark;
  do {
    remark = 0;
    for (ir = IR(J->cur.nins-1); ir->o == IR_PHI; ir--) {
      IRIns *irl = IR(ir->op1), *irr = IR(ir->op2);
      if (((irl->t.irt ^ irr->t.irt) & IRT_MARK))
        remark = 1;
      else if (irl->prev == irr->prev)
        continue;
      irt_setmark(IR(ir->op1)->t);
      irt_setmark(IR(ir->op2)->t);
    }
  } while (remark);
}

/* Sweep instructions and tag sunken allocations and stores. */
static void sink_sweep_ins(jit_State *J)
{
  IRIns *ir, *irfirst = IR(J->cur.nk);
  for (ir = IR(J->cur.nins-1) ; ir >= irfirst; ir--) {
    switch (ir->o) {
    case IR_ASTORE: case IR_HSTORE: case IR_FSTORE: case IR_XSTORE: {
      IRIns *ira = sink_checkalloc(J, ir);
      if (ira && !irt_ismarked(ira->t)) {
        int delta = (int)(ir - ira);
        ir->prev = REGSP(RID_SINK, delta > 255 ? 255 : delta);
      } else {
        ir->prev = REGSP_INIT;
      }
      break;
      }
    case IR_NEWREF:
      if (!irt_ismarked(IR(ir->op1)->t)) {
        ir->prev = REGSP(RID_SINK, 0);
      } else {
        irt_clearmark(ir->t);
```

```
        ir->prev = REGSP_INIT;
      }
      break;
#if LJ_HASFFI
    case IR_CNEW: case IR_CNEWI:
#endif
    case IR_TNEW: case IR_TDUP:
      if (!irt_ismarked(ir->t)) {
        ir->t.irt &= ~IRT_GUARD;
        ir->prev = REGSP(RID_SINK, 0);
        J->cur.sinktags = 1;  /* Signal present SINK tags to assembler. */
      } else {
        irt_clearmark(ir->t);
        ir->prev = REGSP_INIT;
      }
      break;
    case IR_PHI: {
      IRIns *ira = IR(ir->op2);
      if (!irt_ismarked(ira->t) &&
          (ira->o == IR_TNEW || ira->o == IR_TDUP ||
           (LJ_HASFFI && (ira->o == IR_CNEW || ira->o == IR_CNEWI)))) {
        ir->prev = REGSP(RID_SINK, 0);
      } else {
        ir->prev = REGSP_INIT;
      }
      break;
      }
    default:
      irt_clearmark(ir->t);
      ir->prev = REGSP_INIT;
      break;
    }
  }
}

/* Allocation sinking and store sinking.
**
** 1. Mark all non-sinkable allocations.
** 2. Then sink all remaining allocations and the related stores.
*/
void lj_opt_sink(jit_State *J)
{
  const uint32_t need = (JIT_F_OPT_SINK|JIT_F_OPT_FWD|
                         JIT_F_OPT_DCE|JIT_F_OPT_CSE|JIT_F_OPT_FOLD);
  if ((J->flags & need) == need &&
      (J->chain[IR_TNEW] || J->chain[IR_TDUP] ||
       (LJ_HASFFI && (J->chain[IR_CNEW] || J->chain[IR_CNEWI])))) {
    if (!J->loopref)
      sink_mark_snap(J, &J->cur.snap[J->cur.nsnap-1]);
    sink_mark_ins(J);
    if (J->loopref)
      sink_remark_phi(J);
    sink_sweep_ins(J);
  }
}

#undef IR

#endif
```

One Level Up Top Level

src/lj_asm_arm.h - luajit-2.0-src

Global variables defined

- asm_compmap

Functions defined

- asm_add
- asm_ahustore
- asm_ahuvload
- asm_aref
- asm_bitop
- asm_bitshift
- asm_bswap
- asm_callround
- asm_callx
- asm_cnew
- asm_comp
- asm_conv
- asm_exitstub_gen
- asm_exitstub_setup
- asm_fload
- asm_fparith
- asm_fpcomp
- asm_fpmath
- asm_fpmin_max
- asm_fpunary
- asm_fref
- asm_fstore
- asm_fuseabase
- asm_fuseahuref
- asm_fuselsl2
- asm_fusemadd
- asm_fuseopm

- asm_fusexref
- asm_fxloadins
- asm_fxstoreins
- asm_gc_check
- asm_gencall
- asm_guardcc
- asm_head_lreg
- asm_head_root_base
- asm_head_side_base
- asm_hiop
- asm_href
- asm_hrefk
- asm_int64comp
- asm_intcomp
- asm_intmin_max
- asm_intmul
- asm_intneg
- asm_intop
- asm_intop_s
- asm_loop_fixup
- asm_min_max
- asm_mul
- asm_neg
- asm_obar
- asm_prof
- asm_retf
- asm_setup_call_slots
- asm_setup_target
- asm_setupresult
- asm_sfpcomp
- asm_sfpmin_max
- asm_sload
- asm_stack_check

- asm_stack_restore
- asm_strref
- asm_strto
- asm_sub
- asm_swapops
- asm_tail_fixup
- asm_tail_prep
- asm_tbar
- asm_tobit
- asm_tointg
- asm_tvptr
- asm_uref
- asm_xload
- asm_xstore_
- lj_asm_patchexit
- noconflict
- ra_alloc2
- ra_hintalloc
- ra_scratchpair

Macros defined

- CONFLICT_SEARCH_LIM
- asm_abs
- asm_abs
- asm_addov
- asm_atan2
- asm_atan2
- asm_band
- asm_bnot
- asm_bor
- asm_brol
- asm_bror
- asm_bsar
- asm_bshl

- asm_bshr
- asm_bxor
- asm_cnew
- asm_div
- asm_div
- asm_equal
- asm_fpmath
- asm_ldexp
- asm_ldexp
- asm_max
- asm_min
- asm_mod
- asm_mulov
- asm_pow
- asm_pow
- asm_subov
- asm_tobit
- asm_xstore

Source code

```
/*
** ARM IR assembler (SSA IR -> machine code).
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

/* -- Register allocator extensions --------------------------------------- */

/* Allocate a register with a hint. */
static Reg ra_hintalloc(ASMState *as, IRRef ref, Reg hint, RegSet allow)
{
  Reg r = IR(ref)->r;
  if (ra_noreg(r)) {
    if (!ra_hashint(r) && !iscrossref(as, ref))
      ra_sethint(IR(ref)->r, hint);  /* Propagate register hint. */
    r = ra_allocref(as, ref, allow);
  }
  ra_noweak(as, r);
  return r;
}

/* Allocate a scratch register pair. */
static Reg ra_scratchpair(ASMState *as, RegSet allow)
{
  RegSet pick1 = as->freeset & allow;
  RegSet pick2 = pick1 & (pick1 >> 1) & RSET_GPREVEN;
  Reg r;
  if (pick2) {
    r = rset_picktop(pick2);
  } else {
    RegSet pick = pick1 & (allow >> 1) & RSET_GPREVEN;
    if (pick) {
```

```
      r = rset_picktop(pick);
      ra_restore(as, regcost_ref(as->cost[r+1]));
    } else {
      pick = pick1 & (allow << 1) & RSET_GPRODD;
      if (pick) {
        r = ra_restore(as, regcost_ref(as->cost[rset_picktop(pick)-1]));
      } else {
        r = ra_evict(as, allow & (allow >> 1) & RSET_GPREVEN);
        ra_restore(as, regcost_ref(as->cost[r+1]));
      }
    }
  }
  lua_assert(rset_test(RSET_GPREVEN, r));
  ra_modified(as, r);
  ra_modified(as, r+1);
  RA_DBGX((as, "scratchpair    $r $r", r, r+1));
  return r;
}

#if !LJ_SOFTFP
/* Allocate two source registers for three-operand instructions. */
static Reg ra_alloc2(ASMState *as, IRIns *ir, RegSet allow)
{
  IRIns *irl = IR(ir->op1), *irr = IR(ir->op2);
  Reg left = irl->r, right = irr->r;
  if (ra_hasreg(left)) {
    ra_noweak(as, left);
    if (ra_noreg(right))
      right = ra_allocref(as, ir->op2, rset_exclude(allow, left));
    else
      ra_noweak(as, right);
  } else if (ra_hasreg(right)) {
    ra_noweak(as, right);
    left = ra_allocref(as, ir->op1, rset_exclude(allow, right));
  } else if (ra_hashint(right)) {
    right = ra_allocref(as, ir->op2, allow);
    left = ra_alloc1(as, ir->op1, rset_exclude(allow, right));
  } else {
    left = ra_allocref(as, ir->op1, allow);
    right = ra_alloc1(as, ir->op2, rset_exclude(allow, left));
  }
  return left | (right << 8);
}
#endif

/* -- Guard handling ------------------------------------------------------ */

/* Generate an exit stub group at the bottom of the reserved MCode memory. */
static MCode *asm_exitstub_gen(ASMState *as, ExitNo group)
{
  MCode *mxp = as->mcbot;
  int i;
  if (mxp + 4*4+4*EXITSTUBS_PER_GROUP >= as->mctop)
    asm_mclimit(as);
  /* str lr, [sp]; bl ->vm_exit_handler; .long DISPATCH_address, group. */
  *mxp++ = ARMI_STR|ARMI_LS_P|ARMI_LS_U|ARMF_D(RID_LR)|ARMF_N(RID_SP);
  *mxp = ARMI_BL|((((MCode *)(void *)lj_vm_exit_handler-mxp)-2)&0x00ffffffu);
  mxp++;
  *mxp++ = (MCode)i32ptr(J2GG(as->J)->dispatch);  /* DISPATCH address */
  *mxp++ = group*EXITSTUBS_PER_GROUP;
  for (i = 0; i < EXITSTUBS_PER_GROUP; i++)
    *mxp++ = ARMI_B|((-6-i)&0x00ffffffu);
  lj_mcode_sync(as->mcbot, mxp);
  lj_mcode_commitbot(as->J, mxp);
  as->mcbot = mxp;
  as->mclim = as->mcbot + MCLIM_REDZONE;
  return mxp - EXITSTUBS_PER_GROUP;
}

 /* Setup all needed exit stubs. */
 static void asm_exitstub_setup(ASMState *as, ExitNo nexits)
 {
   ExitNo i;
   if (nexits >= EXITSTUBS_PER_GROUP*LJ_MAX_EXITSTUBGR)
     lj_trace_err(as->J, LJ_TRERR_SNAPOV);
   for (i = 0; i < (nexits+EXITSTUBS_PER_GROUP-1)/EXITSTUBS_PER_GROUP; i++)
```

```
    if (as->J->exitstubgroup[i] == NULL)
      as->J->exitstubgroup[i] = asm_exitstub_gen(as, i);
}

/* Emit conditional branch to exit for guard. */
static void asm_guardcc(ASMState *as, ARMCC cc)
{
  MCode *target = exitstub_addr(as->J, as->snapno);
  MCode *p = as->mcp;
  if (LJ_UNLIKELY(p == as->invmcp)) {
    as->loopinv = 1;
    *p = ARMI_BL | ((target-p-2) & 0x00ffffffu);
    emit_branch(as, ARMF_CC(ARMI_B, cc^1), p+1);
    return;
  }
  emit_branch(as, ARMF_CC(ARMI_BL, cc), target);
}

/* -- Operand fusion ------------------------------------------------------ */

/* Limit linear search to this distance. Avoids O(n^2) behavior. */
#define CONFLICT_SEARCH_LIM        31

/* Check if there's no conflicting instruction between curins and ref. */
static int noconflict(ASMState *as, IRRef ref, IROp conflict)
{
  IRIns *ir = as->ir;
  IRRef i = as->curins;
  if (i > ref + CONFLICT_SEARCH_LIM)
    return 0;  /* Give up, ref is too far away. */
  while (--i > ref)
    if (ir[i].o == conflict)
      return 0;  /* Conflict found. */
  return 1;  /* Ok, no conflict. */
}

/* Fuse the array base of colocated arrays. */
static int32_t asm_fuseabase(ASMState *as, IRRef ref)
{
  IRIns *ir = IR(ref);
  if (ir->o == IR_TNEW && ir->op1 <= LJ_MAX_COLOSIZE &&
      !neverfuse(as) && noconflict(as, ref, IR_NEWREF))
    return (int32_t)sizeof(GCtab);
  return 0;
}

/* Fuse array/hash/upvalue reference into register+offset operand. */
static Reg asm_fuseahuref(ASMState *as, IRRef ref, int32_t *ofsp, RegSet allow,
                          int lim)
{
  IRIns *ir = IR(ref);
  if (ra_noreg(ir->r)) {
    if (ir->o == IR_AREF) {
      if (mayfuse(as, ref)) {
        if (irref_isk(ir->op2)) {
          IRRef tab = IR(ir->op1)->op1;
          int32_t ofs = asm_fuseabase(as, tab);
          IRRef refa = ofs ? tab : ir->op1;
          ofs += 8*IR(ir->op2)->i;
          if (ofs > -lim && ofs < lim) {
            *ofsp = ofs;
            return ra_alloc1(as, refa, allow);
          }
        }
      }
    } else if (ir->o == IR_HREFK) {
      if (mayfuse(as, ref)) {
        int32_t ofs = (int32_t)(IR(ir->op2)->op2 * sizeof(Node));
        if (ofs < lim) {
          *ofsp = ofs;
          return ra_alloc1(as, ir->op1, allow);
        }
      }
    } else if (ir->o == IR_UREFC) {
      if (irref_isk(ir->op1)) {
        GCfunc *fn = ir_kfunc(IR(ir->op1));
```

```
        int32_t ofs = i32ptr(&gcref(fn->l.uvptr[(ir->op2 >> 8)])->uv.tv);
        *ofsp = (ofs & 255);  /* Mask out less bits to allow LDRD. */
        return ra_allock(as, (ofs & ~255), allow);
      }
    }
  }
  *ofsp = 0;
  return ra_alloc1(as, ref, allow);
}

/* Fuse m operand into arithmetic/logic instructions. */
static uint32_t asm_fuseopm(ASMState *as, ARMIns ai, IRRef ref, RegSet allow)
{
  IRIns *ir = IR(ref);
  if (ra_hasreg(ir->r)) {
    ra_noweak(as, ir->r);
    return ARMF_M(ir->r);
  } else if (irref_isk(ref)) {
    uint32_t k = emit_isk12(ai, ir->i);
    if (k)
      return k;
  } else if (mayfuse(as, ref)) {
    if (ir->o >= IR_BSHL && ir->o <= IR_BROR) {
      Reg m = ra_alloc1(as, ir->op1, allow);
      ARMShift sh = ir->o == IR_BSHL ? ARMSH_LSL :
                    ir->o == IR_BSHR ? ARMSH_LSR :
                    ir->o == IR_BSAR ? ARMSH_ASR : ARMSH_ROR;
      if (irref_isk(ir->op2)) {
        return m | ARMF_SH(sh, (IR(ir->op2)->i & 31));
      } else {
        Reg s = ra_alloc1(as, ir->op2, rset_exclude(allow, m));
        return m | ARMF_RSH(sh, s);
      }
    } else if (ir->o == IR_ADD && ir->op1 == ir->op2) {
      Reg m = ra_alloc1(as, ir->op1, allow);
      return m | ARMF_SH(ARMSH_LSL, 1);
    }
  }
  return ra_allocref(as, ref, allow);
}

/* Fuse shifts into loads/stores. Only bother with BSHL 2 => lsl #2. */
static IRRef asm_fuselsl2(ASMState *as, IRRef ref)
{
  IRIns *ir = IR(ref);
  if (ra_noreg(ir->r) && mayfuse(as, ref) && ir->o == IR_BSHL &&
      irref_isk(ir->op2) && IR(ir->op2)->i == 2)
    return ir->op1;
  return 0;  /* No fusion. */
}

/* Fuse XLOAD/XSTORE reference into load/store operand. */
static void asm_fusexref(ASMState *as, ARMIns ai, Reg rd, IRRef ref,
                         RegSet allow, int32_t ofs)
{
  IRIns *ir = IR(ref);
  Reg base;
  if (ra_noreg(ir->r) && canfuse(as, ir)) {
    int32_t lim = (!LJ_SOFTFP && (ai & 0x08000000)) ? 1024 :
                  (ai & 0x04000000) ? 4096 : 256;
    if (ir->o == IR_ADD) {
      int32_t ofs2;
      if (irref_isk(ir->op2) &&
          (ofs2 = ofs + IR(ir->op2)->i) > -lim && ofs2 < lim &&
          (!(!LJ_SOFTFP && (ai & 0x08000000)) || !(ofs2 & 3))) {
        ofs = ofs2;
        ref = ir->op1;
      } else if (ofs == 0 && !(!LJ_SOFTFP && (ai & 0x08000000))) {
        IRRef lref = ir->op1, rref = ir->op2;
        Reg rn, rm;
        if ((ai & 0x04000000)) {
          IRRef sref = asm_fuselsl2(as, rref);
          if (sref) {
            rref = sref;
            ai |= ARMF_SH(ARMSH_LSL, 2);
          } else if ((sref = asm_fuselsl2(as, lref)) != 0) {
```

```
260             lref = rref;
261             rref = sref;
262             ai |= ARMF_SH(ARMSH_LSL, 2);
263           }
264         }
265         rn = ra_alloc1(as, lref, allow);
266         rm = ra_alloc1(as, rref, rset_exclude(allow, rn));
267         if ((ai & 0x04000000)) ai |= ARMI_LS_R;
268         emit_dnm(as, ai|ARMI_LS_P|ARMI_LS_U, rd, rn, rm);
269         return;
270       }
271     } else if (ir->o == IR_STRREF && !(!LJ_SOFTFP && (ai & 0x08000000))) {
272       lua_assert(ofs == 0);
273       ofs = (int32_t)sizeof(GCstr);
274       if (irref_isk(ir->op2)) {
275         ofs += IR(ir->op2)->i;
276         ref = ir->op1;
277       } else if (irref_isk(ir->op1)) {
278         ofs += IR(ir->op1)->i;
279         ref = ir->op2;
280       } else {
281         /* NYI: Fuse ADD with constant. */
282         Reg rn = ra_alloc1(as, ir->op1, allow);
283         uint32_t m = asm_fuseopm(as, 0, ir->op2, rset_exclude(allow, rn));
284         if ((ai & 0x04000000))
285           emit_lso(as, ai, rd, rd, ofs);
286         else
287           emit_lsox(as, ai, rd, rd, ofs);
288         emit_dn(as, ARMI_ADD^m, rd, rn);
289         return;
290       }
291       if (ofs <= -lim || ofs >= lim) {
292         Reg rn = ra_alloc1(as, ref, allow);
293         Reg rm = ra_allock(as, ofs, rset_exclude(allow, rn));
294         if ((ai & 0x04000000)) ai |= ARMI_LS_R;
295         emit_dnm(as, ai|ARMI_LS_P|ARMI_LS_U, rd, rn, rm);
296         return;
297       }
298     }
299   }
300   base = ra_alloc1(as, ref, allow);
301 #if !LJ_SOFTFP
302   if ((ai & 0x08000000))
303     emit_vlso(as, ai, rd, base, ofs);
304   else
305 #endif
306   if ((ai & 0x04000000))
307     emit_lso(as, ai, rd, base, ofs);
308   else
309     emit_lsox(as, ai, rd, base, ofs);
310 }
311
312 #if !LJ_SOFTFP
313 /* Fuse to multiply-add/sub instruction. */
314 static int asm_fusemadd(ASMState *as, IRIns *ir, ARMIns ai, ARMIns air)
315 {
316   IRRef lref = ir->op1, rref = ir->op2;
317   IRIns *irm;
318   if (lref != rref &&
319       ((mayfuse(as, lref) && (irm = IR(lref), irm->o == IR_MUL) &&
320         ra_noreg(irm->r)) ||
321        (mayfuse(as, rref) && (irm = IR(rref), irm->o == IR_MUL) &&
322         (rref = lref, ai = air, ra_noreg(irm->r))))) {
323     Reg dest = ra_dest(as, ir, RSET_FPR);
324     Reg add = ra_hintalloc(as, rref, dest, RSET_FPR);
325     Reg right, left = ra_alloc2(as, irm,
326                         rset_exclude(rset_exclude(RSET_FPR, dest), add));
327     right = (left >> 8); left &= 255;
328     emit_dnm(as, ai, (dest & 15), (left & 15), (right & 15));
329     if (dest != add) emit_dm(as, ARMI_VMOV_D, (dest & 15), (add & 15));
330     return 1;
331   }
332   return 0;
333 }
334 #endif
335
```

```
/* -- Calls --------------------------------------------------------- */

/* Generate a call to a C function. */
static void asm_gencall(ASMState *as, const CCallInfo *ci, IRRef *args)
{
  uint32_t n, nargs = CCI_XNARGS(ci);
  int32_t ofs = 0;
#if LJ_SOFTFP
  Reg gpr = REGARG_FIRSTGPR;
#else
  Reg gpr, fpr = REGARG_FIRSTFPR, fprodd = 0;
#endif
  if ((void *)ci->func)
    emit_call(as, (void *)ci->func);
#if !LJ_SOFTFP
  for (gpr = REGARG_FIRSTGPR; gpr <= REGARG_LASTGPR; gpr++)
    as->cost[gpr] = REGCOST(~0u, ASMREF_L);
  gpr = REGARG_FIRSTGPR;
#endif
  for (n = 0; n < nargs; n++) {  /* Setup args. */
    IRRef ref = args[n];
    IRIns *ir = IR(ref);
#if !LJ_SOFTFP
    if (ref && irt_isfp(ir->t)) {
      RegSet of = as->freeset;
      Reg src;
      if (!LJ_ABI_SOFTFP && !(ci->flags & CCI_VARARG)) {
        if (irt_isnum(ir->t)) {
          if (fpr <= REGARG_LASTFPR) {
            ra_leftov(as, fpr, ref);
            fpr++;
            continue;
          }
        } else if (fprodd) {  /* Ick. */
          src = ra_alloc1(as, ref, RSET_FPR);
          emit_dm(as, ARMI_VMOV_S, (fprodd & 15), (src & 15) | 0x00400000);
          fprodd = 0;
          continue;
        } else if (fpr <= REGARG_LASTFPR) {
          ra_leftov(as, fpr, ref);
          fprodd = fpr++;
          continue;
        }
        /* Workaround to protect argument GPRs from being used for remat. */
        as->freeset &= ~RSET_RANGE(REGARG_FIRSTGPR, REGARG_LASTGPR+1);
        src = ra_alloc1(as, ref, RSET_FPR);  /* May alloc GPR to remat FPR. */
        as->freeset |= (of & RSET_RANGE(REGARG_FIRSTGPR, REGARG_LASTGPR+1));
        fprodd = 0;
        goto stackfp;
      }
      /* Workaround to protect argument GPRs from being used for remat. */
      as->freeset &= ~RSET_RANGE(REGARG_FIRSTGPR, REGARG_LASTGPR+1);
      src = ra_alloc1(as, ref, RSET_FPR);  /* May alloc GPR to remat FPR. */
      as->freeset |= (of & RSET_RANGE(REGARG_FIRSTGPR, REGARG_LASTGPR+1));
      if (irt_isnum(ir->t)) gpr = (gpr+1) & ~1u;
      if (gpr <= REGARG_LASTGPR) {
        lua_assert(rset_test(as->freeset, gpr));  /* Must have been evicted. */
        if (irt_isnum(ir->t)) {
          lua_assert(rset_test(as->freeset, gpr+1));  /* Ditto. */
          emit_dnm(as, ARMI_VMOV_RR_D, gpr, gpr+1, (src & 15));
          gpr += 2;
        } else {
          emit_dn(as, ARMI_VMOV_R_S, gpr, (src & 15));
          gpr++;
        }
      } else {
      stackfp:
        if (irt_isnum(ir->t)) ofs = (ofs + 4) & ~4;
        emit_spstore(as, ir, src, ofs);
        ofs += irt_isnum(ir->t) ? 8 : 4;
      }
    } else
#endif
    {
      if (gpr <= REGARG_LASTGPR) {
        lua_assert(rset_test(as->freeset, gpr));  /* Must have been evicted. */
```

```
        if (ref) ra_leftov(as, gpr, ref);
        gpr++;
      } else {
        if (ref) {
          Reg r = ra_alloc1(as, ref, RSET_GPR);
          emit_spstore(as, ir, r, ofs);
        }
        ofs += 4;
      }
    }
  }
}

/* Setup result reg/sp for call. Evict scratch regs. */
static void asm_setupresult(ASMState *as, IRIns *ir, const CCallInfo *ci)
{
  RegSet drop = RSET_SCRATCH;
  int hiop = ((ir+1)->o == IR_HIOP);
  if (ra_hasreg(ir->r))
    rset_clear(drop, ir->r);  /* Dest reg handled below. */
  if (hiop && ra_hasreg((ir+1)->r))
    rset_clear(drop, (ir+1)->r);  /* Dest reg handled below. */
  ra_evictset(as, drop);  /* Evictions must be performed first. */
  if (ra_used(ir)) {
    lua_assert(!irt_ispri(ir->t));
    if (!LJ_SOFTFP && irt_isfp(ir->t)) {
      if (LJ_ABI_SOFTFP || (ci->flags & (CCI_CASTU64|CCI_VARARG))) {
        Reg dest = (ra_dest(as, ir, RSET_FPR) & 15);
        if (irt_isnum(ir->t))
          emit_dnm(as, ARMI_VMOV_D_RR, RID_RETLO, RID_RETHI, dest);
        else
          emit_dn(as, ARMI_VMOV_S_R, RID_RET, dest);
      } else {
        ra_destreg(as, ir, RID_FPRET);
      }
    } else if (hiop) {
      ra_destpair(as, ir);
    } else {
      ra_destreg(as, ir, RID_RET);
    }
  }
  UNUSED(ci);
}

static void asm_callx(ASMState *as, IRIns *ir)
{
  IRRef args[CCI_NARGS_MAX*2];
  CCallInfo ci;
  IRRef func;
  IRIns *irf;
  ci.flags = asm_callx_flags(as, ir);
  asm_collectargs(as, ir, &ci, args);
  asm_setupresult(as, ir, &ci);
  func = ir->op2; irf = IR(func);
  if (irf->o == IR_CARG) { func = irf->op1; irf = IR(func); }
  if (irref_isk(func)) {  /* Call to constant address. */
    ci.func = (ASMFunction)(void *)(irf->i);
  } else {  /* Need a non-argument register for indirect calls. */
    Reg freg = ra_alloc1(as, func, RSET_RANGE(RID_R4, RID_R12+1));
    emit_m(as, ARMI_BLXr, freg);
    ci.func = (ASMFunction)(void *)0;
  }
  asm_gencall(as, &ci, args);
}

/* -- Returns ----------------------------------------------------------- */

/* Return to lower frame. Guard that it goes to the right spot. */
static void asm_retf(ASMState *as, IRIns *ir)
{
  Reg base = ra_alloc1(as, REF_BASE, RSET_GPR);
  void *pc = ir_kptr(IR(ir->op2));
  int32_t delta = 1+LJ_FR2+bc_a(*((const BCIns *)pc - 1));
  as->topslot -= (BCReg)delta;
  if ((int32_t)as->topslot < 0) as->topslot = 0;
  irt_setmark(IR(REF_BASE)->t);  /* Children must not coalesce with BASE reg. */
```

```
  /* Need to force a spill on REF_BASE now to update the stack slot. */
  emit_lso(as, ARMI_STR, base, RID_SP, ra_spill(as, IR(REF_BASE)));
  emit_setgl(as, base, jit_base);
  emit_addptr(as, base, -8*delta);
  asm_guardcc(as, CC_NE);
  emit_nm(as, ARMI_CMP, RID_TMP,
          ra_allock(as, i32ptr(pc), rset_exclude(RSET_GPR, base)));
  emit_lso(as, ARMI_LDR, RID_TMP, base, -4);
}

/* -- Type conversions ---------------------------------------------------- */

#if !LJ_SOFTFP
static void asm_tointg(ASMState *as, IRIns *ir, Reg left)
{
  Reg tmp = ra_scratch(as, rset_exclude(RSET_FPR, left));
  Reg dest = ra_dest(as, ir, RSET_GPR);
  asm_guardcc(as, CC_NE);
  emit_d(as, ARMI_VMRS, 0);
  emit_dm(as, ARMI_VCMP_D, (tmp & 15), (left & 15));
  emit_dm(as, ARMI_VCVT_F64_S32, (tmp & 15), (tmp & 15));
  emit_dn(as, ARMI_VMOV_R_S, dest, (tmp & 15));
  emit_dm(as, ARMI_VCVT_S32_F64, (tmp & 15), (left & 15));
}

static void asm_tobit(ASMState *as, IRIns *ir)
{
  RegSet allow = RSET_FPR;
  Reg left = ra_alloc1(as, ir->op1, allow);
  Reg right = ra_alloc1(as, ir->op2, rset_clear(allow, left));
  Reg tmp = ra_scratch(as, rset_clear(allow, right));
  Reg dest = ra_dest(as, ir, RSET_GPR);
  emit_dn(as, ARMI_VMOV_R_S, dest, (tmp & 15));
  emit_dnm(as, ARMI_VADD_D, (tmp & 15), (left & 15), (right & 15));
}
#else
#define asm_tobit(as, ir)        lua_assert(0)
#endif

static void asm_conv(ASMState *as, IRIns *ir)
{
  IRType st = (IRType)(ir->op2 & IRCONV_SRCMASK);
#if !LJ_SOFTFP
  int stfp = (st == IRT_NUM || st == IRT_FLOAT);
#endif
  IRRef lref = ir->op1;
  /* 64 bit integer conversions are handled by SPLIT. */
  lua_assert(!irt_isint64(ir->t) && !(st == IRT_I64 || st == IRT_U64));
#if LJ_SOFTFP
  /* FP conversions are handled by SPLIT. */
  lua_assert(!irt_isfp(ir->t) && !(st == IRT_NUM || st == IRT_FLOAT));
  /* Can't check for same types: SPLIT uses CONV int.int + BXOR for sfp NEG. */
#else
  lua_assert(irt_type(ir->t) != st);
  if (irt_isfp(ir->t)) {
    Reg dest = ra_dest(as, ir, RSET_FPR);
    if (stfp) {  /* FP to FP conversion. */
      emit_dm(as, st == IRT_NUM ? ARMI_VCVT_F32_F64 : ARMI_VCVT_F64_F32,
              (dest & 15), (ra_alloc1(as, lref, RSET_FPR) & 15));
    } else {  /* Integer to FP conversion. */
      Reg left = ra_alloc1(as, lref, RSET_GPR);
      ARMIns ai = irt_isfloat(ir->t) ?
        (st == IRT_INT ? ARMI_VCVT_F32_S32 : ARMI_VCVT_F32_U32) :
        (st == IRT_INT ? ARMI_VCVT_F64_S32 : ARMI_VCVT_F64_U32);
      emit_dm(as, ai, (dest & 15), (dest & 15));
      emit_dn(as, ARMI_VMOV_S_R, left, (dest & 15));
    }
  } else if (stfp) {  /* FP to integer conversion. */
    if (irt_isguard(ir->t)) {
      /* Checked conversions are only supported from number to int. */
      lua_assert(irt_isint(ir->t) && st == IRT_NUM);
      asm_tointg(as, ir, ra_alloc1(as, lref, RSET_FPR));
    } else {
      Reg left = ra_alloc1(as, lref, RSET_FPR);
      Reg tmp = ra_scratch(as, rset_exclude(RSET_FPR, left));
      Reg dest = ra_dest(as, ir, RSET_GPR);
```

```
      ARMIns ai;
      emit_dn(as, ARMI_VMOV_R_S, dest, (tmp & 15));
      ai = irt_isint(ir->t) ?
        (st == IRT_NUM ? ARMI_VCVT_S32_F64 : ARMI_VCVT_S32_F32) :
        (st == IRT_NUM ? ARMI_VCVT_U32_F64 : ARMI_VCVT_U32_F32);
      emit_dm(as, ai, (tmp & 15), (left & 15));
    }
  } else
#endif
  {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    if (st >= IRT_I8 && st <= IRT_U16) {  /* Extend to 32 bit integer. */
      Reg left = ra_alloc1(as, lref, RSET_GPR);
      lua_assert(irt_isint(ir->t) || irt_isu32(ir->t));
      if ((as->flags & JIT_F_ARMV6)) {
        ARMIns ai = st == IRT_I8 ? ARMI_SXTB :
                    st == IRT_U8 ? ARMI_UXTB :
                    st == IRT_I16 ? ARMI_SXTH : ARMI_UXTH;
        emit_dm(as, ai, dest, left);
      } else if (st == IRT_U8) {
        emit_dn(as, ARMI_AND|ARMI_K12|255, dest, left);
      } else {
        uint32_t shift = st == IRT_I8 ? 24 : 16;
        ARMShift sh = st == IRT_U16 ? ARMSH_LSR : ARMSH_ASR;
        emit_dm(as, ARMI_MOV|ARMF_SH(sh, shift), dest, RID_TMP);
        emit_dm(as, ARMI_MOV|ARMF_SH(ARMSH_LSL, shift), RID_TMP, left);
      }
    } else {  /* Handle 32/32 bit no-op (cast). */
      ra_leftov(as, dest, lref);  /* Do nothing, but may need to move regs. */
    }
  }
}

static void asm_strto(ASMState *as, IRIns *ir)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_strscan_num];
  IRRef args[2];
  Reg rlo = 0, rhi = 0, tmp;
  int destused = ra_used(ir);
  int32_t ofs = 0;
  ra_evictset(as, RSET_SCRATCH);
#if LJ_SOFTFP
  if (destused) {
    if (ra_hasspill(ir->s) && ra_hasspill((ir+1)->s) &&
        (ir->s & 1) == 0 && ir->s + 1 == (ir+1)->s) {
      int i;
      for (i = 0; i < 2; i++) {
        Reg r = (ir+i)->r;
        if (ra_hasreg(r)) {
          ra_free(as, r);
          ra_modified(as, r);
          emit_spload(as, ir+i, r, sps_scale((ir+i)->s));
        }
      }
      ofs = sps_scale(ir->s);
      destused = 0;
    } else {
      rhi = ra_dest(as, ir+1, RSET_GPR);
      rlo = ra_dest(as, ir, rset_exclude(RSET_GPR, rhi));
    }
  }
  asm_guardcc(as, CC_EQ);
  if (destused) {
    emit_lso(as, ARMI_LDR, rhi, RID_SP, 4);
    emit_lso(as, ARMI_LDR, rlo, RID_SP, 0);
  }
#else
  UNUSED(rhi);
  if (destused) {
    if (ra_hasspill(ir->s)) {
      ofs = sps_scale(ir->s);
      destused = 0;
      if (ra_hasreg(ir->r)) {
        ra_free(as, ir->r);
        ra_modified(as, ir->r);
        emit_spload(as, ir, ir->r, ofs);
```

```
640       }
641     } else {
642       rlo = ra_dest(as, ir, RSET_FPR);
643     }
644   }
645   asm_guardcc(as, CC_EQ);
646   if (destused)
647     emit_vlso(as, ARMI_VLDR_D, rlo, RID_SP, 0);
648 #endif
649   emit_n(as, ARMI_CMP|ARMI_K12|0, RID_RET);  /* Test return status. */
650   args[0] = ir->op1;      /* GCstr *str */
651   args[1] = ASMREF_TMP1;  /* TValue *n  */
652   asm_gencall(as, ci, args);
653   tmp = ra_releasetmp(as, ASMREF_TMP1);
654   if (ofs == 0)
655     emit_dm(as, ARMI_MOV, tmp, RID_SP);
656   else
657     emit_opk(as, ARMI_ADD, tmp, RID_SP, ofs, RSET_GPR);
658 }
659
660 /* -- Memory references --------------------------------------------------- */
661
662 /* Get pointer to TValue. */
663 static void asm_tvptr(ASMState *as, Reg dest, IRRef ref)
664 {
665   IRIns *ir = IR(ref);
666   if (irt_isnum(ir->t)) {
667     if (irref_isk(ref)) {
668       /* Use the number constant itself as a TValue. */
669       ra_allockreg(as, i32ptr(ir_knum(ir)), dest);
670     } else {
671 #if LJ_SOFTFP
672       lua_assert(0);
673 #else
674       /* Otherwise force a spill and use the spill slot. */
675       emit_opk(as, ARMI_ADD, dest, RID_SP, ra_spill(as, ir), RSET_GPR);
676 #endif
677     }
678   } else {
679     /* Otherwise use [sp] and [sp+4] to hold the TValue. */
680     RegSet allow = rset_exclude(RSET_GPR, dest);
681     Reg type;
682     emit_dm(as, ARMI_MOV, dest, RID_SP);
683     if (!irt_ispri(ir->t)) {
684       Reg src = ra_alloc1(as, ref, allow);
685       emit_lso(as, ARMI_STR, src, RID_SP, 0);
686     }
687     if (LJ_SOFTFP && (ir+1)->o == IR_HIOP)
688       type = ra_alloc1(as, ref+1, allow);
689     else
690       type = ra_allock(as, irt_toitype(ir->t), allow);
691     emit_lso(as, ARMI_STR, type, RID_SP, 4);
692   }
693 }
694
695 static void asm_aref(ASMState *as, IRIns *ir)
696 {
697   Reg dest = ra_dest(as, ir, RSET_GPR);
698   Reg idx, base;
699   if (irref_isk(ir->op2)) {
700     IRRef tab = IR(ir->op1)->op1;
701     int32_t ofs = asm_fuseabase(as, tab);
702     IRRef refa = ofs ? tab : ir->op1;
703     uint32_t k = emit_isk12(ARMI_ADD, ofs + 8*IR(ir->op2)->i);
704     if (k) {
705       base = ra_alloc1(as, refa, RSET_GPR);
706       emit_dn(as, ARMI_ADD^k, dest, base);
707       return;
708     }
709   }
710   base = ra_alloc1(as, ir->op1, RSET_GPR);
711   idx = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, base));
712   emit_dnm(as, ARMI_ADD|ARMF_SH(ARMSH_LSL, 3), dest, base, idx);
713 }
714
715 /* Inlined hash lookup. Specialized for key type and for const keys.
```

```
** The equivalent C code is:
**   Node *n = hashkey(t, key);
**   do {
**     if (lj_obj_equal(&n->key, key)) return &n->val;
**   } while ((n = nextnode(n)));
**   return niltv(L);
*/
static void asm_href(ASMState *as, IRIns *ir, IROp merge)
{
  RegSet allow = RSET_GPR;
  int destused = ra_used(ir);
  Reg dest = ra_dest(as, ir, allow);
  Reg tab = ra_alloc1(as, ir->op1, rset_clear(allow, dest));
  Reg key = 0, keyhi = 0, keynumhi = RID_NONE, tmp = RID_TMP;
  IRRef refkey = ir->op2;
  IRIns *irkey = IR(refkey);
  IRType1 kt = irkey->t;
  int32_t k = 0, khi = emit_isk12(ARMI_CMP, irt_toitype(kt));
  uint32_t khash;
  MCLabel l_end, l_loop;
  rset_clear(allow, tab);
  if (!irref_isk(refkey) || irt_isstr(kt)) {
#if LJ_SOFTFP
    key = ra_alloc1(as, refkey, allow);
    rset_clear(allow, key);
    if (irkey[1].o == IR_HIOP) {
      if (ra_hasreg((irkey+1)->r)) {
        keynumhi = (irkey+1)->r;
        keyhi = RID_TMP;
        ra_noweak(as, keynumhi);
      } else {
        keyhi = keynumhi = ra_allocref(as, refkey+1, allow);
      }
      rset_clear(allow, keynumhi);
      khi = 0;
    }
#else
    if (irt_isnum(kt)) {
      key = ra_scratch(as, allow);
      rset_clear(allow, key);
      keyhi = keynumhi = ra_scratch(as, allow);
      rset_clear(allow, keyhi);
      khi = 0;
    } else {
      key = ra_alloc1(as, refkey, allow);
      rset_clear(allow, key);
    }
#endif
  } else if (irt_isnum(kt)) {
    int32_t val = (int32_t)ir_knum(irkey)->u32.lo;
    k = emit_isk12(ARMI_CMP, val);
    if (!k) {
      key = ra_allock(as, val, allow);
      rset_clear(allow, key);
    }
    val = (int32_t)ir_knum(irkey)->u32.hi;
    khi = emit_isk12(ARMI_CMP, val);
    if (!khi) {
      keyhi = ra_allock(as, val, allow);
      rset_clear(allow, keyhi);
    }
  } else if (!irt_ispri(kt)) {
    k = emit_isk12(ARMI_CMP, irkey->i);
    if (!k) {
      key = ra_alloc1(as, refkey, allow);
      rset_clear(allow, key);
    }
  }
  if (!irt_ispri(kt))
    tmp = ra_scratchpair(as, allow);

  /* Key not found in chain: jump to exit (if merged) or load niltv. */
  l_end = emit_label(as);
  as->invmcp = NULL;
  if (merge == IR_NE)
    asm_guardcc(as, CC_AL);
```

```
  else if (destused)
    emit_loada(as, dest, niltvg(J2G(as->J)));

  /* Follow hash chain until the end. */
  l_loop = --as->mcp;
  emit_n(as, ARMI_CMP|ARMI_K12|0, dest);
  emit_lso(as, ARMI_LDR, dest, dest, (int32_t)offsetof(Node, next));

  /* Type and value comparison. */
  if (merge == IR_EQ)
    asm_guardcc(as, CC_EQ);
  else
    emit_branch(as, ARMF_CC(ARMI_B, CC_EQ), l_end);
  if (!irt_ispri(kt)) {
    emit_nm(as, ARMF_CC(ARMI_CMP, CC_EQ)^k, tmp, key);
    emit_nm(as, ARMI_CMP^khi, tmp+1, keyhi);
    emit_lsox(as, ARMI_LDRD, tmp, dest, (int32_t)offsetof(Node, key));
  } else {
    emit_n(as, ARMI_CMP^khi, tmp);
    emit_lso(as, ARMI_LDR, tmp, dest, (int32_t)offsetof(Node, key.it));
  }
  *l_loop = ARMF_CC(ARMI_B, CC_NE) | ((as->mcp-l_loop-2) & 0x00ffffffu);

  /* Load main position relative to tab->node into dest. */
  khash = irref_isk(refkey) ? ir_khash(irkey) : 1;
  if (khash == 0) {
    emit_lso(as, ARMI_LDR, dest, tab, (int32_t)offsetof(GCtab, node));
  } else {
    emit_dnm(as, ARMI_ADD|ARMF_SH(ARMSH_LSL, 3), dest, dest, tmp);
    emit_dnm(as, ARMI_ADD|ARMF_SH(ARMSH_LSL, 1), tmp, tmp, tmp);
    if (irt_isstr(kt)) {  /* Fetch of str->hash is cheaper than ra_allock. */
      emit_dnm(as, ARMI_AND, tmp, tmp+1, RID_TMP);
      emit_lso(as, ARMI_LDR, dest, tab, (int32_t)offsetof(GCtab, node));
      emit_lso(as, ARMI_LDR, tmp+1, key, (int32_t)offsetof(GCstr, hash));
      emit_lso(as, ARMI_LDR, RID_TMP, tab, (int32_t)offsetof(GCtab, hmask));
    } else if (irref_isk(refkey)) {
      emit_opk(as, ARMI_AND, tmp, RID_TMP, (int32_t)khash,
               rset_exclude(rset_exclude(RSET_GPR, tab), dest));
      emit_lso(as, ARMI_LDR, dest, tab, (int32_t)offsetof(GCtab, node));
      emit_lso(as, ARMI_LDR, RID_TMP, tab, (int32_t)offsetof(GCtab, hmask));
    } else {  /* Must match with hash*() in lj_tab.c. */
      if (ra_hasreg(keynumhi)) {  /* Canonicalize +-0.0 to 0.0. */
        if (keyhi == RID_TMP)
          emit_dm(as, ARMF_CC(ARMI_MOV, CC_NE), keyhi, keynumhi);
        emit_d(as, ARMF_CC(ARMI_MOV, CC_EQ)|ARMI_K12|0, keyhi);
      }
      emit_dnm(as, ARMI_AND, tmp, tmp, RID_TMP);
      emit_dnm(as, ARMI_SUB|ARMF_SH(ARMSH_ROR, 32-HASH_ROT3), tmp, tmp, tmp+1);
      emit_lso(as, ARMI_LDR, dest, tab, (int32_t)offsetof(GCtab, node));
      emit_dnm(as, ARMI_EOR|ARMF_SH(ARMSH_ROR, 32-((HASH_ROT2+HASH_ROT1)&31)),
               tmp, tmp+1, tmp);
      emit_lso(as, ARMI_LDR, RID_TMP, tab, (int32_t)offsetof(GCtab, hmask));
      emit_dnm(as, ARMI_SUB|ARMF_SH(ARMSH_ROR, 32-HASH_ROT1), tmp+1, tmp+1, tmp);
      if (ra_hasreg(keynumhi)) {
        emit_dnm(as, ARMI_EOR, tmp+1, tmp, key);
        emit_dnm(as, ARMI_ORR|ARMI_S, RID_TMP, tmp, key);  /* Test for +-0.0. */
        emit_dnm(as, ARMI_ADD, tmp, keynumhi, keynumhi);
#if !LJ_SOFTFP
        emit_dnm(as, ARMI_VMOV_RR_D, key, keynumhi,
                 (ra_alloc1(as, refkey, RSET_FPR) & 15));
#endif
      } else {
        emit_dnm(as, ARMI_EOR, tmp+1, tmp, key);
        emit_opk(as, ARMI_ADD, tmp, key, (int32_t)HASH_BIAS,
                 rset_exclude(rset_exclude(RSET_GPR, tab), key));
      }
    }
  }
}

static void asm_hrefk(ASMState *as, IRIns *ir)
{
  IRIns *kslot = IR(ir->op2);
  IRIns *irkey = IR(kslot->op1);
  int32_t ofs = (int32_t)(kslot->op2 * sizeof(Node));
  int32_t kofs = ofs + (int32_t)offsetof(Node, key);
```

```
  Reg dest = (ra_used(ir) || ofs > 4095) ? ra_dest(as, ir, RSET_GPR) : RID_NONE;
  Reg node = ra_alloc1(as, ir->op1, RSET_GPR);
  Reg key = RID_NONE, type = RID_TMP, idx = node;
  RegSet allow = rset_exclude(RSET_GPR, node);
  lua_assert(ofs % sizeof(Node) == 0);
  if (ofs > 4095) {
    idx = dest;
    rset_clear(allow, dest);
    kofs = (int32_t)offsetof(Node, key);
  } else if (ra_hasreg(dest)) {
    emit_opk(as, ARMI_ADD, dest, node, ofs, allow);
  }
  asm_guardcc(as, CC_NE);
  if (!irt_ispri(irkey->t)) {
    RegSet even = (as->freeset & allow);
    even = even & (even >> 1) & RSET_GPREVEN;
    if (even) {
      key = ra_scratch(as, even);
      if (rset_test(as->freeset, key+1)) {
        type = key+1;
        ra_modified(as, type);
      }
    } else {
      key = ra_scratch(as, allow);
    }
    rset_clear(allow, key);
  }
  rset_clear(allow, type);
  if (irt_isnum(irkey->t)) {
    emit_opk(as, ARMF_CC(ARMI_CMP, CC_EQ), 0, type,
             (int32_t)ir_knum(irkey)->u32.hi, allow);
    emit_opk(as, ARMI_CMP, 0, key,
             (int32_t)ir_knum(irkey)->u32.lo, allow);
  } else {
    if (ra_hasreg(key))
      emit_opk(as, ARMF_CC(ARMI_CMP, CC_EQ), 0, key, irkey->i, allow);
    emit_n(as, ARMI_CMN|ARMI_K12|-irt_toitype(irkey->t), type);
  }
  emit_lso(as, ARMI_LDR, type, idx, kofs+4);
  if (ra_hasreg(key)) emit_lso(as, ARMI_LDR, key, idx, kofs);
  if (ofs > 4095)
    emit_opk(as, ARMI_ADD, dest, node, ofs, RSET_GPR);
}

static void asm_uref(ASMState *as, IRIns *ir)
{
  /* NYI: Check that UREFO is still open and not aliasing a slot. */
  Reg dest = ra_dest(as, ir, RSET_GPR);
  if (irref_isk(ir->op1)) {
    GCfunc *fn = ir_kfunc(IR(ir->op1));
    MRef *v = &gcref(fn->l.uvptr[(ir->op2 >> 8)])->uv.v;
    emit_lsptr(as, ARMI_LDR, dest, v);
  } else {
    Reg uv = ra_scratch(as, RSET_GPR);
    Reg func = ra_alloc1(as, ir->op1, RSET_GPR);
    if (ir->o == IR_UREFC) {
      asm_guardcc(as, CC_NE);
      emit_n(as, ARMI_CMP|ARMI_K12|1, RID_TMP);
      emit_opk(as, ARMI_ADD, dest, uv,
               (int32_t)offsetof(GCupval, tv), RSET_GPR);
      emit_lso(as, ARMI_LDRB, RID_TMP, uv, (int32_t)offsetof(GCupval, closed));
    } else {
      emit_lso(as, ARMI_LDR, dest, uv, (int32_t)offsetof(GCupval, v));
    }
    emit_lso(as, ARMI_LDR, uv, func,
             (int32_t)offsetof(GCfuncL, uvptr) + 4*(int32_t)(ir->op2 >> 8));
  }
}

static void asm_fref(ASMState *as, IRIns *ir)
{
  UNUSED(as); UNUSED(ir);
  lua_assert(!ra_used(ir));
}

static void asm_strref(ASMState *as, IRIns *ir)
```

```
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  IRRef ref = ir->op2, refk = ir->op1;
  Reg r;
  if (irref_isk(ref)) {
    IRRef tmp = refk; refk = ref; ref = tmp;
  } else if (!irref_isk(refk)) {
    uint32_t k, m = ARMI_K12|sizeof(GCstr);
    Reg right, left = ra_alloc1(as, ir->op1, RSET_GPR);
    IRIns *irr = IR(ir->op2);
    if (ra_hasreg(irr->r)) {
      ra_noweak(as, irr->r);
      right = irr->r;
    } else if (mayfuse(as, irr->op2) &&
               irr->o == IR_ADD && irref_isk(irr->op2) &&
               (k = emit_isk12(ARMI_ADD,
                               (int32_t)sizeof(GCstr) + IR(irr->op2)->i))) {
      m = k;
      right = ra_alloc1(as, irr->op1, rset_exclude(RSET_GPR, left));
    } else {
      right = ra_allocref(as, ir->op2, rset_exclude(RSET_GPR, left));
    }
    emit_dn(as, ARMI_ADD^m, dest, dest);
    emit_dnm(as, ARMI_ADD, dest, left, right);
    return;
  }
  r = ra_alloc1(as, ref, RSET_GPR);
  emit_opk(as, ARMI_ADD, dest, r,
           sizeof(GCstr) + IR(refk)->i, rset_exclude(RSET_GPR, r));
}

/* -- Loads and stores ---------------------------------------------------- */

static ARMIns asm_fxloadins(IRIns *ir)
{
  switch (irt_type(ir->t)) {
  case IRT_I8: return ARMI_LDRSB;
  case IRT_U8: return ARMI_LDRB;
  case IRT_I16: return ARMI_LDRSH;
  case IRT_U16: return ARMI_LDRH;
  case IRT_NUM: lua_assert(!LJ_SOFTFP); return ARMI_VLDR_D;
  case IRT_FLOAT: if (!LJ_SOFTFP) return ARMI_VLDR_S;
  default: return ARMI_LDR;
  }
}

static ARMIns asm_fxstoreins(IRIns *ir)
{
  switch (irt_type(ir->t)) {
  case IRT_I8: case IRT_U8: return ARMI_STRB;
  case IRT_I16: case IRT_U16: return ARMI_STRH;
  case IRT_NUM: lua_assert(!LJ_SOFTFP); return ARMI_VSTR_D;
  case IRT_FLOAT: if (!LJ_SOFTFP) return ARMI_VSTR_S;
  default: return ARMI_STR;
  }
}

 static void asm_fload(ASMState *as, IRIns *ir)
 {
   Reg dest = ra_dest(as, ir, RSET_GPR);
   Reg idx = ra_alloc1(as, ir->op1, RSET_GPR);
   ARMIns ai = asm_fxloadins(ir);
   int32_t ofs;
   if (ir->op2 == IRFL_TAB_ARRAY) {
     ofs = asm_fuseabase(as, ir->op1);
     if (ofs) {  /* Turn the t->array load into an add for colocated arrays. */
       emit_dn(as, ARMI_ADD|ARMI_K12|ofs, dest, idx);
       return;
     }
   }
   ofs = field_ofs[ir->op2];
   if ((ai & 0x04000000))
     emit_lso(as, ai, dest, idx, ofs);
   else
     emit_lsox(as, ai, dest, idx, ofs);
 }
```

```
1020
1021 static void asm_fstore(ASMState *as, IRIns *ir)
1022 {
1023   if (ir->r != RID_SINK) {
1024     Reg src = ra_alloc1(as, ir->op2, RSET_GPR);
1025     IRIns *irf = IR(ir->op1);
1026     Reg idx = ra_alloc1(as, irf->op1, rset_exclude(RSET_GPR, src));
1027     int32_t ofs = field_ofs[irf->op2];
1028     ARMIns ai = asm_fxstoreins(ir);
1029     if ((ai & 0x04000000))
1030       emit_lso(as, ai, src, idx, ofs);
1031     else
1032       emit_lsox(as, ai, src, idx, ofs);
1033   }
1034 }
1035
1036 static void asm_xload(ASMState *as, IRIns *ir)
1037 {
1038   Reg dest = ra_dest(as, ir,
1039                      (!LJ_SOFTFP && irt_isfp(ir->t)) ? RSET_FPR : RSET_GPR);
1040   lua_assert(!(ir->op2 & IRXLOAD_UNALIGNED));
1041   asm_fusexref(as, asm_fxloadins(ir), dest, ir->op1, RSET_GPR, 0);
1042 }
1043
1044 static void asm_xstore_(ASMState *as, IRIns *ir, int32_t ofs)
1045 {
1046   if (ir->r != RID_SINK) {
1047     Reg src = ra_alloc1(as, ir->op2,
1048                         (!LJ_SOFTFP && irt_isfp(ir->t)) ? RSET_FPR : RSET_GPR);
1049     asm_fusexref(as, asm_fxstoreins(ir), src, ir->op1,
1050                  rset_exclude(RSET_GPR, src), ofs);
1051   }
1052 }
1053
1054 #define asm_xstore(as, ir)        asm_xstore_(as, ir, 0)
1055
1056 static void asm_ahuvload(ASMState *as, IRIns *ir)
1057 {
1058   int hiop = (LJ_SOFTFP && (ir+1)->o == IR_HIOP);
1059   IRType t = hiop ? IRT_NUM : irt_type(ir->t);
1060   Reg dest = RID_NONE, type = RID_NONE, idx;
1061   RegSet allow = RSET_GPR;
1062   int32_t ofs = 0;
1063   if (hiop && ra_used(ir+1)) {
1064     type = ra_dest(as, ir+1, allow);
1065     rset_clear(allow, type);
1066   }
1067   if (ra_used(ir)) {
1068     lua_assert((LJ_SOFTFP ? 0 : irt_isnum(ir->t)) ||
1069                irt_isint(ir->t) || irt_isaddr(ir->t));
1070     dest = ra_dest(as, ir, (!LJ_SOFTFP && t == IRT_NUM) ? RSET_FPR : allow);
1071     rset_clear(allow, dest);
1072   }
1073   idx = asm_fuseahuref(as, ir->op1, &ofs, allow,
1074                        (!LJ_SOFTFP && t == IRT_NUM) ? 1024 : 4096);
1075   if (!hiop || type == RID_NONE) {
1076     rset_clear(allow, idx);
1077     if (ofs < 256 && ra_hasreg(dest) && (dest & 1) == 0 &&
1078         rset_test((as->freeset & allow), dest+1)) {
1079       type = dest+1;
1080       ra_modified(as, type);
1081     } else {
1082       type = RID_TMP;
1083     }
1084   }
1085   asm_guardcc(as, t == IRT_NUM ? CC_HS : CC_NE);
1086   emit_n(as, ARMI_CMN|ARMI_K12|-irt_toitype_(t), type);
1087   if (ra_hasreg(dest)) {
1088 #if !LJ_SOFTFP
1089     if (t == IRT_NUM)
1090       emit_vlso(as, ARMI_VLDR_D, dest, idx, ofs);
1091     else
1092 #endif
1093       emit_lso(as, ARMI_LDR, dest, idx, ofs);
1094   }
1095   emit_lso(as, ARMI_LDR, type, idx, ofs+4);
```

```
}

static void asm_ahustore(ASMState *as, IRIns *ir)
{
  if (ir->r != RID_SINK) {
    RegSet allow = RSET_GPR;
    Reg idx, src = RID_NONE, type = RID_NONE;
    int32_t ofs = 0;
#if !LJ_SOFTFP
    if (irt_isnum(ir->t)) {
      src = ra_alloc1(as, ir->op2, RSET_FPR);
      idx = asm_fuseahuref(as, ir->op1, &ofs, allow, 1024);
      emit_vlso(as, ARMI_VSTR_D, src, idx, ofs);
    } else
#endif
    {
      int hiop = (LJ_SOFTFP && (ir+1)->o == IR_HIOP);
      if (!irt_ispri(ir->t)) {
        src = ra_alloc1(as, ir->op2, allow);
        rset_clear(allow, src);
      }
      if (hiop)
        type = ra_alloc1(as, (ir+1)->op2, allow);
      else
        type = ra_allock(as, (int32_t)irt_toitype(ir->t), allow);
      idx = asm_fuseahuref(as, ir->op1, &ofs, rset_exclude(allow, type), 4096);
      if (ra_hasreg(src)) emit_lso(as, ARMI_STR, src, idx, ofs);
      emit_lso(as, ARMI_STR, type, idx, ofs+4);
    }
  }
}

static void asm_sload(ASMState *as, IRIns *ir)
{
  int32_t ofs = 8*((int32_t)ir->op1-1) + ((ir->op2 & IRSLOAD_FRAME) ? 4 : 0);
  int hiop = (LJ_SOFTFP && (ir+1)->o == IR_HIOP);
  IRType t = hiop ? IRT_NUM : irt_type(ir->t);
  Reg dest = RID_NONE, type = RID_NONE, base;
  RegSet allow = RSET_GPR;
  lua_assert(!(ir->op2 & IRSLOAD_PARENT));  /* Handled by asm_head_side(). */
  lua_assert(irt_isguard(ir->t) || !(ir->op2 & IRSLOAD_TYPECHECK));
#if LJ_SOFTFP
  lua_assert(!(ir->op2 & IRSLOAD_CONVERT));  /* Handled by LJ_SOFTFP SPLIT. */
  if (hiop && ra_used(ir+1)) {
    type = ra_dest(as, ir+1, allow);
    rset_clear(allow, type);
  }
#else
  if ((ir->op2 & IRSLOAD_CONVERT) && irt_isguard(ir->t) && t == IRT_INT) {
    dest = ra_scratch(as, RSET_FPR);
    asm_tointg(as, ir, dest);
    t = IRT_NUM;  /* Continue with a regular number type check. */
  } else
#endif
  if (ra_used(ir)) {
    Reg tmp = RID_NONE;
    if ((ir->op2 & IRSLOAD_CONVERT))
      tmp = ra_scratch(as, t == IRT_INT ? RSET_FPR : RSET_GPR);
    lua_assert((LJ_SOFTFP ? 0 : irt_isnum(ir->t)) ||
               irt_isint(ir->t) || irt_isaddr(ir->t));
    dest = ra_dest(as, ir, (!LJ_SOFTFP && t == IRT_NUM) ? RSET_FPR : allow);
    rset_clear(allow, dest);
    base = ra_alloc1(as, REF_BASE, allow);
    if ((ir->op2 & IRSLOAD_CONVERT)) {
      if (t == IRT_INT) {
        emit_dn(as, ARMI_VMOV_R_S, dest, (tmp & 15));
        emit_dm(as, ARMI_VCVT_S32_F64, (tmp & 15), (tmp & 15));
        t = IRT_NUM;  /* Check for original type. */
      } else {
        emit_dm(as, ARMI_VCVT_F64_S32, (dest & 15), (dest & 15));
        emit_dn(as, ARMI_VMOV_S_R, tmp, (dest & 15));
        t = IRT_INT;  /* Check for original type. */
      }
      dest = tmp;
    }
    goto dotypecheck;
```

```
  }
  base = ra_alloc1(as, REF_BASE, allow);
dotypecheck:
  rset_clear(allow, base);
  if ((ir->op2 & IRSLOAD_TYPECHECK)) {
    if (ra_noreg(type)) {
      if (ofs < 256 && ra_hasreg(dest) && (dest & 1) == 0 &&
          rset_test((as->freeset & allow), dest+1)) {
        type = dest+1;
        ra_modified(as, type);
      } else {
        type = RID_TMP;
      }
    }
    asm_guardcc(as, t == IRT_NUM ? CC_HS : CC_NE);
    emit_n(as, ARMI_CMN|ARMI_K12|-irt_toitype_(t), type);
  }
  if (ra_hasreg(dest)) {
#if !LJ_SOFTFP
    if (t == IRT_NUM) {
      if (ofs < 1024) {
        emit_vlso(as, ARMI_VLDR_D, dest, base, ofs);
      } else {
        if (ra_hasreg(type)) emit_lso(as, ARMI_LDR, type, base, ofs+4);
        emit_vlso(as, ARMI_VLDR_D, dest, RID_TMP, 0);
        emit_opk(as, ARMI_ADD, RID_TMP, base, ofs, allow);
        return;
      }
    } else
#endif
      emit_lso(as, ARMI_LDR, dest, base, ofs);
  }
  if (ra_hasreg(type)) emit_lso(as, ARMI_LDR, type, base, ofs+4);
}

/* -- Allocations --------------------------------------------------------- */

#if LJ_HASFFI
static void asm_cnew(ASMState *as, IRIns *ir)
{
  CTState *cts = ctype_ctsG(J2G(as->J));
  CTypeID id = (CTypeID)IR(ir->op1)->i;
  CTSize sz;
  CTInfo info = lj_ctype_info(cts, id, &sz);
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_mem_newgco];
  IRRef args[4];
  RegSet allow = (RSET_GPR & ~RSET_SCRATCH);
  RegSet drop = RSET_SCRATCH;
  lua_assert(sz != CTSIZE_INVALID || (ir->o == IR_CNEW && ir->op2 != REF_NIL));

  as->gcsteps++;
  if (ra_hasreg(ir->r))
    rset_clear(drop, ir->r);  /* Dest reg handled below. */
  ra_evictset(as, drop);
  if (ra_used(ir))
    ra_destreg(as, ir, RID_RET);  /* GCcdata * */

  /* Initialize immutable cdata object. */
  if (ir->o == IR_CNEWI) {
    int32_t ofs = sizeof(GCcdata);
    lua_assert(sz == 4 || sz == 8);
    if (sz == 8) {
      ofs += 4; ir++;
      lua_assert(ir->o == IR_HIOP);
    }
    for (;;) {
      Reg r = ra_alloc1(as, ir->op2, allow);
      emit_lso(as, ARMI_STR, r, RID_RET, ofs);
      rset_clear(allow, r);
      if (ofs == sizeof(GCcdata)) break;
      ofs -= 4; ir--;
    }
  } else if (ir->op2 != REF_NIL) {  /* Create VLA/VLS/aligned cdata. */
    ci = &lj_ir_callinfo[IRCALL_lj_cdata_newv];
    args[0] = ASMREF_L;     /* lua_State *L */
    args[1] = ir->op1;      /* CTypeID id   */
```

```
    args[2] = ir->op2;      /* CTSize sz    */
    args[3] = ASMREF_TMP1;  /* CTSize align */
    asm_gencall(as, ci, args);
    emit_loadi(as, ra_releasetmp(as, ASMREF_TMP1), (int32_t)ctype_align(info));
    return;
  }

  /* Initialize gct and ctypeid. lj_mem_newgco() already sets marked. */
  {
    uint32_t k = emit_isk12(ARMI_MOV, id);
    Reg r = k ? RID_R1 : ra_allock(as, id, allow);
    emit_lso(as, ARMI_STRB, RID_TMP, RID_RET, offsetof(GCcdata, gct));
    emit_lsox(as, ARMI_STRH, r, RID_RET, offsetof(GCcdata, ctypeid));
    emit_d(as, ARMI_MOV|ARMI_K12|~LJ_TCDATA, RID_TMP);
    if (k) emit_d(as, ARMI_MOV^k, RID_R1);
  }
  args[0] = ASMREF_L;     /* lua_State *L */
  args[1] = ASMREF_TMP1;  /* MSize size   */
  asm_gencall(as, ci, args);
  ra_allockreg(as, (int32_t)(sz+sizeof(GCcdata)),
               ra_releasetmp(as, ASMREF_TMP1));
}
#else
#define asm_cnew(as, ir)        ((void)0)
#endif

/* -- Write barriers ----------------------------------------------------- */

static void asm_tbar(ASMState *as, IRIns *ir)
{
  Reg tab = ra_alloc1(as, ir->op1, RSET_GPR);
  Reg link = ra_scratch(as, rset_exclude(RSET_GPR, tab));
  Reg gr = ra_allock(as, i32ptr(J2G(as->J)),
                     rset_exclude(rset_exclude(RSET_GPR, tab), link));
  Reg mark = RID_TMP;
  MCLabel l_end = emit_label(as);
  emit_lso(as, ARMI_STR, link, tab, (int32_t)offsetof(GCtab, gclist));
  emit_lso(as, ARMI_STRB, mark, tab, (int32_t)offsetof(GCtab, marked));
  emit_lso(as, ARMI_STR, tab, gr,
           (int32_t)offsetof(global_State, gc.grayagain));
  emit_dn(as, ARMI_BIC|ARMI_K12|LJ_GC_BLACK, mark, mark);
  emit_lso(as, ARMI_LDR, link, gr,
           (int32_t)offsetof(global_State, gc.grayagain));
  emit_branch(as, ARMF_CC(ARMI_B, CC_EQ), l_end);
  emit_n(as, ARMI_TST|ARMI_K12|LJ_GC_BLACK, mark);
  emit_lso(as, ARMI_LDRB, mark, tab, (int32_t)offsetof(GCtab, marked));
}

static void asm_obar(ASMState *as, IRIns *ir)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_gc_barrieruv];
  IRRef args[2];
  MCLabel l_end;
  Reg obj, val, tmp;
  /* No need for other object barriers (yet). */
  lua_assert(IR(ir->op1)->o == IR_UREFC);
  ra_evictset(as, RSET_SCRATCH);
  l_end = emit_label(as);
  args[0] = ASMREF_TMP1;  /* global_State *g */
  args[1] = ir->op1;      /* TValue *tv      */
  asm_gencall(as, ci, args);
  if ((l_end[-1] >> 28) == CC_AL)
    l_end[-1] = ARMF_CC(l_end[-1], CC_NE);
  else
    emit_branch(as, ARMF_CC(ARMI_B, CC_EQ), l_end);
  ra_allockreg(as, i32ptr(J2G(as->J)), ra_releasetmp(as, ASMREF_TMP1));
  obj = IR(ir->op1)->r;
  tmp = ra_scratch(as, rset_exclude(RSET_GPR, obj));
  emit_n(as, ARMF_CC(ARMI_TST, CC_NE)|ARMI_K12|LJ_GC_BLACK, tmp);
  emit_n(as, ARMI_TST|ARMI_K12|LJ_GC_WHITES, RID_TMP);
  val = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, obj));
  emit_lso(as, ARMI_LDRB, tmp, obj,
           (int32_t)offsetof(GCupval, marked)-(int32_t)offsetof(GCupval, tv));
  emit_lso(as, ARMI_LDRB, RID_TMP, val, (int32_t)offsetof(GChead, marked));
}

```

```
/* -- Arithmetic and logic operations ------------------------------------ */

#if !LJ_SOFTFP
static void asm_fparith(ASMState *as, IRIns *ir, ARMIns ai)
{
  Reg dest = ra_dest(as, ir, RSET_FPR);
  Reg right, left = ra_alloc2(as, ir, RSET_FPR);
  right = (left >> 8); left &= 255;
  emit_dnm(as, ai, (dest & 15), (left & 15), (right & 15));
}

static void asm_fpunary(ASMState *as, IRIns *ir, ARMIns ai)
{
  Reg dest = ra_dest(as, ir, RSET_FPR);
  Reg left = ra_hintalloc(as, ir->op1, dest, RSET_FPR);
  emit_dm(as, ai, (dest & 15), (left & 15));
}

static void asm_callround(ASMState *as, IRIns *ir, int id)
{
  /* The modified regs must match with the *.dasc implementation. */
  RegSet drop = RID2RSET(RID_R0)|RID2RSET(RID_R1)|RID2RSET(RID_R2)|
                RID2RSET(RID_R3)|RID2RSET(RID_R12);
  RegSet of;
  Reg dest, src;
  ra_evictset(as, drop);
  dest = ra_dest(as, ir, RSET_FPR);
  emit_dnm(as, ARMI_VMOV_D_RR, RID_RETLO, RID_RETHI, (dest & 15));
  emit_call(as, id == IRFPM_FLOOR ? (void *)lj_vm_floor_sf :
                id == IRFPM_CEIL ? (void *)lj_vm_ceil_sf :
                                   (void *)lj_vm_trunc_sf);
  /* Workaround to protect argument GPRs from being used for remat. */
  of = as->freeset;
  as->freeset &= ~RSET_RANGE(RID_R0, RID_R1+1);
  as->cost[RID_R0] = as->cost[RID_R1] = REGCOST(~0u, ASMREF_L);
  src = ra_alloc1(as, ir->op1, RSET_FPR);  /* May alloc GPR to remat FPR. */
  as->freeset |= (of & RSET_RANGE(RID_R0, RID_R1+1));
  emit_dnm(as, ARMI_VMOV_RR_D, RID_R0, RID_R1, (src & 15));
}

static void asm_fpmath(ASMState *as, IRIns *ir)
{
  if (ir->op2 == IRFPM_EXP2 && asm_fpjoin_pow(as, ir))
    return;
  if (ir->op2 <= IRFPM_TRUNC)
    asm_callround(as, ir, ir->op2);
  else if (ir->op2 == IRFPM_SQRT)
    asm_fpunary(as, ir, ARMI_VSQRT_D);
  else
    asm_callid(as, ir, IRCALL_lj_vm_floor + ir->op2);
}
#else
#define asm_fpmath(as, ir)        lua_assert(0)
#endif

static int asm_swapops(ASMState *as, IRRef lref, IRRef rref)
{
  IRIns *ir;
  if (irref_isk(rref))
    return 0;  /* Don't swap constants to the left. */
  if (irref_isk(lref))
    return 1;  /* But swap constants to the right. */
  ir = IR(rref);
  if ((ir->o >= IR_BSHL && ir->o <= IR_BROR) ||
      (ir->o == IR_ADD && ir->op1 == ir->op2))
    return 0;  /* Don't swap fusable operands to the left. */
  ir = IR(lref);
  if ((ir->o >= IR_BSHL && ir->o <= IR_BROR) ||
      (ir->o == IR_ADD && ir->op1 == ir->op2))
    return 1;  /* But swap fusable operands to the right. */
  return 0;  /* Otherwise don't swap. */
}

static void asm_intop(ASMState *as, IRIns *ir, ARMIns ai)
{
  IRRef lref = ir->op1, rref = ir->op2;
```

```
  Reg left, dest = ra_dest(as, ir, RSET_GPR);
  uint32_t m;
  if (asm_swapops(as, lref, rref)) {
    IRRef tmp = lref; lref = rref; rref = tmp;
    if ((ai & ~ARMI_S) == ARMI_SUB || (ai & ~ARMI_S) == ARMI_SBC)
      ai ^= (ARMI_SUB^ARMI_RSB);
  }
  left = ra_hintalloc(as, lref, dest, RSET_GPR);
  m = asm_fuseopm(as, ai, rref, rset_exclude(RSET_GPR, left));
  if (irt_isguard(ir->t)) {  /* For IR_ADDOV etc. */
    asm_guardcc(as, CC_VS);
    ai |= ARMI_S;
  }
  emit_dn(as, ai^m, dest, left);
}

static void asm_intop_s(ASMState *as, IRIns *ir, ARMIns ai)
{
  if (as->flagmcp == as->mcp) {  /* Drop cmp r, #0. */
    as->flagmcp = NULL;
    as->mcp++;
    ai |= ARMI_S;
  }
  asm_intop(as, ir, ai);
}

static void asm_intneg(ASMState *as, IRIns *ir, ARMIns ai)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg left = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
  emit_dn(as, ai|ARMI_K12|0, dest, left);
}

/* NYI: use add/shift for MUL(OV) with constants. FOLD only does 2^k. */
static void asm_intmul(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg left = ra_alloc1(as, ir->op1, rset_exclude(RSET_GPR, dest));
  Reg right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
  Reg tmp = RID_NONE;
  /* ARMv5 restriction: dest != left and dest_hi != left. */
  if (dest == left && left != right) { left = right; right = dest; }
  if (irt_isguard(ir->t)) {  /* IR_MULOV */
    if (!(as->flags & JIT_F_ARMV6) && dest == left)
      tmp = left = ra_scratch(as, rset_exclude(RSET_GPR, left));
    asm_guardcc(as, CC_NE);
    emit_nm(as, ARMI_TEQ|ARMF_SH(ARMSH_ASR, 31), RID_TMP, dest);
    emit_dnm(as, ARMI_SMULL|ARMF_S(right), dest, RID_TMP, left);
  } else {
    if (!(as->flags & JIT_F_ARMV6) && dest == left) tmp = left = RID_TMP;
    emit_nm(as, ARMI_MUL|ARMF_S(right), dest, left);
  }
  /* Only need this for the dest == left == right case. */
  if (ra_hasreg(tmp)) emit_dm(as, ARMI_MOV, tmp, right);
}

static void asm_add(ASMState *as, IRIns *ir)
{
#if !LJ_SOFTFP
  if (irt_isnum(ir->t)) {
    if (!asm_fusemadd(as, ir, ARMI_VMLA_D, ARMI_VMLA_D))
      asm_fparith(as, ir, ARMI_VADD_D);
    return;
  }
#endif
  asm_intop_s(as, ir, ARMI_ADD);
}

static void asm_sub(ASMState *as, IRIns *ir)
{
#if !LJ_SOFTFP
  if (irt_isnum(ir->t)) {
    if (!asm_fusemadd(as, ir, ARMI_VNMLS_D, ARMI_VMLS_D))
      asm_fparith(as, ir, ARMI_VSUB_D);
    return;
  }
```

```
#endif
  asm_intop_s(as, ir, ARMI_SUB);
}

static void asm_mul(ASMState *as, IRIns *ir)
{
#if !LJ_SOFTFP
  if (irt_isnum(ir->t)) {
    asm_fparith(as, ir, ARMI_VMUL_D);
    return;
  }
#endif
  asm_intmul(as, ir);
}

#define asm_addov(as, ir)        asm_add(as, ir)
#define asm_subov(as, ir)        asm_sub(as, ir)
#define asm_mulov(as, ir)        asm_mul(as, ir)

#if LJ_SOFTFP
#define asm_div(as, ir)                lua_assert(0)
#define asm_pow(as, ir)                lua_assert(0)
#define asm_abs(as, ir)                lua_assert(0)
#define asm_atan2(as, ir)        lua_assert(0)
#define asm_ldexp(as, ir)        lua_assert(0)
#else
#define asm_div(as, ir)                asm_fparith(as, ir, ARMI_VDIV_D)
#define asm_pow(as, ir)                asm_callid(as, ir, IRCALL_lj_vm_powi)
#define asm_abs(as, ir)                asm_fpunary(as, ir, ARMI_VABS_D)
#define asm_atan2(as, ir)        asm_callid(as, ir, IRCALL_atan2)
#define asm_ldexp(as, ir)        asm_callid(as, ir, IRCALL_ldexp)
#endif

#define asm_mod(as, ir)                asm_callid(as, ir, IRCALL_lj_vm_modi)

static void asm_neg(ASMState *as, IRIns *ir)
{
#if !LJ_SOFTFP
  if (irt_isnum(ir->t)) {
    asm_fpunary(as, ir, ARMI_VNEG_D);
    return;
  }
#endif
  asm_intneg(as, ir, ARMI_RSB);
}

static void asm_bitop(ASMState *as, IRIns *ir, ARMIns ai)
{
  if (as->flagmcp == as->mcp) {  /* Try to drop cmp r, #0. */
    uint32_t cc = (as->mcp[1] >> 28);
    as->flagmcp = NULL;
    if (cc <= CC_NE) {
      as->mcp++;
      ai |= ARMI_S;
    } else if (cc == CC_GE) {
      *++as->mcp ^= ((CC_GE^CC_PL) << 28);
      ai |= ARMI_S;
    } else if (cc == CC_LT) {
      *++as->mcp ^= ((CC_LT^CC_MI) << 28);
      ai |= ARMI_S;
    }  /* else: other conds don't work with bit ops. */
  }
  if (ir->op2 == 0) {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    uint32_t m = asm_fuseopm(as, ai, ir->op1, RSET_GPR);
    emit_d(as, ai^m, dest);
  } else {
    /* NYI: Turn BAND !k12 into uxtb, uxth or bfc or shl+shr. */
    asm_intop(as, ir, ai);
  }
}

#define asm_bnot(as, ir)        asm_bitop(as, ir, ARMI_MVN)

static void asm_bswap(ASMState *as, IRIns *ir)
{
```

```
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg left = ra_alloc1(as, ir->op1, RSET_GPR);
  if ((as->flags & JIT_F_ARMV6)) {
    emit_dm(as, ARMI_REV, dest, left);
  } else {
    Reg tmp2 = dest;
    if (tmp2 == left)
      tmp2 = ra_scratch(as, rset_exclude(rset_exclude(RSET_GPR, dest), left));
    emit_dnm(as, ARMI_EOR|ARMF_SH(ARMSH_LSR, 8), dest, tmp2, RID_TMP);
    emit_dm(as, ARMI_MOV|ARMF_SH(ARMSH_ROR, 8), tmp2, left);
    emit_dn(as, ARMI_BIC|ARMI_K12|256*8|255, RID_TMP, RID_TMP);
    emit_dnm(as, ARMI_EOR|ARMF_SH(ARMSH_ROR, 16), RID_TMP, left, left);
  }
}

#define asm_band(as, ir)        asm_bitop(as, ir, ARMI_AND)
#define asm_bor(as, ir)                asm_bitop(as, ir, ARMI_ORR)
#define asm_bxor(as, ir)        asm_bitop(as, ir, ARMI_EOR)

static void asm_bitshift(ASMState *as, IRIns *ir, ARMShift sh)
{
  if (irref_isk(ir->op2)) {  /* Constant shifts. */
    /* NYI: Turn SHL+SHR or BAND+SHR into uxtb, uxth or ubfx. */
    /* NYI: Turn SHL+ASR into sxtb, sxth or sbfx. */
    Reg dest = ra_dest(as, ir, RSET_GPR);
    Reg left = ra_alloc1(as, ir->op1, RSET_GPR);
    int32_t shift = (IR(ir->op2)->i & 31);
    emit_dm(as, ARMI_MOV|ARMF_SH(sh, shift), dest, left);
  } else {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    Reg left = ra_alloc1(as, ir->op1, RSET_GPR);
    Reg right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
    emit_dm(as, ARMI_MOV|ARMF_RSH(sh, right), dest, left);
  }
}

#define asm_bshl(as, ir)        asm_bitshift(as, ir, ARMSH_LSL)
#define asm_bshr(as, ir)        asm_bitshift(as, ir, ARMSH_LSR)
#define asm_bsar(as, ir)        asm_bitshift(as, ir, ARMSH_ASR)
#define asm_bror(as, ir)        asm_bitshift(as, ir, ARMSH_ROR)
#define asm_brol(as, ir)        lua_assert(0)

static void asm_intmin_max(ASMState *as, IRIns *ir, int cc)
{
  uint32_t kcmp = 0, kmov = 0;
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg left = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
  Reg right = 0;
  if (irref_isk(ir->op2)) {
    kcmp = emit_isk12(ARMI_CMP, IR(ir->op2)->i);
    if (kcmp) kmov = emit_isk12(ARMI_MOV, IR(ir->op2)->i);
  }
  if (!kmov) {
    kcmp = 0;
    right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
  }
  if (kmov || dest != right) {
    emit_dm(as, ARMF_CC(ARMI_MOV, cc)^kmov, dest, right);
    cc ^= 1;  /* Must use opposite conditions for paired moves. */
  } else {
    cc ^= (CC_LT^CC_GT);  /* Otherwise may swap CC_LT <-> CC_GT. */
  }
  if (dest != left) emit_dm(as, ARMF_CC(ARMI_MOV, cc), dest, left);
  emit_nm(as, ARMI_CMP^kcmp, left, right);
}

#if LJ_SOFTFP
static void asm_sfpmin_max(ASMState *as, IRIns *ir, int cc)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_softfp_cmp];
  RegSet drop = RSET_SCRATCH;
  Reg r;
  IRRef args[4];
  args[0] = ir->op1; args[1] = (ir+1)->op1;
  args[2] = ir->op2; args[3] = (ir+1)->op2;
  /* __aeabi_cdcmple preserves r0-r3. */
```

```
  if (ra_hasreg(ir->r)) rset_clear(drop, ir->r);
  if (ra_hasreg((ir+1)->r)) rset_clear(drop, (ir+1)->r);
  if (!rset_test(as->freeset, RID_R2) &&
      regcost_ref(as->cost[RID_R2]) == args[2]) rset_clear(drop, RID_R2);
  if (!rset_test(as->freeset, RID_R3) &&
      regcost_ref(as->cost[RID_R3]) == args[3]) rset_clear(drop, RID_R3);
  ra_evictset(as, drop);
  ra_destpair(as, ir);
  emit_dm(as, ARMF_CC(ARMI_MOV, cc), RID_RETHI, RID_R3);
  emit_dm(as, ARMF_CC(ARMI_MOV, cc), RID_RETLO, RID_R2);
  emit_call(as, (void *)ci->func);
  for (r = RID_R0; r <= RID_R3; r++)
    ra_leftov(as, r, args[r-RID_R0]);
}
#else
static void asm_fpmin_max(ASMState *as, IRIns *ir, int cc)
{
  Reg dest = (ra_dest(as, ir, RSET_FPR) & 15);
  Reg right, left = ra_alloc2(as, ir, RSET_FPR);
  right = ((left >> 8) & 15); left &= 15;
  if (dest != left) emit_dm(as, ARMF_CC(ARMI_VMOV_D, cc^1), dest, left);
  if (dest != right) emit_dm(as, ARMF_CC(ARMI_VMOV_D, cc), dest, right);
  emit_d(as, ARMI_VMRS, 0);
  emit_dm(as, ARMI_VCMP_D, left, right);
}
#endif

static void asm_min_max(ASMState *as, IRIns *ir, int cc, int fcc)
{
#if LJ_SOFTFP
  UNUSED(fcc);
#else
  if (irt_isnum(ir->t))
    asm_fpmin_max(as, ir, fcc);
  else
#endif
    asm_intmin_max(as, ir, cc);
}

#define asm_min(as, ir)                asm_min_max(as, ir, CC_GT, CC_HI)
#define asm_max(as, ir)                asm_min_max(as, ir, CC_LT, CC_LO)

/* -- Comparisons ---------------------------------------------------------- */

/* Map of comparisons to flags. ORDER IR. */
static const uint8_t asm_compmap[IR_ABC+1] = {
  /* op  FP swp  int cc   FP cc */
  /* LT       */ CC_GE + (CC_HS << 4),
  /* GE    x  */ CC_LT + (CC_HI << 4),
  /* LE       */ CC_GT + (CC_HI << 4),
  /* GT    x  */ CC_LE + (CC_HS << 4),
  /* ULT   x  */ CC_HS + (CC_LS << 4),
  /* UGE      */ CC_LO + (CC_LO << 4),
  /* ULE   x  */ CC_HI + (CC_LO << 4),
  /* UGT      */ CC_LS + (CC_LS << 4),
  /* EQ       */ CC_NE + (CC_NE << 4),
  /* NE       */ CC_EQ + (CC_EQ << 4),
  /* ABC      */ CC_LS + (CC_LS << 4)  /* Same as UGT. */
};

#if LJ_SOFTFP
/* FP comparisons. */
static void asm_sfpcomp(ASMState *as, IRIns *ir)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_softfp_cmp];
  RegSet drop = RSET_SCRATCH;
  Reg r;
  IRRef args[4];
  int swp = (((ir->o ^ (ir->o >> 2)) & ~(ir->o >> 3) & 1) << 1);
  args[swp^0] = ir->op1; args[swp^1] = (ir+1)->op1;
  args[swp^2] = ir->op2; args[swp^3] = (ir+1)->op2;
  /* __aeabi_cdcmple preserves r0-r3. This helps to reduce spills. */
  for (r = RID_R0; r <= RID_R3; r++)
    if (!rset_test(as->freeset, r) &&
        regcost_ref(as->cost[r]) == args[r-RID_R0]) rset_clear(drop, r);
  ra_evictset(as, drop);
```

```
  asm_guardcc(as, (asm_compmap[ir->o] >> 4));
  emit_call(as, (void *)ci->func);
  for (r = RID_R0; r <= RID_R3; r++)
    ra_leftov(as, r, args[r-RID_R0]);
}
#else
/* FP comparisons. */
static void asm_fpcomp(ASMState *as, IRIns *ir)
{
  Reg left, right;
  ARMIns ai;
  int swp = ((ir->o ^ (ir->o >> 2)) & ~(ir->o >> 3) & 1);
  if (!swp && irref_isk(ir->op2) && ir_knum(IR(ir->op2))->u64 == 0) {
    left = (ra_alloc1(as, ir->op1, RSET_FPR) & 15);
    right = 0;
    ai = ARMI_VCMPZ_D;
  } else {
    left = ra_alloc2(as, ir, RSET_FPR);
    if (swp) {
      right = (left & 15); left = ((left >> 8) & 15);
    } else {
      right = ((left >> 8) & 15); left &= 15;
    }
    ai = ARMI_VCMP_D;
  }
  asm_guardcc(as, (asm_compmap[ir->o] >> 4));
  emit_d(as, ARMI_VMRS, 0);
  emit_dm(as, ai, left, right);
}
#endif

/* Integer comparisons. */
static void asm_intcomp(ASMState *as, IRIns *ir)
{
  ARMCC cc = (asm_compmap[ir->o] & 15);
  IRRef lref = ir->op1, rref = ir->op2;
  Reg left;
  uint32_t m;
  int cmpprev0 = 0;
  lua_assert(irt_isint(ir->t) || irt_isu32(ir->t) || irt_isaddr(ir->t));
  if (asm_swapops(as, lref, rref)) {
    Reg tmp = lref; lref = rref; rref = tmp;
    if (cc >= CC_GE) cc ^= 7;  /* LT <-> GT, LE <-> GE */
    else if (cc > CC_NE) cc ^= 11;  /* LO <-> HI, LS <-> HS */
  }
  if (irref_isk(rref) && IR(rref)->i == 0) {
    IRIns *irl = IR(lref);
    cmpprev0 = (irl+1 == ir);
    /* Combine comp(BAND(left, right), 0) into tst left, right. */
    if (cmpprev0 && irl->o == IR_BAND && !ra_used(irl)) {
      IRRef blref = irl->op1, brref = irl->op2;
      uint32_t m2 = 0;
      Reg bleft;
      if (asm_swapops(as, blref, brref)) {
        Reg tmp = blref; blref = brref; brref = tmp;
      }
      if (irref_isk(brref)) {
        m2 = emit_isk12(ARMI_AND, IR(brref)->i);
        if ((m2 & (ARMI_AND^ARMI_BIC)))
          goto notst;  /* Not beneficial if we miss a constant operand. */
      }
      if (cc == CC_GE) cc = CC_PL;
      else if (cc == CC_LT) cc = CC_MI;
      else if (cc > CC_NE) goto notst;  /* Other conds don't work with tst. */
      bleft = ra_alloc1(as, blref, RSET_GPR);
      if (!m2) m2 = asm_fuseopm(as, 0, brref, rset_exclude(RSET_GPR, bleft));
      asm_guardcc(as, cc);
      emit_n(as, ARMI_TST^m2, bleft);
      return;
    }
  }
notst:
  left = ra_alloc1(as, lref, RSET_GPR);
  m = asm_fuseopm(as, ARMI_CMP, rref, rset_exclude(RSET_GPR, left));
  asm_guardcc(as, cc);
  emit_n(as, ARMI_CMP^m, left);
```

```
  /* Signed comparison with zero and referencing previous ins? */
  if (cmpprev0 && (cc <= CC_NE || cc >= CC_GE))
    as->flagmcp = as->mcp;  /* Allow elimination of the compare. */
}

static void asm_comp(ASMState *as, IRIns *ir)
{
#if !LJ_SOFTFP
  if (irt_isnum(ir->t))
    asm_fpcomp(as, ir);
  else
#endif
    asm_intcomp(as, ir);
}

#define asm_equal(as, ir)        asm_comp(as, ir)

#if LJ_HASFFI
/* 64 bit integer comparisons. */
static void asm_int64comp(ASMState *as, IRIns *ir)
{
  int signedcomp = (ir->o <= IR_GT);
  ARMCC cclo, cchi;
  Reg leftlo, lefthi;
  uint32_t mlo, mhi;
  RegSet allow = RSET_GPR, oldfree;

  /* Always use unsigned comparison for loword. */
  cclo = asm_compmap[ir->o + (signedcomp ? 4 : 0)] & 15;
  leftlo = ra_alloc1(as, ir->op1, allow);
  oldfree = as->freeset;
  mlo = asm_fuseopm(as, ARMI_CMP, ir->op2, rset_clear(allow, leftlo));
  allow &= ~(oldfree & ~as->freeset);  /* Update for allocs of asm_fuseopm. */

  /* Use signed or unsigned comparison for hiword. */
  cchi = asm_compmap[ir->o] & 15;
  lefthi = ra_alloc1(as, (ir+1)->op1, allow);
  mhi = asm_fuseopm(as, ARMI_CMP, (ir+1)->op2, rset_clear(allow, lefthi));

  /* All register allocations must be performed _before_ this point. */
  if (signedcomp) {
    MCLabel l_around = emit_label(as);
    asm_guardcc(as, cclo);
    emit_n(as, ARMI_CMP^mlo, leftlo);
    emit_branch(as, ARMF_CC(ARMI_B, CC_NE), l_around);
    if (cchi == CC_GE || cchi == CC_LE) cchi ^= 6;  /* GE -> GT, LE -> LT */
    asm_guardcc(as, cchi);
  } else {
    asm_guardcc(as, cclo);
    emit_n(as, ARMF_CC(ARMI_CMP, CC_EQ)^mlo, leftlo);
  }
  emit_n(as, ARMI_CMP^mhi, lefthi);
}
#endif

/* -- Support for 64 bit ops in 32 bit mode ------------------------------ */

/* Hiword op of a split 64 bit op. Previous op must be the loword op. */
static void asm_hiop(ASMState *as, IRIns *ir)
{
#if LJ_HASFFI || LJ_SOFTFP
  /* HIOP is marked as a store because it needs its own DCE logic. */
  int uselo = ra_used(ir-1), usehi = ra_used(ir);  /* Loword/hiword used? */
  if (LJ_UNLIKELY(!(as->flags & JIT_F_OPT_DCE))) uselo = usehi = 1;
  if ((ir-1)->o <= IR_NE) {  /* 64 bit integer or FP comparisons. ORDER IR. */
    as->curins--;  /* Always skip the loword comparison. */
#if LJ_SOFTFP
    if (!irt_isint(ir->t)) {
      asm_sfpcomp(as, ir-1);
      return;
    }
#endif
#if LJ_HASFFI
    asm_int64comp(as, ir-1);
#endif
    return;
```

```
#if LJ_SOFTFP
  } else if ((ir-1)->o == IR_MIN || (ir-1)->o == IR_MAX) {
    as->curins--;  /* Always skip the loword min/max. */
    if (uselo || usehi)
      asm_sfpmin_max(as, ir-1, (ir-1)->o == IR_MIN ? CC_HI : CC_LO);
    return;
#elif LJ_HASFFI
  } else if ((ir-1)->o == IR_CONV) {
    as->curins--;  /* Always skip the CONV. */
    if (usehi || uselo)
      asm_conv64(as, ir);
    return;
#endif
  } else if ((ir-1)->o == IR_XSTORE) {
    if ((ir-1)->r != RID_SINK)
      asm_xstore_(as, ir, 4);
    return;
  }
  if (!usehi) return;  /* Skip unused hiword op for all remaining ops. */
  switch ((ir-1)->o) {
#if LJ_HASFFI
  case IR_ADD:
    as->curins--;
    asm_intop(as, ir, ARMI_ADC);
    asm_intop(as, ir-1, ARMI_ADD|ARMI_S);
    break;
  case IR_SUB:
    as->curins--;
    asm_intop(as, ir, ARMI_SBC);
    asm_intop(as, ir-1, ARMI_SUB|ARMI_S);
    break;
  case IR_NEG:
    as->curins--;
    asm_intneg(as, ir, ARMI_RSC);
    asm_intneg(as, ir-1, ARMI_RSB|ARMI_S);
    break;
#endif
#if LJ_SOFTFP
  case IR_SLOAD: case IR_ALOAD: case IR_HLOAD: case IR_ULOAD: case IR_VLOAD:
  case IR_STRTO:
    if (!uselo)
      ra_allocref(as, ir->op1, RSET_GPR);  /* Mark lo op as used. */
    break;
#endif
  case IR_CALLN:
  case IR_CALLS:
  case IR_CALLXS:
    if (!uselo)
      ra_allocref(as, ir->op1, RID2RSET(RID_RETLO));  /* Mark lo op as used. */
    break;
#if LJ_SOFTFP
  case IR_ASTORE: case IR_HSTORE: case IR_USTORE: case IR_TOSTR:
#endif
  case IR_CNEWI:
    /* Nothing to do here. Handled by lo op itself. */
    break;
  default: lua_assert(0); break;
  }
#else
  UNUSED(as); UNUSED(ir); lua_assert(0);
#endif
}

/* -- Profiling ----------------------------------------------------------- */

static void asm_prof(ASMState *as, IRIns *ir)
{
  UNUSED(ir);
  asm_guardcc(as, CC_NE);
  emit_n(as, ARMI_TST|ARMI_K12|HOOK_PROFILE, RID_TMP);
  emit_lsptr(as, ARMI_LDRB, RID_TMP, (void *)&J2G(as->J)->hookmask);
}

/* -- Stack handling ------------------------------------------------------ */

/* Check Lua stack size for overflow. Use exit handler as fallback. */
```

```
1932 static void asm_stack_check(ASMState *as, BCReg topslot,
1933                            IRIns *irp, RegSet allow, ExitNo exitno)
1934 {
1935   Reg pbase;
1936   uint32_t k;
1937   if (irp) {
1938     if (!ra_hasspill(irp->s)) {
1939       pbase = irp->r;
1940       lua_assert(ra_hasreg(pbase));
1941     } else if (allow) {
1942       pbase = rset_pickbot(allow);
1943     } else {
1944       pbase = RID_RET;
1945       emit_lso(as, ARMI_LDR, RID_RET, RID_SP, 0);  /* Restore temp. register. */
1946     }
1947   } else {
1948     pbase = RID_BASE;
1949   }
1950   emit_branch(as, ARMF_CC(ARMI_BL, CC_LS), exitstub_addr(as->J, exitno));
1951   k = emit_isk12(0, (int32_t)(8*topslot));
1952   lua_assert(k);
1953   emit_n(as, ARMI_CMP^k, RID_TMP);
1954   emit_dnm(as, ARMI_SUB, RID_TMP, RID_TMP, pbase);
1955   emit_lso(as, ARMI_LDR, RID_TMP, RID_TMP,
1956            (int32_t)offsetof(lua_State, maxstack));
1957   if (irp) {  /* Must not spill arbitrary registers in head of side trace. */
1958     int32_t i = i32ptr(&J2G(as->J)->cur_L);
1959     if (ra_hasspill(irp->s))
1960       emit_lso(as, ARMI_LDR, pbase, RID_SP, sps_scale(irp->s));
1961     emit_lso(as, ARMI_LDR, RID_TMP, RID_TMP, (i & 4095));
1962     if (ra_hasspill(irp->s) && !allow)
1963       emit_lso(as, ARMI_STR, RID_RET, RID_SP, 0);  /* Save temp. register. */
1964     emit_loadi(as, RID_TMP, (i & ~4095));
1965   } else {
1966     emit_getgl(as, RID_TMP, cur_L);
1967   }
1968 }
1969
1970 /* Restore Lua stack from on-trace state. */
1971 static void asm_stack_restore(ASMState *as, SnapShot *snap)
1972 {
1973   SnapEntry *map = &as->T->snapmap[snap->mapofs];
1974   SnapEntry *flinks = &as->T->snapmap[snap_nextofs(as->T, snap)-1];
1975   MSize n, nent = snap->nent;
1976   /* Store the value of all modified slots to the Lua stack. */
1977   for (n = 0; n < nent; n++) {
1978     SnapEntry sn = map[n];
1979     BCReg s = snap_slot(sn);
1980     int32_t ofs = 8*((int32_t)s-1);
1981     IRRef ref = snap_ref(sn);
1982     IRIns *ir = IR(ref);
1983     if ((sn & SNAP_NORESTORE))
1984       continue;
1985     if (irt_isnum(ir->t)) {
1986 #if LJ_SOFTFP
1987       RegSet odd = rset_exclude(RSET_GPRODD, RID_BASE);
1988       Reg tmp;
1989       lua_assert(irref_isk(ref));  /* LJ_SOFTFP: must be a number constant. */
1990       tmp = ra_allock(as, (int32_t)ir_knum(ir)->u32.lo,
1991                       rset_exclude(RSET_GPREVEN, RID_BASE));
1992       emit_lso(as, ARMI_STR, tmp, RID_BASE, ofs);
1993       if (rset_test(as->freeset, tmp+1)) odd = RID2RSET(tmp+1);
1994       tmp = ra_allock(as, (int32_t)ir_knum(ir)->u32.hi, odd);
1995       emit_lso(as, ARMI_STR, tmp, RID_BASE, ofs+4);
1996 #else
1997       Reg src = ra_alloc1(as, ref, RSET_FPR);
1998       emit_vlso(as, ARMI_VSTR_D, src, RID_BASE, ofs);
1999 #endif
2000     } else {
2001       RegSet odd = rset_exclude(RSET_GPRODD, RID_BASE);
2002       Reg type;
2003       lua_assert(irt_ispri(ir->t) || irt_isaddr(ir->t) || irt_isinteger(ir->t));
2004       if (!irt_ispri(ir->t)) {
2005         Reg src = ra_alloc1(as, ref, rset_exclude(RSET_GPREVEN, RID_BASE));
2006         emit_lso(as, ARMI_STR, src, RID_BASE, ofs);
2007         if (rset_test(as->freeset, src+1)) odd = RID2RSET(src+1);
```

```
2008        }
2009        if ((sn & (SNAP_CONT|SNAP_FRAME))) {
2010          if (s == 0) continue;  /* Do not overwrite link to previous frame. */
2011          type = ra_allock(as, (int32_t)(*flinks--), odd);
2012  #if LJ_SOFTFP
2013        } else if ((sn & SNAP_SOFTFPNUM)) {
2014          type = ra_alloc1(as, ref+1, rset_exclude(RSET_GPRODD, RID_BASE));
2015  #endif
2016        } else {
2017          type = ra_allock(as, (int32_t)irt_toitype(ir->t), odd);
2018        }
2019        emit_lso(as, ARMI_STR, type, RID_BASE, ofs+4);
2020      }
2021      checkmclim(as);
2022    }
2023    lua_assert(map + nent == flinks);
2024  }
2025
2026  /* -- GC handling ---------------------------------------------------------- */
2027
2028  /* Check GC threshold and do one or more GC steps. */
2029  static void asm_gc_check(ASMState *as)
2030  {
2031    const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_gc_step_jit];
2032    IRRef args[2];
2033    MCLabel l_end;
2034    Reg tmp1, tmp2;
2035    ra_evictset(as, RSET_SCRATCH);
2036    l_end = emit_label(as);
2037    /* Exit trace if in GCSatomic or GCSfinalize. Avoids syncing GC objects. */
2038    asm_guardcc(as, CC_NE);  /* Assumes asm_snap_prep() already done. */
2039    emit_n(as, ARMI_CMP|ARMI_K12|0, RID_RET);
2040    args[0] = ASMREF_TMP1;  /* global_State *g */
2041    args[1] = ASMREF_TMP2;  /* MSize steps     */
2042    asm_gencall(as, ci, args);
2043    tmp1 = ra_releasetmp(as, ASMREF_TMP1);
2044    tmp2 = ra_releasetmp(as, ASMREF_TMP2);
2045    emit_loadi(as, tmp2, as->gcsteps);
2046    /* Jump around GC step if GC total < GC threshold. */
2047    emit_branch(as, ARMF_CC(ARMI_B, CC_LS), l_end);
2048    emit_nm(as, ARMI_CMP, RID_TMP, tmp2);
2049    emit_lso(as, ARMI_LDR, tmp2, tmp1,
2050             (int32_t)offsetof(global_State, gc.threshold));
2051    emit_lso(as, ARMI_LDR, RID_TMP, tmp1,
2052             (int32_t)offsetof(global_State, gc.total));
2053    ra_allockreg(as, i32ptr(J2G(as->J)), tmp1);
2054    as->gcsteps = 0;
2055    checkmclim(as);
2056  }
2057
2058  /* -- Loop handling -------------------------------------------------------- */
2059
2060  /* Fixup the loop branch. */
2061  static void asm_loop_fixup(ASMState *as)
2062  {
2063    MCode *p = as->mctop;
2064    MCode *target = as->mcp;
2065    if (as->loopinv) {  /* Inverted loop branch? */
2066      /* asm_guardcc already inverted the bcc and patched the final bl. */
2067      p[-2] |= ((uint32_t)(target-p) & 0x00ffffffu);
2068    } else {
2069      p[-1] = ARMI_B | ((uint32_t)((target-p)-1) & 0x00ffffffu);
2070    }
2071  }
2072
2073  /* -- Head of trace -------------------------------------------------------- */
2074
2075  /* Reload L register from g->cur_L. */
2076  static void asm_head_lreg(ASMState *as)
2077  {
2078    IRIns *ir = IR(ASMREF_L);
2079    if (ra_used(ir)) {
2080      Reg r = ra_dest(as, ir, RSET_GPR);
2081      emit_getgl(as, r, cur_L);
2082      ra_evictk(as);
2083    }
```

```
}

/* Coalesce BASE register for a root trace. */
static void asm_head_root_base(ASMState *as)
{
  IRIns *ir;
  asm_head_lreg(as);
  ir = IR(REF_BASE);
  if (ra_hasreg(ir->r) && (rset_test(as->modset, ir->r) || irt_ismarked(ir->t)))
    ra_spill(as, ir);
  ra_destreg(as, ir, RID_BASE);
}

/* Coalesce BASE register for a side trace. */
static RegSet asm_head_side_base(ASMState *as, IRIns *irp, RegSet allow)
{
  IRIns *ir;
  asm_head_lreg(as);
  ir = IR(REF_BASE);
  if (ra_hasreg(ir->r) && (rset_test(as->modset, ir->r) || irt_ismarked(ir->t)))
    ra_spill(as, ir);
  if (ra_hasspill(irp->s)) {
    rset_clear(allow, ra_dest(as, ir, allow));
  } else {
    Reg r = irp->r;
    lua_assert(ra_hasreg(r));
    rset_clear(allow, r);
    if (r != ir->r && !rset_test(as->freeset, r))
      ra_restore(as, regcost_ref(as->cost[r]));
    ra_destreg(as, ir, r);
  }
  return allow;
}

/* -- Tail of trace ------------------------------------------------------ */

/* Fixup the tail code. */
static void asm_tail_fixup(ASMState *as, TraceNo lnk)
{
  MCode *p = as->mctop;
  MCode *target;
  int32_t spadj = as->T->spadjust;
  if (spadj == 0) {
    as->mctop = --p;
  } else {
    /* Patch stack adjustment. */
    uint32_t k = emit_isk12(ARMI_ADD, spadj);
    lua_assert(k);
    p[-2] = (ARMI_ADD^k) | ARMF_D(RID_SP) | ARMF_N(RID_SP);
  }
  /* Patch exit branch. */
  target = lnk ? traceref(as->J, lnk)->mcode : (MCode *)lj_vm_exit_interp;
  p[-1] = ARMI_B|(((target-p)-1)&0x00ffffffu);
}

/* Prepare tail of code. */
static void asm_tail_prep(ASMState *as)
{
  MCode *p = as->mctop - 1;  /* Leave room for exit branch. */
  if (as->loopref) {
    as->invmcp = as->mcp = p;
  } else {
    as->mcp = p-1;  /* Leave room for stack pointer adjustment. */
    as->invmcp = NULL;
  }
  *p = 0;  /* Prevent load/store merging. */
}

/* -- Trace setup -------------------------------------------------------- */

/* Ensure there are enough stack slots for call arguments. */
static Reg asm_setup_call_slots(ASMState *as, IRIns *ir, const CCallInfo *ci)
{
  IRRef args[CCI_NARGS_MAX*2];
  uint32_t i, nargs = CCI_XNARGS(ci);
  int nslots = 0, ngpr = REGARG_NUMGPR, nfpr = REGARG_NUMFPR, fprodd = 0;
```

```
2160   asm_collectargs(as, ir, ci, args);
2161   for (i = 0; i < nargs; i++) {
2162     if (!LJ_SOFTFP && args[i] && irt_isfp(IR(args[i])->t)) {
2163       if (!LJ_ABI_SOFTFP && !(ci->flags & CCI_VARARG)) {
2164         if (irt_isnum(IR(args[i])->t)) {
2165           if (nfpr > 0) nfpr--;
2166           else fprodd = 0, nslots = (nslots + 3) & ~1;
2167         } else {
2168           if (fprodd) fprodd--;
2169           else if (nfpr > 0) fprodd = 1, nfpr--;
2170           else nslots++;
2171         }
2172       } else if (irt_isnum(IR(args[i])->t)) {
2173         ngpr &= ~1;
2174         if (ngpr > 0) ngpr -= 2; else nslots += 2;
2175       } else {
2176         if (ngpr > 0) ngpr--; else nslots++;
2177       }
2178     } else {
2179       if (ngpr > 0) ngpr--; else nslots++;
2180     }
2181   }
2182   if (nslots > as->evenspill)  /* Leave room for args in stack slots. */
2183     as->evenspill = nslots;
2184   return REGSP_HINT(RID_RET);
2185 }
2186
2187 static void asm_setup_target(ASMState *as)
2188 {
2189   /* May need extra exit for asm_stack_check on side traces. */
2190   asm_exitstub_setup(as, as->T->nsnap + (as->parent ? 1 : 0));
2191 }
2192
2193 /* -- Trace patching ------------------------------------------------------ */
2194
2195 /* Patch exit jumps of existing machine code to a new target. */
2196 void lj_asm_patchexit(jit_State *J, GCtrace *T, ExitNo exitno, MCode *target)
2197 {
2198   MCode *p = T->mcode;
2199   MCode *pe = (MCode *)((char *)p + T->szmcode);
2200   MCode *cstart = NULL, *cend = p;
2201   MCode *mcarea = lj_mcode_patch(J, p, 0);
2202   MCode *px = exitstub_addr(J, exitno) - 2;
2203   for (; p < pe; p++) {
2204     /* Look for bl_cc exitstub, replace with b_cc target. */
2205     uint32_t ins = *p;
2206     if ((ins & 0x0f000000u) == 0x0b000000u && ins < 0xf0000000u &&
2207         ((ins ^ (px-p)) & 0x00ffffffu) == 0) {
2208       *p = (ins & 0xfe000000u) | (((target-p)-2) & 0x00ffffffu);
2209       cend = p+1;
2210       if (!cstart) cstart = p;
2211     }
2212   }
2213   lua_assert(cstart != NULL);
2214   lj_mcode_sync(cstart, cend);
2215   lj_mcode_patch(J, mcarea, 1);
2216 }
2217
```

One Level Up Top Level

src/lj_mcode.h - luajit-2.0-src

Macros defined

- _LJ_MCODE_H
- lj_mcode_commitbot

Source code

```
/*
** Machine code management.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_MCODE_H
#define _LJ_MCODE_H

#include "lj_obj.h"

#if LJ_HASJIT || LJ_HASFFI
LJ_FUNC void lj_mcode_sync(void *start, void *end);
#endif

#if LJ_HASJIT

#include "lj_jit.h"

LJ_FUNC void lj_mcode_free(jit_State *J);
LJ_FUNC MCode *lj_mcode_reserve(jit_State *J, MCode **lim);
LJ_FUNC void lj_mcode_commit(jit_State *J, MCode *m);
LJ_FUNC void lj_mcode_abort(jit_State *J);
LJ_FUNC MCode *lj_mcode_patch(jit_State *J, MCode *ptr, int finish);
LJ_FUNC_NORET void lj_mcode_limiterr(jit_State *J, size_t need);

#define lj_mcode_commitbot(J, m)        (J->mcbot = (m))

#endif

#endif
```

src/lj_emit_arm.h - luajit-2.0-src

Global variables defined

- emit_invai

Data types defined

- MCLabel

Functions defined

- emit_addptr
- emit_branch
- emit_call
- emit_d
- emit_dm
- emit_dn
- emit_dnm
- emit_isk12
- emit_kdelta1
- emit_kdelta2
- emit_loadi
- emit_loadn
- emit_loadofs
- emit_lso
- emit_lsox
- emit_lsptr
- emit_m
- emit_movrr
- emit_n
- emit_nm
- emit_opk
- emit_storeofs
- emit_vlso

Macros defined

- emit_canremat
- emit_getgl
- emit_jmp
- emit_label
- emit_loada
- emit_setgl
- emit_setvmstate
- emit_spsub

Source code

```
/*
** ARM instruction emitter.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

/* -- Constant encoding ---------------------------------------------------- */

static uint8_t emit_invai[16] = {
  /* AND */ (ARMI_AND^ARMI_BIC) >> 21,
  /* EOR */ 0,
  /* SUB */ (ARMI_SUB^ARMI_ADD) >> 21,
  /* RSB */ 0,
  /* ADD */ (ARMI_ADD^ARMI_SUB) >> 21,
  /* ADC */ (ARMI_ADC^ARMI_SBC) >> 21,
  /* SBC */ (ARMI_SBC^ARMI_ADC) >> 21,
  /* RSC */ 0,
  /* TST */ 0,
  /* TEQ */ 0,
  /* CMP */ (ARMI_CMP^ARMI_CMN) >> 21,
  /* CMN */ (ARMI_CMN^ARMI_CMP) >> 21,
  /* ORR */ 0,
  /* MOV */ (ARMI_MOV^ARMI_MVN) >> 21,
  /* BIC */ (ARMI_BIC^ARMI_AND) >> 21,
  /* MVN */ (ARMI_MVN^ARMI_MOV) >> 21
};

/* Encode constant in K12 format for data processing instructions. */
static uint32_t emit_isk12(ARMIns ai, int32_t n)
{
  uint32_t invai, i, m = (uint32_t)n;
  /* K12: unsigned 8 bit value, rotated in steps of two bits. */
  for (i = 0; i < 4096; i += 256, m = lj_rol(m, 2))
    if (m <= 255) return ARMI_K12|m|i;
  /* Otherwise try negation/complement with the inverse instruction. */
  invai = emit_invai[((ai >> 21) & 15)];
  if (!invai) return 0;  /* Failed. No inverse instruction. */
  m = ~(uint32_t)n;
  if (invai == ((ARMI_SUB^ARMI_ADD) >> 21) ||
      invai == (ARMI_CMP^ARMI_CMN) >> 21) m++;
  for (i = 0; i < 4096; i += 256, m = lj_rol(m, 2))
    if (m <= 255) return ARMI_K12|(invai<<21)|m|i;
  return 0;  /* Failed. */
}

/* -- Emit basic instructions ---------------------------------------------- */

static void emit_dnm(ASMState *as, ARMIns ai, Reg rd, Reg rn, Reg rm)
{
  *--as->mcp = ai | ARMF_D(rd) | ARMF_N(rn) | ARMF_M(rm);
}

static void emit_dm(ASMState *as, ARMIns ai, Reg rd, Reg rm)
{
  *--as->mcp = ai | ARMF_D(rd) | ARMF_M(rm);
```

```
}

static void emit_dn(ASMState *as, ARMIns ai, Reg rd, Reg rn)
{
  *--as->mcp = ai | ARMF_D(rd) | ARMF_N(rn);
}

static void emit_nm(ASMState *as, ARMIns ai, Reg rn, Reg rm)
{
  *--as->mcp = ai | ARMF_N(rn) | ARMF_M(rm);
}

static void emit_d(ASMState *as, ARMIns ai, Reg rd)
{
  *--as->mcp = ai | ARMF_D(rd);
}

static void emit_n(ASMState *as, ARMIns ai, Reg rn)
{
  *--as->mcp = ai | ARMF_N(rn);
}

static void emit_m(ASMState *as, ARMIns ai, Reg rm)
{
  *--as->mcp = ai | ARMF_M(rm);
}

static void emit_lsox(ASMState *as, ARMIns ai, Reg rd, Reg rn, int32_t ofs)
{
  lua_assert(ofs >= -255 && ofs <= 255);
  if (ofs < 0) ofs = -ofs; else ai |= ARMI_LS_U;
  *--as->mcp = ai | ARMI_LS_P | ARMI_LSX_I | ARMF_D(rd) | ARMF_N(rn) |
               ((ofs & 0xf0) << 4) | (ofs & 0x0f);
}

static void emit_lso(ASMState *as, ARMIns ai, Reg rd, Reg rn, int32_t ofs)
{
  lua_assert(ofs >= -4095 && ofs <= 4095);
  /* Combine LDR/STR pairs to LDRD/STRD. */
  if (*as->mcp == (ai|ARMI_LS_P|ARMI_LS_U|ARMF_D(rd^1)|ARMF_N(rn)|(ofs^4)) &&
      (ai & ~(ARMI_LDR^ARMI_STR)) == ARMI_STR && rd != rn &&
      (uint32_t)ofs <= 252 && !(ofs & 3) && !((rd ^ (ofs >>2)) & 1) &&
      as->mcp != as->mcloop) {
    as->mcp++;
    emit_lsox(as, ai == ARMI_LDR ? ARMI_LDRD : ARMI_STRD, rd&~1, rn, ofs&~4);
    return;
  }
  if (ofs < 0) ofs = -ofs; else ai |= ARMI_LS_U;
  *--as->mcp = ai | ARMI_LS_P | ARMF_D(rd) | ARMF_N(rn) | ofs;
}

#if !LJ_SOFTFP
static void emit_vlso(ASMState *as, ARMIns ai, Reg rd, Reg rn, int32_t ofs)
{
  lua_assert(ofs >= -1020 && ofs <= 1020 && (ofs&3) == 0);
  if (ofs < 0) ofs = -ofs; else ai |= ARMI_LS_U;
  *--as->mcp = ai | ARMI_LS_P | ARMF_D(rd & 15) | ARMF_N(rn) | (ofs >> 2);
}
#endif

/* -- Emit loads/stores ---------------------------------------------------- */

/* Prefer spills of BASE/L. */
#define emit_canremat(ref)        ((ref) < ASMREF_L)

/* Try to find a one step delta relative to another constant. */
static int emit_kdelta1(ASMState *as, Reg d, int32_t i)
{
  RegSet work = ~as->freeset & RSET_GPR;
  while (work) {
    Reg r = rset_picktop(work);
    IRRef ref = regcost_ref(as->cost[r]);
    lua_assert(r != d);
    if (emit_canremat(ref)) {
      int32_t delta = i - (ra_iskref(ref) ? ra_krefk(as, ref) : IR(ref)->i);
      uint32_t k = emit_isk12(ARMI_ADD, delta);
```

```
      if (k) {
        if (k == ARMI_K12)
          emit_dm(as, ARMI_MOV, d, r);
        else
          emit_dn(as, ARMI_ADD^k, d, r);
        return 1;
      }
    }
    rset_clear(work, r);
  }
  return 0;  /* Failed. */
}

/* Try to find a two step delta relative to another constant. */
static int emit_kdelta2(ASMState *as, Reg d, int32_t i)
{
  RegSet work = ~as->freeset & RSET_GPR;
  while (work) {
    Reg r = rset_picktop(work);
    IRRef ref = regcost_ref(as->cost[r]);
    lua_assert(r != d);
    if (emit_canremat(ref)) {
      int32_t other = ra_iskref(ref) ? ra_krefk(as, ref) : IR(ref)->i;
      if (other) {
        int32_t delta = i - other;
        uint32_t sh, inv = 0, k2, k;
        if (delta < 0) { delta = -delta; inv = ARMI_ADD^ARMI_SUB; }
        sh = lj_ffs(delta) & ~1;
        k2 = emit_isk12(0, delta & (255 << sh));
        k = emit_isk12(0, delta & ~(255 << sh));
        if (k) {
          emit_dn(as, ARMI_ADD^k2^inv, d, d);
          emit_dn(as, ARMI_ADD^k^inv, d, r);
          return 1;
        }
      }
    }
    rset_clear(work, r);
  }
  return 0;  /* Failed. */
}

/* Load a 32 bit constant into a GPR. */
static void emit_loadi(ASMState *as, Reg r, int32_t i)
{
  uint32_t k = emit_isk12(ARMI_MOV, i);
  lua_assert(rset_test(as->freeset, r) || r == RID_TMP);
  if (k) {
    /* Standard K12 constant. */
    emit_d(as, ARMI_MOV^k, r);
  } else if ((as->flags & JIT_F_ARMV6T2) && (uint32_t)i < 0x00010000u) {
    /* 16 bit loword constant for ARMv6T2. */
    emit_d(as, ARMI_MOVW|(i & 0x0fff)|((i & 0xf000)<<4), r);
  } else if (emit_kdelta1(as, r, i)) {
    /* One step delta relative to another constant. */
  } else if ((as->flags & JIT_F_ARMV6T2)) {
    /* 32 bit hiword/loword constant for ARMv6T2. */
    emit_d(as, ARMI_MOVT|((i>>16) & 0x0fff)|(((i>>16) & 0xf000)<<4), r);
    emit_d(as, ARMI_MOVW|(i & 0x0fff)|((i & 0xf000)<<4), r);
  } else if (emit_kdelta2(as, r, i)) {
    /* Two step delta relative to another constant. */
  } else {
    /* Otherwise construct the constant with up to 4 instructions. */
    /* NYI: use mvn+bic, use pc-relative loads. */
    for (;;) {
      uint32_t sh = lj_ffs(i) & ~1;
      int32_t m = i & (255 << sh);
      i &= ~(255 << sh);
      if (i == 0) {
        emit_d(as, ARMI_MOV ^ emit_isk12(0, m), r);
        break;
      }
      emit_dn(as, ARMI_ORR ^ emit_isk12(0, m), r, r);
    }
  }
}
```

```
207
208 #define emit_loada(as, r, addr)               emit_loadi(as, (r), i32ptr((addr)))
209
210 static Reg ra_allock(ASMState *as, int32_t k, RegSet allow);
211
212 /* Get/set from constant pointer. */
213 static void emit_lsptr(ASMState *as, ARMIns ai, Reg r, void *p)
214 {
215   int32_t i = i32ptr(p);
216   emit_lso(as, ai, r, ra_allock(as, (i & ~4095), rset_exclude(RSET_GPR, r)),
217            (i & 4095));
218 }
219
220 #if !LJ_SOFTFP
221 /* Load a number constant into an FPR. */
222 static void emit_loadn(ASMState *as, Reg r, cTValue *tv)
223 {
224   int32_t i;
225   if ((as->flags & JIT_F_VFPV3) && !tv->u32.lo) {
226     uint32_t hi = tv->u32.hi;
227     uint32_t b = ((hi >> 22) & 0x1ff);
228     if (!(hi & 0xffff) && (b == 0x100 || b == 0x0ff)) {
229       *--as->mcp = ARMI_VMOVI_D | ARMF_D(r & 15) |
230                    ((tv->u32.hi >> 12) & 0x00080000) |
231                    ((tv->u32.hi >> 4) & 0x00070000) |
232                    ((tv->u32.hi >> 16) & 0x0000000f);
233       return;
234     }
235   }
236   i = i32ptr(tv);
237   emit_vlso(as, ARMI_VLDR_D, r,
238             ra_allock(as, (i & ~1020), RSET_GPR), (i & 1020));
239 }
240 #endif
241
242 /* Get/set global_State fields. */
243 #define emit_getgl(as, r, field) \
244   emit_lsptr(as, ARMI_LDR, (r), (void *)&J2G(as->J)->field)
245 #define emit_setgl(as, r, field) \
246   emit_lsptr(as, ARMI_STR, (r), (void *)&J2G(as->J)->field)
247
248 /* Trace number is determined from pc of exit instruction. */
249 #define emit_setvmstate(as, i)                UNUSED(i)
250
251 /* -- Emit control-flow instructions ------------------------------------- */
252
253 /* Label for internal jumps. */
254 typedef MCode *MCLabel;
255
256 /* Return label pointing to current PC. */
257 #define emit_label(as)                ((as)->mcp)
258
259 static void emit_branch(ASMState *as, ARMIns ai, MCode *target)
260 {
261   MCode *p = as->mcp;
262   ptrdiff_t delta = (target - p) - 1;
263   lua_assert(((delta + 0x00800000) >> 24) == 0);
264   *--p = ai | ((uint32_t)delta & 0x00ffffffu);
265   as->mcp = p;
266 }
267
268 #define emit_jmp(as, target) emit_branch(as, ARMI_B, (target))
269
270 static void emit_call(ASMState *as, void *target)
271 {
272   MCode *p = --as->mcp;
273   ptrdiff_t delta = ((char *)target - (char *)p) - 8;
274   if ((((delta>>2) + 0x00800000) >> 24) == 0) {
275     if ((delta & 1))
276       *p = ARMI_BLX | ((uint32_t)(delta>>2) & 0x00ffffffu) | ((delta&2) << 27);
277     else
278       *p = ARMI_BL | ((uint32_t)(delta>>2) & 0x00ffffffu);
279   } else {  /* Target out of range: need indirect call. But don't use R0-R3. */
280     Reg r = ra_allock(as, i32ptr(target), RSET_RANGE(RID_R4, RID_R12+1));
281     *p = ARMI_BLXr | ARMF_M(r);
282   }
```

```
283 }
284
285 /* -- Emit generic operations ---------------------------------------- */
286
287 /* Generic move between two regs. */
288 static void emit_movrr(ASMState *as, IRIns *ir, Reg dst, Reg src)
289 {
290 #if LJ_SOFTFP
291   lua_assert(!irt_isnum(ir->t)); UNUSED(ir);
292 #else
293   if (dst >= RID_MAX_GPR) {
294     emit_dm(as, irt_isnum(ir->t) ? ARMI_VMOV_D : ARMI_VMOV_S,
295             (dst & 15), (src & 15));
296     return;
297   }
298 #endif
299   if (as->mcp != as->mcloop) {  /* Swap early registers for loads/stores. */
300     MCode ins = *as->mcp, swp = (src^dst);
301     if ((ins & 0x0c000000) == 0x04000000 && (ins & 0x02000010) != 0x02000010) {
302       if (!((ins ^ (dst << 16)) & 0x000f0000))
303         *as->mcp = ins ^ (swp << 16);  /* Swap N in load/store. */
304       if (!(ins & 0x00100000) && !((ins ^ (dst << 12)) & 0x0000f000))
305         *as->mcp = ins ^ (swp << 12);  /* Swap D in store. */
306     }
307   }
308   emit_dm(as, ARMI_MOV, dst, src);
309 }
310
311 /* Generic load of register with base and (small) offset address. */
312 static void emit_loadofs(ASMState *as, IRIns *ir, Reg r, Reg base, int32_t ofs)
313 {
314 #if LJ_SOFTFP
315   lua_assert(!irt_isnum(ir->t)); UNUSED(ir);
316 #else
317   if (r >= RID_MAX_GPR)
318     emit_vlso(as, irt_isnum(ir->t) ? ARMI_VLDR_D : ARMI_VLDR_S, r, base, ofs);
319   else
320 #endif
321     emit_lso(as, ARMI_LDR, r, base, ofs);
322 }
323
324 /* Generic store of register with base and (small) offset address. */
325 static void emit_storeofs(ASMState *as, IRIns *ir, Reg r, Reg base, int32_t ofs)
326 {
327 #if LJ_SOFTFP
328   lua_assert(!irt_isnum(ir->t)); UNUSED(ir);
329 #else
330   if (r >= RID_MAX_GPR)
331     emit_vlso(as, irt_isnum(ir->t) ? ARMI_VSTR_D : ARMI_VSTR_S, r, base, ofs);
332   else
333 #endif
334     emit_lso(as, ARMI_STR, r, base, ofs);
335 }
336
337 /* Emit an arithmetic/logic operation with a constant operand. */
338 static void emit_opk(ASMState *as, ARMIns ai, Reg dest, Reg src,
339                      int32_t i, RegSet allow)
340 {
341   uint32_t k = emit_isk12(ai, i);
342   if (k)
343     emit_dn(as, ai^k, dest, src);
344   else
345     emit_dnm(as, ai, dest, src, ra_allock(as, i, allow));
346 }
347
348 /* Add offset to pointer. */
349 static void emit_addptr(ASMState *as, Reg r, int32_t ofs)
350 {
351   if (ofs)
352     emit_opk(as, ARMI_ADD, r, r, ofs, rset_exclude(RSET_GPR, r));
353 }
354
355 #define emit_spsub(as, ofs)        emit_addptr(as, RID_SP, -(ofs))
356
```

One Level Up Top Level

src/lj_obj.c - luajit-2.0-src

Global variables defined

- lj_obj_itypename
- lj_obj_typename

Functions defined

- lj_obj_equal
- lj_obj_ptr

Macros defined

- LUA_CORE
- lj_obj_c

Source code

```
1  /*
2  ** Miscellaneous object handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lj_obj_c
7  #define LUA_CORE
8
9  #include "lj_obj.h"
10
11 /* Object type names. */
12 LJ_DATADEF const char *const lj_obj_typename[] = {  /* ORDER LUA_T */
13   "no value", "nil", "boolean", "userdata", "number", "string",
14   "table", "function", "userdata", "thread", "proto", "cdata"
15 };
16
17 LJ_DATADEF const char *const lj_obj_itypename[] = {  /* ORDER LJ_T */
18   "nil", "boolean", "boolean", "userdata", "string", "upval", "thread",
19   "proto", "function", "trace", "cdata", "table", "userdata", "number"
20 };
21
22 /* Compare two objects without calling metamethods. */
23 int LJ_FASTCALL lj_obj_equal(cTValue *o1, cTValue *o2)
24 {
25   if (itype(o1) == itype(o2)) {
26     if (tvispri(o1))
27       return 1;
28     if (!tvisnum(o1))
29       return gcrefeq(o1->gcr, o2->gcr);
30   } else if (!tvisnumber(o1) || !tvisnumber(o2)) {
31     return 0;
32   }
33   return numberVnum(o1) == numberVnum(o2);
34 }
35
36 /* Return pointer to object or its object data. */
37 const void * LJ_FASTCALL lj_obj_ptr(cTValue *o)
38 {
39   if (tvisudata(o))
40     return uddata(udataV(o));
41   else if (tvislightud(o))
42     return lightudV(o);
43   else if (LJ_HASFFI && tviscdata(o))
44     return cdataptr(cdataV(o));
```

```
  else if (tvisgcv(o))
    return gcV(o);
  else
    return NULL;
}

```

One Level Up Top Level

src/lj_target_ppc.h - luajit-2.0-src

Data types defined

- ExitState
- PPCCC
- PPCCC
- PPCIns
- PPCIns

Functions defined

- exitstub_trace_addr_

Macros defined

- EXITSTATE_CHECKEXIT
- FPRDEF
- GPRDEF
- PPCF_A
- PPCF_B
- PPCF_C
- PPCF_CC
- PPCF_DOT
- PPCF_MB
- PPCF_ME
- PPCF_T
- PPCF_Y
- REGARG_FIRSTFPR
- REGARG_FIRSTGPR
- REGARG_LASTFPR
- REGARG_LASTGPR
- REGARG_NUMFPR
- REGARG_NUMGPR
- RIDENUM
- RID_MIN_KREF
- RID_NUM_KREF

- RSET_ALL
- RSET_FIXED
- RSET_FPR
- RSET_GPR
- RSET_INIT
- RSET_SCRATCH
- RSET_SCRATCH_FPR
- RSET_SCRATCH_GPR
- SPOFS_TMP
- SPOFS_TMPHI
- SPOFS_TMPLO
- SPOFS_TMPW
- SPS_FIRST
- SPS_FIXED
- VRIDDEF
- _LJ_TARGET_PPC_H
- exitstub_trace_addr
- sps_align
- sps_scale

Source code

```
1  /*
2  ** Definitions for PPC CPUs.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_TARGET_PPC_H
7  #define _LJ_TARGET_PPC_H
8
9  /* -- Registers IDs ------------------------------------------------------- */
10
11  #define GPRDEF(_) \
12    _(R0) _(SP) _(SYS1) _(R3) _(R4) _(R5) _(R6) _(R7) \
13    _(R8) _(R9) _(R10) _(R11) _(R12) _(SYS2) _(R14) _(R15) \
14    _(R16) _(R17) _(R18) _(R19) _(R20) _(R21) _(R22) _(R23) \
15    _(R24) _(R25) _(R26) _(R27) _(R28) _(R29) _(R30) _(R31)
16  #define FPRDEF(_) \
17    _(F0) _(F1) _(F2) _(F3) _(F4) _(F5) _(F6) _(F7) \
18    _(F8) _(F9) _(F10) _(F11) _(F12) _(F13) _(F14) _(F15) \
19    _(F16) _(F17) _(F18) _(F19) _(F20) _(F21) _(F22) _(F23) \
20    _(F24) _(F25) _(F26) _(F27) _(F28) _(F29) _(F30) _(F31)
21  #define VRIDDEF(_)
22
23  #define RIDENUM(name)        RID_##name,
24
25  enum {
26    GPRDEF(RIDENUM)              /* General-purpose registers (GPRs). */
27    FPRDEF(RIDENUM)              /* Floating-point registers (FPRs). */
28    RID_MAX,
29    RID_TMP = RID_R0,
```

```
  /* Calling conventions. */
  RID_RET = RID_R3,
  RID_RETHI = RID_R3,
  RID_RETLO = RID_R4,
  RID_FPRET = RID_F1,

  /* These definitions must match with the *.dasc file(s): */
  RID_BASE = RID_R14,               /* Interpreter BASE. */
  RID_LPC = RID_R16,               /* Interpreter PC. */
  RID_DISPATCH = RID_R17,        /* Interpreter DISPATCH table. */
  RID_LREG = RID_R18,               /* Interpreter L. */
  RID_JGL = RID_R31,               /* On-trace: global_State + 32768. */

  /* Register ranges [min, max) and number of registers. */
  RID_MIN_GPR = RID_R0,
  RID_MAX_GPR = RID_R31+1,
  RID_MIN_FPR = RID_F0,
  RID_MAX_FPR = RID_F31+1,
  RID_NUM_GPR = RID_MAX_GPR - RID_MIN_GPR,
  RID_NUM_FPR = RID_MAX_FPR - RID_MIN_FPR
};

#define RID_NUM_KREF                RID_NUM_GPR
#define RID_MIN_KREF                RID_R0

/* -- Register sets ------------------------------------------------------ */

/* Make use of all registers, except TMP, SP, SYS1, SYS2 and JGL. */
#define RSET_FIXED \
  (RID2RSET(RID_TMP)|RID2RSET(RID_SP)|RID2RSET(RID_SYS1)|\
   RID2RSET(RID_SYS2)|RID2RSET(RID_JGL))
#define RSET_GPR        (RSET_RANGE(RID_MIN_GPR, RID_MAX_GPR) - RSET_FIXED)
#define RSET_FPR        RSET_RANGE(RID_MIN_FPR, RID_MAX_FPR)
#define RSET_ALL        (RSET_GPR|RSET_FPR)
#define RSET_INIT        RSET_ALL

#define RSET_SCRATCH_GPR        (RSET_RANGE(RID_R3, RID_R12+1))
#define RSET_SCRATCH_FPR        (RSET_RANGE(RID_F0, RID_F13+1))
#define RSET_SCRATCH                (RSET_SCRATCH_GPR|RSET_SCRATCH_FPR)
#define REGARG_FIRSTGPR               RID_R3
#define REGARG_LASTGPR               RID_R10
#define REGARG_NUMGPR                8
#define REGARG_FIRSTFPR               RID_F1
#define REGARG_LASTFPR               RID_F8
#define REGARG_NUMFPR                8

/* -- Spill slots -------------------------------------------------------- */

/* Spill slots are 32 bit wide. An even/odd pair is used for FPRs.
**
** SPS_FIXED: Available fixed spill slots in interpreter frame.
** This definition must match with the *.dasc file(s).
**
** SPS_FIRST: First spill slot for general use.
** [sp+12] tmplo word \
** [sp+ 8] tmphi word / tmp dword, parameter area for callee
** [sp+ 4] tmpw, LR of callee
** [sp+ 0] stack chain
*/
#define SPS_FIXED        7
#define SPS_FIRST        4

/* Stack offsets for temporary slots. Used for FP<->int conversions etc. */
#define SPOFS_TMPW        4
#define SPOFS_TMP        8
#define SPOFS_TMPHI        8
#define SPOFS_TMPLO        12

#define sps_scale(slot)                (4 * (int32_t)(slot))
 #define sps_align(slot)                (((slot) - SPS_FIXED + 3) & ~3)

 /* -- Exit state --------------------------------------------------------- */

 /* This definition must match with the *.dasc file(s). */
 typedef struct {
```

```
  lua_Number fpr[RID_NUM_FPR];        /* Floating-point registers. */
  intptr_t gpr[RID_NUM_GPR];        /* General-purpose registers. */
  int32_t spill[256];                /* Spill slots. */
} ExitState;

/* Highest exit + 1 indicates stack check. */
#define EXITSTATE_CHECKEXIT        1

/* Return the address of a per-trace exit stub. */
static LJ_AINLINE uint32_t *exitstub_trace_addr_(uint32_t *p, uint32_t exitno)
{
  while (*p == 0x60000000) p++;  /* Skip PPCI_NOP. */
  return p + 3 + exitno;
}
/* Avoid dependence on lj_jit.h if only including lj_target.h. */
#define exitstub_trace_addr(T, exitno) \
  exitstub_trace_addr_((MCode *)((char *)(T)->mcode + (T)->szmcode), (exitno))

/* -- Instructions --------------------------------------------------------- */

/* Instruction fields. */
#define PPCF_CC(cc)        ((((cc) & 3) << 16) | (((cc) & 4) << 22))
#define PPCF_T(r)        ((r) << 21)
#define PPCF_A(r)        ((r) << 16)
#define PPCF_B(r)        ((r) << 11)
#define PPCF_C(r)        ((r) << 6)
#define PPCF_MB(n)        ((n) << 6)
#define PPCF_ME(n)        ((n) << 1)
#define PPCF_Y                0x00200000
#define PPCF_DOT        0x00000001

typedef enum PPCIns {
  /* Integer instructions. */
  PPCI_MR = 0x7c000378,
  PPCI_NOP = 0x60000000,

  PPCI_LI = 0x38000000,
  PPCI_LIS = 0x3c000000,

  PPCI_ADD = 0x7c000214,
  PPCI_ADDC = 0x7c000014,
  PPCI_ADDO = 0x7c000614,
  PPCI_ADDE = 0x7c000114,
  PPCI_ADDZE = 0x7c000194,
  PPCI_ADDME = 0x7c0001d4,
  PPCI_ADDI = 0x38000000,
  PPCI_ADDIS = 0x3c000000,
  PPCI_ADDIC = 0x30000000,
  PPCI_ADDICDOT = 0x34000000,

  PPCI_SUBF = 0x7c000050,
  PPCI_SUBFC = 0x7c000010,
  PPCI_SUBFO = 0x7c000450,
  PPCI_SUBFE = 0x7c000110,
  PPCI_SUBFZE = 0x7c000190,
  PPCI_SUBFME = 0x7c0001d0,
  PPCI_SUBFIC = 0x20000000,

  PPCI_NEG = 0x7c0000d0,

  PPCI_AND = 0x7c000038,
  PPCI_ANDC = 0x7c000078,
  PPCI_NAND = 0x7c0003b8,
  PPCI_ANDIDOT = 0x70000000,
  PPCI_ANDISDOT = 0x74000000,

  PPCI_OR = 0x7c000378,
  PPCI_NOR = 0x7c0000f8,
  PPCI_ORI = 0x60000000,
  PPCI_ORIS = 0x64000000,

  PPCI_XOR = 0x7c000278,
  PPCI_EQV = 0x7c000238,
  PPCI_XORI = 0x68000000,
  PPCI_XORIS = 0x6c000000,

```

```
  PPCI_CMPW = 0x7c000000,
  PPCI_CMPLW = 0x7c000040,
  PPCI_CMPWI = 0x2c000000,
  PPCI_CMPLWI = 0x28000000,

  PPCI_MULLW = 0x7c0001d6,
  PPCI_MULLI = 0x1c000000,
  PPCI_MULLWO = 0x7c0005d6,

  PPCI_EXTSB = 0x7c000774,
  PPCI_EXTSH = 0x7c000734,

  PPCI_SLW = 0x7c000030,
  PPCI_SRW = 0x7c000430,
  PPCI_SRAW = 0x7c000630,
  PPCI_SRAWI = 0x7c000670,

  PPCI_RLWNM = 0x5c000000,
  PPCI_RLWINM = 0x54000000,
  PPCI_RLWIMI = 0x50000000,

  PPCI_B = 0x48000000,
  PPCI_BL = 0x48000001,
  PPCI_BC = 0x40800000,
  PPCI_BCL = 0x40800001,
  PPCI_BCTR = 0x4e800420,
  PPCI_BCTRL = 0x4e800421,

  PPCI_CRANDC = 0x4c000102,
  PPCI_CRXOR = 0x4c000182,
  PPCI_CRAND = 0x4c000202,
  PPCI_CREQV = 0x4c000242,
  PPCI_CRORC = 0x4c000342,
  PPCI_CROR = 0x4c000382,

  PPCI_MFLR = 0x7c0802a6,
  PPCI_MTCTR = 0x7c0903a6,

  PPCI_MCRXR = 0x7c000400,

  /* Load/store instructions. */
  PPCI_LWZ = 0x80000000,
  PPCI_LBZ = 0x88000000,
  PPCI_STW = 0x90000000,
  PPCI_STB = 0x98000000,
  PPCI_LHZ = 0xa0000000,
  PPCI_LHA = 0xa8000000,
  PPCI_STH = 0xb0000000,

  PPCI_STWU = 0x94000000,

  PPCI_LFS = 0xc0000000,
  PPCI_LFD = 0xc8000000,
  PPCI_STFS = 0xd0000000,
  PPCI_STFD = 0xd8000000,

  PPCI_LWZX = 0x7c00002e,
  PPCI_LBZX = 0x7c0000ae,
  PPCI_STWX = 0x7c00012e,
  PPCI_STBX = 0x7c0001ae,
  PPCI_LHZX = 0x7c00022e,
  PPCI_LHAX = 0x7c0002ae,
  PPCI_STHX = 0x7c00032e,

  PPCI_LWBRX = 0x7c00042c,
  PPCI_STWBRX = 0x7c00052c,

  PPCI_LFSX = 0x7c00042e,
  PPCI_LFDX = 0x7c0004ae,
  PPCI_STFSX = 0x7c00052e,
  PPCI_STFDX = 0x7c0005ae,

  /* FP instructions. */
  PPCI_FMR = 0xfc000090,
  PPCI_FNEG = 0xfc000050,
  PPCI_FABS = 0xfc000210,
```

```
258
259   PPCI_FRSP = 0xfc000018,
260   PPCI_FCTIWZ = 0xfc00001e,
261
262   PPCI_FADD = 0xfc00002a,
263   PPCI_FSUB = 0xfc000028,
264   PPCI_FMUL = 0xfc000032,
265   PPCI_FDIV = 0xfc000024,
266   PPCI_FSQRT = 0xfc00002c,
267
268   PPCI_FMADD = 0xfc00003a,
269   PPCI_FMSUB = 0xfc000038,
270   PPCI_FNMSUB = 0xfc00003c,
271
272   PPCI_FCMPU = 0xfc000000,
273   PPCI_FSEL = 0xfc00002e,
274 } PPCIns;
275
276 typedef enum PPCCC {
277   CC_GE, CC_LE, CC_NE, CC_NS, CC_LT, CC_GT, CC_EQ, CC_SO
278 } PPCCC;
279
280 #endif
```

One Level Up Top Level

src/lj_alloc.c - luajit-2.0-src

Global variables defined

- ntavm

Data types defined

- PNTAVM
- bindex_t
- binmap_t
- flag_t
- malloc_chunk
- malloc_segment
- malloc_state
- malloc_tree_chunk
- mchunk
- mchunkptr
- msegment
- msegmentptr
- mstate
- sbinptr
- tbinptr
- tchunk
- tchunkptr

Functions defined

- static LJ_AINLINE void *CALL_MMAP(size_t size)
- static LJ_AINLINE void *CALL_MMAP(size_t size)
- static LJ_AINLINE void *CALL_MMAP(size_t size)
- static LJ_AINLINE void *CALL_MMAP(size_t size)
- static LJ_AINLINE void *CALL_MMAP(size_t size)
- static LJ_AINLINE void *CALL_MREMAP_(void *ptr, size_t osz, size_t nsz,
- static LJ_AINLINE int CALL_MUNMAP(void *ptr, size_t size)
- static LJ_AINLINE int CALL_MUNMAP(void *ptr, size_t size)
- static LJ_AINLINE void *DIRECT_MMAP(size_t size)

- static LJ_AINLINE void *DIRECT_MMAP(size_t size)
- static void INIT_MMAP(void)
- add_segment
- alloc_sys
- alloc_trim
- direct_alloc
- direct_resize
- has_segment_link
- init_bins
- init_top
- lj_alloc_create
- lj_alloc_destroy
- lj_alloc_f
- lj_alloc_free
- lj_alloc_malloc
- lj_alloc_realloc
- prepend_alloc
- release_unused_segments
- segment_holding
- tmalloc_large
- tmalloc_small

Macros defined

- CALL_MREMAP
- CALL_MREMAP
- CALL_MREMAP_MAYMOVE
- CALL_MREMAP_MV
- CALL_MREMAP_MV
- CALL_MREMAP_NOMOVE
- CHUNK_ALIGN_MASK
- CHUNK_OVERHEAD
- CINUSE_BIT
- CMFAIL
- DEFAULT_GRANULARITY

- DEFAULT_MMAP_THRESHOLD
- DEFAULT_TRIM_THRESHOLD
- DIRECT_CHUNK_OVERHEAD
- DIRECT_FOOT_PAD
- DIRECT_MMAP
- FENCEPOST_HEAD
- FOUR_SIZE_T_SIZES
- INIT_MMAP
- INIT_MMAP
- INUSE_BITS
- IS_DIRECT_BIT
- LUA_CORE
- MALLOC_ALIGNMENT
- MAP_ANONYMOUS
- MAX_RELEASE_CHECK_RATE
- MAX_REQUEST
- MAX_SIZE_T
- MAX_SMALL_REQUEST
- MAX_SMALL_SIZE
- MCHUNK_SIZE
- MFAIL
- MIN_CHUNK_SIZE
- MIN_LARGE_SIZE
- MIN_REQUEST
- MIN_SMALL_INDEX
- MMAP_FLAGS
- MMAP_PROT
- MMAP_REGION_END
- MMAP_REGION_START
- MMAP_REGION_START
- MMAP_REGION_START
- MMAP_REGION_START
- MMAP_REGION_START

- NSMALLBINS
- NTAVM_ZEROBITS
- NTREEBINS
- PINUSE_BIT
- SIX_SIZE_T_SIZES
- SIZE_T_BITSIZE
- SIZE_T_ONE
- SIZE_T_SIZE
- SIZE_T_TWO
- SIZE_T_ZERO
- SMALLBIN_SHIFT
- SMALLBIN_WIDTH
- TOP_FOOT_SIZE
- TREEBIN_SHIFT
- TWO_SIZE_T_SIZES
- WIN32_LEAN_AND_MEAN
- _GNU_SOURCE
- align_as_chunk
- align_offset
- bit_for_tree_index
- chunk2mem
- chunk_minus_offset
- chunk_plus_offset
- chunksize
- cinuse
- clear_cinuse
- clear_pinuse
- clear_smallmap
- clear_treemap
- compute_tree_index
- get_foot
- granularity_align
- idx2bit

- insert_chunk
- insert_large_chunk
- insert_small_chunk
- is_direct
- is_initialized
- is_small
- left_bits
- leftmost_child
- leftshift_for_tree_index
- lj_alloc_c
- mark_smallmap
- mark_treemap
- mem2chunk
- minsize_for_tree_index
- mmap_align
- mmap_align
- next_chunk
- next_pinuse
- overhead_for
- pad_request
- page_align
- pinuse
- prev_chunk
- replace_dv
- request2size
- segment_holds
- set_foot
- set_free_with_pinuse
- set_inuse
- set_inuse_and_pinuse
- set_size_and_pinuse_of_free_chunk
- set_size_and_pinuse_of_inuse_chunk
- small_index

- small_index2size
- smallbin_at
- smallmap_is_marked
- treebin_at
- treemap_is_marked
- unlink_chunk
- unlink_first_small_chunk
- unlink_large_chunk
- unlink_small_chunk

Source code

```
/*
** Bundled memory allocator.
**
** Beware: this is a HEAVILY CUSTOMIZED version of dlmalloc.
** The original bears the following remark:
**
**   This is a version (aka dlmalloc) of malloc/free/realloc written by
**   Doug Lea and released to the public domain, as explained at
**   http://creativecommons.org/licenses/publicdomain.
 **
 **   * Version pre-2.8.4 Wed Mar 29 19:46:29 2006    (dl at gee)
 **
 ** No additional copyright is claimed over the customizations.
 ** Please do NOT bother the original author about this version here!
 **
 ** If you want to use dlmalloc in another project, you should get
 ** the original from: ftp://gee.cs.oswego.edu/pub/misc/
 ** For thread-safe derivatives, take a look at:
 ** - ptmalloc: http://www.malloc.de/
 ** - nedmalloc: http://www.nedprod.com/programs/portable/nedmalloc/
 */

#define lj_alloc_c
#define LUA_CORE

/* To get the mremap prototype. Must be defined before any system includes. */
#if defined(__linux__) && !defined(_GNU_SOURCE)
#define _GNU_SOURCE
#endif

#include "lj_def.h"
#include "lj_arch.h"
#include "lj_alloc.h"

#ifndef LUAJIT_USE_SYSMALLOC

#define MAX_SIZE_T               (~(size_t)0)
#define MALLOC_ALIGNMENT        ((size_t)8U)

#define DEFAULT_GRANULARITY        ((size_t)128U * (size_t)1024U)
#define DEFAULT_TRIM_THRESHOLD        ((size_t)2U * (size_t)1024U * (size_t)1024U)
#define DEFAULT_MMAP_THRESHOLD        ((size_t)128U * (size_t)1024U)
#define MAX_RELEASE_CHECK_RATE        255

/* ------------------- size_t and alignment properties -------------------- */

/* The byte and bit size of a size_t */
#define SIZE_T_SIZE               (sizeof(size_t))
#define SIZE_T_BITSIZE               (sizeof(size_t) << 3)

/* Some constants coerced to size_t */
/* Annoying but necessary to avoid errors on some platforms */
```

```c
#define SIZE_T_ZERO               ((size_t)0)
#define SIZE_T_ONE                ((size_t)1)
#define SIZE_T_TWO                ((size_t)2)
#define TWO_SIZE_T_SIZES         (SIZE_T_SIZE<<1)
#define FOUR_SIZE_T_SIZES         (SIZE_T_SIZE<<2)
#define SIX_SIZE_T_SIZES         (FOUR_SIZE_T_SIZES+TWO_SIZE_T_SIZES)

/* The bit mask value corresponding to MALLOC_ALIGNMENT */
#define CHUNK_ALIGN_MASK         (MALLOC_ALIGNMENT - SIZE_T_ONE)

/* the number of bytes to offset an address to align it */
#define align_offset(A)\
((((size_t)(A) & CHUNK_ALIGN_MASK) == 0)? 0 :\
  ((MALLOC_ALIGNMENT - ((size_t)(A) & CHUNK_ALIGN_MASK)) & CHUNK_ALIGN_MASK))

/* ------------------------- MMAP support ------------------------------ */

#define MFAIL                        ((void *)(MAX_SIZE_T))
#define CMFAIL                        ((char *)(MFAIL)) /* defined for convenience */

#define IS_DIRECT_BIT                (SIZE_T_ONE)

#if LJ_TARGET_WINDOWS

#define WIN32_LEAN_AND_MEAN
#include <windows.h>

#if LJ_64 && !LJ_GC64

/* Undocumented, but hey, that's what we all love so much about Windows. */
typedef long (*PNTAVM)(HANDLE handle, void **addr, ULONG zbits,
                       size_t *size, ULONG alloctype, ULONG prot);
static PNTAVM ntavm;

/* Number of top bits of the lower 32 bits of an address that must be zero.
** Apparently 0 gives us full 64 bit addresses and 1 gives us the lower 2GB.
*/
#define NTAVM_ZEROBITS                1

static void INIT_MMAP(void)
{
  ntavm = (PNTAVM)GetProcAddress(GetModuleHandleA("ntdll.dll"),
                                 "NtAllocateVirtualMemory");
}

/* Win64 32 bit MMAP via NtAllocateVirtualMemory. */
static LJ_AINLINE void *CALL_MMAP(size_t size)
{
  DWORD olderr = GetLastError();
  void *ptr = NULL;
  long st = ntavm(INVALID_HANDLE_VALUE, &ptr, NTAVM_ZEROBITS, &size,
                  MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
  SetLastError(olderr);
  return st == 0 ? ptr : MFAIL;
}

/* For direct MMAP, use MEM_TOP_DOWN to minimize interference */
static LJ_AINLINE void *DIRECT_MMAP(size_t size)
{
  DWORD olderr = GetLastError();
  void *ptr = NULL;
  long st = ntavm(INVALID_HANDLE_VALUE, &ptr, NTAVM_ZEROBITS, &size,
                  MEM_RESERVE|MEM_COMMIT|MEM_TOP_DOWN, PAGE_READWRITE);
  SetLastError(olderr);
  return st == 0 ? ptr : MFAIL;
}

#else

#define INIT_MMAP()               ((void)0)

/* Win32 MMAP via VirtualAlloc */
static LJ_AINLINE void *CALL_MMAP(size_t size)
{
  DWORD olderr = GetLastError();
  void *ptr = VirtualAlloc(0, size, MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
```

```
  SetLastError(olderr);
  return ptr ? ptr : MFAIL;
}

/* For direct MMAP, use MEM_TOP_DOWN to minimize interference */
static LJ_AINLINE void *DIRECT_MMAP(size_t size)
{
  DWORD olderr = GetLastError();
  void *ptr = VirtualAlloc(0, size, MEM_RESERVE|MEM_COMMIT|MEM_TOP_DOWN,
                           PAGE_READWRITE);
  SetLastError(olderr);
  return ptr ? ptr : MFAIL;
}

#endif

/* This function supports releasing coalesed segments */
static LJ_AINLINE int CALL_MUNMAP(void *ptr, size_t size)
{
  DWORD olderr = GetLastError();
  MEMORY_BASIC_INFORMATION minfo;
  char *cptr = (char *)ptr;
  while (size) {
    if (VirtualQuery(cptr, &minfo, sizeof(minfo)) == 0)
      return -1;
    if (minfo.BaseAddress != cptr || minfo.AllocationBase != cptr ||
        minfo.State != MEM_COMMIT || minfo.RegionSize > size)
      return -1;
    if (VirtualFree(cptr, 0, MEM_RELEASE) == 0)
      return -1;
    cptr += minfo.RegionSize;
    size -= minfo.RegionSize;
  }
  SetLastError(olderr);
  return 0;
}

#else

#include <errno.h>
#include <sys/mman.h>

#define MMAP_PROT                (PROT_READ|PROT_WRITE)
#if !defined(MAP_ANONYMOUS) && defined(MAP_ANON)
#define MAP_ANONYMOUS                MAP_ANON
#endif
#define MMAP_FLAGS               (MAP_PRIVATE|MAP_ANONYMOUS)

#if LJ_64 && !LJ_GC64
/* 64 bit mode with 32 bit pointers needs special support for allocating
** memory in the lower 2GB.
*/

#if defined(MAP_32BIT)

#if defined(__sun__)
#define MMAP_REGION_START        ((uintptr_t)0x1000)
#else
/* Actually this only gives us max. 1GB in current Linux kernels. */
#define MMAP_REGION_START        ((uintptr_t)0)
#endif

static LJ_AINLINE void *CALL_MMAP(size_t size)
{
  int olderr = errno;
  void *ptr = mmap((void *)MMAP_REGION_START, size, MMAP_PROT, MAP_32BIT|MMAP_FLAGS, -1, 0);
  errno = olderr;
  return ptr;
}

#elif LJ_TARGET_OSX || LJ_TARGET_PS4 || defined(__FreeBSD__) || defined(__FreeBSD_kernel__) ||
defined(__NetBSD__) || defined(__OpenBSD__) || defined(__DragonFly__) || defined(__sun__)

/* OSX and FreeBSD mmap() use a naive first-fit linear search.
** That's perfect for us. Except that -pagezero_size must be set for OSX,
** otherwise the lower 4GB are blocked. And the 32GB RLIMIT_DATA needs
```

```
** to be reduced to 250MB on FreeBSD.
*/
#if LJ_TARGET_OSX || defined(__DragonFly__)
#define MMAP_REGION_START	((uintptr_t)0x10000)
#elif LJ_TARGET_PS4
#define MMAP_REGION_START	((uintptr_t)0x4000)
#else
#define MMAP_REGION_START	((uintptr_t)0x10000000)
#endif
#define MMAP_REGION_END		((uintptr_t)0x80000000)

#if (defined(__FreeBSD__) || defined(__FreeBSD_kernel__)) && !LJ_TARGET_PS4
#include <sys/resource.h>
#endif

static LJ_AINLINE void *CALL_MMAP(size_t size)
{
  int olderr = errno;
  /* Hint for next allocation. Doesn't need to be thread-safe. */
  static uintptr_t alloc_hint = MMAP_REGION_START;
  int retry = 0;
#if (defined(__FreeBSD__) || defined(__FreeBSD_kernel__)) && !LJ_TARGET_PS4
  static int rlimit_modified = 0;
  if (LJ_UNLIKELY(rlimit_modified == 0)) {
    struct rlimit rlim;
    rlim.rlim_cur = rlim.rlim_max = MMAP_REGION_START;
    setrlimit(RLIMIT_DATA, &rlim);  /* Ignore result. May fail below. */
    rlimit_modified = 1;
  }
#endif
  for (;;) {
    void *p = mmap((void *)alloc_hint, size, MMAP_PROT, MMAP_FLAGS, -1, 0);
    if ((uintptr_t)p >= MMAP_REGION_START &&
        (uintptr_t)p + size < MMAP_REGION_END) {
      alloc_hint = (uintptr_t)p + size;
      errno = olderr;
      return p;
    }
    if (p != CMFAIL) munmap(p, size);
#if defined(__sun__) || defined(__DragonFly__)
    alloc_hint += 0x1000000;  /* Need near-exhaustive linear scan. */
    if (alloc_hint + size < MMAP_REGION_END) continue;
#endif
    if (retry) break;
    retry = 1;
    alloc_hint = MMAP_REGION_START;
  }
  errno = olderr;
  return CMFAIL;
}

#else

#error "NYI: need an equivalent of MAP_32BIT for this 64 bit OS"

#endif

#else

/* 32 bit mode and GC64 mode is easy. */
static LJ_AINLINE void *CALL_MMAP(size_t size)
{
  int olderr = errno;
  void *ptr = mmap(NULL, size, MMAP_PROT, MMAP_FLAGS, -1, 0);
  errno = olderr;
  return ptr;
}

#endif

#define INIT_MMAP()		((void)0)
#define DIRECT_MMAP(s)		CALL_MMAP(s)

static LJ_AINLINE int CALL_MUNMAP(void *ptr, size_t size)
{
  int olderr = errno;
```

```
  int ret = munmap(ptr, size);
  errno = olderr;
  return ret;
}

#if LJ_TARGET_LINUX
/* Need to define _GNU_SOURCE to get the mremap prototype. */
static LJ_AINLINE void *CALL_MREMAP_(void *ptr, size_t osz, size_t nsz,
                                     int flags)
{
  int olderr = errno;
  ptr = mremap(ptr, osz, nsz, flags);
  errno = olderr;
  return ptr;
}

#define CALL_MREMAP(addr, osz, nsz, mv) CALL_MREMAP_((addr), (osz), (nsz), (mv))
#define CALL_MREMAP_NOMOVE         0
#define CALL_MREMAP_MAYMOVE        1
#if LJ_64 && !LJ_GC64
#define CALL_MREMAP_MV               CALL_MREMAP_NOMOVE
#else
#define CALL_MREMAP_MV               CALL_MREMAP_MAYMOVE
#endif
#endif

#endif

#ifndef CALL_MREMAP
#define CALL_MREMAP(addr, osz, nsz, mv) ((void)osz, MFAIL)
#endif

/* ----------------------  Chunk representations ----------------------- */

struct malloc_chunk {
  size_t               prev_foot;  /* Size of previous chunk (if free).  */
  size_t               head;       /* Size and inuse bits. */
  struct malloc_chunk *fd;         /* double links -- used only if free. */
  struct malloc_chunk *bk;
};

typedef struct malloc_chunk  mchunk;
typedef struct malloc_chunk *mchunkptr;
typedef struct malloc_chunk *sbinptr;  /* The type of bins of chunks */
typedef size_t bindex_t;               /* Described below */
typedef unsigned int binmap_t;         /* Described below */
typedef unsigned int flag_t;           /* The type of various bit flag sets */

/* ------------------- Chunks sizes and alignments ----------------------- */

#define MCHUNK_SIZE               (sizeof(mchunk))

#define CHUNK_OVERHEAD               (SIZE_T_SIZE)

/* Direct chunks need a second word of overhead ... */
#define DIRECT_CHUNK_OVERHEAD        (TWO_SIZE_T_SIZES)
/* ... and additional padding for fake next-chunk at foot */
#define DIRECT_FOOT_PAD               (FOUR_SIZE_T_SIZES)

/* The smallest size we can malloc is an aligned minimal chunk */
#define MIN_CHUNK_SIZE\
  ((MCHUNK_SIZE + CHUNK_ALIGN_MASK) & ~CHUNK_ALIGN_MASK)

/* conversion from malloc headers to user pointers, and back */
#define chunk2mem(p)                ((void *)((char *)(p) + TWO_SIZE_T_SIZES))
#define mem2chunk(mem)               ((mchunkptr)((char *)(mem) - TWO_SIZE_T_SIZES))
/* chunk associated with aligned address A */
#define align_as_chunk(A)        (mchunkptr)((A) + align_offset(chunk2mem(A)))

/* Bounds on request (not chunk) sizes. */
#define MAX_REQUEST               ((~MIN_CHUNK_SIZE+1) << 2)
#define MIN_REQUEST               (MIN_CHUNK_SIZE - CHUNK_OVERHEAD - SIZE_T_ONE)

/* pad request bytes into a usable size */
#define pad_request(req) \
   (((req) + CHUNK_OVERHEAD + CHUNK_ALIGN_MASK) & ~CHUNK_ALIGN_MASK)
```

```
356
357 /* pad request, checking for minimum (but not maximum) */
358 #define request2size(req) \
359   (((req) < MIN_REQUEST)? MIN_CHUNK_SIZE : pad_request(req))
360
361 /* ------------------ Operations on head and foot fields ----------------- */
362
363 #define PINUSE_BIT               (SIZE_T_ONE)
364 #define CINUSE_BIT               (SIZE_T_TWO)
365 #define INUSE_BITS               (PINUSE_BIT|CINUSE_BIT)
366
367 /* Head value for fenceposts */
368 #define FENCEPOST_HEAD                (INUSE_BITS|SIZE_T_SIZE)
369
370 /* extraction of fields from head words */
371 #define cinuse(p)               ((p)->head & CINUSE_BIT)
372 #define pinuse(p)               ((p)->head & PINUSE_BIT)
373 #define chunksize(p)               ((p)->head & ~(INUSE_BITS))
374
375 #define clear_pinuse(p)               ((p)->head &= ~PINUSE_BIT)
376 #define clear_cinuse(p)               ((p)->head &= ~CINUSE_BIT)
377
378 /* Treat space at ptr +/- offset as a chunk */
379 #define chunk_plus_offset(p, s)                ((mchunkptr)(((char *)(p)) + (s)))
380 #define chunk_minus_offset(p, s)        ((mchunkptr)(((char *)(p)) - (s)))
381
382 /* Ptr to next or previous physical malloc_chunk. */
383 #define next_chunk(p)        ((mchunkptr)(((char *)(p)) + ((p)->head & ~INUSE_BITS)))
384 #define prev_chunk(p)        ((mchunkptr)(((char *)(p)) - ((p)->prev_foot) ))
385
386 /* extract next chunk's pinuse bit */
387 #define next_pinuse(p)        ((next_chunk(p)->head) & PINUSE_BIT)
388
389 /* Get/set size at footer */
390 #define get_foot(p, s)        (((mchunkptr)((char *)(p) + (s)))->prev_foot)
391 #define set_foot(p, s)        (((mchunkptr)((char *)(p) + (s)))->prev_foot = (s))
392
393 /* Set size, pinuse bit, and foot */
394 #define set_size_and_pinuse_of_free_chunk(p, s)\
395   ((p)->head = (s|PINUSE_BIT), set_foot(p, s))
396
397 /* Set size, pinuse bit, foot, and clear next pinuse */
398 #define set_free_with_pinuse(p, s, n)\
399   (clear_pinuse(n), set_size_and_pinuse_of_free_chunk(p, s))
400
401 #define is_direct(p)\
402   (!((p)->head & PINUSE_BIT) && ((p)->prev_foot & IS_DIRECT_BIT))
403
404 /* Get the internal overhead associated with chunk p */
405 #define overhead_for(p)\
406 (is_direct(p)? DIRECT_CHUNK_OVERHEAD : CHUNK_OVERHEAD)
407
408 /* ---------------------- Overlaid data structures ----------------------- */
409
410 struct malloc_tree_chunk {
411   /* The first four fields must be compatible with malloc_chunk */
412   size_t                    prev_foot;
413   size_t                    head;
414   struct malloc_tree_chunk *fd;
415   struct malloc_tree_chunk *bk;
416
417   struct malloc_tree_chunk *child[2];
418   struct malloc_tree_chunk *parent;
419   bindex_t                  index;
420 };
421
422 typedef struct malloc_tree_chunk  tchunk;
423 typedef struct malloc_tree_chunk *tchunkptr;
424 typedef struct malloc_tree_chunk *tbinptr; /* The type of bins of trees */
425
426 /* A little helper macro for trees */
427 #define leftmost_child(t) ((t)->child[0] != 0? (t)->child[0] : (t)->child[1])
428
429 /* ----------------------------- Segments -------------------------------- */
430
431 struct malloc_segment {
```

```
  char        *base;            /* base address */
  size_t       size;            /* allocated size */
  struct malloc_segment *next;   /* ptr to next segment */
};

typedef struct malloc_segment  msegment;
typedef struct malloc_segment *msegmentptr;

/* ---------------------------- malloc_state ----------------------------- */

/* Bin types, widths and sizes */
#define NSMALLBINS               (32U)
#define NTREEBINS                (32U)
#define SMALLBIN_SHIFT                (3U)
#define SMALLBIN_WIDTH                (SIZE_T_ONE << SMALLBIN_SHIFT)
#define TREEBIN_SHIFT                (8U)
#define MIN_LARGE_SIZE                (SIZE_T_ONE << TREEBIN_SHIFT)
#define MAX_SMALL_SIZE                (MIN_LARGE_SIZE - SIZE_T_ONE)
#define MAX_SMALL_REQUEST  (MAX_SMALL_SIZE - CHUNK_ALIGN_MASK - CHUNK_OVERHEAD)

struct malloc_state {
  binmap_t   smallmap;
  binmap_t   treemap;
  size_t     dvsize;
  size_t     topsize;
  mchunkptr  dv;
  mchunkptr  top;
  size_t     trim_check;
  size_t     release_checks;
  mchunkptr  smallbins[(NSMALLBINS+1)*2];
  tbinptr    treebins[NTREEBINS];
  msegment   seg;
};

typedef struct malloc_state *mstate;

#define is_initialized(M)        ((M)->top != 0)

/* ------------------------- system alloc setup ------------------------- */

/* page-align a size */
#define page_align(S)\
(((S) + (LJ_PAGESIZE - SIZE_T_ONE)) & ~(LJ_PAGESIZE - SIZE_T_ONE))

/* granularity-align a size */
#define granularity_align(S)\
  (((S) + (DEFAULT_GRANULARITY - SIZE_T_ONE))\
   & ~(DEFAULT_GRANULARITY - SIZE_T_ONE))

#if LJ_TARGET_WINDOWS
#define mmap_align(S)        granularity_align(S)
#else
#define mmap_align(S)        page_align(S)
#endif

/*  True if segment S holds address A */
#define segment_holds(S, A)\
  ((char *)(A) >= S->base && (char *)(A) < S->base + S->size)

/* Return segment holding given address */
static msegmentptr segment_holding(mstate m, char *addr)
{
  msegmentptr sp = &m->seg;
  for (;;) {
    if (addr >= sp->base && addr < sp->base + sp->size)
      return sp;
    if ((sp = sp->next) == 0)
      return 0;
  }
}

/* Return true if segment contains a segment link */
static int has_segment_link(mstate m, msegmentptr ss)
{
  msegmentptr sp = &m->seg;
  for (;;) {
```

```
    if ((char *)sp >= ss->base && (char *)sp < ss->base + ss->size)
      return 1;
    if ((sp = sp->next) == 0)
      return 0;
  }
}

/*
  TOP_FOOT_SIZE is padding at the end of a segment, including space
  that may be needed to place segment records and fenceposts when new
  noncontiguous segments are added.
*/
#define TOP_FOOT_SIZE\
  (align_offset(chunk2mem(0))+pad_request(sizeof(struct malloc_segment))+MIN_CHUNK_SIZE)

/* ---------------------------- Indexing Bins ---------------------------- */

#define is_small(s)               (((s) >> SMALLBIN_SHIFT) < NSMALLBINS)
#define small_index(s)              ((s)  >> SMALLBIN_SHIFT)
#define small_index2size(i)       ((i)  << SMALLBIN_SHIFT)
#define MIN_SMALL_INDEX             (small_index(MIN_CHUNK_SIZE))

/* addressing by index. See above about smallbin repositioning */
#define smallbin_at(M, i)       ((sbinptr)((char *)&((M)->smallbins[(i)<<1])))
#define treebin_at(M,i)             (&((M)->treebins[i]))

/* assign tree index for size S to variable I */
#define compute_tree_index(S, I)\
{\
  unsigned int X = (unsigned int)(S >> TREEBIN_SHIFT);\
  if (X == 0) {\
    I = 0;\
  } else if (X > 0xFFFF) {\
    I = NTREEBINS-1;\
  } else {\
    unsigned int K = lj_fls(X);\
    I =  (bindex_t)((K << 1) + ((S >> (K + (TREEBIN_SHIFT-1)) & 1)));\
  }\
}

/* Bit representing maximum resolved size in a treebin at i */
#define bit_for_tree_index(i) \
   (i == NTREEBINS-1)? (SIZE_T_BITSIZE-1) : (((i) >> 1) + TREEBIN_SHIFT - 2)

/* Shift placing maximum resolved bit in a treebin at i as sign bit */
#define leftshift_for_tree_index(i) \
   ((i == NTREEBINS-1)? 0 : \
    ((SIZE_T_BITSIZE-SIZE_T_ONE) - (((i) >> 1) + TREEBIN_SHIFT - 2)))

/* The size of the smallest chunk held in bin with index i */
#define minsize_for_tree_index(i) \
   ((SIZE_T_ONE << (((i) >> 1) + TREEBIN_SHIFT)) |  \
   (((size_t)((i) & SIZE_T_ONE)) << (((i) >> 1) + TREEBIN_SHIFT - 1)))

/* ----------------------- Operations on bin maps ----------------------- */

/* bit corresponding to given index */
#define idx2bit(i)                ((binmap_t)(1) << (i))

/* Mark/Clear bits with given index */
#define mark_smallmap(M,i)        ((M)->smallmap |=  idx2bit(i))
#define clear_smallmap(M,i)        ((M)->smallmap &= ~idx2bit(i))
#define smallmap_is_marked(M,i)        ((M)->smallmap &   idx2bit(i))

#define mark_treemap(M,i)        ((M)->treemap  |=  idx2bit(i))
#define clear_treemap(M,i)        ((M)->treemap  &= ~idx2bit(i))
#define treemap_is_marked(M,i)        ((M)->treemap  &   idx2bit(i))

/* mask with all bits to left of least bit of x on */
#define left_bits(x)                ((x<<1) | (~(x<<1)+1))

/* Set cinuse bit and pinuse bit of next chunk */
#define set_inuse(M,p,s)\
  ((p)->head = (((p)->head & PINUSE_BIT)|s|CINUSE_BIT),\
  ((mchunkptr)(((char *)(p)) + (s)))->head |= PINUSE_BIT)

```

```
584 /* Set cinuse and pinuse of this chunk and pinuse of next chunk */
585 #define set_inuse_and_pinuse(M,p,s)\
586   ((p)->head = (s|PINUSE_BIT|CINUSE_BIT),\
587   ((mchunkptr)(((char *)(p)) + (s)))->head |= PINUSE_BIT)
588
589 /* Set size, cinuse and pinuse bit of this chunk */
590 #define set_size_and_pinuse_of_inuse_chunk(M, p, s)\
591   ((p)->head = (s|PINUSE_BIT|CINUSE_BIT))
592
593 /* ----------------------- Operations on smallbins ----------------------- */
594
595 /* Link a free chunk into a smallbin  */
596 #define insert_small_chunk(M, P, S) {\
597   bindex_t I = small_index(S);\
598   mchunkptr B = smallbin_at(M, I);\
599   mchunkptr F = B;\
600   if (!smallmap_is_marked(M, I))\
601     mark_smallmap(M, I);\
602   else\
603     F = B->fd;\
604   B->fd = P;\
605   F->bk = P;\
606   P->fd = F;\
607   P->bk = B;\
608 }
609
610 /* Unlink a chunk from a smallbin  */
611 #define unlink_small_chunk(M, P, S) {\
612   mchunkptr F = P->fd;\
613   mchunkptr B = P->bk;\
614   bindex_t I = small_index(S);\
615   if (F == B) {\
616     clear_smallmap(M, I);\
617   } else {\
618     F->bk = B;\
619     B->fd = F;\
620   }\
621 }
622
623 /* Unlink the first chunk from a smallbin */
624 #define unlink_first_small_chunk(M, B, P, I) {\
625   mchunkptr F = P->fd;\
626   if (B == F) {\
627     clear_smallmap(M, I);\
628   } else {\
629     B->fd = F;\
630     F->bk = B;\
631   }\
632 }
633
634 /* Replace dv node, binning the old one */
635 /* Used only when dvsize known to be small */
636 #define replace_dv(M, P, S) {\
637   size_t DVS = M->dvsize;\
638   if (DVS != 0) {\
639     mchunkptr DV = M->dv;\
640     insert_small_chunk(M, DV, DVS);\
641   }\
642   M->dvsize = S;\
643   M->dv = P;\
644 }
645
646 /* ------------------------ Operations on trees ------------------------- */
647
648 /* Insert chunk into tree */
649 #define insert_large_chunk(M, X, S) {\
650   tbinptr *H;\
651   bindex_t I;\
652   compute_tree_index(S, I);\
653   H = treebin_at(M, I);\
654   X->index = I;\
655   X->child[0] = X->child[1] = 0;\
656   if (!treemap_is_marked(M, I)) {\
657     mark_treemap(M, I);\
658     *H = X;\
659     X->parent = (tchunkptr)H;\
```

```
    X->fd = X->bk = X;\
  } else {\
    tchunkptr T = *H;\
    size_t K = S << leftshift_for_tree_index(I);\
    for (;;) {\
      if (chunksize(T) != S) {\
        tchunkptr *C = &(T->child[(K >> (SIZE_T_BITSIZE-SIZE_T_ONE)) & 1]);\
        K <<= 1;\
        if (*C != 0) {\
          T = *C;\
        } else {\
          *C = X;\
          X->parent = T;\
          X->fd = X->bk = X;\
          break;\
        }\
      } else {\
        tchunkptr F = T->fd;\
        T->fd = F->bk = X;\
        X->fd = F;\
        X->bk = T;\
        X->parent = 0;\
        break;\
      }\
    }\
  }\
}

#define unlink_large_chunk(M, X) {\
  tchunkptr XP = X->parent;\
  tchunkptr R;\
  if (X->bk != X) {\
    tchunkptr F = X->fd;\
    R = X->bk;\
    F->bk = R;\
    R->fd = F;\
  } else {\
    tchunkptr *RP;\
    if (((R = *(RP = &(X->child[1]))) != 0) ||\
        ((R = *(RP = &(X->child[0]))) != 0)) {\
      tchunkptr *CP;\
      while ((*(CP = &(R->child[1])) != 0) ||\
             (*(CP = &(R->child[0])) != 0)) {\
        R = *(RP = CP);\
      }\
      *RP = 0;\
    }\
  }\
  if (XP != 0) {\
    tbinptr *H = treebin_at(M, X->index);\
    if (X == *H) {\
      if ((*H = R) == 0) \
        clear_treemap(M, X->index);\
    } else {\
      if (XP->child[0] == X) \
        XP->child[0] = R;\
      else \
        XP->child[1] = R;\
    }\
    if (R != 0) {\
      tchunkptr C0, C1;\
      R->parent = XP;\
      if ((C0 = X->child[0]) != 0) {\
        R->child[0] = C0;\
        C0->parent = R;\
      }\
      if ((C1 = X->child[1]) != 0) {\
        R->child[1] = C1;\
        C1->parent = R;\
      }\
    }\
  }\
}

/* Relays to large vs small bin operations */

```

```
#define insert_chunk(M, P, S)\
  if (is_small(S)) { insert_small_chunk(M, P, S)\
  } else { tchunkptr TP = (tchunkptr)(P); insert_large_chunk(M, TP, S); }

#define unlink_chunk(M, P, S)\
  if (is_small(S)) { unlink_small_chunk(M, P, S)\
  } else { tchunkptr TP = (tchunkptr)(P); unlink_large_chunk(M, TP); }

/* ----------------------  Direct-mmapping chunks ---------------------- */

static void *direct_alloc(size_t nb)
{
  size_t mmsize = mmap_align(nb + SIX_SIZE_T_SIZES + CHUNK_ALIGN_MASK);
  if (LJ_LIKELY(mmsize > nb)) {     /* Check for wrap around 0 */
    char *mm = (char *)(DIRECT_MMAP(mmsize));
    if (mm != CMFAIL) {
      size_t offset = align_offset(chunk2mem(mm));
      size_t psize = mmsize - offset - DIRECT_FOOT_PAD;
      mchunkptr p = (mchunkptr)(mm + offset);
      p->prev_foot = offset | IS_DIRECT_BIT;
      p->head = psize|CINUSE_BIT;
      chunk_plus_offset(p, psize)->head = FENCEPOST_HEAD;
      chunk_plus_offset(p, psize+SIZE_T_SIZE)->head = 0;
      return chunk2mem(p);
    }
  }
  return NULL;
}

static mchunkptr direct_resize(mchunkptr oldp, size_t nb)
{
  size_t oldsize = chunksize(oldp);
  if (is_small(nb)) /* Can't shrink direct regions below small size */
    return NULL;
  /* Keep old chunk if big enough but not too big */
  if (oldsize >= nb + SIZE_T_SIZE &&
      (oldsize - nb) <= (DEFAULT_GRANULARITY >> 1)) {
    return oldp;
  } else {
    size_t offset = oldp->prev_foot & ~IS_DIRECT_BIT;
    size_t oldmmsize = oldsize + offset + DIRECT_FOOT_PAD;
    size_t newmmsize = mmap_align(nb + SIX_SIZE_T_SIZES + CHUNK_ALIGN_MASK);
    char *cp = (char *)CALL_MREMAP((char *)oldp - offset,
                                   oldmmsize, newmmsize, CALL_MREMAP_MV);
    if (cp != CMFAIL) {
      mchunkptr newp = (mchunkptr)(cp + offset);
      size_t psize = newmmsize - offset - DIRECT_FOOT_PAD;
      newp->head = psize|CINUSE_BIT;
      chunk_plus_offset(newp, psize)->head = FENCEPOST_HEAD;
      chunk_plus_offset(newp, psize+SIZE_T_SIZE)->head = 0;
      return newp;
    }
  }
  return NULL;
}

/* ------------------------- mspace management ------------------------- */

/* Initialize top chunk and its size */
static void init_top(mstate m, mchunkptr p, size_t psize)
{
  /* Ensure alignment */
  size_t offset = align_offset(chunk2mem(p));
  p = (mchunkptr)((char *)p + offset);
  psize -= offset;

  m->top = p;
  m->topsize = psize;
  p->head = psize | PINUSE_BIT;
  /* set size of fake trailing chunk holding overhead space only once */
  chunk_plus_offset(p, psize)->head = TOP_FOOT_SIZE;
  m->trim_check = DEFAULT_TRIM_THRESHOLD; /* reset on each update */
}

/* Initialize bins for a new mstate that is otherwise zeroed out */
static void init_bins(mstate m)
```

```
{
  /* Establish circular links for smallbins */
  bindex_t i;
  for (i = 0; i < NSMALLBINS; i++) {
    sbinptr bin = smallbin_at(m,i);
    bin->fd = bin->bk = bin;
  }
}

/* Allocate chunk and prepend remainder with chunk in successor base. */
static void *prepend_alloc(mstate m, char *newbase, char *oldbase, size_t nb)
{
  mchunkptr p = align_as_chunk(newbase);
  mchunkptr oldfirst = align_as_chunk(oldbase);
  size_t psize = (size_t)((char *)oldfirst - (char *)p);
  mchunkptr q = chunk_plus_offset(p, nb);
  size_t qsize = psize - nb;
  set_size_and_pinuse_of_inuse_chunk(m, p, nb);

  /* consolidate remainder with first chunk of old base */
  if (oldfirst == m->top) {
    size_t tsize = m->topsize += qsize;
    m->top = q;
    q->head = tsize | PINUSE_BIT;
  } else if (oldfirst == m->dv) {
    size_t dsize = m->dvsize += qsize;
    m->dv = q;
    set_size_and_pinuse_of_free_chunk(q, dsize);
  } else {
    if (!cinuse(oldfirst)) {
      size_t nsize = chunksize(oldfirst);
      unlink_chunk(m, oldfirst, nsize);
      oldfirst = chunk_plus_offset(oldfirst, nsize);
      qsize += nsize;
    }
    set_free_with_pinuse(q, qsize, oldfirst);
    insert_chunk(m, q, qsize);
  }

  return chunk2mem(p);
}

/* Add a segment to hold a new noncontiguous region */
static void add_segment(mstate m, char *tbase, size_t tsize)
{
  /* Determine locations and sizes of segment, fenceposts, old top */
  char *old_top = (char *)m->top;
  msegmentptr oldsp = segment_holding(m, old_top);
  char *old_end = oldsp->base + oldsp->size;
  size_t ssize = pad_request(sizeof(struct malloc_segment));
  char *rawsp = old_end - (ssize + FOUR_SIZE_T_SIZES + CHUNK_ALIGN_MASK);
  size_t offset = align_offset(chunk2mem(rawsp));
  char *asp = rawsp + offset;
  char *csp = (asp < (old_top + MIN_CHUNK_SIZE))? old_top : asp;
  mchunkptr sp = (mchunkptr)csp;
  msegmentptr ss = (msegmentptr)(chunk2mem(sp));
  mchunkptr tnext = chunk_plus_offset(sp, ssize);
  mchunkptr p = tnext;

  /* reset top to new space */
  init_top(m, (mchunkptr)tbase, tsize - TOP_FOOT_SIZE);

  /* Set up segment record */
  set_size_and_pinuse_of_inuse_chunk(m, sp, ssize);
  *ss = m->seg; /* Push current record */
  m->seg.base = tbase;
  m->seg.size = tsize;
  m->seg.next = ss;

  /* Insert trailing fenceposts */
  for (;;) {
    mchunkptr nextp = chunk_plus_offset(p, SIZE_T_SIZE);
    p->head = FENCEPOST_HEAD;
    if ((char *)(&(nextp->head)) < old_end)
      p = nextp;
    else
```

```
888       break;
889     }
890
891     /* Insert the rest of old top into a bin as an ordinary free chunk */
892     if (csp != old_top) {
893       mchunkptr q = (mchunkptr)old_top;
894       size_t psize = (size_t)(csp - old_top);
895       mchunkptr tn = chunk_plus_offset(q, psize);
896       set_free_with_pinuse(q, psize, tn);
897       insert_chunk(m, q, psize);
898     }
899   }
900
901   /* ------------------------- System allocation -------------------------- */
902
903   static void *alloc_sys(mstate m, size_t nb)
904   {
905     char *tbase = CMFAIL;
906     size_t tsize = 0;
907
908     /* Directly map large chunks */
909     if (LJ_UNLIKELY(nb >= DEFAULT_MMAP_THRESHOLD)) {
910       void *mem = direct_alloc(nb);
911       if (mem != 0)
912         return mem;
913     }
914
915     {
916       size_t req = nb + TOP_FOOT_SIZE + SIZE_T_ONE;
917       size_t rsize = granularity_align(req);
918       if (LJ_LIKELY(rsize > nb)) { /* Fail if wraps around zero */
919         char *mp = (char *)(CALL_MMAP(rsize));
920         if (mp != CMFAIL) {
921           tbase = mp;
922           tsize = rsize;
923         }
924       }
925     }
926
927     if (tbase != CMFAIL) {
928       msegmentptr sp = &m->seg;
929       /* Try to merge with an existing segment */
930       while (sp != 0 && tbase != sp->base + sp->size)
931         sp = sp->next;
932       if (sp != 0 && segment_holds(sp, m->top)) { /* append */
933         sp->size += tsize;
934         init_top(m, m->top, m->topsize + tsize);
935       } else {
936         sp = &m->seg;
937         while (sp != 0 && sp->base != tbase + tsize)
938           sp = sp->next;
939         if (sp != 0) {
940           char *oldbase = sp->base;
941           sp->base = tbase;
942           sp->size += tsize;
943           return prepend_alloc(m, tbase, oldbase, nb);
944         } else {
945           add_segment(m, tbase, tsize);
946         }
947       }
948
949       if (nb < m->topsize) { /* Allocate from new or extended top space */
950         size_t rsize = m->topsize -= nb;
951         mchunkptr p = m->top;
952         mchunkptr r = m->top = chunk_plus_offset(p, nb);
953         r->head = rsize | PINUSE_BIT;
954         set_size_and_pinuse_of_inuse_chunk(m, p, nb);
955         return chunk2mem(p);
956       }
957     }
958
959     return NULL;
960   }
961
962   /* ----------------------  system deallocation -------------------------- */
963
```

```c
/* Unmap and unlink any mmapped segments that don't contain used chunks */
static size_t release_unused_segments(mstate m)
{
  size_t released = 0;
  size_t nsegs = 0;
  msegmentptr pred = &m->seg;
  msegmentptr sp = pred->next;
  while (sp != 0) {
    char *base = sp->base;
    size_t size = sp->size;
    msegmentptr next = sp->next;
    nsegs++;
    {
      mchunkptr p = align_as_chunk(base);
      size_t psize = chunksize(p);
      /* Can unmap if first chunk holds entire segment and not pinned */
      if (!cinuse(p) && (char *)p + psize >= base + size - TOP_FOOT_SIZE) {
        tchunkptr tp = (tchunkptr)p;
        if (p == m->dv) {
          m->dv = 0;
          m->dvsize = 0;
        } else {
          unlink_large_chunk(m, tp);
        }
        if (CALL_MUNMAP(base, size) == 0) {
          released += size;
          /* unlink obsoleted record */
          sp = pred;
          sp->next = next;
        } else { /* back out if cannot unmap */
          insert_large_chunk(m, tp, psize);
        }
      }
    }
    pred = sp;
    sp = next;
  }
  /* Reset check counter */
  m->release_checks = nsegs > MAX_RELEASE_CHECK_RATE ?
                      nsegs : MAX_RELEASE_CHECK_RATE;
  return released;
}

static int alloc_trim(mstate m, size_t pad)
{
  size_t released = 0;
  if (pad < MAX_REQUEST && is_initialized(m)) {
    pad += TOP_FOOT_SIZE; /* ensure enough room for segment overhead */

    if (m->topsize > pad) {
      /* Shrink top space in granularity-size units, keeping at least one */
      size_t unit = DEFAULT_GRANULARITY;
      size_t extra = ((m->topsize - pad + (unit - SIZE_T_ONE)) / unit -
                      SIZE_T_ONE) * unit;
      msegmentptr sp = segment_holding(m, (char *)m->top);

      if (sp->size >= extra &&
          !has_segment_link(m, sp)) { /* can't shrink if pinned */
        size_t newsize = sp->size - extra;
        /* Prefer mremap, fall back to munmap */
        if ((CALL_MREMAP(sp->base, sp->size, newsize, CALL_MREMAP_NOMOVE) != MFAIL) ||
            (CALL_MUNMAP(sp->base + newsize, extra) == 0)) {
          released = extra;
        }
      }

      if (released != 0) {
        sp->size -= released;
        init_top(m, m->top, m->topsize - released);
      }
    }

    /* Unmap any unused mmapped segments */
    released += release_unused_segments(m);

    /* On failure, disable autotrim to avoid repeated failed future calls */
```

```
    if (released == 0 && m->topsize > m->trim_check)
      m->trim_check = MAX_SIZE_T;
  }

  return (released != 0)? 1 : 0;
}

/* ---------------------------- malloc support --------------------------- */

/* allocate a large request from the best fitting chunk in a treebin */
static void *tmalloc_large(mstate m, size_t nb)
{
  tchunkptr v = 0;
  size_t rsize = ~nb+1; /* Unsigned negation */
  tchunkptr t;
  bindex_t idx;
  compute_tree_index(nb, idx);

  if ((t = *treebin_at(m, idx)) != 0) {
    /* Traverse tree for this bin looking for node with size == nb */
    size_t sizebits = nb << leftshift_for_tree_index(idx);
    tchunkptr rst = 0;  /* The deepest untaken right subtree */
    for (;;) {
      tchunkptr rt;
      size_t trem = chunksize(t) - nb;
      if (trem < rsize) {
        v = t;
        if ((rsize = trem) == 0)
          break;
      }
      rt = t->child[1];
      t = t->child[(sizebits >> (SIZE_T_BITSIZE-SIZE_T_ONE)) & 1];
      if (rt != 0 && rt != t)
        rst = rt;
      if (t == 0) {
        t = rst; /* set t to least subtree holding sizes > nb */
        break;
      }
      sizebits <<= 1;
    }
  }

  if (t == 0 && v == 0) { /* set t to root of next non-empty treebin */
    binmap_t leftbits = left_bits(idx2bit(idx)) & m->treemap;
    if (leftbits != 0)
      t = *treebin_at(m, lj_ffs(leftbits));
  }

  while (t != 0) { /* find smallest of tree or subtree */
    size_t trem = chunksize(t) - nb;
    if (trem < rsize) {
      rsize = trem;
      v = t;
    }
    t = leftmost_child(t);
  }

  /*  If dv is a better fit, return NULL so malloc will use it */
  if (v != 0 && rsize < (size_t)(m->dvsize - nb)) {
    mchunkptr r = chunk_plus_offset(v, nb);
    unlink_large_chunk(m, v);
    if (rsize < MIN_CHUNK_SIZE) {
      set_inuse_and_pinuse(m, v, (rsize + nb));
    } else {
      set_size_and_pinuse_of_inuse_chunk(m, v, nb);
      set_size_and_pinuse_of_free_chunk(r, rsize);
      insert_chunk(m, r, rsize);
    }
    return chunk2mem(v);
  }
  return NULL;
}

/* allocate a small request from the best fitting chunk in a treebin */
static void *tmalloc_small(mstate m, size_t nb)
{
```

```
  tchunkptr t, v;
  mchunkptr r;
  size_t rsize;
  bindex_t i = lj_ffs(m->treemap);

  v = t = *treebin_at(m, i);
  rsize = chunksize(t) - nb;

  while ((t = leftmost_child(t)) != 0) {
    size_t trem = chunksize(t) - nb;
    if (trem < rsize) {
      rsize = trem;
      v = t;
    }
  }

  r = chunk_plus_offset(v, nb);
  unlink_large_chunk(m, v);
  if (rsize < MIN_CHUNK_SIZE) {
    set_inuse_and_pinuse(m, v, (rsize + nb));
  } else {
    set_size_and_pinuse_of_inuse_chunk(m, v, nb);
    set_size_and_pinuse_of_free_chunk(r, rsize);
    replace_dv(m, r, rsize);
  }
  return chunk2mem(v);
}

/* ---------------------------------------------------------------------- */

void *lj_alloc_create(void)
{
  size_t tsize = DEFAULT_GRANULARITY;
  char *tbase;
  INIT_MMAP();
  tbase = (char *)(CALL_MMAP(tsize));
  if (tbase != CMFAIL) {
    size_t msize = pad_request(sizeof(struct malloc_state));
    mchunkptr mn;
    mchunkptr msp = align_as_chunk(tbase);
    mstate m = (mstate)(chunk2mem(msp));
    memset(m, 0, msize);
    msp->head = (msize|PINUSE_BIT|CINUSE_BIT);
    m->seg.base = tbase;
    m->seg.size = tsize;
    m->release_checks = MAX_RELEASE_CHECK_RATE;
    init_bins(m);
    mn = next_chunk(mem2chunk(m));
    init_top(m, mn, (size_t)((tbase + tsize) - (char *)mn) - TOP_FOOT_SIZE);
    return m;
  }
  return NULL;
}

void lj_alloc_destroy(void *msp)
{
  mstate ms = (mstate)msp;
  msegmentptr sp = &ms->seg;
  while (sp != 0) {
    char *base = sp->base;
    size_t size = sp->size;
    sp = sp->next;
    CALL_MUNMAP(base, size);
  }
}

static LJ_NOINLINE void *lj_alloc_malloc(void *msp, size_t nsize)
{
  mstate ms = (mstate)msp;
  void *mem;
  size_t nb;
  if (nsize <= MAX_SMALL_REQUEST) {
    bindex_t idx;
    binmap_t smallbits;
    nb = (nsize < MIN_REQUEST)? MIN_CHUNK_SIZE : pad_request(nsize);
    idx = small_index(nb);
```

```
1192      smallbits = ms->smallmap >> idx;
1193
1194      if ((smallbits & 0x3U) != 0) { /* Remainderless fit to a smallbin. */
1195        mchunkptr b, p;
1196        idx += ~smallbits & 1;       /* Uses next bin if idx empty */
1197        b = smallbin_at(ms, idx);
1198        p = b->fd;
1199        unlink_first_small_chunk(ms, b, p, idx);
1200        set_inuse_and_pinuse(ms, p, small_index2size(idx));
1201        mem = chunk2mem(p);
1202        return mem;
1203      } else if (nb > ms->dvsize) {
1204        if (smallbits != 0) { /* Use chunk in next nonempty smallbin */
1205          mchunkptr b, p, r;
1206          size_t rsize;
1207          binmap_t leftbits = (smallbits << idx) & left_bits(idx2bit(idx));
1208          bindex_t i = lj_ffs(leftbits);
1209          b = smallbin_at(ms, i);
1210          p = b->fd;
1211          unlink_first_small_chunk(ms, b, p, i);
1212          rsize = small_index2size(i) - nb;
1213          /* Fit here cannot be remainderless if 4byte sizes */
1214          if (SIZE_T_SIZE != 4 && rsize < MIN_CHUNK_SIZE) {
1215            set_inuse_and_pinuse(ms, p, small_index2size(i));
1216          } else {
1217            set_size_and_pinuse_of_inuse_chunk(ms, p, nb);
1218            r = chunk_plus_offset(p, nb);
1219            set_size_and_pinuse_of_free_chunk(r, rsize);
1220            replace_dv(ms, r, rsize);
1221          }
1222          mem = chunk2mem(p);
1223          return mem;
1224        } else if (ms->treemap != 0 && (mem = tmalloc_small(ms, nb)) != 0) {
1225          return mem;
1226        }
1227      }
1228    } else if (nsize >= MAX_REQUEST) {
1229      nb = MAX_SIZE_T; /* Too big to allocate. Force failure (in sys alloc) */
1230    } else {
1231      nb = pad_request(nsize);
1232      if (ms->treemap != 0 && (mem = tmalloc_large(ms, nb)) != 0) {
1233        return mem;
1234      }
1235    }
1236
1237    if (nb <= ms->dvsize) {
1238      size_t rsize = ms->dvsize - nb;
1239      mchunkptr p = ms->dv;
1240      if (rsize >= MIN_CHUNK_SIZE) { /* split dv */
1241        mchunkptr r = ms->dv = chunk_plus_offset(p, nb);
1242        ms->dvsize = rsize;
1243        set_size_and_pinuse_of_free_chunk(r, rsize);
1244        set_size_and_pinuse_of_inuse_chunk(ms, p, nb);
1245      } else { /* exhaust dv */
1246        size_t dvs = ms->dvsize;
1247        ms->dvsize = 0;
1248        ms->dv = 0;
1249        set_inuse_and_pinuse(ms, p, dvs);
1250      }
1251      mem = chunk2mem(p);
1252      return mem;
1253    } else if (nb < ms->topsize) { /* Split top */
1254      size_t rsize = ms->topsize -= nb;
1255      mchunkptr p = ms->top;
1256      mchunkptr r = ms->top = chunk_plus_offset(p, nb);
1257      r->head = rsize | PINUSE_BIT;
1258      set_size_and_pinuse_of_inuse_chunk(ms, p, nb);
1259      mem = chunk2mem(p);
1260      return mem;
1261    }
1262    return alloc_sys(ms, nb);
1263  }
1264
1265  static LJ_NOINLINE void *lj_alloc_free(void *msp, void *ptr)
1266  {
1267    if (ptr != 0) {
```

```
    mchunkptr p = mem2chunk(ptr);
    mstate fm = (mstate)msp;
    size_t psize = chunksize(p);
    mchunkptr next = chunk_plus_offset(p, psize);
    if (!pinuse(p)) {
      size_t prevsize = p->prev_foot;
      if ((prevsize & IS_DIRECT_BIT) != 0) {
        prevsize &= ~IS_DIRECT_BIT;
        psize += prevsize + DIRECT_FOOT_PAD;
        CALL_MUNMAP((char *)p - prevsize, psize);
        return NULL;
      } else {
        mchunkptr prev = chunk_minus_offset(p, prevsize);
        psize += prevsize;
        p = prev;
        /* consolidate backward */
        if (p != fm->dv) {
          unlink_chunk(fm, p, prevsize);
        } else if ((next->head & INUSE_BITS) == INUSE_BITS) {
          fm->dvsize = psize;
          set_free_with_pinuse(p, psize, next);
          return NULL;
        }
      }
    }
    if (!cinuse(next)) {  /* consolidate forward */
      if (next == fm->top) {
        size_t tsize = fm->topsize += psize;
        fm->top = p;
        p->head = tsize | PINUSE_BIT;
        if (p == fm->dv) {
          fm->dv = 0;
          fm->dvsize = 0;
        }
        if (tsize > fm->trim_check)
          alloc_trim(fm, 0);
        return NULL;
      } else if (next == fm->dv) {
        size_t dsize = fm->dvsize += psize;
        fm->dv = p;
        set_size_and_pinuse_of_free_chunk(p, dsize);
        return NULL;
      } else {
        size_t nsize = chunksize(next);
        psize += nsize;
        unlink_chunk(fm, next, nsize);
        set_size_and_pinuse_of_free_chunk(p, psize);
        if (p == fm->dv) {
          fm->dvsize = psize;
          return NULL;
        }
      }
    } else {
      set_free_with_pinuse(p, psize, next);
    }

    if (is_small(psize)) {
      insert_small_chunk(fm, p, psize);
    } else {
      tchunkptr tp = (tchunkptr)p;
      insert_large_chunk(fm, tp, psize);
      if (--fm->release_checks == 0)
        release_unused_segments(fm);
    }
  }
  return NULL;
}

static LJ_NOINLINE void *lj_alloc_realloc(void *msp, void *ptr, size_t nsize)
{
  if (nsize >= MAX_REQUEST) {
    return NULL;
  } else {
    mstate m = (mstate)msp;
    mchunkptr oldp = mem2chunk(ptr);
    size_t oldsize = chunksize(oldp);
```

```
1344      mchunkptr next = chunk_plus_offset(oldp, oldsize);
1345      mchunkptr newp = 0;
1346      size_t nb = request2size(nsize);
1347
1348      /* Try to either shrink or extend into top. Else malloc-copy-free */
1349      if (is_direct(oldp)) {
1350        newp = direct_resize(oldp, nb);  /* this may return NULL. */
1351      } else if (oldsize >= nb) { /* already big enough */
1352        size_t rsize = oldsize - nb;
1353        newp = oldp;
1354        if (rsize >= MIN_CHUNK_SIZE) {
1355          mchunkptr rem = chunk_plus_offset(newp, nb);
1356          set_inuse(m, newp, nb);
1357          set_inuse(m, rem, rsize);
1358          lj_alloc_free(m, chunk2mem(rem));
1359        }
1360      } else if (next == m->top && oldsize + m->topsize > nb) {
1361        /* Expand into top */
1362        size_t newsize = oldsize + m->topsize;
1363        size_t newtopsize = newsize - nb;
1364        mchunkptr newtop = chunk_plus_offset(oldp, nb);
1365        set_inuse(m, oldp, nb);
1366        newtop->head = newtopsize |PINUSE_BIT;
1367        m->top = newtop;
1368        m->topsize = newtopsize;
1369        newp = oldp;
1370      }
1371
1372      if (newp != 0) {
1373        return chunk2mem(newp);
1374      } else {
1375        void *newmem = lj_alloc_malloc(m, nsize);
1376        if (newmem != 0) {
1377          size_t oc = oldsize - overhead_for(oldp);
1378          memcpy(newmem, ptr, oc < nsize ? oc : nsize);
1379          lj_alloc_free(m, ptr);
1380        }
1381        return newmem;
1382      }
1383    }
1384  }
1385
1386  void *lj_alloc_f(void *msp, void *ptr, size_t osize, size_t nsize)
1387  {
1388    (void)osize;
1389    if (nsize == 0) {
1390      return lj_alloc_free(msp, ptr);
1391    } else if (ptr == NULL) {
1392      return lj_alloc_malloc(msp, nsize);
1393    } else {
1394      return lj_alloc_realloc(msp, ptr, nsize);
1395    }
1396  }
1397
1398  #endif
```

One Level Up Top Level

src/ljamalg.c - luajit-2.0-src

Macros defined

- LUA_CORE
- WINVER
- _GNU_SOURCE
- ljamalg_c

Source code

```
1  /*
2  ** LuaJIT core and libraries amalgamation.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  /*
7  +-------------------------------------------------------------------------+
8  | WARNING: Compiling the amalgamation needs a lot of virtual memory       |
9  | (around 300 MB with GCC 4.x)! If you don't have enough physical memory  |
10 | your machine will start swapping to disk and the compile will not finish |
11 | within a reasonable amount of time.                                     |
12 | So either compile on a bigger machine or use the non-amalgamated build. |
13 +-------------------------------------------------------------------------+
14 */
15
16 #define ljamalg_c
17 #define LUA_CORE
18
19 /* To get the mremap prototype. Must be defined before any system includes. */
20 #if defined(__linux__) && !defined(_GNU_SOURCE)
21 #define _GNU_SOURCE
22 #endif
23
24 #ifndef WINVER
25 #define WINVER 0x0501
26 #endif
27
28 #include "lua.h"
29 #include "lauxlib.h"
30
31 #include "lj_gc.c"
32 #include "lj_err.c"
33 #include "lj_char.c"
34 #include "lj_bc.c"
35 #include "lj_obj.c"
36 #include "lj_buf.c"
37 #include "lj_str.c"
38 #include "lj_tab.c"
39 #include "lj_func.c"
40 #include "lj_udata.c"
41 #include "lj_meta.c"
42 #include "lj_debug.c"
43 #include "lj_state.c"
44 #include "lj_dispatch.c"
45 #include "lj_vmevent.c"
46 #include "lj_vmmath.c"
47 #include "lj_strscan.c"
48 #include "lj_strfmt.c"
49 #include "lj_api.c"
50 #include "lj_profile.c"
51 #include "lj_lex.c"
52 #include "lj_parse.c"
53 #include "lj_bcread.c"
54 #include "lj_bcwrite.c"
55 #include "lj_load.c"
56 #include "lj_ctype.c"
```

```
57 #include "lj_cdata.c"
58 #include "lj_cconv.c"
59 #include "lj_ccall.c"
60 #include "lj_ccallback.c"
61 #include "lj_carith.c"
62 #include "lj_clib.c"
63 #include "lj_cparse.c"
64 #include "lj_lib.c"
65 #include "lj_ir.c"
66 #include "lj_opt_mem.c"
67 #include "lj_opt_fold.c"
68 #include "lj_opt_narrow.c"
69 #include "lj_opt_dce.c"
70 #include "lj_opt_loop.c"
71 #include "lj_opt_split.c"
72 #include "lj_opt_sink.c"
73 #include "lj_mcode.c"
74 #include "lj_snap.c"
75 #include "lj_record.c"
76 #include "lj_crecord.c"
77 #include "lj_ffrecord.c"
78 #include "lj_asm.c"
79 #include "lj_trace.c"
80 #include "lj_gdbjit.c"
81 #include "lj_alloc.c"
82
83 #include "lib_aux.c"
84 #include "lib_base.c"
85 #include "lib_math.c"
86 #include "lib_string.c"
87 #include "lib_table.c"
88 #include "lib_io.c"
89 #include "lib_os.c"
90 #include "lib_package.c"
91 #include "lib_debug.c"
92 #include "lib_bit.c"
93 #include "lib_jit.c"
94 #include "lib_ffi.c"
95 #include "lib_init.c"
96
```

One Level Up Top Level

src/lj_bcread.c - luajit-2.0-src

Functions defined

Macros defined

Source code

```
1 /*
2 ** Bytecode reader.
```

```
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_bcread_c
#define LUA_CORE

#include "lj_obj.h"
#include "lj_gc.h"
#include "lj_err.h"
#include "lj_buf.h"
#include "lj_str.h"
#include "lj_tab.h"
#include "lj_bc.h"
#if LJ_HASFFI
#include "lj_ctype.h"
#include "lj_cdata.h"
#include "lualib.h"
#endif
#include "lj_lex.h"
#include "lj_bcdump.h"
#include "lj_state.h"
#include "lj_strfmt.h"

/* Reuse some lexer fields for our own purposes. */
#define bcread_flags(ls)	ls->level
#define bcread_swap(ls) \
  ((bcread_flags(ls) & BCDUMP_F_BE) != LJ_BE*BCDUMP_F_BE)
#define bcread_oldtop(L, ls)	restorestack(L, ls->lastline)
#define bcread_savetop(L, ls, top) \
  ls->lastline = (BCLine)savestack(L, (top))

/* -- Input buffer handling ---------------------------------------------- */

/* Throw reader error. */
static LJ_NOINLINE void bcread_error(LexState *ls, ErrMsg em)
{
  lua_State *L = ls->L;
  const char *name = ls->chunkarg;
  if (*name == BCDUMP_HEAD1) name = "(binary)";
  else if (*name == '@' || *name == '=') name++;
  lj_strfmt_pushf(L, "%s: %s", name, err2msg(em));
  lj_err_throw(L, LUA_ERRSYNTAX);
}

/* Refill buffer. */
static LJ_NOINLINE void bcread_fill(LexState *ls, MSize len, int need)
{
  lua_assert(len != 0);
  if (len > LJ_MAX_BUF || ls->c < 0)
    bcread_error(ls, LJ_ERR_BCBAD);
  do {
    const char *buf;
    size_t sz;
    char *p = sbufB(&ls->sb);
    MSize n = (MSize)(ls->pe - ls->p);
    if (n) {  /* Copy remainder to buffer. */
      if (sbuflen(&ls->sb)) {  /* Move down in buffer. */
        lua_assert(ls->pe == sbufP(&ls->sb));
        if (ls->p != p) memmove(p, ls->p, n);
      } else {  /* Copy from buffer provided by reader. */
        p = lj_buf_need(&ls->sb, len);
        memcpy(p, ls->p, n);
      }
      ls->p = p;
      ls->pe = p + n;
    }
    setsbufP(&ls->sb, p + n);
    buf = ls->rfunc(ls->L, ls->rdata, &sz);  /* Get more data from reader. */
    if (buf == NULL || sz == 0) {  /* EOF? */
      if (need) bcread_error(ls, LJ_ERR_BCBAD);
      ls->c = -1;  /* Only bad if we get called again. */
      break;
    }
    if (n) {  /* Append to buffer. */
      n += (MSize)sz;
      p = lj_buf_need(&ls->sb, n < len ? len : n);
```

```
      memcpy(sbufP(&ls->sb), buf, sz);
      setsbufP(&ls->sb, p + n);
      ls->p = p;
      ls->pe = p + n;
    } else {  /* Return buffer provided by reader. */
      ls->p = buf;
      ls->pe = buf + sz;
    }
  } while (ls->p + len > ls->pe);
}

/* Need a certain number of bytes. */
static LJ_AINLINE void bcread_need(LexState *ls, MSize len)
{
  if (LJ_UNLIKELY(ls->p + len > ls->pe))
    bcread_fill(ls, len, 1);
}

/* Want to read up to a certain number of bytes, but may need less. */
static LJ_AINLINE void bcread_want(LexState *ls, MSize len)
{
  if (LJ_UNLIKELY(ls->p + len > ls->pe))
    bcread_fill(ls, len, 0);
}

/* Return memory block from buffer. */
static LJ_AINLINE uint8_t *bcread_mem(LexState *ls, MSize len)
{
  uint8_t *p = (uint8_t *)ls->p;
  ls->p += len;
  lua_assert(ls->p <= ls->pe);
  return p;
}

/* Copy memory block from buffer. */
static void bcread_block(LexState *ls, void *q, MSize len)
{
  memcpy(q, bcread_mem(ls, len), len);
}

/* Read byte from buffer. */
static LJ_AINLINE uint32_t bcread_byte(LexState *ls)
{
  lua_assert(ls->p < ls->pe);
  return (uint32_t)(uint8_t)*ls->p++;
}

/* Read ULEB128 value from buffer. */
static LJ_AINLINE uint32_t bcread_uleb128(LexState *ls)
{
  uint32_t v = lj_buf_ruleb128(&ls->p);
  lua_assert(ls->p <= ls->pe);
  return v;
}

/* Read top 32 bits of 33 bit ULEB128 value from buffer. */
static uint32_t bcread_uleb128_33(LexState *ls)
{
  const uint8_t *p = (const uint8_t *)ls->p;
  uint32_t v = (*p++ >> 1);
  if (LJ_UNLIKELY(v >= 0x40)) {
    int sh = -1;
    v &= 0x3f;
    do {
      v |= ((*p & 0x7f) << (sh += 7));
    } while (*p++ >= 0x80);
  }
  ls->p = (char *)p;
  lua_assert(ls->p <= ls->pe);
  return v;
}

/* -- Bytecode reader ------------------------------------------------------ */

/* Read debug info of a prototype. */
static void bcread_dbg(LexState *ls, GCproto *pt, MSize sizedbg)
```

```
{
  void *lineinfo = (void *)proto_lineinfo(pt);
  bcread_block(ls, lineinfo, sizedbg);
  /* Swap lineinfo if the endianess differs. */
  if (bcread_swap(ls) && pt->numline >= 256) {
    MSize i, n = pt->sizebc-1;
    if (pt->numline < 65536) {
      uint16_t *p = (uint16_t *)lineinfo;
      for (i = 0; i < n; i++) p[i] = (uint16_t)((p[i] >> 8)|(p[i] << 8));
    } else {
      uint32_t *p = (uint32_t *)lineinfo;
      for (i = 0; i < n; i++) p[i] = lj_bswap(p[i]);
    }
  }
}

/* Find pointer to varinfo. */
static const void *bcread_varinfo(GCproto *pt)
{
  const uint8_t *p = proto_uvinfo(pt);
  MSize n = pt->sizeuv;
  if (n) while (*p++ || --n) ;
  return p;
}

/* Read a single constant key/value of a template table. */
static void bcread_ktabk(LexState *ls, TValue *o)
{
  MSize tp = bcread_uleb128(ls);
  if (tp >= BCDUMP_KTAB_STR) {
    MSize len = tp - BCDUMP_KTAB_STR;
    const char *p = (const char *)bcread_mem(ls, len);
    setstrV(ls->L, o, lj_str_new(ls->L, p, len));
  } else if (tp == BCDUMP_KTAB_INT) {
    setintV(o, (int32_t)bcread_uleb128(ls));
  } else if (tp == BCDUMP_KTAB_NUM) {
    o->u32.lo = bcread_uleb128(ls);
    o->u32.hi = bcread_uleb128(ls);
  } else {
    lua_assert(tp <= BCDUMP_KTAB_TRUE);
    setpriV(o, ~tp);
  }
}

/* Read a template table. */
static GCtab *bcread_ktab(LexState *ls)
{
  MSize narray = bcread_uleb128(ls);
  MSize nhash = bcread_uleb128(ls);
  GCtab *t = lj_tab_new(ls->L, narray, hsize2hbits(nhash));
  if (narray) {  /* Read array entries. */
    MSize i;
    TValue *o = tvref(t->array);
    for (i = 0; i < narray; i++, o++)
      bcread_ktabk(ls, o);
  }
  if (nhash) {  /* Read hash entries. */
    MSize i;
    for (i = 0; i < nhash; i++) {
      TValue key;
      bcread_ktabk(ls, &key);
      lua_assert(!tvisnil(&key));
      bcread_ktabk(ls, lj_tab_set(ls->L, t, &key));
    }
  }
  return t;
}

/* Read GC constants of a prototype. */
static void bcread_kgc(LexState *ls, GCproto *pt, MSize sizekgc)
{
  MSize i;
  GCRef *kr = mref(pt->k, GCRef) - (ptrdiff_t)sizekgc;
  for (i = 0; i < sizekgc; i++, kr++) {
    MSize tp = bcread_uleb128(ls);
    if (tp >= BCDUMP_KGC_STR) {
```

```
      MSize len = tp - BCDUMP_KGC_STR;
      const char *p = (const char *)bcread_mem(ls, len);
      setgcref(*kr, obj2gco(lj_str_new(ls->L, p, len)));
    } else if (tp == BCDUMP_KGC_TAB) {
      setgcref(*kr, obj2gco(bcread_ktab(ls)));
#if LJ_HASFFI
    } else if (tp != BCDUMP_KGC_CHILD) {
      CTypeID id = tp == BCDUMP_KGC_COMPLEX ? CTID_COMPLEX_DOUBLE :
                   tp == BCDUMP_KGC_I64 ? CTID_INT64 : CTID_UINT64;
      CTSize sz = tp == BCDUMP_KGC_COMPLEX ? 16 : 8;
      GCcdata *cd = lj_cdata_new(ls->L, id, sz);
      TValue *p = (TValue *)cdataptr(cd);
      setgcref(*kr, obj2gco(cd));
      p[0].u32.lo = bcread_uleb128(ls);
      p[0].u32.hi = bcread_uleb128(ls);
      if (tp == BCDUMP_KGC_COMPLEX) {
        p[1].u32.lo = bcread_uleb128(ls);
        p[1].u32.hi = bcread_uleb128(ls);
      }
#endif
    } else {
      lua_State *L = ls->L;
      lua_assert(tp == BCDUMP_KGC_CHILD);
      if (L->top <= bcread_oldtop(L, ls))  /* Stack underflow? */
        bcread_error(ls, LJ_ERR_BCBAD);
      L->top--;
      setgcref(*kr, obj2gco(protoV(L->top)));
    }
  }
}

/* Read number constants of a prototype. */
static void bcread_knum(LexState *ls, GCproto *pt, MSize sizekn)
{
  MSize i;
  TValue *o = mref(pt->k, TValue);
  for (i = 0; i < sizekn; i++, o++) {
    int isnum = (ls->p[0] & 1);
    uint32_t lo = bcread_uleb128_33(ls);
    if (isnum) {
      o->u32.lo = lo;
      o->u32.hi = bcread_uleb128(ls);
    } else {
      setintV(o, lo);
    }
  }
}

/* Read bytecode instructions. */
static void bcread_bytecode(LexState *ls, GCproto *pt, MSize sizebc)
{
  BCIns *bc = proto_bc(pt);
  bc[0] = BCINS_AD((pt->flags & PROTO_VARARG) ? BC_FUNCV : BC_FUNCF,
                   pt->framesize, 0);
  bcread_block(ls, bc+1, (sizebc-1)*(MSize)sizeof(BCIns));
  /* Swap bytecode instructions if the endianess differs. */
  if (bcread_swap(ls)) {
    MSize i;
    for (i = 1; i < sizebc; i++) bc[i] = lj_bswap(bc[i]);
  }
}

/* Read upvalue refs. */
static void bcread_uv(LexState *ls, GCproto *pt, MSize sizeuv)
{
  if (sizeuv) {
    uint16_t *uv = proto_uv(pt);
    bcread_block(ls, uv, sizeuv*2);
    /* Swap upvalue refs if the endianess differs. */
    if (bcread_swap(ls)) {
      MSize i;
      for (i = 0; i < sizeuv; i++)
        uv[i] = (uint16_t)((uv[i] >> 8)|(uv[i] << 8));
    }
  }
}
```

```
307
308 /* Read a prototype. */
309 GCproto *lj_bcread_proto(LexState *ls)
310 {
311   GCproto *pt;
312   MSize framesize, numparams, flags, sizeuv, sizekgc, sizekn, sizebc, sizept;
313   MSize ofsk, ofsuv, ofsdbg;
314   MSize sizedbg = 0;
315   BCLine firstline = 0, numline = 0;
316
317   /* Read prototype header. */
318   flags = bcread_byte(ls);
319   numparams = bcread_byte(ls);
320   framesize = bcread_byte(ls);
321   sizeuv = bcread_byte(ls);
322   sizekgc = bcread_uleb128(ls);
323   sizekn = bcread_uleb128(ls);
324   sizebc = bcread_uleb128(ls) + 1;
325   if (!(bcread_flags(ls) & BCDUMP_F_STRIP)) {
326     sizedbg = bcread_uleb128(ls);
327     if (sizedbg) {
328       firstline = bcread_uleb128(ls);
329       numline = bcread_uleb128(ls);
330     }
331   }
332
333   /* Calculate total size of prototype including all colocated arrays. */
334   sizept = (MSize)sizeof(GCproto) +
335            sizebc*(MSize)sizeof(BCIns) +
336            sizekgc*(MSize)sizeof(GCRef);
337   sizept = (sizept + (MSize)sizeof(TValue)-1) & ~((MSize)sizeof(TValue)-1);
338   ofsk = sizept; sizept += sizekn*(MSize)sizeof(TValue);
339   ofsuv = sizept; sizept += ((sizeuv+1)&~1)*2;
340   ofsdbg = sizept; sizept += sizedbg;
341
342   /* Allocate prototype object and initialize its fields. */
343   pt = (GCproto *)lj_mem_newgco(ls->L, (MSize)sizept);
344   pt->gct = ~LJ_TPROTO;
345   pt->numparams = (uint8_t)numparams;
346   pt->framesize = (uint8_t)framesize;
347   pt->sizebc = sizebc;
348   setmref(pt->k, (char *)pt + ofsk);
349   setmref(pt->uv, (char *)pt + ofsuv);
350   pt->sizekgc = 0;  /* Set to zero until fully initialized. */
351   pt->sizekn = sizekn;
352   pt->sizept = sizept;
353   pt->sizeuv = (uint8_t)sizeuv;
354   pt->flags = (uint8_t)flags;
355   pt->trace = 0;
356   setgcref(pt->chunkname, obj2gco(ls->chunkname));
357
358   /* Close potentially uninitialized gap between bc and kgc. */
359   *(uint32_t *)((char *)pt + ofsk - sizeof(GCRef)*(sizekgc+1)) = 0;
360
361   /* Read bytecode instructions and upvalue refs. */
362   bcread_bytecode(ls, pt, sizebc);
363   bcread_uv(ls, pt, sizeuv);
364
365   /* Read constants. */
366   bcread_kgc(ls, pt, sizekgc);
367   pt->sizekgc = sizekgc;
368   bcread_knum(ls, pt, sizekn);
369
370   /* Read and initialize debug info. */
371   pt->firstline = firstline;
372   pt->numline = numline;
373   if (sizedbg) {
374     MSize sizeli = (sizebc-1) << (numline < 256 ? 0 : numline < 65536 ? 1 : 2);
375     setmref(pt->lineinfo, (char *)pt + ofsdbg);
376     setmref(pt->uvinfo, (char *)pt + ofsdbg + sizeli);
377     bcread_dbg(ls, pt, sizedbg);
378     setmref(pt->varinfo, bcread_varinfo(pt));
379   } else {
380     setmref(pt->lineinfo, NULL);
381     setmref(pt->uvinfo, NULL);
382     setmref(pt->varinfo, NULL);
```

```
  }
  return pt;
}

/* Read and check header of bytecode dump. */
static int bcread_header(LexState *ls)
{
  uint32_t flags;
  bcread_want(ls, 3+5+5);
  if (bcread_byte(ls) != BCDUMP_HEAD2 ||
      bcread_byte(ls) != BCDUMP_HEAD3 ||
      bcread_byte(ls) != BCDUMP_VERSION) return 0;
  bcread_flags(ls) = flags = bcread_uleb128(ls);
  if ((flags & ~(BCDUMP_F_KNOWN)) != 0) return 0;
  if ((flags & BCDUMP_F_FR2) != LJ_FR2*BCDUMP_F_FR2) return 0;
  if ((flags & BCDUMP_F_FFI)) {
#if LJ_HASFFI
    lua_State *L = ls->L;
    if (!ctype_ctsG(G(L))) {
      ptrdiff_t oldtop = savestack(L, L->top);
      luaopen_ffi(L);  /* Load FFI library on-demand. */
      L->top = restorestack(L, oldtop);
    }
#else
    return 0;
#endif
  }
  if ((flags & BCDUMP_F_STRIP)) {
    ls->chunkname = lj_str_newz(ls->L, ls->chunkarg);
  } else {
    MSize len = bcread_uleb128(ls);
    bcread_need(ls, len);
    ls->chunkname = lj_str_new(ls->L, (const char *)bcread_mem(ls, len), len);
  }
  return 1;  /* Ok. */
}

/* Read a bytecode dump. */
GCproto *lj_bcread(LexState *ls)
{
  lua_State *L = ls->L;
  lua_assert(ls->c == BCDUMP_HEAD1);
  bcread_savetop(L, ls, L->top);
  lj_buf_reset(&ls->sb);
  /* Check for a valid bytecode dump header. */
  if (!bcread_header(ls))
    bcread_error(ls, LJ_ERR_BCFMT);
  for (;;) {  /* Process all prototypes in the bytecode dump. */
    GCproto *pt;
    MSize len;
    const char *startp;
    /* Read length. */
    if (ls->p < ls->pe && ls->p[0] == 0) {  /* Shortcut EOF. */
      ls->p++;
      break;
    }
    bcread_want(ls, 5);
    len = bcread_uleb128(ls);
    if (!len) break;  /* EOF */
    bcread_need(ls, len);
    startp = ls->p;
    pt = lj_bcread_proto(ls);
    if (ls->p != startp + len)
      bcread_error(ls, LJ_ERR_BCBAD);
    setprotoV(L, L->top, pt);
    incr_top(L);
  }
  if ((int32_t)(2*(uint32_t)(ls->pe - ls->p)) > 0 ||
      L->top-1 != bcread_oldtop(L, ls))
    bcread_error(ls, LJ_ERR_BCBAD);
  /* Pop off last prototype. */
  L->top--;
  return protoV(L->top);
}
```

src/lj_bcdump.h - luajit-2.0-src

Macros defined

- BCDUMP_F_BE
- BCDUMP_F_FFI
- BCDUMP_F_FR2
- BCDUMP_F_KNOWN
- BCDUMP_F_STRIP
- BCDUMP_HEAD1
- BCDUMP_HEAD2
- BCDUMP_HEAD3
- BCDUMP_VERSION
- _LJ_BCDUMP_H

Source code

```
1  /*
2  ** Bytecode dump definitions.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_BCDUMP_H
7  #define _LJ_BCDUMP_H
8
9  #include "lj_obj.h"
10  #include "lj_lex.h"
11
12  /* -- Bytecode dump format ------------------------------------------------ */
13
14  /*
15  ** dump   = header proto+ 0U
16  ** header = ESC 'L' 'J' versionB flagsU [namelenU nameB*]
17  ** proto  = lengthU pdata
18  ** pdata  = phead bcinsW* uvdataH* kgc* knum* [debugB*]
19  ** phead  = flagsB numparamsB framesizeB numuvB numkgcU numknU numbcU
20  **          [debuglenU [firstlineU numlineU]]
21  ** kgc    = kgctypeU { ktab | (loU hiU) | (rloU rhiU iloU ihiU) | strB* }
22  ** knum   = intU0 | (loU1 hiU)
23  ** ktab   = narrayU nhashU karray* khash*
24  ** karray = ktabk
25  ** khash  = ktabk ktabk
26  ** ktabk  = ktabtypeU { intU | (loU hiU) | strB* }
27  **
28  ** B = 8 bit, H = 16 bit, W = 32 bit, U = ULEB128 of W, U0/U1 = ULEB128 of W+1
29  */
30
31  /* Bytecode dump header. */
32  #define BCDUMP_HEAD1                0x1b
33  #define BCDUMP_HEAD2                0x4c
34  #define BCDUMP_HEAD3                0x4a
35
36  /* If you perform *any* kind of private modifications to the bytecode itself
37  ** or to the dump format, you *must* set BCDUMP_VERSION to 0x80 or higher.
38  */
39  #define BCDUMP_VERSION                2
40
41  /* Compatibility flags. */
42  #define BCDUMP_F_BE                0x01
```

```
43 #define BCDUMP_F_STRIP               0x02
44 #define BCDUMP_F_FFI                 0x04
45 #define BCDUMP_F_FR2                 0x08
46
47 #define BCDUMP_F_KNOWN               (BCDUMP_F_FR2*2-1)
48
49 /* Type codes for the GC constants of a prototype. Plus length for strings. */
50 enum {
51   BCDUMP_KGC_CHILD, BCDUMP_KGC_TAB, BCDUMP_KGC_I64, BCDUMP_KGC_U64,
52   BCDUMP_KGC_COMPLEX, BCDUMP_KGC_STR
53 };
54
55 /* Type codes for the keys/values of a constant table. */
56 enum {
57   BCDUMP_KTAB_NIL, BCDUMP_KTAB_FALSE, BCDUMP_KTAB_TRUE,
58   BCDUMP_KTAB_INT, BCDUMP_KTAB_NUM, BCDUMP_KTAB_STR
59 };
60
61 /* -- Bytecode reader/writer ----------------------------------------------- */
62
63 LJ_FUNC int lj_bcwrite(lua_State *L, GCproto *pt, lua_Writer writer,
64                        void *data, int strip);
65 LJ_FUNC GCproto *lj_bcread_proto(LexState *ls);
66 LJ_FUNC GCproto *lj_bcread(LexState *ls);
67
68 #endif
```

One Level Up Top Level

src/lj_bcwrite.c - luajit-2.0-src

Data types defined

- BCWriteCtx
- BCWriteCtx

Functions defined

- bcwrite_bytecode
- bcwrite_footer
- bcwrite_header
- bcwrite_kgc
- bcwrite_knum
- bcwrite_ktab
- bcwrite_ktabk
- bcwrite_proto
- cpwriter
- lj_bcwrite

Macros defined

- LUA_CORE
- lj_bcwrite_c

Source code

```
1  /*
2  ** Bytecode writer.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lj_bcwrite_c
7  #define LUA_CORE
8
9  #include "lj_obj.h"
10  #include "lj_gc.h"
11  #include "lj_buf.h"
12  #include "lj_bc.h"
13  #if LJ_HASFFI
14  #include "lj_ctype.h"
15  #endif
16  #if LJ_HASJIT
17  #include "lj_dispatch.h"
18  #include "lj_jit.h"
19  #endif
20  #include "lj_strfmt.h"
21  #include "lj_bcdump.h"
22  #include "lj_vm.h"
23
24  /* Context for bytecode writer. */
25  typedef struct BCWriteCtx {
26    SBuf sb;                        /* Output buffer. */
```

```
  GCproto *pt;                        /* Root prototype. */
  lua_Writer wfunc;                  /* Writer callback. */
  void *wdata;                        /* Writer callback data. */
  int strip;                         /* Strip debug info. */
  int status;                         /* Status from writer callback. */
} BCWriteCtx;

/* -- Bytecode writer ----------------------------------------------------- */

/* Write a single constant key/value of a template table. */
static void bcwrite_ktabk(BCWriteCtx *ctx, cTValue *o, int narrow)
{
  char *p = lj_buf_more(&ctx->sb, 1+10);
  if (tvisstr(o)) {
    const GCstr *str = strV(o);
    MSize len = str->len;
    p = lj_buf_more(&ctx->sb, 5+len);
    p = lj_strfmt_wuleb128(p, BCDUMP_KTAB_STR+len);
    p = lj_buf_wmem(p, strdata(str), len);
  } else if (tvisint(o)) {
    *p++ = BCDUMP_KTAB_INT;
    p = lj_strfmt_wuleb128(p, intV(o));
  } else if (tvisnum(o)) {
    if (!LJ_DUALNUM && narrow) {  /* Narrow number constants to integers. */
      lua_Number num = numV(o);
      int32_t k = lj_num2int(num);
      if (num == (lua_Number)k) {  /* -0 is never a constant. */
        *p++ = BCDUMP_KTAB_INT;
        p = lj_strfmt_wuleb128(p, k);
        setsbufP(&ctx->sb, p);
        return;
      }
    }
    *p++ = BCDUMP_KTAB_NUM;
    p = lj_strfmt_wuleb128(p, o->u32.lo);
    p = lj_strfmt_wuleb128(p, o->u32.hi);
  } else {
    lua_assert(tvispri(o));
    *p++ = BCDUMP_KTAB_NIL+~itype(o);
  }
  setsbufP(&ctx->sb, p);
}

/* Write a template table. */
static void bcwrite_ktab(BCWriteCtx *ctx, char *p, const GCtab *t)
{
  MSize narray = 0, nhash = 0;
  if (t->asize > 0) {  /* Determine max. length of array part. */
    ptrdiff_t i;
    TValue *array = tvref(t->array);
    for (i = (ptrdiff_t)t->asize-1; i >= 0; i--)
      if (!tvisnil(&array[i]))
        break;
    narray = (MSize)(i+1);
  }
  if (t->hmask > 0) {  /* Count number of used hash slots. */
    MSize i, hmask = t->hmask;
    Node *node = noderef(t->node);
    for (i = 0; i <= hmask; i++)
      nhash += !tvisnil(&node[i].val);
  }
  /* Write number of array slots and hash slots. */
  p = lj_strfmt_wuleb128(p, narray);
  p = lj_strfmt_wuleb128(p, nhash);
  setsbufP(&ctx->sb, p);
  if (narray) {  /* Write array entries (may contain nil). */
    MSize i;
    TValue *o = tvref(t->array);
    for (i = 0; i < narray; i++, o++)
      bcwrite_ktabk(ctx, o, 1);
  }
  if (nhash) {  /* Write hash entries. */
    MSize i = nhash;
    Node *node = noderef(t->node) + t->hmask;
    for (;; node--)
      if (!tvisnil(&node->val)) {
```

```
        bcwrite_ktabk(ctx, &node->key, 0);
        bcwrite_ktabk(ctx, &node->val, 1);
        if (--i == 0) break;
      }
  }
}

/* Write GC constants of a prototype. */
static void bcwrite_kgc(BCWriteCtx *ctx, GCproto *pt)
{
  MSize i, sizekgc = pt->sizekgc;
  GCRef *kr = mref(pt->k, GCRef) - (ptrdiff_t)sizekgc;
  for (i = 0; i < sizekgc; i++, kr++) {
    GCobj *o = gcref(*kr);
    MSize tp, need = 1;
    char *p;
    /* Determine constant type and needed size. */
    if (o->gch.gct == ~LJ_TSTR) {
      tp = BCDUMP_KGC_STR + gco2str(o)->len;
      need = 5+gco2str(o)->len;
    } else if (o->gch.gct == ~LJ_TPROTO) {
      lua_assert((pt->flags & PROTO_CHILD));
      tp = BCDUMP_KGC_CHILD;
#if LJ_HASFFI
    } else if (o->gch.gct == ~LJ_TCDATA) {
      CTypeID id = gco2cd(o)->ctypeid;
      need = 1+4*5;
      if (id == CTID_INT64) {
        tp = BCDUMP_KGC_I64;
      } else if (id == CTID_UINT64) {
        tp = BCDUMP_KGC_U64;
      } else {
        lua_assert(id == CTID_COMPLEX_DOUBLE);
        tp = BCDUMP_KGC_COMPLEX;
      }
#endif
    } else {
      lua_assert(o->gch.gct == ~LJ_TTAB);
      tp = BCDUMP_KGC_TAB;
      need = 1+2*5;
    }
    /* Write constant type. */
    p = lj_buf_more(&ctx->sb, need);
    p = lj_strfmt_wuleb128(p, tp);
    /* Write constant data (if any). */
    if (tp >= BCDUMP_KGC_STR) {
      p = lj_buf_wmem(p, strdata(gco2str(o)), gco2str(o)->len);
    } else if (tp == BCDUMP_KGC_TAB) {
      bcwrite_ktab(ctx, p, gco2tab(o));
      continue;
#if LJ_HASFFI
    } else if (tp != BCDUMP_KGC_CHILD) {
      cTValue *q = (TValue *)cdataptr(gco2cd(o));
      p = lj_strfmt_wuleb128(p, q[0].u32.lo);
      p = lj_strfmt_wuleb128(p, q[0].u32.hi);
      if (tp == BCDUMP_KGC_COMPLEX) {
        p = lj_strfmt_wuleb128(p, q[1].u32.lo);
        p = lj_strfmt_wuleb128(p, q[1].u32.hi);
      }
#endif
    }
    setsbufP(&ctx->sb, p);
  }
}

/* Write number constants of a prototype. */
static void bcwrite_knum(BCWriteCtx *ctx, GCproto *pt)
{
  MSize i, sizekn = pt->sizekn;
  cTValue *o = mref(pt->k, TValue);
  char *p = lj_buf_more(&ctx->sb, 10*sizekn);
  for (i = 0; i < sizekn; i++, o++) {
    int32_t k;
    if (tvisint(o)) {
      k = intV(o);
      goto save_int;
```

```
    } else {
      /* Write a 33 bit ULEB128 for the int (lsb=0) or loword (lsb=1). */
      if (!LJ_DUALNUM) {  /* Narrow number constants to integers. */
        lua_Number num = numV(o);
        k = lj_num2int(num);
        if (num == (lua_Number)k) {  /* -0 is never a constant. */
        save_int:
          p = lj_strfmt_wuleb128(p, 2*(uint32_t)k | ((uint32_t)k&0x80000000u));
          if (k < 0)
            p[-1] = (p[-1] & 7) | ((k>>27) & 0x18);
          continue;
        }
      }
      p = lj_strfmt_wuleb128(p, 1+(2*o->u32.lo | (o->u32.lo & 0x80000000u)));
      if (o->u32.lo >= 0x80000000u)
        p[-1] = (p[-1] & 7) | ((o->u32.lo>>27) & 0x18);
      p = lj_strfmt_wuleb128(p, o->u32.hi);
    }
  }
  setsbufP(&ctx->sb, p);
}

/* Write bytecode instructions. */
static char *bcwrite_bytecode(BCWriteCtx *ctx, char *p, GCproto *pt)
{
  MSize nbc = pt->sizebc-1;  /* Omit the [JI]FUNC* header. */
#if LJ_HASJIT
  uint8_t *q = (uint8_t *)p;
#endif
  p = lj_buf_wmem(p, proto_bc(pt)+1, nbc*(MSize)sizeof(BCIns));
  UNUSED(ctx);
#if LJ_HASJIT
  /* Unpatch modified bytecode containing ILOOP/JLOOP etc. */
  if ((pt->flags & PROTO_ILOOP) || pt->trace) {
    jit_State *J = L2J(sbufL(&ctx->sb));
    MSize i;
    for (i = 0; i < nbc; i++, q += sizeof(BCIns)) {
      BCOp op = (BCOp)q[LJ_ENDIAN_SELECT(0, 3)];
      if (op == BC_IFORL || op == BC_IITERL || op == BC_ILOOP ||
          op == BC_JFORI) {
        q[LJ_ENDIAN_SELECT(0, 3)] = (uint8_t)(op-BC_IFORL+BC_FORL);
      } else if (op == BC_JFORL || op == BC_JITERL || op == BC_JLOOP) {
        BCReg rd = q[LJ_ENDIAN_SELECT(2, 1)] + (q[LJ_ENDIAN_SELECT(3, 0)] << 8);
        BCIns ins = traceref(J, rd)->startins;
        q[LJ_ENDIAN_SELECT(0, 3)] = (uint8_t)(op-BC_JFORL+BC_FORL);
        q[LJ_ENDIAN_SELECT(2, 1)] = bc_c(ins);
        q[LJ_ENDIAN_SELECT(3, 0)] = bc_b(ins);
      }
    }
  }
#endif
  return p;
}

/* Write prototype. */
static void bcwrite_proto(BCWriteCtx *ctx, GCproto *pt)
{
  MSize sizedbg = 0;
  char *p;

  /* Recursively write children of prototype. */
  if ((pt->flags & PROTO_CHILD)) {
    ptrdiff_t i, n = pt->sizekgc;
    GCRef *kr = mref(pt->k, GCRef) - 1;
    for (i = 0; i < n; i++, kr--) {
      GCobj *o = gcref(*kr);
      if (o->gch.gct == ~LJ_TPROTO)
        bcwrite_proto(ctx, gco2pt(o));
    }
  }

  /* Start writing the prototype info to a buffer. */
  p = lj_buf_need(&ctx->sb,
                  5+4+6*5+(pt->sizebc-1)*(MSize)sizeof(BCIns)+pt->sizeuv*2);
  p += 5;  /* Leave room for final size. */

```

```
  /* Write prototype header. */
  *p++ = (pt->flags & (PROTO_CHILD|PROTO_VARARG|PROTO_FFI));
  *p++ = pt->numparams;
  *p++ = pt->framesize;
  *p++ = pt->sizeuv;
  p = lj_strfmt_wuleb128(p, pt->sizekgc);
  p = lj_strfmt_wuleb128(p, pt->sizekn);
  p = lj_strfmt_wuleb128(p, pt->sizebc-1);
  if (!ctx->strip) {
    if (proto_lineinfo(pt))
      sizedbg = pt->sizept - (MSize)((char *)proto_lineinfo(pt) - (char *)pt);
    p = lj_strfmt_wuleb128(p, sizedbg);
    if (sizedbg) {
      p = lj_strfmt_wuleb128(p, pt->firstline);
      p = lj_strfmt_wuleb128(p, pt->numline);
    }
  }

  /* Write bytecode instructions and upvalue refs. */
  p = bcwrite_bytecode(ctx, p, pt);
  p = lj_buf_wmem(p, proto_uv(pt), pt->sizeuv*2);
  setsbufP(&ctx->sb, p);

  /* Write constants. */
  bcwrite_kgc(ctx, pt);
  bcwrite_knum(ctx, pt);

  /* Write debug info, if not stripped. */
  if (sizedbg) {
    p = lj_buf_more(&ctx->sb, sizedbg);
    p = lj_buf_wmem(p, proto_lineinfo(pt), sizedbg);
    setsbufP(&ctx->sb, p);
  }

  /* Pass buffer to writer function. */
  if (ctx->status == 0) {
    MSize n = sbuflen(&ctx->sb) - 5;
    MSize nn = (lj_fls(n)+8)*9 >> 6;
    char *q = sbufB(&ctx->sb) + (5 - nn);
    p = lj_strfmt_wuleb128(q, n);  /* Fill in final size. */
    lua_assert(p == sbufB(&ctx->sb) + 5);
    ctx->status = ctx->wfunc(sbufL(&ctx->sb), q, nn+n, ctx->wdata);
  }
}

/* Write header of bytecode dump. */
static void bcwrite_header(BCWriteCtx *ctx)
{
  GCstr *chunkname = proto_chunkname(ctx->pt);
  const char *name = strdata(chunkname);
  MSize len = chunkname->len;
  char *p = lj_buf_need(&ctx->sb, 5+5+len);
  *p++ = BCDUMP_HEAD1;
  *p++ = BCDUMP_HEAD2;
  *p++ = BCDUMP_HEAD3;
  *p++ = BCDUMP_VERSION;
  *p++ = (ctx->strip ? BCDUMP_F_STRIP : 0) +
         LJ_BE*BCDUMP_F_BE +
         ((ctx->pt->flags & PROTO_FFI) ? BCDUMP_F_FFI : 0) +
         LJ_FR2*BCDUMP_F_FR2;
  if (!ctx->strip) {
    p = lj_strfmt_wuleb128(p, len);
    p = lj_buf_wmem(p, name, len);
  }
  ctx->status = ctx->wfunc(sbufL(&ctx->sb), sbufB(&ctx->sb),
                           (MSize)(p - sbufB(&ctx->sb)), ctx->wdata);
}

/* Write footer of bytecode dump. */
static void bcwrite_footer(BCWriteCtx *ctx)
{
  if (ctx->status == 0) {
    uint8_t zero = 0;
    ctx->status = ctx->wfunc(sbufL(&ctx->sb), &zero, 1, ctx->wdata);
  }
}
```

```
331
332 /* Protected callback for bytecode writer. */
333 static TValue *cpwriter(lua_State *L, lua_CFunction dummy, void *ud)
334 {
335   BCWriteCtx *ctx = (BCWriteCtx *)ud;
336   UNUSED(L); UNUSED(dummy);
337   lj_buf_need(&ctx->sb, 1024);  /* Avoids resize for most prototypes. */
338   bcwrite_header(ctx);
339   bcwrite_proto(ctx, ctx->pt);
340   bcwrite_footer(ctx);
341   return NULL;
342 }
343
344 /* Write bytecode for a prototype. */
345 int lj_bcwrite(lua_State *L, GCproto *pt, lua_Writer writer, void *data,
346                int strip)
347 {
348   BCWriteCtx ctx;
349   int status;
350   ctx.pt = pt;
351   ctx.wfunc = writer;
352   ctx.wdata = data;
353   ctx.strip = strip;
354   ctx.status = 0;
355   lj_buf_init(L, &ctx.sb);
356   status = lj_vm_cpcall(L, NULL, &ctx, cpwriter);
357   if (status == 0) status = ctx.status;
358   lj_buf_free(G(sbufL(&ctx.sb)), &ctx.sb);
359   return status;
360 }
361
```

One Level Up Top Level

src/lj_ff.h - luajit-2.0-src

Data types defined

- FastFunc

Macros defined

- FFDEF
- _LJ_FF_H

Source code

```
1  /*
2  ** Fast function IDs.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_FF_H
7  #define _LJ_FF_H
8
9  /* Fast function ID. */
10  typedef enum {
11    FF_LUA_ = FF_LUA,         /* Lua function (must be 0). */
12    FF_C_ = FF_C,                 /* Regular C function (must be 1). */
13  #define FFDEF(name)         FF_##name,
14  #include "lj_ffdef.h"
15    FF__MAX
16  } FastFunc;
17
18  #endif
```

src/host/buildvm_libbc.h - luajit-2.0-src

Global variables defined

- libbc_code
- libbc_endian
- libbc_map

Source code

```
/* This is a generated file. DO NOT EDIT! */

static const int libbc_endian = 0;

static const uint8_t libbc_code[] = {
#if LJ_FR2
0,1,2,0,0,1,2,24,1,0,0,76,1,2,0,241,135,158,166,3,220,203,178,130,4,0,1,2,0,
0,1,2,24,1,0,0,76,1,2,0,243,244,148,165,20,198,190,199,252,3,0,1,2,0,0,0,3,
16,0,5,0,21,1,0,0,76,1,2,0,0,2,10,0,0,0,15,16,0,12,0,16,1,9,0,41,2,1,0,21,3,
 0,0,41,4,1,0,77,2,8,128,18,6,1,0,18,8,5,0,59,9,5,0,66,6,3,2,10,6,0,0,88,7,1,
 128,76,6,2,0,79,2,248,127,75,0,1,0,0,2,11,0,0,0,16,16,0,12,0,16,1,9,0,43,2,
 0,0,18,3,0,0,41,4,0,0,88,5,7,128,18,7,1,0,18,9,5,0,18,10,6,0,66,7,3,2,10,7,
 0,0,88,8,1,128,76,7,2,0,70,5,3,3,82,5,247,127,75,0,1,0,0,1,2,0,0,0,3,16,0,12,
 0,21,1,0,0,76,1,2,0,0,2,10,0,0,2,30,16,0,12,0,21,2,0,0,11,1,0,0,88,3,7,128,
 8,2,0,0,88,3,23,128,59,3,2,0,43,4,0,0,64,4,2,0,76,3,2,0,88,3,18,128,16,1,14,
 0,41,3,1,0,3,3,1,0,88,3,14,128,3,1,2,0,88,3,12,128,59,3,1,0,22,4,1,1,18,5,2,
 0,41,6,1,0,77,4,4,128,23,8,1,7,59,9,7,0,64,9,8,0,79,4,252,127,43,4,0,0,64,4,
 2,0,76,3,2,0,75,0,1,0,0,2,0
 #else
 0,1,2,0,0,1,2,24,1,0,0,76,1,2,0,241,135,158,166,3,220,203,178,130,4,0,1,2,0,
 0,1,2,24,1,0,0,76,1,2,0,243,244,148,165,20,198,190,199,252,3,0,1,2,0,0,0,3,
 16,0,5,0,21,1,0,0,76,1,2,0,0,2,9,0,0,0,15,16,0,12,0,16,1,9,0,41,2,1,0,21,3,
 0,0,41,4,1,0,77,2,8,128,18,6,1,0,18,7,5,0,59,8,5,0,66,6,3,2,10,6,0,0,88,7,1,
 128,76,6,2,0,79,2,248,127,75,0,1,0,0,2,10,0,0,0,16,16,0,12,0,16,1,9,0,43,2,
 0,0,18,3,0,0,41,4,0,0,88,5,7,128,18,7,1,0,18,8,5,0,18,9,6,0,66,7,3,2,10,7,0,
 0,88,8,1,128,76,7,2,0,70,5,3,3,82,5,247,127,75,0,1,0,0,1,2,0,0,0,3,16,0,12,
 0,21,1,0,0,76,1,2,0,0,2,10,0,0,2,30,16,0,12,0,21,2,0,0,11,1,0,0,88,3,7,128,
 8,2,0,0,88,3,23,128,59,3,2,0,43,4,0,0,64,4,2,0,76,3,2,0,88,3,18,128,16,1,14,
 0,41,3,1,0,3,3,1,0,88,3,14,128,3,1,2,0,88,3,12,128,59,3,1,0,22,4,1,1,18,5,2,
 0,41,6,1,0,77,4,4,128,23,8,1,7,59,9,7,0,64,9,8,0,79,4,252,127,43,4,0,0,64,4,
 2,0,76,3,2,0,75,0,1,0,0,2,0
 #endif
 };

 static const struct { const char *name; int ofs; } libbc_map[] = {
 {"math_deg",0},
 {"math_rad",25},
 {"string_len",50},
 {"table_foreachi",69},
 {"table_foreach",136},
 {"table_getn",207},
 {"table_remove",226},
 {NULL,355}
 };

```

src/lj_recdef.h - luajit-2.0-src

Global variables defined

- recff_func
- recff_idmap

Source code

```
/* This is a generated file. DO NOT EDIT! */

static const uint16_t recff_idmap[] = {
0,
0x0100,
0x0200,
0x0300,
0,
0x0400+(0),
0x0500,
0x0400+(1),
0x0600,
0x0700,
0x0800,
0,
0x0900,
0x0a00,
0x0b00,
0,
0x0c00,
0x0d00,
0x0e00,
0,
0x0f00,
0x1000,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0x1100,
0x1200+(IRFPM_FLOOR),
0x1200+(IRFPM_CEIL),
0x1300+(IRFPM_SQRT),
0x1300+(IRFPM_LOG10),
0x1300+(IRFPM_EXP),
0x1300+(IRFPM_SIN),
0x1300+(IRFPM_COS),
0x1300+(IRFPM_TAN),
0x1400+(FF_math_asin),
0x1400+(FF_math_acos),
0x1400+(FF_math_atan),
0x1500+(IRCALL_sinh),
0x1500+(IRCALL_cosh),
0x1500+(IRCALL_tanh),
0,
0x1600,
0x1700,
0x1800,
0x1900,
```

```
0,
0x1a00,
0x1b00+(IR_MIN),
0x1b00+(IR_MAX),
0x1c00,
0,
0x1d00,
0x1e00+(IR_BNOT),
0x1e00+(IR_BSWAP),
0x1f00+(IR_BSHL),
0x1f00+(IR_BSHR),
0x1f00+(IR_BSAR),
0x1f00+(IR_BROL),
0x1f00+(IR_BROR),
0x2000+(IR_BAND),
0x2000+(IR_BOR),
0x2000+(IR_BXOR),
0x2100,
0x2200+(0),
0x2300,
0x2200+(1),
0x2400,
0x2500+(IRCALL_lj_buf_putstr_reverse),
0x2500+(IRCALL_lj_buf_putstr_lower),
0x2500+(IRCALL_lj_buf_putstr_upper),
0,
0x2600,
0,
0,
0,
0,
0x2700,
0,
0x2800,
0x2900,
0,
0x2a00,
0x2b00,
0,
 0,
 0x2c00+(0),
 0x2d00+(0),
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0x2c00+(GCROOT_IO_OUTPUT),
 0x2d00+(GCROOT_IO_OUTPUT),
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0,
 0x2e00,
 0,
 0,
 0,
 0,
 0,
```

```
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0x2f00+(0),
0x2f00+(1),
0x3000+(MM_eq),
0x3000+(MM_len),
0x3000+(MM_lt),
0x3000+(MM_le),
0x3000+(MM_concat),
0x3100,
0x3000+(MM_add),
0x3000+(MM_sub),
0x3000+(MM_mul),
0x3000+(MM_div),
0x3000+(MM_mod),
0x3000+(MM_pow),
0x3000+(MM_unm),
0,
0,
0,
0x3200+(1),
0x3200+(0),
0,
0,
0,
0,
0x3300,
0x3300,
0x3400,
0,
0x3500,
0x3600+(FF_ffi_sizeof),
0x3600+(FF_ffi_alignof),
0x3600+(FF_ffi_offsetof),
0x3700,
0x3800,
0x3900,
0x3a00,
0x3b00,
0,
0x3c00
};

static const RecordFunc recff_func[] = {
recff_nyi,
recff_c,
recff_assert,
recff_type,
recff_xpairs,
```

```
recff_ipairs_aux,
recff_getmetatable,
recff_setmetatable,
recff_getfenv,
recff_rawget,
recff_rawset,
recff_rawequal,
recff_select,
recff_tonumber,
recff_tostring,
recff_pcall,
recff_xpcall,
recff_math_abs,
recff_math_round,
recff_math_unary,
recff_math_atrig,
recff_math_htrig,
recff_math_modf,
recff_math_log,
recff_math_atan2,
recff_math_pow,
recff_math_ldexp,
recff_math_minmax,
recff_math_random,
recff_bit_tobit,
recff_bit_unary,
recff_bit_shift,
recff_bit_nary,
recff_bit_tohex,
recff_string_range,
recff_string_char,
recff_string_rep,
recff_string_op,
recff_string_find,
recff_string_format,
recff_table_insert,
recff_table_concat,
recff_table_new,
recff_table_clear,
recff_io_write,
recff_io_flush,
recff_debug_getmetatable,
recff_cdata_index,
recff_cdata_arith,
recff_cdata_call,
recff_clib_index,
recff_ffi_new,
recff_ffi_typeof,
recff_ffi_istype,
recff_ffi_xof,
recff_ffi_errno,
recff_ffi_string,
recff_ffi_copy,
recff_ffi_fill,
recff_ffi_abi,
recff_ffi_gc
};

```

One Level Up Top Level

src/lj_ffrecord.c - luajit-2.0-src

Data types defined

- RecordFunc

Functions defined

- argv2int
- argv2str
- lj_ffrecord_func
- lj_ffrecord_select_mode
- recdef_lookup
- recff_assert
- recff_bit_nary
- recff_bit_shift
- recff_bit_tobit
- recff_bit_tohex
- recff_bit_unary
- recff_bufhdr
- recff_debug_getmetatable
- recff_getfenv
- recff_getmetatable
- recff_io_flush
- recff_io_fp
- recff_io_write
- recff_ipairs_aux
- recff_math_abs
- recff_math_atan2
- recff_math_atrig
- recff_math_htrig
- recff_math_ldexp
- recff_math_log
- recff_math_minmax
- recff_math_modf

- recff_math_pow
- recff_math_random
- recff_math_round
- recff_math_unary
- recff_metacall
- recff_metacall_cp
- recff_nyi
- recff_pcall
- recff_rawequal
- recff_rawget
- recff_rawlen
- recff_rawset
- recff_select
- recff_setmetatable
- recff_stitch
- recff_string_char
- recff_string_find
- recff_string_format
- recff_string_op
- recff_string_range
- recff_string_rep
- recff_string_start
- recff_table_clear
- recff_table_concat
- recff_table_insert
- recff_table_new
- recff_tonumber
- recff_tostring
- recff_type
- recff_xpairs
- recff_xpcall
- recff_xpcall_cp
- results_wanted

Macros defined

- IR
- IR
- LUA_CORE
- emitir
- emitir
- lj_ffrecord_c
- recff_c
- recff_nyiu

Source code

```
1  /*
2  ** Fast function call recorder.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lj_ffrecord_c
7  #define LUA_CORE
8
9  #include "lj_obj.h"
10
11 #if LJ_HASJIT
12
13 #include "lj_err.h"
14 #include "lj_str.h"
15 #include "lj_tab.h"
16 #include "lj_frame.h"
17 #include "lj_bc.h"
18 #include "lj_ff.h"
19 #include "lj_ir.h"
20 #include "lj_jit.h"
21 #include "lj_ircall.h"
22 #include "lj_iropt.h"
23 #include "lj_trace.h"
24 #include "lj_record.h"
25 #include "lj_ffrecord.h"
26 #include "lj_crecord.h"
27 #include "lj_dispatch.h"
28 #include "lj_vm.h"
29 #include "lj_strscan.h"
30 #include "lj_strfmt.h"
31
32 /* Some local macros to save typing. Undef'd at the end. */
33 #define IR(ref)                         (&J->cur.ir[(ref)])
34
35 /* Pass IR on to next optimization in chain (FOLD). */
36 #define emitir(ot, a, b)        (lj_ir_set(J, (ot), (a), (b)), lj_opt_fold(J))
37
38 /* -- Fast function recording handlers ----------------------------------- */
39
40 /* Conventions for fast function call handlers:
41 **
42 ** The argument slots start at J->base[0]. All of them are guaranteed to be
43 ** valid and type-specialized references. J->base[J->maxslot] is set to 0
44 ** as a sentinel. The runtime argument values start at rd->argv[0].
45 **
46 ** In general fast functions should check for presence of all of their
47 ** arguments and for the correct argument types. Some simplifications
48 ** are allowed if the interpreter throws instead. But even if recording
49 ** is aborted, the generated IR must be consistent (no zero-refs).
50 **
51 ** The number of results in rd->nres is set to 1. Handlers that return
52 ** a different number of results need to override it. A negative value
```

```
** prevents return processing (e.g. for pending calls).
**
** Results need to be stored starting at J->base[0]. Return processing
** moves them to the right slots later.
**
** The per-ffid auxiliary data is the value of the 2nd part of the
** LJLIB_REC() annotation. This allows handling similar functionality
** in a common handler.
*/

/* Type of handler to record a fast function. */
typedef void (LJ_FASTCALL *RecordFunc)(jit_State *J, RecordFFData *rd);

/* Get runtime value of int argument. */
static int32_t argv2int(jit_State *J, TValue *o)
{
  if (!lj_strscan_numberobj(o))
    lj_trace_err(J, LJ_TRERR_BADTYPE);
  return tvisint(o) ? intV(o) : lj_num2int(numV(o));
}

/* Get runtime value of string argument. */
static GCstr *argv2str(jit_State *J, TValue *o)
{
  if (LJ_LIKELY(tvisstr(o))) {
    return strV(o);
  } else {
    GCstr *s;
    if (!tvisnumber(o))
      lj_trace_err(J, LJ_TRERR_BADTYPE);
    s = lj_strfmt_number(J->L, o);
    setstrV(J->L, o, s);
    return s;
  }
}

/* Return number of results wanted by caller. */
static ptrdiff_t results_wanted(jit_State *J)
{
  TValue *frame = J->L->base-1;
  if (frame_islua(frame))
    return (ptrdiff_t)bc_b(frame_pc(frame)[-1]) - 1;
  else
    return -1;
}

/* Trace stitching: add continuation below frame to start a new trace. */
static void recff_stitch(jit_State *J)
{
  ASMFunction cont = lj_cont_stitch;
  TraceNo traceno = J->cur.traceno;
  lua_State *L = J->L;
  TValue *base = L->base;
  const BCIns *pc = frame_pc(base-1);
  TValue *pframe = frame_prevl(base-1);
  TRef trcont;

  lua_assert(!LJ_FR2);  /* TODO_FR2: handle frame shift. */
  /* Move func + args up in Lua stack and insert continuation. */
  memmove(&base[1], &base[-1], sizeof(TValue)*(J->maxslot+1));
  setframe_ftsz(base+1, ((char *)(base+1) - (char *)pframe) + FRAME_CONT);
  setcont(base, cont);
  setframe_pc(base, pc);
  if (LJ_DUALNUM) setintV(base-1, traceno); else base[-1].u64 = traceno;
  L->base += 2;
  L->top += 2;

  /* Ditto for the IR. */
  memmove(&J->base[1], &J->base[-1], sizeof(TRef)*(J->maxslot+1));
#if LJ_64
  trcont = lj_ir_kptr(J, (void *)((int64_t)cont-(int64_t)lj_vm_asm_begin));
#else
  trcont = lj_ir_kptr(J, (void *)cont);
#endif
  J->base[0] = trcont | TREF_CONT;
  J->base[-1] = LJ_DUALNUM ? lj_ir_kint(J,traceno) : lj_ir_knum_u64(J,traceno);
```

```
  J->base += 2;
  J->baseslot += 2;
  J->framedepth++;

  lj_record_stop(J, LJ_TRLINK_STITCH, 0);

  /* Undo Lua stack changes. */
  memmove(&base[-1], &base[1], sizeof(TValue)*(J->maxslot+1));
  setframe_pc(base-1, pc);
  L->base -= 2;
  L->top -= 2;
}

/* Fallback handler for fast functions that are not recorded (yet). */
static void LJ_FASTCALL recff_nyi(jit_State *J, RecordFFData *rd)
{
  if (J->cur.nins < (IRRef)J->param[JIT_P_minstitch] + REF_BASE) {
    lj_trace_err_info(J, LJ_TRERR_TRACEUV);
  } else {
    /* Can only stitch from Lua call. */
    if (J->framedepth && frame_islua(J->L->base-1)) {
      BCOp op = bc_op(*frame_pc(J->L->base-1));
      /* Stitched trace cannot start with *M op with variable # of args. */
      if (!(op == BC_CALLM || op == BC_CALLMT ||
            op == BC_RETM || op == BC_TSETM)) {
        switch (J->fn->c.ffid) {
        case FF_error:
        case FF_debug_sethook:
        case FF_jit_flush:
          break;  /* Don't stitch across special builtins. */
        default:
          recff_stitch(J);  /* Use trace stitching. */
          rd->nres = -1;
          return;
        }
      }
    }
    /* Otherwise stop trace and return to interpreter. */
    lj_record_stop(J, LJ_TRLINK_RETURN, 0);
    rd->nres = -1;
  }
}

/* Fallback handler for unsupported variants of fast functions. */
#define recff_nyiu        recff_nyi

/* Must stop the trace for classic C functions with arbitrary side-effects. */
#define recff_c           recff_nyi

/* Emit BUFHDR for the global temporary buffer. */
static TRef recff_bufhdr(jit_State *J)
{
  return emitir(IRT(IR_BUFHDR, IRT_P32),
                lj_ir_kptr(J, &J2G(J)->tmpbuf), IRBUFHDR_RESET);
}

/* -- Base library fast functions ----------------------------------------- */

static void LJ_FASTCALL recff_assert(jit_State *J, RecordFFData *rd)
{
  /* Arguments already specialized. The interpreter throws for nil/false. */
  rd->nres = J->maxslot;  /* Pass through all arguments. */
}

static void LJ_FASTCALL recff_type(jit_State *J, RecordFFData *rd)
{
  /* Arguments already specialized. Result is a constant string. Neat, huh? */
  uint32_t t;
  if (tvisnumber(&rd->argv[0]))
    t = ~LJ_TNUMX;
  else if (LJ_64 && !LJ_GC64 && tvislightud(&rd->argv[0]))
    t = ~LJ_TLIGHTUD;
  else
    t = ~itype(&rd->argv[0]);
  J->base[0] = lj_ir_kstr(J, strV(&J->fn->c.upvalue[t]));
  UNUSED(rd);
```

```
}

static void LJ_FASTCALL recff_getmetatable(jit_State *J, RecordFFData *rd)
{
  TRef tr = J->base[0];
  if (tr) {
    RecordIndex ix;
    ix.tab = tr;
    copyTV(J->L, &ix.tabv, &rd->argv[0]);
    if (lj_record_mm_lookup(J, &ix, MM_metatable))
      J->base[0] = ix.mobj;
    else
      J->base[0] = ix.mt;
  }  /* else: Interpreter will throw. */
}

static void LJ_FASTCALL recff_setmetatable(jit_State *J, RecordFFData *rd)
{
  TRef tr = J->base[0];
  TRef mt = J->base[1];
  if (tref_istab(tr) && (tref_istab(mt) || (mt && tref_isnil(mt)))) {
    TRef fref, mtref;
    RecordIndex ix;
    ix.tab = tr;
    copyTV(J->L, &ix.tabv, &rd->argv[0]);
    lj_record_mm_lookup(J, &ix, MM_metatable); /* Guard for no __metatable. */
    fref = emitir(IRT(IR_FREF, IRT_P32), tr, IRFL_TAB_META);
    mtref = tref_isnil(mt) ? lj_ir_knull(J, IRT_TAB) : mt;
    emitir(IRT(IR_FSTORE, IRT_TAB), fref, mtref);
    if (!tref_isnil(mt))
      emitir(IRT(IR_TBAR, IRT_TAB), tr, 0);
    J->base[0] = tr;
    J->needsnap = 1;
  }  /* else: Interpreter will throw. */
}

static void LJ_FASTCALL recff_rawget(jit_State *J, RecordFFData *rd)
{
  RecordIndex ix;
  ix.tab = J->base[0]; ix.key = J->base[1];
  if (tref_istab(ix.tab) && ix.key) {
    ix.val = 0; ix.idxchain = 0;
    settabV(J->L, &ix.tabv, tabV(&rd->argv[0]));
    copyTV(J->L, &ix.keyv, &rd->argv[1]);
    J->base[0] = lj_record_idx(J, &ix);
  }  /* else: Interpreter will throw. */
}

static void LJ_FASTCALL recff_rawset(jit_State *J, RecordFFData *rd)
{
  RecordIndex ix;
  ix.tab = J->base[0]; ix.key = J->base[1]; ix.val = J->base[2];
  if (tref_istab(ix.tab) && ix.key && ix.val) {
    ix.idxchain = 0;
    settabV(J->L, &ix.tabv, tabV(&rd->argv[0]));
    copyTV(J->L, &ix.keyv, &rd->argv[1]);
    copyTV(J->L, &ix.valv, &rd->argv[2]);
    lj_record_idx(J, &ix);
    /* Pass through table at J->base[0] as result. */
  }  /* else: Interpreter will throw. */
}

static void LJ_FASTCALL recff_rawequal(jit_State *J, RecordFFData *rd)
{
  TRef tra = J->base[0];
  TRef trb = J->base[1];
  if (tra && trb) {
    int diff = lj_record_objcmp(J, tra, trb, &rd->argv[0], &rd->argv[1]);
    J->base[0] = diff ? TREF_FALSE : TREF_TRUE;
  }  /* else: Interpreter will throw. */
}

#if LJ_52
static void LJ_FASTCALL recff_rawlen(jit_State *J, RecordFFData *rd)
{
  TRef tr = J->base[0];
```

```
  if (tref_isstr(tr))
    J->base[0] = emitir(IRTI(IR_FLOAD), tr, IRFL_STR_LEN);
  else if (tref_istab(tr))
    J->base[0] = lj_ir_call(J, IRCALL_lj_tab_len, tr);
  /* else: Interpreter will throw. */
  UNUSED(rd);
}
#endif

/* Determine mode of select() call. */
int32_t lj_ffrecord_select_mode(jit_State *J, TRef tr, TValue *tv)
{
  if (tref_isstr(tr) && *strVdata(tv) == '#') {  /* select('#', ...) */
    if (strV(tv)->len == 1) {
      emitir(IRTG(IR_EQ, IRT_STR), tr, lj_ir_kstr(J, strV(tv)));
    } else {
      TRef trptr = emitir(IRT(IR_STRREF, IRT_P32), tr, lj_ir_kint(J, 0));
      TRef trchar = emitir(IRT(IR_XLOAD, IRT_U8), trptr, IRXLOAD_READONLY);
      emitir(IRTG(IR_EQ, IRT_INT), trchar, lj_ir_kint(J, '#'));
    }
    return 0;
  } else {  /* select(n, ...) */
    int32_t start = argv2int(J, tv);
    if (start == 0) lj_trace_err(J, LJ_TRERR_BADTYPE);  /* A bit misleading. */
    return start;
  }
}

static void LJ_FASTCALL recff_select(jit_State *J, RecordFFData *rd)
{
  TRef tr = J->base[0];
  if (tr) {
    ptrdiff_t start = lj_ffrecord_select_mode(J, tr, &rd->argv[0]);
    if (start == 0) {  /* select('#', ...) */
      J->base[0] = lj_ir_kint(J, J->maxslot - 1);
    } else if (tref_isk(tr)) {  /* select(k, ...) */
      ptrdiff_t n = (ptrdiff_t)J->maxslot;
      if (start < 0) start += n;
      else if (start > n) start = n;
      rd->nres = n - start;
      if (start >= 1) {
        ptrdiff_t i;
        for (i = 0; i < n - start; i++)
          J->base[i] = J->base[start+i];
      }  /* else: Interpreter will throw. */
    } else {
      recff_nyiu(J, rd);
      return;
    }
  }  /* else: Interpreter will throw. */
}

static void LJ_FASTCALL recff_tonumber(jit_State *J, RecordFFData *rd)
{
  TRef tr = J->base[0];
  TRef base = J->base[1];
  if (tr && !tref_isnil(base)) {
    base = lj_opt_narrow_toint(J, base);
    if (!tref_isk(base) || IR(tref_ref(base))->i != 10) {
      recff_nyiu(J, rd);
      return;
    }
  }
  if (tref_isnumber_str(tr)) {
    if (tref_isstr(tr)) {
      TValue tmp;
      if (!lj_strscan_num(strV(&rd->argv[0]), &tmp)) {
        recff_nyiu(J, rd);  /* Would need an inverted STRTO for this case. */
        return;
      }
      tr = emitir(IRTG(IR_STRTO, IRT_NUM), tr, 0);
    }
#if LJ_HASFFI
  } else if (tref_iscdata(tr)) {
    lj_crecord_tonumber(J, rd);
    return;
```

```
#endif
  } else {
    tr = TREF_NIL;
  }
  J->base[0] = tr;
  UNUSED(rd);
}

static TValue *recff_metacall_cp(lua_State *L, lua_CFunction dummy, void *ud)
{
  jit_State *J = (jit_State *)ud;
  lj_record_tailcall(J, 0, 1);
  UNUSED(L); UNUSED(dummy);
  return NULL;
}

static int recff_metacall(jit_State *J, RecordFFData *rd, MMS mm)
{
  RecordIndex ix;
  ix.tab = J->base[0];
  copyTV(J->L, &ix.tabv, &rd->argv[0]);
  if (lj_record_mm_lookup(J, &ix, mm)) {  /* Has metamethod? */
    int errcode;
    TValue argv0;
    /* Temporarily insert metamethod below object. */
    J->base[1] = J->base[0];
    J->base[0] = ix.mobj;
    copyTV(J->L, &argv0, &rd->argv[0]);
    copyTV(J->L, &rd->argv[1], &rd->argv[0]);
    copyTV(J->L, &rd->argv[0], &ix.mobjv);
    /* Need to protect lj_record_tailcall because it may throw. */
    errcode = lj_vm_cpcall(J->L, NULL, J, recff_metacall_cp);
    /* Always undo Lua stack changes to avoid confusing the interpreter. */
    copyTV(J->L, &rd->argv[0], &argv0);
    if (errcode)
      lj_err_throw(J->L, errcode);  /* Propagate errors. */
    rd->nres = -1;  /* Pending call. */
    return 1;  /* Tailcalled to metamethod. */
  }
  return 0;
}

static void LJ_FASTCALL recff_tostring(jit_State *J, RecordFFData *rd)
{
  TRef tr = J->base[0];
  if (tref_isstr(tr)) {
    /* Ignore __tostring in the string base metatable. */
    /* Pass on result in J->base[0]. */
  } else if (tr && !recff_metacall(J, rd, MM_tostring)) {
    if (tref_isnumber(tr)) {
      J->base[0] = emitir(IRT(IR_TOSTR, IRT_STR), tr,
                          tref_isnum(tr) ? IRTOSTR_NUM : IRTOSTR_INT);
    } else if (tref_ispri(tr)) {
      J->base[0] = lj_ir_kstr(J, lj_strfmt_obj(J->L, &rd->argv[0]));
    } else {
      recff_nyiu(J, rd);
      return;
    }
  }
}

static void LJ_FASTCALL recff_ipairs_aux(jit_State *J, RecordFFData *rd)
{
  RecordIndex ix;
  ix.tab = J->base[0];
  if (tref_istab(ix.tab)) {
    if (!tvisnumber(&rd->argv[1]))  /* No support for string coercion. */
      lj_trace_err(J, LJ_TRERR_BADTYPE);
    setintV(&ix.keyv, numberVint(&rd->argv[1])+1);
    settabV(J->L, &ix.tabv, tabV(&rd->argv[0]));
    ix.val = 0; ix.idxchain = 0;
    ix.key = lj_opt_narrow_toint(J, J->base[1]);
    J->base[0] = ix.key = emitir(IRTI(IR_ADD), ix.key, lj_ir_kint(J, 1));
    J->base[1] = lj_record_idx(J, &ix);
    rd->nres = tref_isnil(J->base[1]) ? 0 : 2;
  }  /* else: Interpreter will throw. */
```

```
}

static void LJ_FASTCALL recff_xpairs(jit_State *J, RecordFFData *rd)
{
  if (!(LJ_52 && recff_metacall(J, rd, MM_ipairs))) {
    TRef tab = J->base[0];
    if (tref_istab(tab)) {
      J->base[0] = lj_ir_kfunc(J, funcV(&J->fn->c.upvalue[0]));
      J->base[1] = tab;
      J->base[2] = rd->data ? lj_ir_kint(J, 0) : TREF_NIL;
      rd->nres = 3;
    }  /* else: Interpreter will throw. */
  }
}

static void LJ_FASTCALL recff_pcall(jit_State *J, RecordFFData *rd)
{
  if (J->maxslot >= 1) {
    lj_record_call(J, 0, J->maxslot - 1);
    rd->nres = -1;  /* Pending call. */
  }  /* else: Interpreter will throw. */
}

static TValue *recff_xpcall_cp(lua_State *L, lua_CFunction dummy, void *ud)
{
  jit_State *J = (jit_State *)ud;
  lj_record_call(J, 1, J->maxslot - 2);
  UNUSED(L); UNUSED(dummy);
  return NULL;
}

static void LJ_FASTCALL recff_xpcall(jit_State *J, RecordFFData *rd)
{
  if (J->maxslot >= 2) {
    TValue argv0, argv1;
    TRef tmp;
    int errcode;
    lua_assert(!LJ_FR2);  /* TODO_FR2: handle different frame setup. */
    /* Swap function and traceback. */
    tmp = J->base[0]; J->base[0] = J->base[1]; J->base[1] = tmp;
    copyTV(J->L, &argv0, &rd->argv[0]);
    copyTV(J->L, &argv1, &rd->argv[1]);
    copyTV(J->L, &rd->argv[0], &argv1);
    copyTV(J->L, &rd->argv[1], &argv0);
    /* Need to protect lj_record_call because it may throw. */
    errcode = lj_vm_cpcall(J->L, NULL, J, recff_xpcall_cp);
    /* Always undo Lua stack swap to avoid confusing the interpreter. */
    copyTV(J->L, &rd->argv[0], &argv0);
    copyTV(J->L, &rd->argv[1], &argv1);
    if (errcode)
      lj_err_throw(J->L, errcode);  /* Propagate errors. */
    rd->nres = -1;  /* Pending call. */
  }  /* else: Interpreter will throw. */
}

static void LJ_FASTCALL recff_getfenv(jit_State *J, RecordFFData *rd)
{
  TRef tr = J->base[0];
  /* Only support getfenv(0) for now. */
  if (tref_isint(tr) && tref_isk(tr) && IR(tref_ref(tr))->i == 0) {
    TRef trl = emitir(IRT(IR_LREF, IRT_THREAD), 0, 0);
    J->base[0] = emitir(IRT(IR_FLOAD, IRT_TAB), trl, IRFL_THREAD_ENV);
    return;
  }
  recff_nyiu(J, rd);
}

/* -- Math library fast functions ----------------------------------------- */

static void LJ_FASTCALL recff_math_abs(jit_State *J, RecordFFData *rd)
{
  TRef tr = lj_ir_tonum(J, J->base[0]);
  J->base[0] = emitir(IRTN(IR_ABS), tr, lj_ir_knum_abs(J));
  UNUSED(rd);
}

```

```c
/* Record rounding functions math.floor and math.ceil. */
static void LJ_FASTCALL recff_math_round(jit_State *J, RecordFFData *rd)
{
  TRef tr = J->base[0];
  if (!tref_isinteger(tr)) {  /* Pass through integers unmodified. */
    tr = emitir(IRTN(IR_FPMATH), lj_ir_tonum(J, tr), rd->data);
    /* Result is integral (or NaN/Inf), but may not fit an int32_t. */
    if (LJ_DUALNUM) {  /* Try to narrow using a guarded conversion to int. */
      lua_Number n = lj_vm_foldfpm(numberVnum(&rd->argv[0]), rd->data);
      if (n == (lua_Number)lj_num2int(n))
        tr = emitir(IRTGI(IR_CONV), tr, IRCONV_INT_NUM|IRCONV_CHECK);
    }
    J->base[0] = tr;
  }
}

/* Record unary math.* functions, mapped to IR_FPMATH opcode. */
static void LJ_FASTCALL recff_math_unary(jit_State *J, RecordFFData *rd)
{
  J->base[0] = emitir(IRTN(IR_FPMATH), lj_ir_tonum(J, J->base[0]), rd->data);
}

/* Record math.log. */
static void LJ_FASTCALL recff_math_log(jit_State *J, RecordFFData *rd)
{
  TRef tr = lj_ir_tonum(J, J->base[0]);
  if (J->base[1]) {
#ifdef LUAJIT_NO_LOG2
    uint32_t fpm = IRFPM_LOG;
#else
    uint32_t fpm = IRFPM_LOG2;
#endif
    TRef trb = lj_ir_tonum(J, J->base[1]);
    tr = emitir(IRTN(IR_FPMATH), tr, fpm);
    trb = emitir(IRTN(IR_FPMATH), trb, fpm);
    trb = emitir(IRTN(IR_DIV), lj_ir_knum_one(J), trb);
    tr = emitir(IRTN(IR_MUL), tr, trb);
  } else {
    tr = emitir(IRTN(IR_FPMATH), tr, IRFPM_LOG);
  }
  J->base[0] = tr;
  UNUSED(rd);
}

/* Record math.atan2. */
static void LJ_FASTCALL recff_math_atan2(jit_State *J, RecordFFData *rd)
{
  TRef tr = lj_ir_tonum(J, J->base[0]);
  TRef tr2 = lj_ir_tonum(J, J->base[1]);
  J->base[0] = emitir(IRTN(IR_ATAN2), tr, tr2);
  UNUSED(rd);
}

/* Record math.ldexp. */
static void LJ_FASTCALL recff_math_ldexp(jit_State *J, RecordFFData *rd)
{
  TRef tr = lj_ir_tonum(J, J->base[0]);
#if LJ_TARGET_X86ORX64
  TRef tr2 = lj_ir_tonum(J, J->base[1]);
#else
  TRef tr2 = lj_opt_narrow_toint(J, J->base[1]);
#endif
  J->base[0] = emitir(IRTN(IR_LDEXP), tr, tr2);
  UNUSED(rd);
}

/* Record math.asin, math.acos, math.atan. */
static void LJ_FASTCALL recff_math_atrig(jit_State *J, RecordFFData *rd)
{
  TRef y = lj_ir_tonum(J, J->base[0]);
  TRef x = lj_ir_knum_one(J);
  uint32_t ffid = rd->data;
  if (ffid != FF_math_atan) {
    TRef tmp = emitir(IRTN(IR_MUL), y, y);
    tmp = emitir(IRTN(IR_SUB), x, tmp);
    tmp = emitir(IRTN(IR_FPMATH), tmp, IRFPM_SQRT);
```

```
585     if (ffid == FF_math_asin) { x = tmp; } else { x = y; y = tmp; }
586   }
587   J->base[0] = emitir(IRTN(IR_ATAN2), y, x);
588 }
589
590 static void LJ_FASTCALL recff_math_htrig(jit_State *J, RecordFFData *rd)
591 {
592   TRef tr = lj_ir_tonum(J, J->base[0]);
593   J->base[0] = emitir(IRTN(IR_CALLN), tr, rd->data);
594 }
595
596 static void LJ_FASTCALL recff_math_modf(jit_State *J, RecordFFData *rd)
597 {
598   TRef tr = J->base[0];
599   if (tref_isinteger(tr)) {
600     J->base[0] = tr;
601     J->base[1] = lj_ir_kint(J, 0);
602   } else {
603     TRef trt;
604     tr = lj_ir_tonum(J, tr);
605     trt = emitir(IRTN(IR_FPMATH), tr, IRFPM_TRUNC);
606     J->base[0] = trt;
607     J->base[1] = emitir(IRTN(IR_SUB), tr, trt);
608   }
609   rd->nres = 2;
610 }
611
612 static void LJ_FASTCALL recff_math_pow(jit_State *J, RecordFFData *rd)
613 {
614   TRef tr = lj_ir_tonum(J, J->base[0]);
615   if (!tref_isnumber_str(J->base[1]))
616     lj_trace_err(J, LJ_TRERR_BADTYPE);
617   J->base[0] = lj_opt_narrow_pow(J, tr, J->base[1], &rd->argv[1]);
618   UNUSED(rd);
619 }
620
621 static void LJ_FASTCALL recff_math_minmax(jit_State *J, RecordFFData *rd)
622 {
623   TRef tr = lj_ir_tonumber(J, J->base[0]);
624   uint32_t op = rd->data;
625   BCReg i;
626   for (i = 1; J->base[i] != 0; i++) {
627     TRef tr2 = lj_ir_tonumber(J, J->base[i]);
628     IRType t = IRT_INT;
629     if (!(tref_isinteger(tr) && tref_isinteger(tr2))) {
630       if (tref_isinteger(tr)) tr = emitir(IRTN(IR_CONV), tr, IRCONV_NUM_INT);
631       if (tref_isinteger(tr2)) tr2 = emitir(IRTN(IR_CONV), tr2, IRCONV_NUM_INT);
632       t = IRT_NUM;
633     }
634     tr = emitir(IRT(op, t), tr, tr2);
635   }
636   J->base[0] = tr;
637 }
638
639 static void LJ_FASTCALL recff_math_random(jit_State *J, RecordFFData *rd)
640 {
641   GCudata *ud = udataV(&J->fn->c.upvalue[0]);
642   TRef tr, one;
643   lj_ir_kgc(J, obj2gco(ud), IRT_UDATA);  /* Prevent collection. */
644   tr = lj_ir_call(J, IRCALL_lj_math_random_step, lj_ir_kptr(J, uddata(ud)));
645   one = lj_ir_knum_one(J);
646   tr = emitir(IRTN(IR_SUB), tr, one);
647   if (J->base[0]) {
648     TRef tr1 = lj_ir_tonum(J, J->base[0]);
649     if (J->base[1]) {  /* d = floor(d*(r2-r1+1.0)) + r1 */
650       TRef tr2 = lj_ir_tonum(J, J->base[1]);
651       tr2 = emitir(IRTN(IR_SUB), tr2, tr1);
652       tr2 = emitir(IRTN(IR_ADD), tr2, one);
653       tr = emitir(IRTN(IR_MUL), tr, tr2);
654       tr = emitir(IRTN(IR_FPMATH), tr, IRFPM_FLOOR);
655       tr = emitir(IRTN(IR_ADD), tr, tr1);
656     } else {  /* d = floor(d*r1) + 1.0 */
657       tr = emitir(IRTN(IR_MUL), tr, tr1);
658       tr = emitir(IRTN(IR_FPMATH), tr, IRFPM_FLOOR);
659       tr = emitir(IRTN(IR_ADD), tr, one);
660     }
```

```
  }
  J->base[0] = tr;
  UNUSED(rd);
}

/* -- Bit library fast functions ---------------------------------------- */

/* Record bit.tobit. */
static void LJ_FASTCALL recff_bit_tobit(jit_State *J, RecordFFData *rd)
{
  TRef tr = J->base[0];
#if LJ_HASFFI
  if (tref_iscdata(tr)) { recff_bit64_tobit(J, rd); return; }
#endif
  J->base[0] = lj_opt_narrow_tobit(J, tr);
  UNUSED(rd);
}

/* Record unary bit.bnot, bit.bswap. */
static void LJ_FASTCALL recff_bit_unary(jit_State *J, RecordFFData *rd)
{
#if LJ_HASFFI
  if (recff_bit64_unary(J, rd))
    return;
#endif
  J->base[0] = emitir(IRTI(rd->data), lj_opt_narrow_tobit(J, J->base[0]), 0);
}

/* Record N-ary bit.band, bit.bor, bit.bxor. */
static void LJ_FASTCALL recff_bit_nary(jit_State *J, RecordFFData *rd)
{
#if LJ_HASFFI
  if (recff_bit64_nary(J, rd))
    return;
#endif
  {
    TRef tr = lj_opt_narrow_tobit(J, J->base[0]);
    uint32_t ot = IRTI(rd->data);
    BCReg i;
    for (i = 1; J->base[i] != 0; i++)
      tr = emitir(ot, tr, lj_opt_narrow_tobit(J, J->base[i]));
    J->base[0] = tr;
  }
}

/* Record bit shifts. */
static void LJ_FASTCALL recff_bit_shift(jit_State *J, RecordFFData *rd)
{
#if LJ_HASFFI
  if (recff_bit64_shift(J, rd))
    return;
#endif
  {
    TRef tr = lj_opt_narrow_tobit(J, J->base[0]);
    TRef tsh = lj_opt_narrow_tobit(J, J->base[1]);
    IROp op = (IROp)rd->data;
    if (!(op < IR_BROL ? LJ_TARGET_MASKSHIFT : LJ_TARGET_MASKROT) &&
        !tref_isk(tsh))
      tsh = emitir(IRTI(IR_BAND), tsh, lj_ir_kint(J, 31));
#ifdef LJ_TARGET_UNIFYROT
    if (op == (LJ_TARGET_UNIFYROT == 1 ? IR_BROR : IR_BROL)) {
      op = LJ_TARGET_UNIFYROT == 1 ? IR_BROL : IR_BROR;
      tsh = emitir(IRTI(IR_NEG), tsh, tsh);
    }
#endif
    J->base[0] = emitir(IRTI(op), tr, tsh);
  }
}

static void LJ_FASTCALL recff_bit_tohex(jit_State *J, RecordFFData *rd)
{
#if LJ_HASFFI
  TRef hdr = recff_bufhdr(J);
  TRef tr = recff_bit64_tohex(J, rd, hdr);
  J->base[0] = emitir(IRT(IR_BUFSTR, IRT_STR), tr, hdr);
#else
```

```
  recff_nyiu(J, rd);  /* Don't bother working around this NYI. */
#endif
}

/* -- String library fast functions ----------------------------------- */

/* Specialize to relative starting position for string. */
static TRef recff_string_start(jit_State *J, GCstr *s, int32_t *st, TRef tr,
                               TRef trlen, TRef tr0)
{
  int32_t start = *st;
  if (start < 0) {
    emitir(IRTGI(IR_LT), tr, tr0);
    tr = emitir(IRTI(IR_ADD), trlen, tr);
    start = start + (int32_t)s->len;
    emitir(start < 0 ? IRTGI(IR_LT) : IRTGI(IR_GE), tr, tr0);
    if (start < 0) {
      tr = tr0;
      start = 0;
    }
  } else if (start == 0) {
    emitir(IRTGI(IR_EQ), tr, tr0);
    tr = tr0;
  } else {
    tr = emitir(IRTI(IR_ADD), tr, lj_ir_kint(J, -1));
    emitir(IRTGI(IR_GE), tr, tr0);
    start--;
  }
  *st = start;
  return tr;
}

/* Handle string.byte (rd->data = 0) and string.sub (rd->data = 1). */
static void LJ_FASTCALL recff_string_range(jit_State *J, RecordFFData *rd)
{
  TRef trstr = lj_ir_tostr(J, J->base[0]);
  TRef trlen = emitir(IRTI(IR_FLOAD), trstr, IRFL_STR_LEN);
  TRef tr0 = lj_ir_kint(J, 0);
  TRef trstart, trend;
  GCstr *str = argv2str(J, &rd->argv[0]);
  int32_t start, end;
  if (rd->data) {  /* string.sub(str, start [,end]) */
    start = argv2int(J, &rd->argv[1]);
    trstart = lj_opt_narrow_toint(J, J->base[1]);
    trend = J->base[2];
    if (tref_isnil(trend)) {
      trend = lj_ir_kint(J, -1);
      end = -1;
    } else {
      trend = lj_opt_narrow_toint(J, trend);
      end = argv2int(J, &rd->argv[2]);
    }
  } else {  /* string.byte(str, [,start [,end]]) */
    if (tref_isnil(J->base[1])) {
      start = 1;
      trstart = lj_ir_kint(J, 1);
    } else {
      start = argv2int(J, &rd->argv[1]);
      trstart = lj_opt_narrow_toint(J, J->base[1]);
    }
    if (J->base[1] && !tref_isnil(J->base[2])) {
      trend = lj_opt_narrow_toint(J, J->base[2]);
      end = argv2int(J, &rd->argv[2]);
    } else {
      trend = trstart;
      end = start;
    }
  }
  if (end < 0) {
    emitir(IRTGI(IR_LT), trend, tr0);
    trend = emitir(IRTI(IR_ADD), emitir(IRTI(IR_ADD), trlen, trend),
                   lj_ir_kint(J, 1));
    end = end+(int32_t)str->len+1;
  } else if ((MSize)end <= str->len) {
    emitir(IRTGI(IR_ULE), trend, trlen);
  } else {
```

```
    emitir(IRTGI(IR_UGT), trend, trlen);
    end = (int32_t)str->len;
    trend = trlen;
  }
  trstart = recff_string_start(J, str, &start, trstart, trlen, tr0);
  if (rd->data) {  /* Return string.sub result. */
    if (end - start >= 0) {
      /* Also handle empty range here, to avoid extra traces. */
      TRef trptr, trslen = emitir(IRTI(IR_SUB), trend, trstart);
      emitir(IRTGI(IR_GE), trslen, tr0);
      trptr = emitir(IRT(IR_STRREF, IRT_P32), trstr, trstart);
      J->base[0] = emitir(IRT(IR_SNEW, IRT_STR), trptr, trslen);
    } else {  /* Range underflow: return empty string. */
      emitir(IRTGI(IR_LT), trend, trstart);
      J->base[0] = lj_ir_kstr(J, &J2G(J)->strempty);
    }
  } else {  /* Return string.byte result(s). */
    ptrdiff_t i, len = end - start;
    if (len > 0) {
      TRef trslen = emitir(IRTI(IR_SUB), trend, trstart);
      emitir(IRTGI(IR_EQ), trslen, lj_ir_kint(J, (int32_t)len));
      if (J->baseslot + len > LJ_MAX_JSLOTS)
        lj_trace_err_info(J, LJ_TRERR_STACKOV);
      rd->nres = len;
      for (i = 0; i < len; i++) {
        TRef tmp = emitir(IRTI(IR_ADD), trstart, lj_ir_kint(J, (int32_t)i));
        tmp = emitir(IRT(IR_STRREF, IRT_P32), trstr, tmp);
        J->base[i] = emitir(IRT(IR_XLOAD, IRT_U8), tmp, IRXLOAD_READONLY);
      }
    } else {  /* Empty range or range underflow: return no results. */
      emitir(IRTGI(IR_LE), trend, trstart);
      rd->nres = 0;
    }
  }
}

static void LJ_FASTCALL recff_string_char(jit_State *J, RecordFFData *rd)
{
  TRef k255 = lj_ir_kint(J, 255);
  BCReg i;
  for (i = 0; J->base[i] != 0; i++) {  /* Convert char values to strings. */
    TRef tr = lj_opt_narrow_toint(J, J->base[i]);
    emitir(IRTGI(IR_ULE), tr, k255);
    J->base[i] = emitir(IRT(IR_TOSTR, IRT_STR), tr, IRTOSTR_CHAR);
  }
  if (i > 1) {  /* Concatenate the strings, if there's more than one. */
    TRef hdr = recff_bufhdr(J), tr = hdr;
    for (i = 0; J->base[i] != 0; i++)
      tr = emitir(IRT(IR_BUFPUT, IRT_P32), tr, J->base[i]);
    J->base[0] = emitir(IRT(IR_BUFSTR, IRT_STR), tr, hdr);
  }
  UNUSED(rd);
}

static void LJ_FASTCALL recff_string_rep(jit_State *J, RecordFFData *rd)
{
  TRef str = lj_ir_tostr(J, J->base[0]);
  TRef rep = lj_opt_narrow_toint(J, J->base[1]);
  TRef hdr, tr, str2 = 0;
  if (!tref_isnil(J->base[2])) {
    TRef sep = lj_ir_tostr(J, J->base[2]);
    int32_t vrep = argv2int(J, &rd->argv[1]);
    emitir(IRTGI(vrep > 1 ? IR_GT : IR_LE), rep, lj_ir_kint(J, 1));
    if (vrep > 1) {
      TRef hdr2 = recff_bufhdr(J);
      TRef tr2 = emitir(IRT(IR_BUFPUT, IRT_P32), hdr2, sep);
      tr2 = emitir(IRT(IR_BUFPUT, IRT_P32), tr2, str);
      str2 = emitir(IRT(IR_BUFSTR, IRT_STR), tr2, hdr2);
    }
  }
  tr = hdr = recff_bufhdr(J);
  if (str2) {
    tr = emitir(IRT(IR_BUFPUT, IRT_P32), tr, str);
    str = str2;
    rep = emitir(IRTI(IR_ADD), rep, lj_ir_kint(J, -1));
  }
```

```
  tr = lj_ir_call(J, IRCALL_lj_buf_putstr_rep, tr, str, rep);
  J->base[0] = emitir(IRT(IR_BUFSTR, IRT_STR), tr, hdr);
}

static void LJ_FASTCALL recff_string_op(jit_State *J, RecordFFData *rd)
{
  TRef str = lj_ir_tostr(J, J->base[0]);
  TRef hdr = recff_bufhdr(J);
  TRef tr = lj_ir_call(J, rd->data, hdr, str);
  J->base[0] = emitir(IRT(IR_BUFSTR, IRT_STR), tr, hdr);
}

static void LJ_FASTCALL recff_string_find(jit_State *J, RecordFFData *rd)
{
  TRef trstr = lj_ir_tostr(J, J->base[0]);
  TRef trpat = lj_ir_tostr(J, J->base[1]);
  TRef trlen = emitir(IRTI(IR_FLOAD), trstr, IRFL_STR_LEN);
  TRef tr0 = lj_ir_kint(J, 0);
  TRef trstart;
  GCstr *str = argv2str(J, &rd->argv[0]);
  GCstr *pat = argv2str(J, &rd->argv[1]);
  int32_t start;
  J->needsnap = 1;
  if (tref_isnil(J->base[2])) {
    trstart = lj_ir_kint(J, 1);
    start = 1;
  } else {
    trstart = lj_opt_narrow_toint(J, J->base[2]);
    start = argv2int(J, &rd->argv[2]);
  }
  trstart = recff_string_start(J, str, &start, trstart, trlen, tr0);
  if ((MSize)start <= str->len) {
    emitir(IRTGI(IR_ULE), trstart, trlen);
  } else {
    emitir(IRTGI(IR_UGT), trstart, trlen);
#if LJ_52
    J->base[0] = TREF_NIL;
    return;
#else
    trstart = trlen;
    start = str->len;
#endif
  }
  /* Fixed arg or no pattern matching chars? (Specialized to pattern string.) */
  if ((J->base[2] && tref_istruecond(J->base[3])) ||
      (emitir(IRTG(IR_EQ, IRT_STR), trpat, lj_ir_kstr(J, pat)),
       !lj_str_haspattern(pat))) {  /* Search for fixed string. */
    TRef trsptr = emitir(IRT(IR_STRREF, IRT_P32), trstr, trstart);
    TRef trpptr = emitir(IRT(IR_STRREF, IRT_P32), trpat, tr0);
    TRef trslen = emitir(IRTI(IR_SUB), trlen, trstart);
    TRef trplen = emitir(IRTI(IR_FLOAD), trpat, IRFL_STR_LEN);
    TRef tr = lj_ir_call(J, IRCALL_lj_str_find, trsptr, trpptr, trslen, trplen);
    TRef trp0 = lj_ir_kkptr(J, NULL);
    if (lj_str_find(strdata(str)+(MSize)start, strdata(pat),
                    str->len-(MSize)start, pat->len)) {
      TRef pos;
      emitir(IRTG(IR_NE, IRT_P32), tr, trp0);
      pos = emitir(IRTI(IR_SUB), tr, emitir(IRT(IR_STRREF, IRT_P32), trstr, tr0));
      J->base[0] = emitir(IRTI(IR_ADD), pos, lj_ir_kint(J, 1));
      J->base[1] = emitir(IRTI(IR_ADD), pos, trplen);
      rd->nres = 2;
    } else {
      emitir(IRTG(IR_EQ, IRT_P32), tr, trp0);
      J->base[0] = TREF_NIL;
    }
  } else {  /* Search for pattern. */
    recff_nyiu(J, rd);
    return;
  }
}

static void LJ_FASTCALL recff_string_format(jit_State *J, RecordFFData *rd)
{
  TRef trfmt = lj_ir_tostr(J, J->base[0]);
  GCstr *fmt = argv2str(J, &rd->argv[0]);
  int arg = 1;
```

```
  TRef hdr, tr;
  FormatState fs;
  SFormat sf;
  /* Specialize to the format string. */
  emitir(IRTG(IR_EQ, IRT_STR), trfmt, lj_ir_kstr(J, fmt));
  tr = hdr = recff_bufhdr(J);
  lj_strfmt_init(&fs, strdata(fmt), fmt->len);
  while ((sf = lj_strfmt_parse(&fs)) != STRFMT_EOF) {  /* Parse format. */
    TRef tra = sf == STRFMT_LIT ? 0 : J->base[arg++];
    TRef trsf = lj_ir_kint(J, (int32_t)sf);
    IRCallID id;
    switch (STRFMT_TYPE(sf)) {
    case STRFMT_LIT:
      tr = emitir(IRT(IR_BUFPUT, IRT_P32), tr,
                  lj_ir_kstr(J, lj_str_new(J->L, fs.str, fs.len)));
      break;
    case STRFMT_INT:
      id = IRCALL_lj_strfmt_putfnum_int;
    handle_int:
      if (!tref_isinteger(tra))
        goto handle_num;
      if (sf == STRFMT_INT) { /* Shortcut for plain %d. */
        tr = emitir(IRT(IR_BUFPUT, IRT_P32), tr,
                    emitir(IRT(IR_TOSTR, IRT_STR), tra, IRTOSTR_INT));
      } else {
#if LJ_HASFFI
        tra = emitir(IRT(IR_CONV, IRT_U64), tra,
                     (IRT_INT|(IRT_U64<<5)|IRCONV_SEXT));
        tr = lj_ir_call(J, IRCALL_lj_strfmt_putfxint, tr, trsf, tra);
        lj_needsplit(J);
#else
        recff_nyiu(J, rd);  /* Don't bother working around this NYI. */
        return;
#endif
      }
      break;
    case STRFMT_UINT:
      id = IRCALL_lj_strfmt_putfnum_uint;
      goto handle_int;
    case STRFMT_NUM:
      id = IRCALL_lj_strfmt_putfnum;
    handle_num:
      tra = lj_ir_tonum(J, tra);
      tr = lj_ir_call(J, id, tr, trsf, tra);
      if (LJ_SOFTFP) lj_needsplit(J);
      break;
    case STRFMT_STR:
      if (!tref_isstr(tra)) {
        recff_nyiu(J, rd);  /* NYI: __tostring and non-string types for %s. */
        return;
      }
      if (sf == STRFMT_STR)  /* Shortcut for plain %s. */
        tr = emitir(IRT(IR_BUFPUT, IRT_P32), tr, tra);
      else if ((sf & STRFMT_T_QUOTED))
        tr = lj_ir_call(J, IRCALL_lj_strfmt_putquoted, tr, tra);
      else
        tr = lj_ir_call(J, IRCALL_lj_strfmt_putfstr, tr, trsf, tra);
      break;
    case STRFMT_CHAR:
      tra = lj_opt_narrow_toint(J, tra);
      if (sf == STRFMT_CHAR)  /* Shortcut for plain %c. */
        tr = emitir(IRT(IR_BUFPUT, IRT_P32), tr,
                    emitir(IRT(IR_TOSTR, IRT_STR), tra, IRTOSTR_CHAR));
      else
        tr = lj_ir_call(J, IRCALL_lj_strfmt_putfchar, tr, trsf, tra);
      break;
    case STRFMT_PTR:  /* NYI */
    case STRFMT_ERR:
    default:
      recff_nyiu(J, rd);
      return;
    }
  }
  J->base[0] = emitir(IRT(IR_BUFSTR, IRT_STR), tr, hdr);
}
```

```
1041 /* -- Table library fast functions ----------------------------------- */
1042
1043 static void LJ_FASTCALL recff_table_insert(jit_State *J, RecordFFData *rd)
1044 {
1045   RecordIndex ix;
1046   ix.tab = J->base[0];
1047   ix.val = J->base[1];
1048   rd->nres = 0;
1049   if (tref_istab(ix.tab) && ix.val) {
1050     if (!J->base[2]) {  /* Simple push: t[#t+1] = v */
1051       TRef trlen = lj_ir_call(J, IRCALL_lj_tab_len, ix.tab);
1052       GCtab *t = tabV(&rd->argv[0]);
1053       ix.key = emitir(IRTI(IR_ADD), trlen, lj_ir_kint(J, 1));
1054       settabV(J->L, &ix.tabv, t);
1055       setintV(&ix.keyv, lj_tab_len(t) + 1);
1056       ix.idxchain = 0;
1057       lj_record_idx(J, &ix);  /* Set new value. */
1058     } else {  /* Complex case: insert in the middle. */
1059       recff_nyiu(J, rd);
1060       return;
1061     }
1062   }  /* else: Interpreter will throw. */
1063 }
1064
1065 static void LJ_FASTCALL recff_table_concat(jit_State *J, RecordFFData *rd)
1066 {
1067   TRef tab = J->base[0];
1068   if (tref_istab(tab)) {
1069     TRef sep = !tref_isnil(J->base[1]) ?
1070                lj_ir_tostr(J, J->base[1]) : lj_ir_knull(J, IRT_STR);
1071     TRef tri = (J->base[1] && !tref_isnil(J->base[2])) ?
1072                lj_opt_narrow_toint(J, J->base[2]) : lj_ir_kint(J, 1);
1073     TRef tre = (J->base[1] && J->base[2] && !tref_isnil(J->base[3])) ?
1074                lj_opt_narrow_toint(J, J->base[3]) :
1075                lj_ir_call(J, IRCALL_lj_tab_len, tab);
1076     TRef hdr = recff_bufhdr(J);
1077     TRef tr = lj_ir_call(J, IRCALL_lj_buf_puttab, hdr, tab, sep, tri, tre);
1078     emitir(IRTG(IR_NE, IRT_PTR), tr, lj_ir_kptr(J, NULL));
1079     J->base[0] = emitir(IRT(IR_BUFSTR, IRT_STR), tr, hdr);
1080   }  /* else: Interpreter will throw. */
1081   UNUSED(rd);
1082 }
1083
1084 static void LJ_FASTCALL recff_table_new(jit_State *J, RecordFFData *rd)
1085 {
1086   TRef tra = lj_opt_narrow_toint(J, J->base[0]);
1087   TRef trh = lj_opt_narrow_toint(J, J->base[1]);
1088   J->base[0] = lj_ir_call(J, IRCALL_lj_tab_new_ah, tra, trh);
1089   UNUSED(rd);
1090 }
1091
1092 static void LJ_FASTCALL recff_table_clear(jit_State *J, RecordFFData *rd)
1093 {
1094   TRef tr = J->base[0];
1095   if (tref_istab(tr)) {
1096     rd->nres = 0;
1097     lj_ir_call(J, IRCALL_lj_tab_clear, tr);
1098     J->needsnap = 1;
1099   }  /* else: Interpreter will throw. */
1100 }
1101
1102 /* -- I/O library fast functions ------------------------------------------ */
1103
1104 /* Get FILE* for I/O function. Any I/O error aborts recording, so there's
1105 ** no need to encode the alternate cases for any of the guards.
1106 */
1107 static TRef recff_io_fp(jit_State *J, TRef *udp, int32_t id)
1108 {
1109   TRef tr, ud, fp;
1110   if (id) {  /* io.func() */
1111     tr = lj_ir_kptr(J, &J2G(J)->gcroot[id]);
1112     ud = emitir(IRT(IR_XLOAD, IRT_UDATA), tr, 0);
1113   } else {  /* fp:method() */
1114     ud = J->base[0];
1115     if (!tref_isudata(ud))
1116       lj_trace_err(J, LJ_TRERR_BADTYPE);
```

```
1117      tr = emitir(IRT(IR_FLOAD, IRT_U8), ud, IRFL_UDATA_UDTYPE);
1118      emitir(IRTGI(IR_EQ), tr, lj_ir_kint(J, UDTYPE_IO_FILE));
1119    }
1120    *udp = ud;
1121    fp = emitir(IRT(IR_FLOAD, IRT_PTR), ud, IRFL_UDATA_FILE);
1122    emitir(IRTG(IR_NE, IRT_PTR), fp, lj_ir_knull(J, IRT_PTR));
1123    return fp;
1124  }
1125
1126  static void LJ_FASTCALL recff_io_write(jit_State *J, RecordFFData *rd)
1127  {
1128    TRef ud, fp = recff_io_fp(J, &ud, rd->data);
1129    TRef zero = lj_ir_kint(J, 0);
1130    TRef one = lj_ir_kint(J, 1);
1131    ptrdiff_t i = rd->data == 0 ? 1 : 0;
1132    for (; J->base[i]; i++) {
1133      TRef str = lj_ir_tostr(J, J->base[i]);
1134      TRef buf = emitir(IRT(IR_STRREF, IRT_P32), str, zero);
1135      TRef len = emitir(IRTI(IR_FLOAD), str, IRFL_STR_LEN);
1136      if (tref_isk(len) && IR(tref_ref(len))->i == 1) {
1137        IRIns *irs = IR(tref_ref(str));
1138        TRef tr = (irs->o == IR_TOSTR && irs->op2 == IRTOSTR_CHAR) ?
1139                  irs->op1 :
1140                  emitir(IRT(IR_XLOAD, IRT_U8), buf, IRXLOAD_READONLY);
1141        tr = lj_ir_call(J, IRCALL_fputc, tr, fp);
1142        if (results_wanted(J) != 0)  /* Check result only if not ignored. */
1143          emitir(IRTGI(IR_NE), tr, lj_ir_kint(J, -1));
1144      } else {
1145        TRef tr = lj_ir_call(J, IRCALL_fwrite, buf, one, len, fp);
1146        if (results_wanted(J) != 0)  /* Check result only if not ignored. */
1147          emitir(IRTGI(IR_EQ), tr, len);
1148      }
1149    }
1150    J->base[0] = LJ_52 ? ud : TREF_TRUE;
1151  }
1152
1153  static void LJ_FASTCALL recff_io_flush(jit_State *J, RecordFFData *rd)
1154  {
1155    TRef ud, fp = recff_io_fp(J, &ud, rd->data);
1156    TRef tr = lj_ir_call(J, IRCALL_fflush, fp);
1157    if (results_wanted(J) != 0)  /* Check result only if not ignored. */
1158      emitir(IRTGI(IR_EQ), tr, lj_ir_kint(J, 0));
1159    J->base[0] = TREF_TRUE;
1160  }
1161
1162  /* -- Debug library fast functions ---------------------------------------- */
1163
1164  static void LJ_FASTCALL recff_debug_getmetatable(jit_State *J, RecordFFData *rd)
1165  {
1166    GCtab *mt;
1167    TRef mtref;
1168    TRef tr = J->base[0];
1169    if (tref_istab(tr)) {
1170      mt = tabref(tabV(&rd->argv[0])->metatable);
1171      mtref = emitir(IRT(IR_FLOAD, IRT_TAB), tr, IRFL_TAB_META);
1172    } else if (tref_isudata(tr)) {
1173      mt = tabref(udataV(&rd->argv[0])->metatable);
1174      mtref = emitir(IRT(IR_FLOAD, IRT_TAB), tr, IRFL_UDATA_META);
1175    } else {
1176      mt = tabref(basemt_obj(J2G(J), &rd->argv[0]));
1177      J->base[0] = mt ? lj_ir_ktab(J, mt) : TREF_NIL;
1178      return;
1179    }
1180    emitir(IRTG(mt ? IR_NE : IR_EQ, IRT_TAB), mtref, lj_ir_knull(J, IRT_TAB));
1181    J->base[0] = mt ? mtref : TREF_NIL;
1182  }
1183
1184  /* -- Record calls to fast functions -------------------------------------- */
1185
1186  #include "lj_recdef.h"
1187
1188  static uint32_t recdef_lookup(GCfunc *fn)
1189  {
1190    if (fn->c.ffid < sizeof(recff_idmap)/sizeof(recff_idmap[0]))
1191      return recff_idmap[fn->c.ffid];
1192    else
```

```
1193      return 0;
1194  }
1195
1196  /* Record entry to a fast function or C function. */
1197  void lj_ffrecord_func(jit_State *J)
1198  {
1199    RecordFFData rd;
1200    uint32_t m = recdef_lookup(J->fn);
1201    rd.data = m & 0xff;
1202    rd.nres = 1;  /* Default is one result. */
1203    rd.argv = J->L->base;
1204    J->base[J->maxslot] = 0;  /* Mark end of arguments. */
1205    (recff_func[m >> 8])(J, &rd);  /* Call recff_* handler. */
1206    if (rd.nres >= 0) {
1207      if (J->postproc == LJ_POST_NONE) J->postproc = LJ_POST_FFRETRY;
1208      lj_record_ret(J, 0, rd.nres);
1209    }
1210  }
1211
1212  #undef IR
1213  #undef emitir
1214
1215  #endif
```

One Level Up Top Level

src/lj_ffrecord.h - luajit-2.0-src

Data types defined

- RecordFFData
- RecordFFData

Macros defined

- _LJ_FFRECORD_H

Source code

```
/*
** Fast function call recorder.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_FFRECORD_H
#define _LJ_FFRECORD_H

#include "lj_obj.h"
 #include "lj_jit.h"

 #if LJ_HASJIT
 /* Data used by handlers to record a fast function. */
 typedef struct RecordFFData {
   TValue *argv;                /* Runtime argument values. */
   ptrdiff_t nres;        /* Number of returned results (defaults to 1). */
   uint32_t data;        /* Per-ffid auxiliary data (opcode, literal etc.). */
 } RecordFFData;

 LJ_FUNC int32_t lj_ffrecord_select_mode(jit_State *J, TRef tr, TValue *tv);
 LJ_FUNC void lj_ffrecord_func(jit_State *J);
 #endif

 #endif
```

src/lj_record.c - luajit-2.0-src

Data types defined

- LoopEvent

Functions defined

- canonicalize_slots
- check_call_unroll
- check_downrec_unroll
- find_kinit
- fori_arg
- fori_load
- getcurrf
- innerloopleft
- lj_record_call
- lj_record_constify
- lj_record_idx
- lj_record_ins
- lj_record_mm_lookup
- lj_record_objcmp
- lj_record_ret
- lj_record_setup
- lj_record_stop
- lj_record_tailcall
- nommstr
- rec_call_setup
- rec_call_specialize
- rec_cat
- rec_check_ir
- rec_check_slots
- rec_comp_fixup
- rec_comp_prep
- rec_for

- rec_for_check
- rec_for_direction
- rec_for_iter
- rec_for_loop
- rec_func_jit
- rec_func_lua
- rec_func_setup
- rec_func_vararg
- rec_idx_abc
- rec_idx_key
- rec_iterl
- rec_loop
- rec_loop_interp
- rec_loop_jit
- rec_mm_arith
- rec_mm_callcomp
- rec_mm_comp
- rec_mm_comp_cdata
- rec_mm_equal
- rec_mm_len
- rec_mm_prep
- rec_profile_ins
- rec_profile_need
- rec_profile_ret
- rec_setup_root
- rec_tnew
- rec_tsetm
- rec_upvalue
- rec_upvalue_constify
- rec_varg
- select_detect
- sload
- sloadt

Macros defined

- IR
- IR
- LUA_CORE
- emitir
- emitir
- emitir_raw
- emitir_raw
- getslot
- lj_record_c
- rav
- rav
- rbv
- rbv
- rcv
- rcv

Source code

```
/*
** Trace recorder (bytecode -> SSA IR).
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_record_c
#define LUA_CORE

#include "lj_obj.h"

 #if LJ_HASJIT

 #include "lj_err.h"
 #include "lj_str.h"
 #include "lj_tab.h"
 #include "lj_meta.h"
 #include "lj_frame.h"
 #if LJ_HASFFI
 #include "lj_ctype.h"
 #endif
 #include "lj_bc.h"
 #include "lj_ff.h"
 #if LJ_HASPROFILE
 #include "lj_debug.h"
 #endif
 #include "lj_ir.h"
 #include "lj_jit.h"
 #include "lj_ircall.h"
 #include "lj_iropt.h"
 #include "lj_trace.h"
 #include "lj_record.h"
 #include "lj_ffrecord.h"
 #include "lj_snap.h"
 #include "lj_dispatch.h"
 #include "lj_vm.h"

```

```
/* Some local macros to save typing. Undef'd at the end. */
#define IR(ref)                         (&J->cur.ir[(ref)])

/* Pass IR on to next optimization in chain (FOLD). */
#define emitir(ot, a, b)        (lj_ir_set(J, (ot), (a), (b)), lj_opt_fold(J))

/* Emit raw IR without passing through optimizations. */
#define emitir_raw(ot, a, b)        (lj_ir_set(J, (ot), (a), (b)), lj_ir_emit(J))

/* -- Sanity checks ------------------------------------------------------- */

#ifdef LUA_USE_ASSERT
/* Sanity check the whole IR -- sloooow. */
static void rec_check_ir(jit_State *J)
{
  IRRef i, nins = J->cur.nins, nk = J->cur.nk;
  lua_assert(nk <= REF_BIAS && nins >= REF_BIAS && nins < 65536);
  for (i = nins-1; i >= nk; i--) {
    IRIns *ir = IR(i);
    uint32_t mode = lj_ir_mode[ir->o];
    IRRef op1 = ir->op1;
    IRRef op2 = ir->op2;
    switch (irm_op1(mode)) {
    case IRMnone: lua_assert(op1 == 0); break;
    case IRMref: lua_assert(op1 >= nk);
      lua_assert(i >= REF_BIAS ? op1 < i : op1 > i); break;
    case IRMlit: break;
    case IRMcst: lua_assert(i < REF_BIAS); continue;
    }
    switch (irm_op2(mode)) {
    case IRMnone: lua_assert(op2 == 0); break;
    case IRMref: lua_assert(op2 >= nk);
      lua_assert(i >= REF_BIAS ? op2 < i : op2 > i); break;
    case IRMlit: break;
    case IRMcst: lua_assert(0); break;
    }
    if (ir->prev) {
      lua_assert(ir->prev >= nk);
      lua_assert(i >= REF_BIAS ? ir->prev < i : ir->prev > i);
      lua_assert(ir->o == IR_NOP || IR(ir->prev)->o == ir->o);
    }
  }
}

/* Compare stack slots and frames of the recorder and the VM. */
static void rec_check_slots(jit_State *J)
{
  BCReg s, nslots = J->baseslot + J->maxslot;
  int32_t depth = 0;
  cTValue *base = J->L->base - J->baseslot;
  lua_assert(J->baseslot >= 1 && J->baseslot < LJ_MAX_JSLOTS);
  lua_assert(J->baseslot == 1 || (J->slot[J->baseslot-1] & TREF_FRAME));
  lua_assert(nslots < LJ_MAX_JSLOTS);
  for (s = 0; s < nslots; s++) {
    TRef tr = J->slot[s];
    if (tr) {
      cTValue *tv = &base[s];
      IRRef ref = tref_ref(tr);
      IRIns *ir;
      lua_assert(ref >= J->cur.nk && ref < J->cur.nins);
      ir = IR(ref);
      lua_assert(irt_t(ir->t) == tref_t(tr));
      if (s == 0) {
         lua_assert(tref_isfunc(tr));
       } else if ((tr & TREF_FRAME)) {
         GCfunc *fn = gco2func(frame_gc(tv));
         BCReg delta = (BCReg)(tv - frame_prev(tv));
         lua_assert(tref_isfunc(tr));
         if (tref_isk(tr)) lua_assert(fn == ir_kfunc(ir));
         lua_assert(s > delta ? (J->slot[s-delta] & TREF_FRAME) : (s == delta));
         depth++;
       } else if ((tr & TREF_CONT)) {
         lua_assert(ir_kptr(ir) == gcrefp(tv->gcr, void));
         lua_assert((J->slot[s+1] & TREF_FRAME));
         depth++;
       } else {
```

```
        if (tvisnumber(tv))
          lua_assert(tref_isnumber(tr));  /* Could be IRT_INT etc., too. */
        else
          lua_assert(itype2irt(tv) == tref_type(tr));
        if (tref_isk(tr)) {  /* Compare constants. */
          TValue tvk;
          lj_ir_kvalue(J->L, &tvk, ir);
          if (!(tvisnum(&tvk) && tvisnan(&tvk)))
            lua_assert(lj_obj_equal(tv, &tvk));
          else
            lua_assert(tvisnum(tv) && tvisnan(tv));
        }
      }
    }
  }
  lua_assert(J->framedepth == depth);
}
#endif

/* -- Type handling and specialization ------------------------------------ */

/* Note: these functions return tagged references (TRef). */

/* Specialize a slot to a specific type. Note: slot can be negative! */
static TRef sloadt(jit_State *J, int32_t slot, IRType t, int mode)
{
  /* Caller may set IRT_GUARD in t. */
  TRef ref = emitir_raw(IRT(IR_SLOAD, t), (int32_t)J->baseslot+slot, mode);
  J->base[slot] = ref;
  return ref;
}

/* Specialize a slot to the runtime type. Note: slot can be negative! */
static TRef sload(jit_State *J, int32_t slot)
{
  IRType t = itype2irt(&J->L->base[slot]);
  TRef ref = emitir_raw(IRTG(IR_SLOAD, t), (int32_t)J->baseslot+slot,
                        IRSLOAD_TYPECHECK);
  if (irtype_ispri(t)) ref = TREF_PRI(t);  /* Canonicalize primitive refs. */
  J->base[slot] = ref;
  return ref;
}

/* Get TRef from slot. Load slot and specialize if not done already. */
#define getslot(J, s)        (J->base[(s)] ? J->base[(s)] : sload(J, (int32_t)(s)))

/* Get TRef for current function. */
static TRef getcurrf(jit_State *J)
{
  if (J->base[-1])
    return J->base[-1];
  lua_assert(J->baseslot == 1);
  return sloadt(J, -1, IRT_FUNC, IRSLOAD_READONLY);
}

/* Compare for raw object equality.
** Returns 0 if the objects are the same.
** Returns 1 if they are different, but the same type.
** Returns 2 for two different types.
** Comparisons between primitives always return 1 -- no caller cares about it.
*/
int lj_record_objcmp(jit_State *J, TRef a, TRef b, cTValue *av, cTValue *bv)
{
  int diff = !lj_obj_equal(av, bv);
  if (!tref_isk2(a, b)) {  /* Shortcut, also handles primitives. */
    IRType ta = tref_isinteger(a) ? IRT_INT : tref_type(a);
    IRType tb = tref_isinteger(b) ? IRT_INT : tref_type(b);
    if (ta != tb) {
      /* Widen mixed number/int comparisons to number/number comparison. */
      if (ta == IRT_INT && tb == IRT_NUM) {
        a = emitir(IRTN(IR_CONV), a, IRCONV_NUM_INT);
        ta = IRT_NUM;
      } else if (ta == IRT_NUM && tb == IRT_INT) {
        b = emitir(IRTN(IR_CONV), b, IRCONV_NUM_INT);
      } else {
        return 2;  /* Two different types are never equal. */
```

```
      }
    }
    emitir(IRTG(diff ? IR_NE : IR_EQ, ta), a, b);
  }
  return diff;
}

/* Constify a value. Returns 0 for non-representable object types. */
TRef lj_record_constify(jit_State *J, cTValue *o)
{
  if (tvisgcv(o))
    return lj_ir_kgc(J, gcV(o), itype2irt(o));
  else if (tvisint(o))
    return lj_ir_kint(J, intV(o));
  else if (tvisnum(o))
    return lj_ir_knumint(J, numV(o));
  else if (tvisbool(o))
    return TREF_PRI(itype2irt(o));
  else
    return 0;  /* Can't represent lightuserdata (pointless). */
}

/* -- Record loop ops ------------------------------------------------------ */

/* Loop event. */
typedef enum {
  LOOPEV_LEAVE,		/* Loop is left or not entered. */
  LOOPEV_ENTERLO,	/* Loop is entered with a low iteration count left. */
  LOOPEV_ENTER		/* Loop is entered. */
} LoopEvent;

/* Canonicalize slots: convert integers to numbers. */
static void canonicalize_slots(jit_State *J)
{
  BCReg s;
  if (LJ_DUALNUM) return;
  for (s = J->baseslot+J->maxslot-1; s >= 1; s--) {
    TRef tr = J->slot[s];
    if (tref_isinteger(tr)) {
      IRIns *ir = IR(tref_ref(tr));
      if (!(ir->o == IR_SLOAD && (ir->op2 & IRSLOAD_READONLY)))
        J->slot[s] = emitir(IRTN(IR_CONV), tr, IRCONV_NUM_INT);
    }
  }
}

/* Stop recording. */
void lj_record_stop(jit_State *J, TraceLink linktype, TraceNo lnk)
{
  lj_trace_end(J);
  J->cur.linktype = (uint8_t)linktype;
  J->cur.link = (uint16_t)lnk;
  /* Looping back at the same stack level? */
  if (lnk == J->cur.traceno && J->framedepth + J->retdepth == 0) {
    if ((J->flags & JIT_F_OPT_LOOP))  /* Shall we try to create a loop? */
      goto nocanon;  /* Do not canonicalize or we lose the narrowing. */
    if (J->cur.root)  /* Otherwise ensure we always link to the root trace. */
      J->cur.link = J->cur.root;
  }
  canonicalize_slots(J);
nocanon:
  /* Note: all loop ops must set J->pc to the following instruction! */
  lj_snap_add(J);  /* Add loop snapshot. */
  J->needsnap = 0;
  J->mergesnap = 1;  /* In case recording continues. */
}

/* Search bytecode backwards for a int/num constant slot initializer. */
static TRef find_kinit(jit_State *J, const BCIns *endpc, BCReg slot, IRType t)
{
  /* This algorithm is rather simplistic and assumes quite a bit about
  ** how the bytecode is generated. It works fine for FORI initializers,
  ** but it won't necessarily work in other cases (e.g. iterator arguments).
  ** It doesn't do anything fancy, either (like backpropagating MOVs).
  */
  const BCIns *pc, *startpc = proto_bc(J->pt);
```

```
  for (pc = endpc-1; pc > startpc; pc--) {
    BCIns ins = *pc;
    BCOp op = bc_op(ins);
    /* First try to find the last instruction that stores to this slot. */
    if (bcmode_a(op) == BCMbase && bc_a(ins) <= slot) {
      return 0;  /* Multiple results, e.g. from a CALL or KNIL. */
    } else if (bcmode_a(op) == BCMdst && bc_a(ins) == slot) {
      if (op == BC_KSHORT || op == BC_KNUM) {  /* Found const. initializer. */
        /* Now try to verify there's no forward jump across it. */
        const BCIns *kpc = pc;
        for (; pc > startpc; pc--)
          if (bc_op(*pc) == BC_JMP) {
            const BCIns *target = pc+bc_j(*pc)+1;
            if (target > kpc && target <= endpc)
              return 0;  /* Conditional assignment. */
          }
        if (op == BC_KSHORT) {
          int32_t k = (int32_t)(int16_t)bc_d(ins);
          return t == IRT_INT ? lj_ir_kint(J, k) : lj_ir_knum(J, (lua_Number)k);
        } else {
          cTValue *tv = proto_knumtv(J->pt, bc_d(ins));
          if (t == IRT_INT) {
            int32_t k = numberVint(tv);
            if (tvisint(tv) || numV(tv) == (lua_Number)k)  /* -0 is ok here. */
              return lj_ir_kint(J, k);
            return 0;  /* Type mismatch. */
          } else {
            return lj_ir_knum(J, numberVnum(tv));
          }
        }
      }
      return 0;  /* Non-constant initializer. */
    }
  }
  return 0;  /* No assignment to this slot found? */
}

/* Load and optionally convert a FORI argument from a slot. */
static TRef fori_load(jit_State *J, BCReg slot, IRType t, int mode)
{
  int conv = (tvisint(&J->L->base[slot]) != (t==IRT_INT)) ? IRSLOAD_CONVERT : 0;
  return sloadt(J, (int32_t)slot,
                t + (((mode & IRSLOAD_TYPECHECK) ||
                      (conv && t == IRT_INT && !(mode >> 16))) ?
                     IRT_GUARD : 0),
                mode + conv);
}

/* Peek before FORI to find a const initializer. Otherwise load from slot. */
static TRef fori_arg(jit_State *J, const BCIns *fori, BCReg slot,
                     IRType t, int mode)
{
  TRef tr = J->base[slot];
  if (!tr) {
    tr = find_kinit(J, fori, slot, t);
    if (!tr)
      tr = fori_load(J, slot, t, mode);
  }
  return tr;
}

/* Return the direction of the FOR loop iterator.
** It's important to exactly reproduce the semantics of the interpreter.
*/
static int rec_for_direction(cTValue *o)
{
  return (tvisint(o) ? intV(o) : (int32_t)o->u32.hi) >= 0;
}

/* Simulate the runtime behavior of the FOR loop iterator. */
static LoopEvent rec_for_iter(IROp *op, cTValue *o, int isforl)
{
  lua_Number stopv = numberVnum(&o[FORL_STOP]);
  lua_Number idxv = numberVnum(&o[FORL_IDX]);
  lua_Number stepv = numberVnum(&o[FORL_STEP]);
  if (isforl)
```

```
    idxv += stepv;
  if (rec_for_direction(&o[FORL_STEP])) {
    if (idxv <= stopv) {
      *op = IR_LE;
      return idxv + 2*stepv > stopv ? LOOPEV_ENTERLO : LOOPEV_ENTER;
    }
    *op = IR_GT; return LOOPEV_LEAVE;
  } else {
    if (stopv <= idxv) {
      *op = IR_GE;
      return idxv + 2*stepv < stopv ? LOOPEV_ENTERLO : LOOPEV_ENTER;
    }
    *op = IR_LT; return LOOPEV_LEAVE;
  }
}

/* Record checks for FOR loop overflow and step direction. */
static void rec_for_check(jit_State *J, IRType t, int dir,
                          TRef stop, TRef step, int init)
{
  if (!tref_isk(step)) {
    /* Non-constant step: need a guard for the direction. */
    TRef zero = (t == IRT_INT) ? lj_ir_kint(J, 0) : lj_ir_knum_zero(J);
    emitir(IRTG(dir ? IR_GE : IR_LT, t), step, zero);
    /* Add hoistable overflow checks for a narrowed FORL index. */
    if (init && t == IRT_INT) {
      if (tref_isk(stop)) {
        /* Constant stop: optimize check away or to a range check for step. */
        int32_t k = IR(tref_ref(stop))->i;
        if (dir) {
          if (k > 0)
            emitir(IRTGI(IR_LE), step, lj_ir_kint(J, (int32_t)0x7fffffff-k));
        } else {
          if (k < 0)
            emitir(IRTGI(IR_GE), step, lj_ir_kint(J, (int32_t)0x80000000-k));
        }
      } else {
        /* Stop+step variable: need full overflow check. */
        TRef tr = emitir(IRTGI(IR_ADDOV), step, stop);
        emitir(IRTI(IR_USE), tr, 0);  /* ADDOV is weak. Avoid dead result. */
      }
    }
  } else if (init && t == IRT_INT && !tref_isk(stop)) {
    /* Constant step: optimize overflow check to a range check for stop. */
    int32_t k = IR(tref_ref(step))->i;
    k = (int32_t)(dir ? 0x7fffffff : 0x80000000) - k;
    emitir(IRTGI(dir ? IR_LE : IR_GE), stop, lj_ir_kint(J, k));
  }
}

/* Record a FORL instruction. */
static void rec_for_loop(jit_State *J, const BCIns *fori, ScEvEntry *scev,
                         int init)
{
  BCReg ra = bc_a(*fori);
  cTValue *tv = &J->L->base[ra];
  TRef idx = J->base[ra+FORL_IDX];
  IRType t = idx ? tref_type(idx) :
             (init || LJ_DUALNUM) ? lj_opt_narrow_forl(J, tv) : IRT_NUM;
  int mode = IRSLOAD_INHERIT +
    ((!LJ_DUALNUM || tvisint(tv) == (t == IRT_INT)) ? IRSLOAD_READONLY : 0);
  TRef stop = fori_arg(J, fori, ra+FORL_STOP, t, mode);
  TRef step = fori_arg(J, fori, ra+FORL_STEP, t, mode);
  int tc, dir = rec_for_direction(&tv[FORL_STEP]);
  lua_assert(bc_op(*fori) == BC_FORI || bc_op(*fori) == BC_JFORI);
  scev->t.irt = t;
  scev->dir = dir;
  scev->stop = tref_ref(stop);
  scev->step = tref_ref(step);
  rec_for_check(J, t, dir, stop, step, init);
  scev->start = tref_ref(find_kinit(J, fori, ra+FORL_IDX, IRT_INT));
  tc = (LJ_DUALNUM &&
        !(scev->start && irref_isk(scev->stop) && irref_isk(scev->step) &&
          tvisint(&tv[FORL_IDX]) == (t == IRT_INT))) ?
        IRSLOAD_TYPECHECK : 0;
  if (tc) {
```

```
    J->base[ra+FORL_STOP] = stop;
    J->base[ra+FORL_STEP] = step;
  }
  if (!idx)
    idx = fori_load(J, ra+FORL_IDX, t,
                    IRSLOAD_INHERIT + tc + (J->scev.start << 16));
  if (!init)
    J->base[ra+FORL_IDX] = idx = emitir(IRT(IR_ADD, t), idx, step);
  J->base[ra+FORL_EXT] = idx;
  scev->idx = tref_ref(idx);
  setmref(scev->pc, fori);
  J->maxslot = ra+FORL_EXT+1;
}

/* Record FORL/JFORL or FORI/JFORI. */
static LoopEvent rec_for(jit_State *J, const BCIns *fori, int isforl)
{
  BCReg ra = bc_a(*fori);
  TValue *tv = &J->L->base[ra];
  TRef *tr = &J->base[ra];
  IROp op;
  LoopEvent ev;
  TRef stop;
  IRType t;
  if (isforl) {  /* Handle FORL/JFORL opcodes. */
    TRef idx = tr[FORL_IDX];
    if (mref(J->scev.pc, const BCIns) == fori && tref_ref(idx) == J->scev.idx) {
      t = J->scev.t.irt;
      stop = J->scev.stop;
      idx = emitir(IRT(IR_ADD, t), idx, J->scev.step);
      tr[FORL_EXT] = tr[FORL_IDX] = idx;
    } else {
      ScEvEntry scev;
      rec_for_loop(J, fori, &scev, 0);
      t = scev.t.irt;
      stop = scev.stop;
    }
  } else {  /* Handle FORI/JFORI opcodes. */
    BCReg i;
    lj_meta_for(J->L, tv);
    t = (LJ_DUALNUM || tref_isint(tr[FORL_IDX])) ? lj_opt_narrow_forl(J, tv) :
                                                   IRT_NUM;
    for (i = FORL_IDX; i <= FORL_STEP; i++) {
      if (!tr[i]) sload(J, ra+i);
      lua_assert(tref_isnumber_str(tr[i]));
      if (tref_isstr(tr[i]))
        tr[i] = emitir(IRTG(IR_STRTO, IRT_NUM), tr[i], 0);
      if (t == IRT_INT) {
        if (!tref_isinteger(tr[i]))
          tr[i] = emitir(IRTGI(IR_CONV), tr[i], IRCONV_INT_NUM|IRCONV_CHECK);
      } else {
        if (!tref_isnum(tr[i]))
          tr[i] = emitir(IRTN(IR_CONV), tr[i], IRCONV_NUM_INT);
      }
    }
    tr[FORL_EXT] = tr[FORL_IDX];
    stop = tr[FORL_STOP];
    rec_for_check(J, t, rec_for_direction(&tv[FORL_STEP]),
                  stop, tr[FORL_STEP], 1);
  }

  ev = rec_for_iter(&op, tv, isforl);
  if (ev == LOOPEV_LEAVE) {
    J->maxslot = ra+FORL_EXT+1;
    J->pc = fori+1;
  } else {
    J->maxslot = ra;
    J->pc = fori+bc_j(*fori)+1;
  }
  lj_snap_add(J);

  emitir(IRTG(op, t), tr[FORL_IDX], stop);

  if (ev == LOOPEV_LEAVE) {
    J->maxslot = ra;
    J->pc = fori+bc_j(*fori)+1;
```

```
  } else {
    J->maxslot = ra+FORL_EXT+1;
    J->pc = fori+1;
  }
  J->needsnap = 1;
  return ev;
}

/* Record ITERL/JITERL. */
static LoopEvent rec_iterl(jit_State *J, const BCIns iterins)
{
  BCReg ra = bc_a(iterins);
  lua_assert(!LJ_FR2);  /* TODO_FR2: handle different frame setup. */
  if (!tref_isnil(getslot(J, ra))) {  /* Looping back? */
    J->base[ra-1] = J->base[ra];  /* Copy result of ITERC to control var. */
    J->maxslot = ra-1+bc_b(J->pc[-1]);
    J->pc += bc_j(iterins)+1;
    return LOOPEV_ENTER;
  } else {
    J->maxslot = ra-3;
    J->pc++;
    return LOOPEV_LEAVE;
  }
}

/* Record LOOP/JLOOP. Now, that was easy. */
static LoopEvent rec_loop(jit_State *J, BCReg ra)
{
  if (ra < J->maxslot) J->maxslot = ra;
  J->pc++;
  return LOOPEV_ENTER;
}

/* Check if a loop repeatedly failed to trace because it didn't loop back. */
static int innerloopleft(jit_State *J, const BCIns *pc)
{
  ptrdiff_t i;
  for (i = 0; i < PENALTY_SLOTS; i++)
    if (mref(J->penalty[i].pc, const BCIns) == pc) {
      if ((J->penalty[i].reason == LJ_TRERR_LLEAVE ||
           J->penalty[i].reason == LJ_TRERR_LINNER) &&
          J->penalty[i].val >= 2*PENALTY_MIN)
        return 1;
      break;
    }
  return 0;
}

/* Handle the case when an interpreted loop op is hit. */
static void rec_loop_interp(jit_State *J, const BCIns *pc, LoopEvent ev)
{
  if (J->parent == 0 && J->exitno == 0) {
    if (pc == J->startpc && J->framedepth + J->retdepth == 0) {
      /* Same loop? */
      if (ev == LOOPEV_LEAVE)  /* Must loop back to form a root trace. */
        lj_trace_err(J, LJ_TRERR_LLEAVE);
      lj_record_stop(J, LJ_TRLINK_LOOP, J->cur.traceno);  /* Looping trace. */
    } else if (ev != LOOPEV_LEAVE) {  /* Entering inner loop? */
      /* It's usually better to abort here and wait until the inner loop
      ** is traced. But if the inner loop repeatedly didn't loop back,
      ** this indicates a low trip count. In this case try unrolling
      ** an inner loop even in a root trace. But it's better to be a bit
      ** more conservative here and only do it for very short loops.
      */
      if (bc_j(*pc) != -1 && !innerloopleft(J, pc))
        lj_trace_err(J, LJ_TRERR_LINNER);  /* Root trace hit an inner loop. */
      if ((ev != LOOPEV_ENTERLO &&
           J->loopref && J->cur.nins - J->loopref > 24) || --J->loopunroll < 0)
        lj_trace_err(J, LJ_TRERR_LUNROLL);  /* Limit loop unrolling. */
      J->loopref = J->cur.nins;
    }
  } else if (ev != LOOPEV_LEAVE) {  /* Side trace enters an inner loop. */
    J->loopref = J->cur.nins;
    if (--J->loopunroll < 0)
      lj_trace_err(J, LJ_TRERR_LUNROLL);  /* Limit loop unrolling. */
  }  /* Side trace continues across a loop that's left or not entered. */
```

```
}

/* Handle the case when an already compiled loop op is hit. */
static void rec_loop_jit(jit_State *J, TraceNo lnk, LoopEvent ev)
{
  if (J->parent == 0 && J->exitno == 0) {  /* Root trace hit an inner loop. */
    /* Better let the inner loop spawn a side trace back here. */
    lj_trace_err(J, LJ_TRERR_LINNER);
  } else if (ev != LOOPEV_LEAVE) {  /* Side trace enters a compiled loop. */
    J->instunroll = 0;  /* Cannot continue across a compiled loop op. */
    if (J->pc == J->startpc && J->framedepth + J->retdepth == 0)
      lj_record_stop(J, LJ_TRLINK_LOOP, J->cur.traceno);  /* Form extra loop. */
    else
      lj_record_stop(J, LJ_TRLINK_ROOT, lnk);  /* Link to the loop. */
  }  /* Side trace continues across a loop that's left or not entered. */
}

/* -- Record profiler hook checks ---------------------------------------- */

#if LJ_HASPROFILE

/* Need to insert profiler hook check? */
static int rec_profile_need(jit_State *J, GCproto *pt, const BCIns *pc)
{
  GCproto *ppt;
  lua_assert(J->prof_mode == 'f' || J->prof_mode == 'l');
  if (!pt)
    return 0;
  ppt = J->prev_pt;
  J->prev_pt = pt;
  if (pt != ppt && ppt) {
    J->prev_line = -1;
    return 1;
  }
  if (J->prof_mode == 'l') {
    BCLine line = lj_debug_line(pt, proto_bcpos(pt, pc));
    BCLine pline = J->prev_line;
    J->prev_line = line;
    if (pline != line)
      return 1;
  }
  return 0;
}

static void rec_profile_ins(jit_State *J, const BCIns *pc)
{
  if (J->prof_mode && rec_profile_need(J, J->pt, pc)) {
    emitir(IRTG(IR_PROF, IRT_NIL), 0, 0);
    lj_snap_add(J);
  }
}

static void rec_profile_ret(jit_State *J)
{
  if (J->prof_mode == 'f') {
    emitir(IRTG(IR_PROF, IRT_NIL), 0, 0);
    J->prev_pt = NULL;
    lj_snap_add(J);
  }
}

#endif

/* -- Record calls and returns ------------------------------------------- */

/* Specialize to the runtime value of the called function or its prototype. */
static TRef rec_call_specialize(jit_State *J, GCfunc *fn, TRef tr)
{
  TRef kfunc;
  if (isluafunc(fn)) {
    GCproto *pt = funcproto(fn);
    /* Too many closures created? Probably not a monomorphic function. */
    if (pt->flags >= PROTO_CLC_POLY) {  /* Specialize to prototype instead. */
      TRef trpt = emitir(IRT(IR_FLOAD, IRT_P32), tr, IRFL_FUNC_PC);
      emitir(IRTG(IR_EQ, IRT_P32), trpt, lj_ir_kptr(J, proto_bc(pt)));
      (void)lj_ir_kgc(J, obj2gco(pt), IRT_PROTO);  /* Prevent GC of proto. */
```

```
645       return tr;
646     }
647   } else {
648     /* Don't specialize to non-monomorphic builtins. */
649     switch (fn->c.ffid) {
650     case FF_coroutine_wrap_aux:
651     case FF_string_gmatch_aux:
652       /* NYI: io_file_iter doesn't have an ffid, yet. */
653       {  /* Specialize to the ffid. */
654         TRef trid = emitir(IRT(IR_FLOAD, IRT_U8), tr, IRFL_FUNC_FFID);
655         emitir(IRTG(IR_EQ, IRT_INT), trid, lj_ir_kint(J, fn->c.ffid));
656       }
657       return tr;
658     default:
659       /* NYI: don't specialize to non-monomorphic C functions. */
660       break;
661     }
662   }
663   /* Otherwise specialize to the function (closure) value itself. */
664   kfunc = lj_ir_kfunc(J, fn);
665   emitir(IRTG(IR_EQ, IRT_FUNC), tr, kfunc);
666   return kfunc;
667 }
668
669 /* Record call setup. */
670 static void rec_call_setup(jit_State *J, BCReg func, ptrdiff_t nargs)
671 {
672   RecordIndex ix;
673   TValue *functv = &J->L->base[func];
674   TRef *fbase = &J->base[func];
675   ptrdiff_t i;
676   lua_assert(!LJ_FR2);  /* TODO_FR2: handle different frame setup. */
677   for (i = 0; i <= nargs; i++)
678     (void)getslot(J, func+i);  /* Ensure func and all args have a reference. */
679   if (!tref_isfunc(fbase[0])) {  /* Resolve __call metamethod. */
680     ix.tab = fbase[0];
681     copyTV(J->L, &ix.tabv, functv);
682     if (!lj_record_mm_lookup(J, &ix, MM_call) || !tref_isfunc(ix.mobj))
683       lj_trace_err(J, LJ_TRERR_NOMM);
684     for (i = ++nargs; i > 0; i--)  /* Shift arguments up. */
685       fbase[i] = fbase[i-1];
686     fbase[0] = ix.mobj;  /* Replace function. */
687     functv = &ix.mobjv;
688   }
689   fbase[0] = TREF_FRAME | rec_call_specialize(J, funcV(functv), fbase[0]);
690   J->maxslot = (BCReg)nargs;
691 }
692
693 /* Record call. */
694 void lj_record_call(jit_State *J, BCReg func, ptrdiff_t nargs)
695 {
696   rec_call_setup(J, func, nargs);
697   /* Bump frame. */
698   J->framedepth++;
699   J->base += func+1;
700   J->baseslot += func+1;
701 }
702
703 /* Record tail call. */
704 void lj_record_tailcall(jit_State *J, BCReg func, ptrdiff_t nargs)
705 {
706   rec_call_setup(J, func, nargs);
707   if (frame_isvarg(J->L->base - 1)) {
708     BCReg cbase = (BCReg)frame_delta(J->L->base - 1);
709     if (--J->framedepth < 0)
710       lj_trace_err(J, LJ_TRERR_NYIRETL);
711     J->baseslot -= (BCReg)cbase;
712     J->base -= cbase;
713     func += cbase;
714   }
715   /* Move func + args down. */
716   memmove(&J->base[-1], &J->base[func], sizeof(TRef)*(J->maxslot+1));
717   /* Note: the new TREF_FRAME is now at J->base[-1] (even for slot #0). */
718   /* Tailcalls can form a loop, so count towards the loop unroll limit. */
719   if (++J->tailcalled > J->loopunroll)
720     lj_trace_err(J, LJ_TRERR_LUNROLL);
```

```
}

/* Check unroll limits for down-recursion. */
static int check_downrec_unroll(jit_State *J, GCproto *pt)
{
  IRRef ptref;
  for (ptref = J->chain[IR_KGC]; ptref; ptref = IR(ptref)->prev)
    if (ir_kgc(IR(ptref)) == obj2gco(pt)) {
      int count = 0;
      IRRef ref;
      for (ref = J->chain[IR_RETF]; ref; ref = IR(ref)->prev)
        if (IR(ref)->op1 == ptref)
          count++;
      if (count) {
        if (J->pc == J->startpc) {
          if (count + J->tailcalled > J->param[JIT_P_recunroll])
            return 1;
        } else {
          lj_trace_err(J, LJ_TRERR_DOWNREC);
        }
      }
    }
  return 0;
}

static TRef rec_cat(jit_State *J, BCReg baseslot, BCReg topslot);

/* Record return. */
void lj_record_ret(jit_State *J, BCReg rbase, ptrdiff_t gotresults)
{
  TValue *frame = J->L->base - 1;
  ptrdiff_t i;
  for (i = 0; i < gotresults; i++)
    (void)getslot(J, rbase+i);  /* Ensure all results have a reference. */
  while (frame_ispcall(frame)) {  /* Immediately resolve pcall() returns. */
    BCReg cbase = (BCReg)frame_delta(frame);
    if (--J->framedepth < 0)
      lj_trace_err(J, LJ_TRERR_NYIRETL);
    lua_assert(J->baseslot > 1);
    gotresults++;
    rbase += cbase;
    J->baseslot -= (BCReg)cbase;
    J->base -= cbase;
    J->base[--rbase] = TREF_TRUE;  /* Prepend true to results. */
    frame = frame_prevd(frame);
  }
  /* Return to lower frame via interpreter for unhandled cases. */
  if (J->framedepth == 0 && J->pt && bc_isret(bc_op(*J->pc)) &&
      (!frame_islua(frame) ||
       (J->parent == 0 && J->exitno == 0 &&
        !bc_isret(bc_op(J->cur.startins))))) {
    /* NYI: specialize to frame type and return directly, not via RET*. */
    for (i = 0; i < (ptrdiff_t)rbase; i++)
      J->base[i] = 0;  /* Purge dead slots. */
    J->maxslot = rbase + (BCReg)gotresults;
    lj_record_stop(J, LJ_TRLINK_RETURN, 0);  /* Return to interpreter. */
    return;
  }
  if (frame_isvarg(frame)) {
    BCReg cbase = (BCReg)frame_delta(frame);
    if (--J->framedepth < 0)  /* NYI: return of vararg func to lower frame. */
      lj_trace_err(J, LJ_TRERR_NYIRETL);
    lua_assert(J->baseslot > 1);
    rbase += cbase;
    J->baseslot -= (BCReg)cbase;
    J->base -= cbase;
    frame = frame_prevd(frame);
  }
  if (frame_islua(frame)) {  /* Return to Lua frame. */
    BCIns callins = *(frame_pc(frame)-1);
    ptrdiff_t nresults = bc_b(callins) ? (ptrdiff_t)bc_b(callins)-1 :gotresults;
    BCReg cbase = bc_a(callins);
    GCproto *pt = funcproto(frame_func(frame - (cbase+1-LJ_FR2)));
    lua_assert(!LJ_FR2);  /* TODO_FR2: handle different frame teardown. */
    if ((pt->flags & PROTO_NOJIT))
      lj_trace_err(J, LJ_TRERR_CJITOFF);
```

```
    if (J->framedepth == 0 && J->pt && frame == J->L->base - 1) {
      if (check_downrec_unroll(J, pt)) {
        J->maxslot = (BCReg)(rbase + gotresults);
        lj_snap_purge(J);
        lj_record_stop(J, LJ_TRLINK_DOWNREC, J->cur.traceno);  /* Down-rec. */
        return;
      }
      lj_snap_add(J);
    }
    for (i = 0; i < nresults; i++)  /* Adjust results. */
      J->base[i-1] = i < gotresults ? J->base[rbase+i] : TREF_NIL;
    J->maxslot = cbase+(BCReg)nresults;
    if (J->framedepth > 0) {  /* Return to a frame that is part of the trace. */
      J->framedepth--;
      lua_assert(J->baseslot > cbase+1);
      J->baseslot -= cbase+1;
      J->base -= cbase+1;
    } else if (J->parent == 0 && J->exitno == 0 &&
               !bc_isret(bc_op(J->cur.startins))) {
      /* Return to lower frame would leave the loop in a root trace. */
      lj_trace_err(J, LJ_TRERR_LLEAVE);
    } else if (J->needsnap) {  /* Tailcalled to ff with side-effects. */
      lj_trace_err(J, LJ_TRERR_NYIRETL);  /* No way to insert snapshot here. */
    } else {  /* Return to lower frame. Guard for the target we return to. */
      TRef trpt = lj_ir_kgc(J, obj2gco(pt), IRT_PROTO);
      TRef trpc = lj_ir_kptr(J, (void *)frame_pc(frame));
      emitir(IRTG(IR_RETF, IRT_P32), trpt, trpc);
      J->retdepth++;
      J->needsnap = 1;
      lua_assert(J->baseslot == 1);
      /* Shift result slots up and clear the slots of the new frame below. */
      memmove(J->base + cbase, J->base-1, sizeof(TRef)*nresults);
      memset(J->base-1, 0, sizeof(TRef)*(cbase+1));
    }
  } else if (frame_iscont(frame)) {  /* Return to continuation frame. */
    ASMFunction cont = frame_contf(frame);
    BCReg cbase = (BCReg)frame_delta(frame);
    if ((J->framedepth -= 2) < 0)
      lj_trace_err(J, LJ_TRERR_NYIRETL);
    J->baseslot -= (BCReg)cbase;
    J->base -= cbase;
    J->maxslot = cbase-2;
    if (cont == lj_cont_ra) {
      /* Copy result to destination slot. */
      BCReg dst = bc_a(*(frame_contpc(frame)-1));
      J->base[dst] = gotresults ? J->base[cbase+rbase] : TREF_NIL;
      if (dst >= J->maxslot) J->maxslot = dst+1;
    } else if (cont == lj_cont_nop) {
      /* Nothing to do here. */
    } else if (cont == lj_cont_cat) {
      BCReg bslot = bc_b(*(frame_contpc(frame)-1));
      TRef tr = gotresults ? J->base[cbase+rbase] : TREF_NIL;
      if (bslot != cbase-2) {  /* Concatenate the remainder. */
        TValue *b = J->L->base, save;  /* Simulate lower frame and result. */
        J->base[cbase-2] = tr;
        copyTV(J->L, &save, b-2);
        if (gotresults) copyTV(J->L, b-2, b+rbase); else setnilV(b-2);
        J->L->base = b - cbase;
        tr = rec_cat(J, bslot, cbase-2);
        b = J->L->base + cbase;  /* Undo. */
        J->L->base = b;
        copyTV(J->L, b-2, &save);
      }
      if (tr) {  /* Store final result. */
        BCReg dst = bc_a(*(frame_contpc(frame)-1));
        J->base[dst] = tr;
        if (dst >= J->maxslot) J->maxslot = dst+1;
      }  /* Otherwise continue with another __concat call. */
    } else {
      /* Result type already specialized. */
      lua_assert(cont == lj_cont_condf || cont == lj_cont_condt);
    }
  } else {
    lj_trace_err(J, LJ_TRERR_NYIRETL);  /* NYI: handle return to C frame. */
  }
  lua_assert(J->baseslot >= 1);
```

```
}

/* -- Metamethod handling ------------------------------------------------ */

/* Prepare to record call to metamethod. */
static BCReg rec_mm_prep(jit_State *J, ASMFunction cont)
{
  BCReg s, top = cont == lj_cont_cat ? J->maxslot : curr_proto(J->L)->framesize;
#if LJ_64
  TRef trcont = lj_ir_kptr(J, (void *)((int64_t)cont-(int64_t)lj_vm_asm_begin));
#else
  TRef trcont = lj_ir_kptr(J, (void *)cont);
#endif
  J->base[top] = trcont | TREF_CONT;
  J->framedepth++;
  for (s = J->maxslot; s < top; s++)
    J->base[s] = 0;  /* Clear frame gap to avoid resurrecting previous refs. */
  return top+1;
}

/* Record metamethod lookup. */
int lj_record_mm_lookup(jit_State *J, RecordIndex *ix, MMS mm)
{
  RecordIndex mix;
  GCtab *mt;
  if (tref_istab(ix->tab)) {
    mt = tabref(tabV(&ix->tabv)->metatable);
    mix.tab = emitir(IRT(IR_FLOAD, IRT_TAB), ix->tab, IRFL_TAB_META);
  } else if (tref_isudata(ix->tab)) {
    int udtype = udataV(&ix->tabv)->udtype;
    mt = tabref(udataV(&ix->tabv)->metatable);
    /* The metatables of special userdata objects are treated as immutable. */
    if (udtype != UDTYPE_USERDATA) {
      cTValue *mo;
      if (LJ_HASFFI && udtype == UDTYPE_FFI_CLIB) {
        /* Specialize to the C library namespace object. */
        emitir(IRTG(IR_EQ, IRT_P32), ix->tab, lj_ir_kptr(J, udataV(&ix->tabv)));
      } else {
        /* Specialize to the type of userdata. */
        TRef tr = emitir(IRT(IR_FLOAD, IRT_U8), ix->tab, IRFL_UDATA_UDTYPE);
        emitir(IRTGI(IR_EQ), tr, lj_ir_kint(J, udtype));
      }
  immutable_mt:
      mo = lj_tab_getstr(mt, mmname_str(J2G(J), mm));
      if (!mo || tvisnil(mo))
        return 0;  /* No metamethod. */
      /* Treat metamethod or index table as immutable, too. */
      if (!(tvisfunc(mo) || tvistab(mo)))
        lj_trace_err(J, LJ_TRERR_BADTYPE);
      copyTV(J->L, &ix->mobjv, mo);
      ix->mobj = lj_ir_kgc(J, gcV(mo), tvisfunc(mo) ? IRT_FUNC : IRT_TAB);
      ix->mtv = mt;
      ix->mt = TREF_NIL;  /* Dummy value for comparison semantics. */
      return 1;  /* Got metamethod or index table. */
    }
    mix.tab = emitir(IRT(IR_FLOAD, IRT_TAB), ix->tab, IRFL_UDATA_META);
  } else {
    /* Specialize to base metatable. Must flush mcode in lua_setmetatable(). */
    mt = tabref(basemt_obj(J2G(J), &ix->tabv));
    if (mt == NULL) {
      ix->mt = TREF_NIL;
      return 0;  /* No metamethod. */
    }
    /* The cdata metatable is treated as immutable. */
    if (LJ_HASFFI && tref_iscdata(ix->tab)) goto immutable_mt;
    ix->mt = mix.tab = lj_ir_ktab(J, mt);
    goto nocheck;
  }
  ix->mt = mt ? mix.tab : TREF_NIL;
  emitir(IRTG(mt ? IR_NE : IR_EQ, IRT_TAB), mix.tab, lj_ir_knull(J, IRT_TAB));
nocheck:
  if (mt) {
    GCstr *mmstr = mmname_str(J2G(J), mm);
    cTValue *mo = lj_tab_getstr(mt, mmstr);
    if (mo && !tvisnil(mo))
      copyTV(J->L, &ix->mobjv, mo);
```

```
    ix->mtv = mt;
    settabV(J->L, &mix.tabv, mt);
    setstrV(J->L, &mix.keyv, mmstr);
    mix.key = lj_ir_kstr(J, mmstr);
    mix.val = 0;
    mix.idxchain = 0;
    ix->mobj = lj_record_idx(J, &mix);
    return !tref_isnil(ix->mobj);  /* 1 if metamethod found, 0 if not. */
  }
  return 0;  /* No metamethod. */
}

/* Record call to arithmetic metamethod. */
static TRef rec_mm_arith(jit_State *J, RecordIndex *ix, MMS mm)
{
  /* Set up metamethod call first to save ix->tab and ix->tabv. */
  BCReg func = rec_mm_prep(J, mm == MM_concat ? lj_cont_cat : lj_cont_ra);
  TRef *base = J->base + func;
  TValue *basev = J->L->base + func;
  base[1] = ix->tab; base[2] = ix->key;
  copyTV(J->L, basev+1, &ix->tabv);
  copyTV(J->L, basev+2, &ix->keyv);
  if (!lj_record_mm_lookup(J, ix, mm)) {  /* Lookup mm on 1st operand. */
    if (mm != MM_unm) {
      ix->tab = ix->key;
      copyTV(J->L, &ix->tabv, &ix->keyv);
      if (lj_record_mm_lookup(J, ix, mm))  /* Lookup mm on 2nd operand. */
        goto ok;
    }
    lj_trace_err(J, LJ_TRERR_NOMM);
  }
ok:
  lua_assert(!LJ_FR2);  /* TODO_FR2: handle different frame setup. */
  base[0] = ix->mobj;
  copyTV(J->L, basev+0, &ix->mobjv);
  lj_record_call(J, func, 2);
  return 0;  /* No result yet. */
}

/* Record call to __len metamethod. */
static TRef rec_mm_len(jit_State *J, TRef tr, TValue *tv)
{
  RecordIndex ix;
  ix.tab = tr;
  copyTV(J->L, &ix.tabv, tv);
  if (lj_record_mm_lookup(J, &ix, MM_len)) {
    BCReg func = rec_mm_prep(J, lj_cont_ra);
    TRef *base = J->base + func;
    TValue *basev = J->L->base + func;
    lua_assert(!LJ_FR2);  /* TODO_FR2: handle different frame setup. */
    base[0] = ix.mobj; copyTV(J->L, basev+0, &ix.mobjv);
     base[1] = tr; copyTV(J->L, basev+1, tv);
#if LJ_52
    base[2] = tr; copyTV(J->L, basev+2, tv);
#else
    base[2] = TREF_NIL; setnilV(basev+2);
#endif
    lj_record_call(J, func, 2);
  } else {
    if (LJ_52 && tref_istab(tr))
      return lj_ir_call(J, IRCALL_lj_tab_len, tr);
    lj_trace_err(J, LJ_TRERR_NOMM);
  }
  return 0;  /* No result yet. */
}

/* Call a comparison metamethod. */
static void rec_mm_callcomp(jit_State *J, RecordIndex *ix, int op)
{
  BCReg func = rec_mm_prep(J, (op&1) ? lj_cont_condf : lj_cont_condt);
  TRef *base = J->base + func;
  TValue *tv = J->L->base + func;
  lua_assert(!LJ_FR2);  /* TODO_FR2: handle different frame setup. */
  base[0] = ix->mobj; base[1] = ix->val; base[2] = ix->key;
  copyTV(J->L, tv+0, &ix->mobjv);
  copyTV(J->L, tv+1, &ix->valv);
```

```
  copyTV(J->L, tv+2, &ix->keyv);
  lj_record_call(J, func, 2);
}

/* Record call to equality comparison metamethod (for tab and udata only). */
static void rec_mm_equal(jit_State *J, RecordIndex *ix, int op)
{
  ix->tab = ix->val;
  copyTV(J->L, &ix->tabv, &ix->valv);
  if (lj_record_mm_lookup(J, ix, MM_eq)) {  /* Lookup mm on 1st operand. */
    cTValue *bv;
    TRef mo1 = ix->mobj;
    TValue mo1v;
    copyTV(J->L, &mo1v, &ix->mobjv);
    /* Avoid the 2nd lookup and the objcmp if the metatables are equal. */
    bv = &ix->keyv;
    if (tvistab(bv) && tabref(tabV(bv)->metatable) == ix->mtv) {
      TRef mt2 = emitir(IRT(IR_FLOAD, IRT_TAB), ix->key, IRFL_TAB_META);
      emitir(IRTG(IR_EQ, IRT_TAB), mt2, ix->mt);
    } else if (tvisudata(bv) && tabref(udataV(bv)->metatable) == ix->mtv) {
      TRef mt2 = emitir(IRT(IR_FLOAD, IRT_TAB), ix->key, IRFL_UDATA_META);
      emitir(IRTG(IR_EQ, IRT_TAB), mt2, ix->mt);
    } else {  /* Lookup metamethod on 2nd operand and compare both. */
      ix->tab = ix->key;
      copyTV(J->L, &ix->tabv, bv);
      if (!lj_record_mm_lookup(J, ix, MM_eq) ||
          lj_record_objcmp(J, mo1, ix->mobj, &mo1v, &ix->mobjv))
        return;
    }
    rec_mm_callcomp(J, ix, op);
  }
}

/* Record call to ordered comparison metamethods (for arbitrary objects). */
static void rec_mm_comp(jit_State *J, RecordIndex *ix, int op)
{
  ix->tab = ix->val;
  copyTV(J->L, &ix->tabv, &ix->valv);
  while (1) {
    MMS mm = (op & 2) ? MM_le : MM_lt;  /* Try __le + __lt or only __lt. */
#if LJ_52
    if (!lj_record_mm_lookup(J, ix, mm)) {  /* Lookup mm on 1st operand. */
      ix->tab = ix->key;
      copyTV(J->L, &ix->tabv, &ix->keyv);
      if (!lj_record_mm_lookup(J, ix, mm))  /* Lookup mm on 2nd operand. */
        goto nomatch;
    }
    rec_mm_callcomp(J, ix, op);
    return;
#else
    if (lj_record_mm_lookup(J, ix, mm)) {  /* Lookup mm on 1st operand. */
      cTValue *bv;
      TRef mo1 = ix->mobj;
      TValue mo1v;
      copyTV(J->L, &mo1v, &ix->mobjv);
      /* Avoid the 2nd lookup and the objcmp if the metatables are equal. */
      bv = &ix->keyv;
      if (tvistab(bv) && tabref(tabV(bv)->metatable) == ix->mtv) {
        TRef mt2 = emitir(IRT(IR_FLOAD, IRT_TAB), ix->key, IRFL_TAB_META);
        emitir(IRTG(IR_EQ, IRT_TAB), mt2, ix->mt);
      } else if (tvisudata(bv) && tabref(udataV(bv)->metatable) == ix->mtv) {
        TRef mt2 = emitir(IRT(IR_FLOAD, IRT_TAB), ix->key, IRFL_UDATA_META);
        emitir(IRTG(IR_EQ, IRT_TAB), mt2, ix->mt);
      } else {  /* Lookup metamethod on 2nd operand and compare both. */
        ix->tab = ix->key;
        copyTV(J->L, &ix->tabv, bv);
        if (!lj_record_mm_lookup(J, ix, mm) ||
            lj_record_objcmp(J, mo1, ix->mobj, &mo1v, &ix->mobjv))
          goto nomatch;
      }
      rec_mm_callcomp(J, ix, op);
      return;
    }
#endif
  nomatch:
    /* Lookup failed. Retry with  __lt and swapped operands. */
```

```
    if (!(op & 2)) break;  /* Already at __lt. Interpreter will throw. */
    ix->tab = ix->key; ix->key = ix->val; ix->val = ix->tab;
    copyTV(J->L, &ix->tabv, &ix->keyv);
    copyTV(J->L, &ix->keyv, &ix->valv);
    copyTV(J->L, &ix->valv, &ix->tabv);
    op ^= 3;
  }
}

#if LJ_HASFFI
/* Setup call to cdata comparison metamethod. */
static void rec_mm_comp_cdata(jit_State *J, RecordIndex *ix, int op, MMS mm)
{
  lj_snap_add(J);
  if (tref_iscdata(ix->val)) {
    ix->tab = ix->val;
    copyTV(J->L, &ix->tabv, &ix->valv);
  } else {
    lua_assert(tref_iscdata(ix->key));
    ix->tab = ix->key;
    copyTV(J->L, &ix->tabv, &ix->keyv);
  }
  lj_record_mm_lookup(J, ix, mm);
  rec_mm_callcomp(J, ix, op);
}
#endif

/* -- Indexed access -------------------------------------------------- */

/* Record bounds-check. */
static void rec_idx_abc(jit_State *J, TRef asizeref, TRef ikey, uint32_t asize)
{
  /* Try to emit invariant bounds checks. */
  if ((J->flags & (JIT_F_OPT_LOOP|JIT_F_OPT_ABC)) ==
      (JIT_F_OPT_LOOP|JIT_F_OPT_ABC)) {
    IRRef ref = tref_ref(ikey);
    IRIns *ir = IR(ref);
    int32_t ofs = 0;
    IRRef ofsref = 0;
    /* Handle constant offsets. */
    if (ir->o == IR_ADD && irref_isk(ir->op2)) {
      ofsref = ir->op2;
      ofs = IR(ofsref)->i;
      ref = ir->op1;
      ir = IR(ref);
    }
    /* Got scalar evolution analysis results for this reference? */
    if (ref == J->scev.idx) {
      int32_t stop;
      lua_assert(irt_isint(J->scev.t) && ir->o == IR_SLOAD);
      stop = numberVint(&(J->L->base - J->baseslot)[ir->op1 + FORL_STOP]);
      /* Runtime value for stop of loop is within bounds? */
      if ((uint64_t)stop + ofs < (uint64_t)asize) {
        /* Emit invariant bounds check for stop. */
        emitir(IRTG(IR_ABC, IRT_P32), asizeref, ofs == 0 ? J->scev.stop :
               emitir(IRTI(IR_ADD), J->scev.stop, ofsref));
        /* Emit invariant bounds check for start, if not const or negative. */
        if (!(J->scev.dir && J->scev.start &&
              (int64_t)IR(J->scev.start)->i + ofs >= 0))
          emitir(IRTG(IR_ABC, IRT_P32), asizeref, ikey);
        return;
      }
    }
  }
  emitir(IRTGI(IR_ABC), asizeref, ikey);  /* Emit regular bounds check. */
}

/* Record indexed key lookup. */
static TRef rec_idx_key(jit_State *J, RecordIndex *ix, IRRef *rbref)
{
  TRef key;
  GCtab *t = tabV(&ix->tabv);
  ix->oldv = lj_tab_get(J->L, t, &ix->keyv);  /* Lookup previous value. */
  *rbref = 0;

  /* Integer keys are looked up in the array part first. */
```

```
  key = ix->key;
  if (tref_isnumber(key)) {
    int32_t k = numberVint(&ix->keyv);
    if (!tvisint(&ix->keyv) && numV(&ix->keyv) != (lua_Number)k)
      k = LJ_MAX_ASIZE;
    if ((MSize)k < LJ_MAX_ASIZE) {  /* Potential array key? */
      TRef ikey = lj_opt_narrow_index(J, key);
      TRef asizeref = emitir(IRTI(IR_FLOAD), ix->tab, IRFL_TAB_ASIZE);
      if ((MSize)k < t->asize) {  /* Currently an array key? */
        TRef arrayref;
        rec_idx_abc(J, asizeref, ikey, t->asize);
        arrayref = emitir(IRT(IR_FLOAD, IRT_P32), ix->tab, IRFL_TAB_ARRAY);
        return emitir(IRT(IR_AREF, IRT_P32), arrayref, ikey);
      } else {  /* Currently not in array (may be an array extension)? */
        emitir(IRTGI(IR_ULE), asizeref, ikey);  /* Inv. bounds check. */
        if (k == 0 && tref_isk(key))
          key = lj_ir_knum_zero(J);  /* Canonicalize 0 or +-0.0 to +0.0. */
        /* And continue with the hash lookup. */
      }
    } else if (!tref_isk(key)) {
      /* We can rule out const numbers which failed the integerness test
      ** above. But all other numbers are potential array keys.
      */
      if (t->asize == 0) {  /* True sparse tables have an empty array part. */
        /* Guard that the array part stays empty. */
        TRef tmp = emitir(IRTI(IR_FLOAD), ix->tab, IRFL_TAB_ASIZE);
        emitir(IRTGI(IR_EQ), tmp, lj_ir_kint(J, 0));
      } else {
        lj_trace_err(J, LJ_TRERR_NYITMIX);
      }
    }
  }

  /* Otherwise the key is located in the hash part. */
  if (t->hmask == 0) {  /* Shortcut for empty hash part. */
    /* Guard that the hash part stays empty. */
    TRef tmp = emitir(IRTI(IR_FLOAD), ix->tab, IRFL_TAB_HMASK);
    emitir(IRTGI(IR_EQ), tmp, lj_ir_kint(J, 0));
    return lj_ir_kkptr(J, niltvg(J2G(J)));
  }
  if (tref_isinteger(key))  /* Hash keys are based on numbers, not ints. */
    key = emitir(IRTN(IR_CONV), key, IRCONV_NUM_INT);
  if (tref_isk(key)) {
    /* Optimize lookup of constant hash keys. */
    MSize hslot = (MSize)((char *)ix->oldv - (char *)&noderef(t->node)[0].val);
    if (t->hmask > 0 && hslot <= t->hmask*(MSize)sizeof(Node) &&
        hslot <= 65535*(MSize)sizeof(Node)) {
      TRef node, kslot, hm;
      *rbref = J->cur.nins;  /* Mark possible rollback point. */
      hm = emitir(IRTI(IR_FLOAD), ix->tab, IRFL_TAB_HMASK);
      emitir(IRTGI(IR_EQ), hm, lj_ir_kint(J, (int32_t)t->hmask));
      node = emitir(IRT(IR_FLOAD, IRT_P32), ix->tab, IRFL_TAB_NODE);
      kslot = lj_ir_kslot(J, key, hslot / sizeof(Node));
      return emitir(IRTG(IR_HREFK, IRT_P32), node, kslot);
    }
  }
  /* Fall back to a regular hash lookup. */
  return emitir(IRT(IR_HREF, IRT_P32), ix->tab, key);
}

/* Determine whether a key is NOT one of the fast metamethod names. */
static int nommstr(jit_State *J, TRef key)
{
  if (tref_isstr(key)) {
    if (tref_isk(key)) {
      GCstr *str = ir_kstr(IR(tref_ref(key)));
      uint32_t mm;
      for (mm = 0; mm <= MM_FAST; mm++)
        if (mmname_str(J2G(J), mm) == str)
          return 0;  /* MUST be one the fast metamethod names. */
    } else {
      return 0;  /* Variable string key MAY be a metamethod name. */
    }
  }
  return 1;  /* CANNOT be a metamethod name. */
}
```

```
1253
1254 /* Record indexed load/store. */
1255 TRef lj_record_idx(jit_State *J, RecordIndex *ix)
1256 {
1257   TRef xref;
1258   IROp xrefop, loadop;
1259   IRRef rbref;
1260   cTValue *oldv;
1261
1262   while (!tref_istab(ix->tab)) { /* Handle non-table lookup. */
1263     /* Never call raw lj_record_idx() on non-table. */
1264     lua_assert(ix->idxchain != 0);
1265     if (!lj_record_mm_lookup(J, ix, ix->val ? MM_newindex : MM_index))
1266       lj_trace_err(J, LJ_TRERR_NOMM);
1267   handlemm:
1268     if (tref_isfunc(ix->mobj)) {  /* Handle metamethod call. */
1269       BCReg func = rec_mm_prep(J, ix->val ? lj_cont_nop : lj_cont_ra);
1270       TRef *base = J->base + func;
1271       TValue *tv = J->L->base + func;
1272       lua_assert(!LJ_FR2);  /* TODO_FR2: handle different frame setup. */
1273       base[0] = ix->mobj; base[1] = ix->tab; base[2] = ix->key;
1274       setfuncV(J->L, tv+0, funcV(&ix->mobjv));
1275       copyTV(J->L, tv+1, &ix->tabv);
1276       copyTV(J->L, tv+2, &ix->keyv);
1277       if (ix->val) {
1278         base[3] = ix->val;
1279         copyTV(J->L, tv+3, &ix->valv);
1280         lj_record_call(J, func, 3);  /* mobj(tab, key, val) */
1281         return 0;
1282       } else {
1283         lj_record_call(J, func, 2);  /* res = mobj(tab, key) */
1284         return 0;  /* No result yet. */
1285       }
1286     }
1287     /* Otherwise retry lookup with metaobject. */
1288     ix->tab = ix->mobj;
1289     copyTV(J->L, &ix->tabv, &ix->mobjv);
1290     if (--ix->idxchain == 0)
1291       lj_trace_err(J, LJ_TRERR_IDXLOOP);
1292   }
1293
1294   /* First catch nil and NaN keys for tables. */
1295   if (tvisnil(&ix->keyv) || (tvisnum(&ix->keyv) && tvisnan(&ix->keyv))) {
1296     if (ix->val)  /* Better fail early. */
1297       lj_trace_err(J, LJ_TRERR_STORENN);
1298     if (tref_isk(ix->key)) {
1299       if (ix->idxchain && lj_record_mm_lookup(J, ix, MM_index))
1300         goto handlemm;
1301       return TREF_NIL;
1302     }
1303   }
1304
1305   /* Record the key lookup. */
1306   xref = rec_idx_key(J, ix, &rbref);
1307   xrefop = IR(tref_ref(xref))->o;
1308   loadop = xrefop == IR_AREF ? IR_ALOAD : IR_HLOAD;
1309   /* The lj_meta_tset() inconsistency is gone, but better play safe. */
1310   oldv = xrefop == IR_KKPTR ? (cTValue *)ir_kptr(IR(tref_ref(xref))) : ix->oldv;
1311
1312   if (ix->val == 0) {  /* Indexed load */
1313     IRType t = itype2irt(oldv);
1314     TRef res;
1315     if (oldv == niltvg(J2G(J))) {
1316       emitir(IRTG(IR_EQ, IRT_P32), xref, lj_ir_kkptr(J, niltvg(J2G(J))));
1317       res = TREF_NIL;
1318     } else {
1319       res = emitir(IRTG(loadop, t), xref, 0);
1320     }
1321     if (tref_ref(res) < rbref)  /* HREFK + load forwarded? */
1322       lj_ir_rollback(J, rbref);  /* Rollback to eliminate hmask guard. */
1323     if (t == IRT_NIL && ix->idxchain && lj_record_mm_lookup(J, ix, MM_index))
1324       goto handlemm;
1325     if (irtype_ispri(t)) res = TREF_PRI(t);  /* Canonicalize primitives. */
1326     return res;
1327   } else {  /* Indexed store. */
1328     GCtab *mt = tabref(tabV(&ix->tabv)->metatable);
```

```
    int keybarrier = tref_isgcv(ix->key) && !tref_isnil(ix->val);
    if (tref_ref(xref) < rbref)  /* HREFK forwarded? */
      lj_ir_rollback(J, rbref);  /* Rollback to eliminate hmask guard. */
    if (tvisnil(oldv)) {  /* Previous value was nil? */
      /* Need to duplicate the hasmm check for the early guards. */
      int hasmm = 0;
      if (ix->idxchain && mt) {
        cTValue *mo = lj_tab_getstr(mt, mmname_str(J2G(J), MM_newindex));
        hasmm = mo && !tvisnil(mo);
      }
      if (hasmm)
        emitir(IRTG(loadop, IRT_NIL), xref, 0);  /* Guard for nil value. */
      else if (xrefop == IR_HREF)
        emitir(IRTG(oldv == niltvg(J2G(J)) ? IR_EQ : IR_NE, IRT_P32),
               xref, lj_ir_kkptr(J, niltvg(J2G(J))));
      if (ix->idxchain && lj_record_mm_lookup(J, ix, MM_newindex)) {
        lua_assert(hasmm);
        goto handlemm;
      }
      lua_assert(!hasmm);
      if (oldv == niltvg(J2G(J))) {  /* Need to insert a new key. */
        TRef key = ix->key;
        if (tref_isinteger(key))  /* NEWREF needs a TValue as a key. */
          key = emitir(IRTN(IR_CONV), key, IRCONV_NUM_INT);
        xref = emitir(IRT(IR_NEWREF, IRT_P32), ix->tab, key);
        keybarrier = 0;  /* NEWREF already takes care of the key barrier. */
      }
    } else if (!lj_opt_fwd_wasnonnil(J, loadop, tref_ref(xref))) {
      /* Cannot derive that the previous value was non-nil, must do checks. */
      if (xrefop == IR_HREF)  /* Guard against store to niltv. */
        emitir(IRTG(IR_NE, IRT_P32), xref, lj_ir_kkptr(J, niltvg(J2G(J))));
      if (ix->idxchain) {  /* Metamethod lookup required? */
        /* A check for NULL metatable is cheaper (hoistable) than a load. */
        if (!mt) {
          TRef mtref = emitir(IRT(IR_FLOAD, IRT_TAB), ix->tab, IRFL_TAB_META);
          emitir(IRTG(IR_EQ, IRT_TAB), mtref, lj_ir_knull(J, IRT_TAB));
        } else {
          IRType t = itype2irt(oldv);
          emitir(IRTG(loadop, t), xref, 0);  /* Guard for non-nil value. */
        }
      }
    } else {
      keybarrier = 0;  /* Previous non-nil value kept the key alive. */
    }
    /* Convert int to number before storing. */
    if (!LJ_DUALNUM && tref_isinteger(ix->val))
      ix->val = emitir(IRTN(IR_CONV), ix->val, IRCONV_NUM_INT);
    emitir(IRT(loadop+IRDELTA_L2S, tref_type(ix->val)), xref, ix->val);
    if (keybarrier || tref_isgcv(ix->val))
      emitir(IRT(IR_TBAR, IRT_NIL), ix->tab, 0);
    /* Invalidate neg. metamethod cache for stores with certain string keys. */
    if (!nommstr(J, ix->key)) {
      TRef fref = emitir(IRT(IR_FREF, IRT_P32), ix->tab, IRFL_TAB_NOMM);
      emitir(IRT(IR_FSTORE, IRT_U8), fref, lj_ir_kint(J, 0));
    }
    J->needsnap = 1;
    return 0;
  }
}

static void rec_tsetm(jit_State *J, BCReg ra, BCReg rn, int32_t i)
{
  RecordIndex ix;
  cTValue *basev = J->L->base;
  copyTV(J->L, &ix.tabv, &basev[ra-1]);
  ix.tab = getslot(J, ra-1);
  ix.idxchain = 0;
  for (; ra < rn; i++, ra++) {
    setintV(&ix.keyv, i);
    ix.key = lj_ir_kint(J, i);
    copyTV(J->L, &ix.valv, &basev[ra]);
    ix.val = getslot(J, ra);
    lj_record_idx(J, &ix);
  }
}
```

```
/* -- Upvalue access ------------------------------------------------------ */

/* Check whether upvalue is immutable and ok to constify. */
static int rec_upvalue_constify(jit_State *J, GCupval *uvp)
{
  if (uvp->immutable) {
    cTValue *o = uvval(uvp);
    /* Don't constify objects that may retain large amounts of memory. */
#if LJ_HASFFI
    if (tviscdata(o)) {
      GCcdata *cd = cdataV(o);
      if (!cdataisv(cd) && !(cd->marked & LJ_GC_CDATA_FIN)) {
        CType *ct = ctype_raw(ctype_ctsG(J2G(J)), cd->ctypeid);
        if (!ctype_hassize(ct->info) || ct->size <= 16)
          return 1;
      }
      return 0;
    }
#else
    UNUSED(J);
#endif
    if (!(tvistab(o) || tvisudata(o) || tvisthread(o)))
      return 1;
  }
  return 0;
}

/* Record upvalue load/store. */
static TRef rec_upvalue(jit_State *J, uint32_t uv, TRef val)
{
  GCupval *uvp = &gcref(J->fn->l.uvptr[uv])->uv;
  TRef fn = getcurrf(J);
  IRRef uref;
  int needbarrier = 0;
  if (rec_upvalue_constify(J, uvp)) {  /* Try to constify immutable upvalue. */
    TRef tr, kfunc;
    lua_assert(val == 0);
    if (!tref_isk(fn)) {  /* Late specialization of current function. */
      if (J->pt->flags >= PROTO_CLC_POLY)
        goto noconstify;
      kfunc = lj_ir_kfunc(J, J->fn);
      emitir(IRTG(IR_EQ, IRT_FUNC), fn, kfunc);
      J->base[-1] = TREF_FRAME | kfunc;
      fn = kfunc;
    }
    tr = lj_record_constify(J, uvval(uvp));
    if (tr)
      return tr;
  }
noconstify:
  /* Note: this effectively limits LJ_MAX_UPVAL to 127. */
  uv = (uv << 8) | (hashrot(uvp->dhash, uvp->dhash + HASH_BIAS) & 0xff);
  if (!uvp->closed) {
    /* In current stack? */
    if (uvval(uvp) >= tvref(J->L->stack) &&
        uvval(uvp) < tvref(J->L->maxstack)) {
      int32_t slot = (int32_t)(uvval(uvp) - (J->L->base - J->baseslot));
      if (slot >= 0) {  /* Aliases an SSA slot? */
        slot -= (int32_t)J->baseslot;  /* Note: slot number may be negative! */
        /* NYI: add IR to guard that it's still aliasing the same slot. */
        if (val == 0) {
          return getslot(J, slot);
        } else {
          J->base[slot] = val;
          if (slot >= (int32_t)J->maxslot) J->maxslot = (BCReg)(slot+1);
          return 0;
        }
      }
    }
    uref = tref_ref(emitir(IRTG(IR_UREFO, IRT_P32), fn, uv));
  } else {
    needbarrier = 1;
    uref = tref_ref(emitir(IRTG(IR_UREFC, IRT_P32), fn, uv));
  }
  if (val == 0) {  /* Upvalue load */
    IRType t = itype2irt(uvval(uvp));
```

```
    TRef res = emitir(IRTG(IR_ULOAD, t), uref, 0);
    if (irtype_ispri(t)) res = TREF_PRI(t);  /* Canonicalize primitive refs. */
    return res;
  } else {  /* Upvalue store. */
    /* Convert int to number before storing. */
    if (!LJ_DUALNUM && tref_isinteger(val))
      val = emitir(IRTN(IR_CONV), val, IRCONV_NUM_INT);
    emitir(IRT(IR_USTORE, tref_type(val)), uref, val);
    if (needbarrier && tref_isgcv(val))
      emitir(IRT(IR_OBAR, IRT_NIL), uref, val);
    J->needsnap = 1;
    return 0;
  }
}

/* -- Record calls to Lua functions --------------------------------------- */

/* Check unroll limits for calls. */
static void check_call_unroll(jit_State *J, TraceNo lnk)
{
  cTValue *frame = J->L->base - 1;
  void *pc = mref(frame_func(frame)->l.pc, void);
  int32_t depth = J->framedepth;
  int32_t count = 0;
  if ((J->pt->flags & PROTO_VARARG)) depth--;  /* Vararg frame still missing. */
  for (; depth > 0; depth--) {  /* Count frames with same prototype. */
    if (frame_iscont(frame)) depth--;
    frame = frame_prev(frame);
    if (mref(frame_func(frame)->l.pc, void) == pc)
      count++;
  }
  if (J->pc == J->startpc) {
    if (count + J->tailcalled > J->param[JIT_P_recunroll]) {
      J->pc++;
      if (J->framedepth + J->retdepth == 0)
        lj_record_stop(J, LJ_TRLINK_TAILREC, J->cur.traceno);  /* Tail-rec. */
      else
        lj_record_stop(J, LJ_TRLINK_UPREC, J->cur.traceno);  /* Up-recursion. */
    }
  } else {
    if (count > J->param[JIT_P_callunroll]) {
      if (lnk) {  /* Possible tail- or up-recursion. */
        lj_trace_flush(J, lnk);  /* Flush trace that only returns. */
        /* Set a small, pseudo-random hotcount for a quick retry of JFUNC*. */
        hotcount_set(J2GG(J), J->pc+1, LJ_PRNG_BITS(J, 4));
      }
      lj_trace_err(J, LJ_TRERR_CUNROLL);
    }
  }
}

/* Record Lua function setup. */
static void rec_func_setup(jit_State *J)
{
  GCproto *pt = J->pt;
  BCReg s, numparams = pt->numparams;
  if ((pt->flags & PROTO_NOJIT))
    lj_trace_err(J, LJ_TRERR_CJITOFF);
  if (J->baseslot + pt->framesize >= LJ_MAX_JSLOTS)
    lj_trace_err(J, LJ_TRERR_STACKOV);
  /* Fill up missing parameters with nil. */
  for (s = J->maxslot; s < numparams; s++)
    J->base[s] = TREF_NIL;
  /* The remaining slots should never be read before they are written. */
  J->maxslot = numparams;
}

/* Record Lua vararg function setup. */
static void rec_func_vararg(jit_State *J)
{
  GCproto *pt = J->pt;
  BCReg s, fixargs, vframe = J->maxslot+1;
  lua_assert((pt->flags & PROTO_VARARG));
  if (J->baseslot + vframe + pt->framesize >= LJ_MAX_JSLOTS)
    lj_trace_err(J, LJ_TRERR_STACKOV);
  J->base[vframe-1] = J->base[-1];  /* Copy function up. */
```

```
  /* Copy fixarg slots up and set their original slots to nil. */
  fixargs = pt->numparams < J->maxslot ? pt->numparams : J->maxslot;
  for (s = 0; s < fixargs; s++) {
    J->base[vframe+s] = J->base[s];
    J->base[s] = TREF_NIL;
  }
  J->maxslot = fixargs;
  J->framedepth++;
  J->base += vframe;
  J->baseslot += vframe;
}

/* Record entry to a Lua function. */
static void rec_func_lua(jit_State *J)
{
  rec_func_setup(J);
  check_call_unroll(J, 0);
}

/* Record entry to an already compiled function. */
static void rec_func_jit(jit_State *J, TraceNo lnk)
{
  GCtrace *T;
  rec_func_setup(J);
  T = traceref(J, lnk);
  if (T->linktype == LJ_TRLINK_RETURN) {  /* Trace returns to interpreter? */
    check_call_unroll(J, lnk);
    /* Temporarily unpatch JFUNC* to continue recording across function. */
    J->patchins = *J->pc;
    J->patchpc = (BCIns *)J->pc;
    *J->patchpc = T->startins;
    return;
  }
  J->instunroll = 0;  /* Cannot continue across a compiled function. */
  if (J->pc == J->startpc && J->framedepth + J->retdepth == 0)
    lj_record_stop(J, LJ_TRLINK_TAILREC, J->cur.traceno);  /* Extra tail-rec. */
  else
    lj_record_stop(J, LJ_TRLINK_ROOT, lnk);  /* Link to the function. */
}

/* -- Vararg handling ------------------------------------------------------ */

/* Detect y = select(x, ...) idiom. */
static int select_detect(jit_State *J)
{
  BCIns ins = J->pc[1];
  if (bc_op(ins) == BC_CALLM && bc_b(ins) == 2 && bc_c(ins) == 1) {
    cTValue *func = &J->L->base[bc_a(ins)];
    if (tvisfunc(func) && funcV(func)->c.ffid == FF_select)
      return 1;
  }
  return 0;
}

/* Record vararg instruction. */
static void rec_varg(jit_State *J, BCReg dst, ptrdiff_t nresults)
{
  int32_t numparams = J->pt->numparams;
  ptrdiff_t nvararg = frame_delta(J->L->base-1) - numparams - 1;
  lua_assert(frame_isvarg(J->L->base-1));
  if (J->framedepth > 0) {  /* Simple case: varargs defined on-trace. */
    ptrdiff_t i;
    if (nvararg < 0) nvararg = 0;
    if (nresults == -1) {
      nresults = nvararg;
      J->maxslot = dst + (BCReg)nvararg;
    } else if (dst + nresults > J->maxslot) {
      J->maxslot = dst + (BCReg)nresults;
    }
    for (i = 0; i < nresults; i++)
      J->base[dst+i] = i < nvararg ? getslot(J, i - nvararg - 1) : TREF_NIL;
  } else {  /* Unknown number of varargs passed to trace. */
    TRef fr = emitir(IRTI(IR_SLOAD), 0, IRSLOAD_READONLY|IRSLOAD_FRAME);
    int32_t frofs = 8*(1+numparams)+FRAME_VARG;
    if (nresults >= 0) {  /* Known fixed number of results. */
      ptrdiff_t i;
```

```
      if (nvararg > 0) {
        ptrdiff_t nload = nvararg >= nresults ? nresults : nvararg;
        TRef vbase;
        if (nvararg >= nresults)
          emitir(IRTGI(IR_GE), fr, lj_ir_kint(J, frofs+8*(int32_t)nresults));
        else
          emitir(IRTGI(IR_EQ), fr,
                 lj_ir_kint(J, (int32_t)frame_ftsz(J->L->base-1)));
        vbase = emitir(IRTI(IR_SUB), REF_BASE, fr);
        vbase = emitir(IRT(IR_ADD, IRT_P32), vbase, lj_ir_kint(J, frofs-8));
        for (i = 0; i < nload; i++) {
          IRType t = itype2irt(&J->L->base[i-1-nvararg]);
          TRef aref = emitir(IRT(IR_AREF, IRT_P32),
                             vbase, lj_ir_kint(J, (int32_t)i));
          TRef tr = emitir(IRTG(IR_VLOAD, t), aref, 0);
          if (irtype_ispri(t)) tr = TREF_PRI(t);  /* Canonicalize primitives. */
          J->base[dst+i] = tr;
        }
      } else {
        emitir(IRTGI(IR_LE), fr, lj_ir_kint(J, frofs));
        nvararg = 0;
      }
      for (i = nvararg; i < nresults; i++)
        J->base[dst+i] = TREF_NIL;
      if (dst + (BCReg)nresults > J->maxslot)
        J->maxslot = dst + (BCReg)nresults;
    } else if (select_detect(J)) {  /* y = select(x, ...) */
      TRef tridx = J->base[dst-1];
      TRef tr = TREF_NIL;
      ptrdiff_t idx = lj_ffrecord_select_mode(J, tridx, &J->L->base[dst-1]);
      if (idx < 0) goto nyivarg;
      if (idx != 0 && !tref_isinteger(tridx))
        tridx = emitir(IRTGI(IR_CONV), tridx, IRCONV_INT_NUM|IRCONV_INDEX);
      if (idx != 0 && tref_isk(tridx)) {
        emitir(IRTGI(idx <= nvararg ? IR_GE : IR_LT),
               fr, lj_ir_kint(J, frofs+8*(int32_t)idx));
        frofs -= 8;  /* Bias for 1-based index. */
      } else if (idx <= nvararg) {  /* Compute size. */
        TRef tmp = emitir(IRTI(IR_ADD), fr, lj_ir_kint(J, -frofs));
        if (numparams)
          emitir(IRTGI(IR_GE), tmp, lj_ir_kint(J, 0));
        tr = emitir(IRTI(IR_BSHR), tmp, lj_ir_kint(J, 3));
        if (idx != 0) {
          tridx = emitir(IRTI(IR_ADD), tridx, lj_ir_kint(J, -1));
          rec_idx_abc(J, tr, tridx, (uint32_t)nvararg);
        }
      } else {
        TRef tmp = lj_ir_kint(J, frofs);
        if (idx != 0) {
          TRef tmp2 = emitir(IRTI(IR_BSHL), tridx, lj_ir_kint(J, 3));
          tmp = emitir(IRTI(IR_ADD), tmp2, tmp);
        } else {
          tr = lj_ir_kint(J, 0);
        }
        emitir(IRTGI(IR_LT), fr, tmp);
      }
      if (idx != 0 && idx <= nvararg) {
        IRType t;
        TRef aref, vbase = emitir(IRTI(IR_SUB), REF_BASE, fr);
        vbase = emitir(IRT(IR_ADD, IRT_P32), vbase, lj_ir_kint(J, frofs-8));
        t = itype2irt(&J->L->base[idx-2-nvararg]);
        aref = emitir(IRT(IR_AREF, IRT_P32), vbase, tridx);
        tr = emitir(IRTG(IR_VLOAD, t), aref, 0);
        if (irtype_ispri(t)) tr = TREF_PRI(t);  /* Canonicalize primitives. */
      }
      J->base[dst-2] = tr;
      J->maxslot = dst-1;
      J->bcskip = 2;  /* Skip CALLM + select. */
    } else {
    nyivarg:
      setintV(&J->errinfo, BC_VARG);
      lj_trace_err_info(J, LJ_TRERR_NYIBC);
    }
  }
}
```

```
1709 /* -- Record allocations ------------------------------------------------ */
1710
1711 static TRef rec_tnew(jit_State *J, uint32_t ah)
1712 {
1713   uint32_t asize = ah & 0x7ff;
1714   uint32_t hbits = ah >> 11;
1715   if (asize == 0x7ff) asize = 0x801;
1716   return emitir(IRTG(IR_TNEW, IRT_TAB), asize, hbits);
1717 }
1718
1719 /* -- Concatenation ---------------------------------------------------- */
1720
1721 static TRef rec_cat(jit_State *J, BCReg baseslot, BCReg topslot)
1722 {
1723   TRef *top = &J->base[topslot];
1724   TValue savetv[5];
1725   BCReg s;
1726   RecordIndex ix;
1727   lua_assert(baseslot < topslot);
1728   for (s = baseslot; s <= topslot; s++)
1729     (void)getslot(J, s);  /* Ensure all arguments have a reference. */
1730   if (tref_isnumber_str(top[0]) && tref_isnumber_str(top[-1])) {
1731     TRef tr, hdr, *trp, *xbase, *base = &J->base[baseslot];
1732     /* First convert numbers to strings. */
1733     for (trp = top; trp >= base; trp--) {
1734       if (tref_isnumber(*trp))
1735         *trp = emitir(IRT(IR_TOSTR, IRT_STR), *trp,
1736                       tref_isnum(*trp) ? IRTOSTR_NUM : IRTOSTR_INT);
1737       else if (!tref_isstr(*trp))
1738         break;
1739     }
1740     xbase = ++trp;
1741     tr = hdr = emitir(IRT(IR_BUFHDR, IRT_P32),
1742                       lj_ir_kptr(J, &J2G(J)->tmpbuf), IRBUFHDR_RESET);
1743     do {
1744       tr = emitir(IRT(IR_BUFPUT, IRT_P32), tr, *trp++);
1745     } while (trp <= top);
1746     tr = emitir(IRT(IR_BUFSTR, IRT_STR), tr, hdr);
1747     J->maxslot = (BCReg)(xbase - J->base);
1748     if (xbase == base) return tr;  /* Return simple concatenation result. */
1749     /* Pass partial result. */
1750     topslot = J->maxslot--;
1751     *xbase = tr;
1752     top = xbase;
1753     setstrV(J->L, &ix.keyv, &J2G(J)->strempty);  /* Simulate string result. */
1754   } else {
1755     J->maxslot = topslot-1;
1756     copyTV(J->L, &ix.keyv, &J->L->base[topslot]);
1757   }
1758   copyTV(J->L, &ix.tabv, &J->L->base[topslot-1]);
1759   ix.tab = top[-1];
1760   ix.key = top[0];
1761   memcpy(savetv, &J->L->base[topslot-1], sizeof(savetv));  /* Save slots. */
1762   rec_mm_arith(J, &ix, MM_concat);  /* Call __concat metamethod. */
1763   memcpy(&J->L->base[topslot-1], savetv, sizeof(savetv));  /* Restore slots. */
1764   return 0;  /* No result yet. */
1765 }
1766
1767 /* -- Record bytecode ops ---------------------------------------------- */
1768
1769 /* Prepare for comparison. */
1770 static void rec_comp_prep(jit_State *J)
1771 {
1772   /* Prevent merging with snapshot #0 (GC exit) since we fixup the PC. */
1773   if (J->cur.nsnap == 1 && J->cur.snap[0].ref == J->cur.nins)
1774     emitir_raw(IRT(IR_NOP, IRT_NIL), 0, 0);
1775   lj_snap_add(J);
1776 }
1777
1778 /* Fixup comparison. */
1779 static void rec_comp_fixup(jit_State *J, const BCIns *pc, int cond)
1780 {
1781   BCIns jmpins = pc[1];
1782   const BCIns *npc = pc + 2 + (cond ? bc_j(jmpins) : 0);
1783   SnapShot *snap = &J->cur.snap[J->cur.nsnap-1];
1784   /* Set PC to opposite target to avoid re-recording the comp. in side trace. */
```

```
  J->cur.snapmap[snap->mapofs + snap->nent] = SNAP_MKPC(npc);
  J->needsnap = 1;
  if (bc_a(jmpins) < J->maxslot) J->maxslot = bc_a(jmpins);
  lj_snap_shrink(J);  /* Shrink last snapshot if possible. */
}

/* Record the next bytecode instruction (_before_ it's executed). */
void lj_record_ins(jit_State *J)
{
  cTValue *lbase;
  RecordIndex ix;
  const BCIns *pc;
  BCIns ins;
  BCOp op;
  TRef ra, rb, rc;

  /* Perform post-processing action before recording the next instruction. */
  if (LJ_UNLIKELY(J->postproc != LJ_POST_NONE)) {
    switch (J->postproc) {
    case LJ_POST_FIXCOMP:  /* Fixup comparison. */
      pc = (const BCIns *)(uintptr_t)J2G(J)->tmptv.u64;
      rec_comp_fixup(J, pc, (!tvistruecond(&J2G(J)->tmptv2) ^ (bc_op(*pc)&1)));
      /* fallthrough */
    case LJ_POST_FIXGUARD:  /* Fixup and emit pending guard. */
    case LJ_POST_FIXGUARDSNAP:  /* Fixup and emit pending guard and snapshot. */
      if (!tvistruecond(&J2G(J)->tmptv2)) {
        J->fold.ins.o ^= 1;  /* Flip guard to opposite. */
        if (J->postproc == LJ_POST_FIXGUARDSNAP) {
          SnapShot *snap = &J->cur.snap[J->cur.nsnap-1];
          J->cur.snapmap[snap->mapofs+snap->nent-1]--;  /* False -> true. */
        }
      }
      lj_opt_fold(J);  /* Emit pending guard. */
      /* fallthrough */
    case LJ_POST_FIXBOOL:
      if (!tvistruecond(&J2G(J)->tmptv2)) {
        BCReg s;
        TValue *tv = J->L->base;
        for (s = 0; s < J->maxslot; s++)  /* Fixup stack slot (if any). */
          if (J->base[s] == TREF_TRUE && tvisfalse(&tv[s])) {
            J->base[s] = TREF_FALSE;
            break;
          }
      }
      break;
    case LJ_POST_FIXCONST:
      {
        BCReg s;
        TValue *tv = J->L->base;
        for (s = 0; s < J->maxslot; s++)  /* Constify stack slots (if any). */
          if (J->base[s] == TREF_NIL && !tvisnil(&tv[s]))
            J->base[s] = lj_record_constify(J, &tv[s]);
      }
      break;
    case LJ_POST_FFRETRY:  /* Suppress recording of retried fast function. */
      if (bc_op(*J->pc) >= BC__MAX)
        return;
      break;
    default: lua_assert(0); break;
    }
    J->postproc = LJ_POST_NONE;
  }

  /* Need snapshot before recording next bytecode (e.g. after a store). */
  if (J->needsnap) {
    J->needsnap = 0;
    lj_snap_purge(J);
    lj_snap_add(J);
    J->mergesnap = 1;
  }

  /* Skip some bytecodes. */
  if (LJ_UNLIKELY(J->bcskip > 0)) {
    J->bcskip--;
    return;
  }
```

```
1861
1862    /* Record only closed loops for root traces. */
1863    pc = J->pc;
1864    if (J->framedepth == 0 &&
1865       (MSize)((char *)pc - (char *)J->bc_min) >= J->bc_extent)
1866      lj_trace_err(J, LJ_TRERR_LLEAVE);
1867
1868  #ifdef LUA_USE_ASSERT
1869    rec_check_slots(J);
1870    rec_check_ir(J);
1871  #endif
1872
1873  #if LJ_HASPROFILE
1874    rec_profile_ins(J, pc);
1875  #endif
1876
1877    /* Keep a copy of the runtime values of var/num/str operands. */
1878  #define rav        (&ix.valv)
1879  #define rbv        (&ix.tabv)
1880  #define rcv        (&ix.keyv)
1881
1882    lbase = J->L->base;
1883    ins = *pc;
1884    op = bc_op(ins);
1885    ra = bc_a(ins);
1886    ix.val = 0;
1887    switch (bcmode_a(op)) {
1888    case BCMvar:
1889      copyTV(J->L, rav, &lbase[ra]); ix.val = ra = getslot(J, ra); break;
1890    default: break;  /* Handled later. */
1891    }
1892    rb = bc_b(ins);
1893    rc = bc_c(ins);
1894    switch (bcmode_b(op)) {
1895    case BCMnone: rb = 0; rc = bc_d(ins); break;  /* Upgrade rc to 'rd'. */
1896    case BCMvar:
1897      copyTV(J->L, rbv, &lbase[rb]); ix.tab = rb = getslot(J, rb); break;
1898    default: break;  /* Handled later. */
1899    }
1900    switch (bcmode_c(op)) {
1901    case BCMvar:
1902      copyTV(J->L, rcv, &lbase[rc]); ix.key = rc = getslot(J, rc); break;
1903    case BCMpri: setpriV(rcv, ~rc); ix.key = rc = TREF_PRI(IRT_NIL+rc); break;
1904    case BCMnum: { cTValue *tv = proto_knumtv(J->pt, rc);
1905      copyTV(J->L, rcv, tv); ix.key = rc = tvisint(tv) ? lj_ir_kint(J, intV(tv)) :
1906      lj_ir_knumint(J, numV(tv)); } break;
1907    case BCMstr: { GCstr *s = gco2str(proto_kgc(J->pt, ~(ptrdiff_t)rc));
1908      setstrV(J->L, rcv, s); ix.key = rc = lj_ir_kstr(J, s); } break;
1909    default: break;  /* Handled later. */
1910    }
1911
1912    switch (op) {
1913
1914    /* -- Comparison ops ---------------------------------------------------- */
1915
1916    case BC_ISLT: case BC_ISGE: case BC_ISLE: case BC_ISGT:
1917  #if LJ_HASFFI
1918      if (tref_iscdata(ra) || tref_iscdata(rc)) {
1919        rec_mm_comp_cdata(J, &ix, op, ((int)op & 2) ? MM_le : MM_lt);
1920        break;
1921      }
1922  #endif
1923      /* Emit nothing for two numeric or string consts. */
1924      if (!(tref_isk2(ra,rc) && tref_isnumber_str(ra) && tref_isnumber_str(rc))) {
1925        IRType ta = tref_isinteger(ra) ? IRT_INT : tref_type(ra);
1926        IRType tc = tref_isinteger(rc) ? IRT_INT : tref_type(rc);
1927        int irop;
1928        if (ta != tc) {
1929          /* Widen mixed number/int comparisons to number/number comparison. */
1930          if (ta == IRT_INT && tc == IRT_NUM) {
1931            ra = emitir(IRTN(IR_CONV), ra, IRCONV_NUM_INT);
1932            ta = IRT_NUM;
1933          } else if (ta == IRT_NUM && tc == IRT_INT) {
1934            rc = emitir(IRTN(IR_CONV), rc, IRCONV_NUM_INT);
1935          } else if (LJ_52) {
1936            ta = IRT_NIL;  /* Force metamethod for different types. */
```

```
        } else if (!((ta == IRT_FALSE || ta == IRT_TRUE) &&
                     (tc == IRT_FALSE || tc == IRT_TRUE))) {
          break;  /* Interpreter will throw for two different types. */
        }
      }
      rec_comp_prep(J);
      irop = (int)op - (int)BC_ISLT + (int)IR_LT;
      if (ta == IRT_NUM) {
        if ((irop & 1)) irop ^= 4;  /* ISGE/ISGT are unordered. */
        if (!lj_ir_numcmp(numberVnum(rav), numberVnum(rcv), (IROp)irop))
          irop ^= 5;
      } else if (ta == IRT_INT) {
        if (!lj_ir_numcmp(numberVnum(rav), numberVnum(rcv), (IROp)irop))
          irop ^= 1;
      } else if (ta == IRT_STR) {
        if (!lj_ir_strcmp(strV(rav), strV(rcv), (IROp)irop)) irop ^= 1;
        ra = lj_ir_call(J, IRCALL_lj_str_cmp, ra, rc);
        rc = lj_ir_kint(J, 0);
        ta = IRT_INT;
      } else {
        rec_mm_comp(J, &ix, (int)op);
        break;
      }
      emitir(IRTG(irop, ta), ra, rc);
      rec_comp_fixup(J, J->pc, ((int)op ^ irop) & 1);
    }
    break;

  case BC_ISEQV: case BC_ISNEV:
  case BC_ISEQS: case BC_ISNES:
  case BC_ISEQN: case BC_ISNEN:
  case BC_ISEQP: case BC_ISNEP:
#if LJ_HASFFI
    if (tref_iscdata(ra) || tref_iscdata(rc)) {
      rec_mm_comp_cdata(J, &ix, op, MM_eq);
      break;
    }
#endif
    /* Emit nothing for two non-table, non-udata consts. */
    if (!(tref_isk2(ra, rc) && !(tref_istab(ra) || tref_isudata(ra)))) {
      int diff;
      rec_comp_prep(J);
      diff = lj_record_objcmp(J, ra, rc, rav, rcv);
      if (diff == 2 || !(tref_istab(ra) || tref_isudata(ra)))
        rec_comp_fixup(J, J->pc, ((int)op & 1) == !diff);
      else if (diff == 1)  /* Only check __eq if different, but same type. */
        rec_mm_equal(J, &ix, (int)op);
    }
    break;

  /* -- Unary test and copy ops --------------------------------------------- */

  case BC_ISTC: case BC_ISFC:
    if ((op & 1) == tref_istruecond(rc))
      rc = 0;  /* Don't store if condition is not true. */
    /* fallthrough */
  case BC_IST: case BC_ISF:  /* Type specialization suffices. */
    if (bc_a(pc[1]) < J->maxslot)
      J->maxslot = bc_a(pc[1]);  /* Shrink used slots. */
    break;

  case BC_ISTYPE: case BC_ISNUM:
    /* These coercions need to correspond with lj_meta_istype(). */
    if (LJ_DUALNUM && rc == ~LJ_TNUMX+1)
      ra = lj_opt_narrow_toint(J, ra);
    else if (rc == ~LJ_TNUMX+2)
      ra = lj_ir_tonum(J, ra);
    else if (rc == ~LJ_TSTR+1)
      ra = lj_ir_tostr(J, ra);
    /* else: type specialization suffices. */
    J->base[bc_a(ins)] = ra;
    break;

  /* -- Unary ops ----------------------------------------------------------- */

  case BC_NOT:
```

```
2013      /* Type specialization already forces const result. */
2014      rc = tref_istruecond(rc) ? TREF_FALSE : TREF_TRUE;
2015      break;
2016
2017    case BC_LEN:
2018      if (tref_isstr(rc))
2019        rc = emitir(IRTI(IR_FLOAD), rc, IRFL_STR_LEN);
2020      else if (!LJ_52 && tref_istab(rc))
2021        rc = lj_ir_call(J, IRCALL_lj_tab_len, rc);
2022      else
2023        rc = rec_mm_len(J, rc, rcv);
2024      break;
2025
2026    /* -- Arithmetic ops ------------------------------------------------ */
2027
2028    case BC_UNM:
2029      if (tref_isnumber_str(rc)) {
2030        rc = lj_opt_narrow_unm(J, rc, rcv);
2031      } else {
2032        ix.tab = rc;
2033        copyTV(J->L, &ix.tabv, rcv);
2034        rc = rec_mm_arith(J, &ix, MM_unm);
2035      }
2036      break;
2037
2038    case BC_ADDNV: case BC_SUBNV: case BC_MULNV: case BC_DIVNV: case BC_MODNV:
2039      /* Swap rb/rc and rbv/rcv. rav is temp. */
2040      ix.tab = rc; ix.key = rc = rb; rb = ix.tab;
2041      copyTV(J->L, rav, rbv);
2042      copyTV(J->L, rbv, rcv);
2043      copyTV(J->L, rcv, rav);
2044      if (op == BC_MODNV)
2045        goto recmod;
2046      /* fallthrough */
2047    case BC_ADDVN: case BC_SUBVN: case BC_MULVN: case BC_DIVVN:
2048    case BC_ADDVV: case BC_SUBVV: case BC_MULVV: case BC_DIVVV: {
2049      MMS mm = bcmode_mm(op);
2050      if (tref_isnumber_str(rb) && tref_isnumber_str(rc))
2051        rc = lj_opt_narrow_arith(J, rb, rc, rbv, rcv,
2052                                 (int)mm - (int)MM_add + (int)IR_ADD);
2053      else
2054        rc = rec_mm_arith(J, &ix, mm);
2055      break;
2056      }
2057
2058    case BC_MODVN: case BC_MODVV:
2059    recmod:
2060      if (tref_isnumber_str(rb) && tref_isnumber_str(rc))
2061        rc = lj_opt_narrow_mod(J, rb, rc, rcv);
2062      else
2063        rc = rec_mm_arith(J, &ix, MM_mod);
2064      break;
2065
2066    case BC_POW:
2067      if (tref_isnumber_str(rb) && tref_isnumber_str(rc))
2068        rc = lj_opt_narrow_pow(J, lj_ir_tonum(J, rb), rc, rcv);
2069      else
2070        rc = rec_mm_arith(J, &ix, MM_pow);
2071      break;
2072
2073    /* -- Miscellaneous ops --------------------------------------------- */
2074
2075    case BC_CAT:
2076      rc = rec_cat(J, rb, rc);
2077      break;
2078
2079    /* -- Constant and move ops ----------------------------------------- */
2080
2081    case BC_MOV:
2082      /* Clear gap of method call to avoid resurrecting previous refs. */
2083      if (ra > J->maxslot) J->base[ra-1] = 0;
2084      break;
2085    case BC_KSTR: case BC_KNUM: case BC_KPRI:
2086      break;
2087    case BC_KSHORT:
2088      rc = lj_ir_kint(J, (int32_t)(int16_t)rc);
```

```
2089      break;
2090    case BC_KNIL:
2091      while (ra <= rc)
2092        J->base[ra++] = TREF_NIL;
2093      if (rc >= J->maxslot) J->maxslot = rc+1;
2094      break;
2095  #if LJ_HASFFI
2096    case BC_KCDATA:
2097      rc = lj_ir_kgc(J, proto_kgc(J->pt, ~(ptrdiff_t)rc), IRT_CDATA);
2098      break;
2099  #endif
2100
2101    /* -- Upvalue and function ops ------------------------------------------ */
2102
2103    case BC_UGET:
2104      rc = rec_upvalue(J, rc, 0);
2105      break;
2106    case BC_USETV: case BC_USETS: case BC_USETN: case BC_USETP:
2107      rec_upvalue(J, ra, rc);
2108      break;
2109
2110    /* -- Table ops --------------------------------------------------------- */
2111
2112    case BC_GGET: case BC_GSET:
2113      settabV(J->L, &ix.tabv, tabref(J->fn->l.env));
2114      ix.tab = emitir(IRT(IR_FLOAD, IRT_TAB), getcurrf(J), IRFL_FUNC_ENV);
2115      ix.idxchain = LJ_MAX_IDXCHAIN;
2116      rc = lj_record_idx(J, &ix);
2117      break;
2118
2119    case BC_TGETB: case BC_TSETB:
2120      setintV(&ix.keyv, (int32_t)rc);
2121      ix.key = lj_ir_kint(J, (int32_t)rc);
2122      /* fallthrough */
2123    case BC_TGETV: case BC_TGETS: case BC_TSETV: case BC_TSETS:
2124      ix.idxchain = LJ_MAX_IDXCHAIN;
2125      rc = lj_record_idx(J, &ix);
2126      break;
2127    case BC_TGETR: case BC_TSETR:
2128      ix.idxchain = 0;
2129      rc = lj_record_idx(J, &ix);
2130      break;
2131
2132    case BC_TSETM:
2133      rec_tsetm(J, ra, (BCReg)(J->L->top - J->L->base), (int32_t)rcv->u32.lo);
2134      break;
2135
2136    case BC_TNEW:
2137      rc = rec_tnew(J, rc);
2138      break;
2139    case BC_TDUP:
2140      rc = emitir(IRTG(IR_TDUP, IRT_TAB),
2141                  lj_ir_ktab(J, gco2tab(proto_kgc(J->pt, ~(ptrdiff_t)rc))), 0);
2142      break;
2143
2144    /* -- Calls and vararg handling ----------------------------------------- */
2145
2146    case BC_ITERC:
2147      J->base[ra] = getslot(J, ra-3-LJ_FR2);
2148      J->base[ra+1] = getslot(J, ra-2-LJ_FR2);
2149      J->base[ra+2] = getslot(J, ra-1-LJ_FR2);
2150      { /* Do the actual copy now because lj_record_call needs the values. */
2151        TValue *b = &J->L->base[ra];
2152        copyTV(J->L, b, b-3-LJ_FR2);
2153        copyTV(J->L, b+1, b-2-LJ_FR2);
2154        copyTV(J->L, b+2, b-1-LJ_FR2);
2155      }
2156      lj_record_call(J, ra, (ptrdiff_t)rc-1);
2157      break;
2158
2159    /* L->top is set to L->base+ra+rc+NARGS-1+1. See lj_dispatch_ins(). */
2160    case BC_CALLM:
2161      rc = (BCReg)(J->L->top - J->L->base) - ra - LJ_FR2;
2162      /* fallthrough */
2163    case BC_CALL:
2164      lj_record_call(J, ra, (ptrdiff_t)rc-1);
```

```
2165      break;
2166
2167    case BC_CALLMT:
2168      rc = (BCReg)(J->L->top - J->L->base) - ra - LJ_FR2;
2169      /* fallthrough */
2170    case BC_CALLT:
2171      lj_record_tailcall(J, ra, (ptrdiff_t)rc-1);
2172      break;
2173
2174    case BC_VARG:
2175      rec_varg(J, ra, (ptrdiff_t)rb-1);
2176      break;
2177
2178    /* -- Returns ------------------------------------------------------- */
2179
2180    case BC_RETM:
2181      /* L->top is set to L->base+ra+rc+NRESULTS-1, see lj_dispatch_ins(). */
2182      rc = (BCReg)(J->L->top - J->L->base) - ra + 1;
2183      /* fallthrough */
2184    case BC_RET: case BC_RET0: case BC_RET1:
2185  #if LJ_HASPROFILE
2186      rec_profile_ret(J);
2187  #endif
2188      lj_record_ret(J, ra, (ptrdiff_t)rc-1);
2189      break;
2190
2191    /* -- Loops and branches -------------------------------------------- */
2192
2193    case BC_FORI:
2194      if (rec_for(J, pc, 0) != LOOPEV_LEAVE)
2195        J->loopref = J->cur.nins;
2196      break;
2197    case BC_JFORI:
2198      lua_assert(bc_op(pc[(ptrdiff_t)rc-BCBIAS_J]) == BC_JFORL);
2199      if (rec_for(J, pc, 0) != LOOPEV_LEAVE)  /* Link to existing loop. */
2200        lj_record_stop(J, LJ_TRLINK_ROOT, bc_d(pc[(ptrdiff_t)rc-BCBIAS_J]));
2201      /* Continue tracing if the loop is not entered. */
2202      break;
2203
2204    case BC_FORL:
2205      rec_loop_interp(J, pc, rec_for(J, pc+((ptrdiff_t)rc-BCBIAS_J), 1));
2206      break;
2207    case BC_ITERL:
2208      rec_loop_interp(J, pc, rec_iterl(J, *pc));
2209      break;
2210    case BC_LOOP:
2211      rec_loop_interp(J, pc, rec_loop(J, ra));
2212      break;
2213
2214    case BC_JFORL:
2215      rec_loop_jit(J, rc, rec_for(J, pc+bc_j(traceref(J, rc)->startins), 1));
2216      break;
2217    case BC_JITERL:
2218      rec_loop_jit(J, rc, rec_iterl(J, traceref(J, rc)->startins));
2219      break;
2220    case BC_JLOOP:
2221      rec_loop_jit(J, rc, rec_loop(J, ra));
2222      break;
2223
2224    case BC_IFORL:
2225    case BC_IITERL:
2226    case BC_ILOOP:
2227    case BC_IFUNCF:
2228    case BC_IFUNCV:
2229      lj_trace_err(J, LJ_TRERR_BLACKL);
2230      break;
2231
2232    case BC_JMP:
2233      if (ra < J->maxslot)
2234        J->maxslot = ra;  /* Shrink used slots. */
2235      break;
2236
2237    /* -- Function headers ---------------------------------------------- */
2238
2239    case BC_FUNCF:
2240      rec_func_lua(J);
```

```
2241      break;
2242    case BC_JFUNCF:
2243      rec_func_jit(J, rc);
2244      break;
2245
2246    case BC_FUNCV:
2247      rec_func_vararg(J);
2248      rec_func_lua(J);
2249      break;
2250    case BC_JFUNCV:
2251      lua_assert(0);  /* Cannot happen. No hotcall counting for varag funcs. */
2252      break;
2253
2254    case BC_FUNCC:
2255    case BC_FUNCCW:
2256      lj_ffrecord_func(J);
2257      break;
2258
2259    default:
2260      if (op >= BC__MAX) {
2261        lj_ffrecord_func(J);
2262        break;
2263      }
2264      /* fallthrough */
2265    case BC_ITERN:
2266    case BC_ISNEXT:
2267    case BC_UCLO:
2268    case BC_FNEW:
2269      setintV(&J->errinfo, (int32_t)op);
2270      lj_trace_err_info(J, LJ_TRERR_NYIBC);
2271      break;
2272    }
2273
2274    /* rc == 0 if we have no result yet, e.g. pending __index metamethod call. */
2275    if (bcmode_a(op) == BCMdst && rc) {
2276      J->base[ra] = rc;
2277      if (ra >= J->maxslot) J->maxslot = ra+1;
2278    }
2279
2280  #undef rav
2281  #undef rbv
2282  #undef rcv
2283
2284    /* Limit the number of recorded IR instructions. */
2285    if (J->cur.nins > REF_FIRST+(IRRef)J->param[JIT_P_maxrecord])
2286      lj_trace_err(J, LJ_TRERR_TRACEOV);
2287  }
2288
2289  /* -- Recording setup ---------------------------------------------------- */
2290
2291  /* Setup recording for a root trace started by a hot loop. */
2292  static const BCIns *rec_setup_root(jit_State *J)
2293  {
2294    /* Determine the next PC and the bytecode range for the loop. */
2295    const BCIns *pcj, *pc = J->pc;
2296    BCIns ins = *pc;
2297    BCReg ra = bc_a(ins);
2298    switch (bc_op(ins)) {
2299    case BC_FORL:
2300      J->bc_extent = (MSize)(-bc_j(ins))*sizeof(BCIns);
2301      pc += 1+bc_j(ins);
2302      J->bc_min = pc;
2303      break;
2304    case BC_ITERL:
2305      lua_assert(bc_op(pc[-1]) == BC_ITERC);
2306      J->maxslot = ra + bc_b(pc[-1]) - 1;
2307      J->bc_extent = (MSize)(-bc_j(ins))*sizeof(BCIns);
2308      pc += 1+bc_j(ins);
2309      lua_assert(bc_op(pc[-1]) == BC_JMP);
2310      J->bc_min = pc;
2311      break;
2312    case BC_LOOP:
2313      /* Only check BC range for real loops, but not for "repeat until true". */
2314      pcj = pc + bc_j(ins);
2315      ins = *pcj;
2316      if (bc_op(ins) == BC_JMP && bc_j(ins) < 0) {
```

```
2317        J->bc_min = pcj+1 + bc_j(ins);
2318        J->bc_extent = (MSize)(-bc_j(ins))*sizeof(BCIns);
2319      }
2320      J->maxslot = ra;
2321      pc++;
2322      break;
2323    case BC_RET:
2324    case BC_RET0:
2325    case BC_RET1:
2326      /* No bytecode range check for down-recursive root traces. */
2327      J->maxslot = ra + bc_d(ins) - 1;
2328      break;
2329    case BC_FUNCF:
2330      /* No bytecode range check for root traces started by a hot call. */
2331      J->maxslot = J->pt->numparams;
2332      pc++;
2333      break;
2334    case BC_CALLM:
2335    case BC_CALL:
2336    case BC_ITERC:
2337      /* No bytecode range check for stitched traces. */
2338      pc++;
2339      break;
2340    default:
2341      lua_assert(0);
2342      break;
2343    }
2344    return pc;
2345  }
2346
2347  /* Setup for recording a new trace. */
2348  void lj_record_setup(jit_State *J)
2349  {
2350    uint32_t i;
2351
2352    /* Initialize state related to current trace. */
2353    memset(J->slot, 0, sizeof(J->slot));
2354    memset(J->chain, 0, sizeof(J->chain));
2355    memset(J->bpropcache, 0, sizeof(J->bpropcache));
2356    J->scev.idx = REF_NIL;
2357    setmref(J->scev.pc, NULL);
2358
2359    J->baseslot = 1;  /* Invoking function is at base[-1]. */
2360    J->base = J->slot + J->baseslot;
2361    J->maxslot = 0;
2362    J->framedepth = 0;
2363    J->retdepth = 0;
2364
2365    J->instunroll = J->param[JIT_P_instunroll];
2366    J->loopunroll = J->param[JIT_P_loopunroll];
2367    J->tailcalled = 0;
2368    J->loopref = 0;
2369
2370    J->bc_min = NULL;  /* Means no limit. */
2371    J->bc_extent = ~(MSize)0;
2372
2373    /* Emit instructions for fixed references. Also triggers initial IR alloc. */
2374    emitir_raw(IRT(IR_BASE, IRT_P32), J->parent, J->exitno);
2375    for (i = 0; i <= 2; i++) {
2376      IRIns *ir = IR(REF_NIL-i);
2377      ir->i = 0;
2378      ir->t.irt = (uint8_t)(IRT_NIL+i);
2379      ir->o = IR_KPRI;
2380      ir->prev = 0;
2381    }
2382    J->cur.nk = REF_TRUE;
2383
2384    J->startpc = J->pc;
2385    setmref(J->cur.startpc, J->pc);
2386    if (J->parent) {  /* Side trace. */
2387      GCtrace *T = traceref(J, J->parent);
2388      TraceNo root = T->root ? T->root : J->parent;
2389      J->cur.root = (uint16_t)root;
2390      J->cur.startins = BCINS_AD(BC_JMP, 0, 0);
2391      /* Check whether we could at least potentially form an extra loop. */
2392      if (J->exitno == 0 && T->snap[0].nent == 0) {
```

```
      /* We can narrow a FORL for some side traces, too. */
      if (J->pc > proto_bc(J->pt) && bc_op(J->pc[-1]) == BC_JFORI &&
          bc_d(J->pc[bc_j(J->pc[-1])-1]) == root) {
        lj_snap_add(J);
        rec_for_loop(J, J->pc-1, &J->scev, 1);
        goto sidecheck;
      }
    } else {
      J->startpc = NULL;  /* Prevent forming an extra loop. */
    }
    lj_snap_replay(J, T);
  sidecheck:
    if (traceref(J, J->cur.root)->nchild >= J->param[JIT_P_maxside] ||
        T->snap[J->exitno].count >= J->param[JIT_P_hotexit] +
                                    J->param[JIT_P_tryside]) {
      lj_record_stop(J, LJ_TRLINK_INTERP, 0);
    }
  } else {  /* Root trace. */
    J->cur.root = 0;
    J->cur.startins = *J->pc;
    J->pc = rec_setup_root(J);
    /* Note: the loop instruction itself is recorded at the end and not
    ** at the start! So snapshot #0 needs to point to the *next* instruction.
    */
    lj_snap_add(J);
    if (bc_op(J->cur.startins) == BC_FORL)
      rec_for_loop(J, J->pc-1, &J->scev, 1);
    else if (bc_op(J->cur.startins) == BC_ITERC)
      J->startpc = NULL;
    if (1 + J->pt->framesize >= LJ_MAX_JSLOTS)
      lj_trace_err(J, LJ_TRERR_STACKOV);
  }
#if LJ_HASPROFILE
  J->prev_pt = NULL;
  J->prev_line = -1;
#endif
#ifdef LUAJIT_ENABLE_CHECKHOOK
  /* Regularly check for instruction/line hooks from compiled code and
  ** exit to the interpreter if the hooks are set.
  **
  ** This is a compile-time option and disabled by default, since the
  ** hook checks may be quite expensive in tight loops.
  **
  ** Note this is only useful if hooks are *not* set most of the time.
  ** Use this only if you want to *asynchronously* interrupt the execution.
  **
  ** You can set the instruction hook via lua_sethook() with a count of 1
  ** from a signal handler or another native thread. Please have a look
  ** at the first few functions in luajit.c for an example (Ctrl-C handler).
  */
  {
    TRef tr = emitir(IRT(IR_XLOAD, IRT_U8),
                     lj_ir_kptr(J, &J2G(J)->hookmask), IRXLOAD_VOLATILE);
    tr = emitir(IRTI(IR_BAND), tr, lj_ir_kint(J, (LUA_MASKLINE|LUA_MASKCOUNT)));
    emitir(IRTGI(IR_EQ), tr, lj_ir_kint(J, 0));
  }
#endif
}

#undef IR
#undef emitir_raw
#undef emitir

#endif
```

One Level Up Top Level

src/lj_record.h - luajit-2.0-src

Data types defined

- RecordIndex
- RecordIndex

Macros defined

- _LJ_RECORD_H

Source code

```
1  /*
2  ** Trace recorder (bytecode -> SSA IR).
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_RECORD_H
7  #define _LJ_RECORD_H
8
9  #include "lj_obj.h"
10  #include "lj_jit.h"
11
12  #if LJ_HASJIT
13  /* Context for recording an indexed load/store. */
14  typedef struct RecordIndex {
15    TValue tabv;               /* Runtime value of table (or indexed object). */
16    TValue keyv;               /* Runtime value of key. */
17    TValue valv;               /* Runtime value of stored value. */
18    TValue mobjv;               /* Runtime value of metamethod object. */
19    GCtab *mtv;               /* Runtime value of metatable object. */
20    cTValue *oldv;        /* Runtime value of previously stored value. */
21    TRef tab;               /* Table (or indexed object) reference. */
22    TRef key;               /* Key reference. */
23    TRef val;               /* Value reference for a store or 0 for a load. */
24    TRef mt;               /* Metatable reference. */
25    TRef mobj;               /* Metamethod object reference. */
26    int idxchain;               /* Index indirections left or 0 for raw lookup. */
27  } RecordIndex;
28
29  LJ_FUNC int lj_record_objcmp(jit_State *J, TRef a, TRef b,
30                               cTValue *av, cTValue *bv);
31  LJ_FUNC void lj_record_stop(jit_State *J, TraceLink linktype, TraceNo lnk);
32  LJ_FUNC TRef lj_record_constify(jit_State *J, cTValue *o);
33
34  LJ_FUNC void lj_record_call(jit_State *J, BCReg func, ptrdiff_t nargs);
35  LJ_FUNC void lj_record_tailcall(jit_State *J, BCReg func, ptrdiff_t nargs);
36  LJ_FUNC void lj_record_ret(jit_State *J, BCReg rbase, ptrdiff_t gotresults);
37
38  LJ_FUNC int lj_record_mm_lookup(jit_State *J, RecordIndex *ix, MMS mm);
39  LJ_FUNC TRef lj_record_idx(jit_State *J, RecordIndex *ix);
40
41  LJ_FUNC void lj_record_ins(jit_State *J);
42  LJ_FUNC void lj_record_setup(jit_State *J);
43  #endif
44
45  #endif
```

src/lj_crecord.c - luajit-2.0-src

Data types defined

- CRecMemList
- CRecMemList

Functions defined

- argv2cdata
- argv2ctype
- crec_alloc
- crec_arith_int64
- crec_arith_meta
- crec_arith_ptr
- crec_bit64_type
- crec_call
- crec_call_args
- crec_constructor
- crec_copy
- crec_copy_emit
- crec_copy_struct
- crec_copy_unroll
- crec_ct2irt
- crec_ct_ct
- crec_ct_tv
- crec_fill
- crec_fill_emit
- crec_fill_unroll
- crec_finalizer
- crec_index_meta
- crec_isnonzero
- crec_reassoc_ofs
- crec_snap_caller
- crec_toint

- crec_tv_ct
- lj_crecord_tonumber
- recff_bit64_nary
- recff_bit64_shift
- recff_bit64_tobit
- recff_bit64_tohex
- recff_bit64_unary
- recff_cdata_arith
- recff_cdata_call
- recff_cdata_index
- recff_clib_index
- recff_ffi_abi
- recff_ffi_copy
- recff_ffi_errno
- recff_ffi_fill
- recff_ffi_gc
- recff_ffi_istype
- recff_ffi_new
- recff_ffi_string
- recff_ffi_typeof
- recff_ffi_xof

Macros defined

- CREC_COPY_MAXLEN
- CREC_COPY_MAXUNROLL
- CREC_COPY_REGWIN
- CREC_COPY_REGWIN
- CREC_COPY_REGWIN
- CREC_FILL_MAXUNROLL
- IR
- IR
- LUA_CORE
- emitconv
- emitconv

- emitir
- emitir
- lj_ffrecord_c

Source code

```
/*
** Trace recorder for C data operations.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_ffrecord_c
#define LUA_CORE

#include "lj_obj.h"

#if LJ_HASJIT && LJ_HASFFI

#include "lj_err.h"
#include "lj_tab.h"
#include "lj_frame.h"
#include "lj_ctype.h"
#include "lj_cdata.h"
#include "lj_cparse.h"
#include "lj_cconv.h"
#include "lj_carith.h"
#include "lj_clib.h"
#include "lj_ccall.h"
#include "lj_ff.h"
#include "lj_ir.h"
#include "lj_jit.h"
#include "lj_ircall.h"
#include "lj_iropt.h"
#include "lj_trace.h"
#include "lj_record.h"
#include "lj_ffrecord.h"
#include "lj_snap.h"
#include "lj_crecord.h"
#include "lj_dispatch.h"
#include "lj_strfmt.h"

/* Some local macros to save typing. Undef'd at the end. */
#define IR(ref)                         (&J->cur.ir[(ref)])

/* Pass IR on to next optimization in chain (FOLD). */
#define emitir(ot, a, b)        (lj_ir_set(J, (ot), (a), (b)), lj_opt_fold(J))

#define emitconv(a, dt, st, flags) \
  emitir(IRT(IR_CONV, (dt)), (a), (st)|((dt) << 5)|(flags))

/* -- C type checks ------------------------------------------------------ */

static GCcdata *argv2cdata(jit_State *J, TRef tr, cTValue *o)
{
  GCcdata *cd;
  TRef trtypeid;
  if (!tref_iscdata(tr))
    lj_trace_err(J, LJ_TRERR_BADTYPE);
  cd = cdataV(o);
  /* Specialize to the CTypeID. */
  trtypeid = emitir(IRT(IR_FLOAD, IRT_U16), tr, IRFL_CDATA_CTYPEID);
  emitir(IRTG(IR_EQ, IRT_INT), trtypeid, lj_ir_kint(J, (int32_t)cd->ctypeid));
  return cd;
}

/* Specialize to the CTypeID held by a cdata constructor. */
static CTypeID crec_constructor(jit_State *J, GCcdata *cd, TRef tr)
{
  CTypeID id;
  lua_assert(tref_iscdata(tr) && cd->ctypeid == CTID_CTYPEID);
  id = *(CTypeID *)cdataptr(cd);
```

```
  tr = emitir(IRT(IR_FLOAD, IRT_INT), tr, IRFL_CDATA_INT);
  emitir(IRTG(IR_EQ, IRT_INT), tr, lj_ir_kint(J, (int32_t)id));
  return id;
}

static CTypeID argv2ctype(jit_State *J, TRef tr, cTValue *o)
{
  if (tref_isstr(tr)) {
    GCstr *s = strV(o);
    CPState cp;
    CTypeID oldtop;
    /* Specialize to the string containing the C type declaration. */
    emitir(IRTG(IR_EQ, IRT_STR), tr, lj_ir_kstr(J, s));
    cp.L = J->L;
    cp.cts = ctype_ctsG(J2G(J));
    oldtop = cp.cts->top;
    cp.srcname = strdata(s);
    cp.p = strdata(s);
    cp.param = NULL;
    cp.mode = CPARSE_MODE_ABSTRACT|CPARSE_MODE_NOIMPLICIT;
    if (lj_cparse(&cp) || cp.cts->top > oldtop)  /* Avoid new struct defs. */
      lj_trace_err(J, LJ_TRERR_BADTYPE);
    return cp.val.id;
  } else {
    GCcdata *cd = argv2cdata(J, tr, o);
    return cd->ctypeid == CTID_CTYPEID ? crec_constructor(J, cd, tr) :
                                         cd->ctypeid;
  }
}

/* Convert CType to IRType (if possible). */
static IRType crec_ct2irt(CTState *cts, CType *ct)
{
  if (ctype_isenum(ct->info)) ct = ctype_child(cts, ct);
   if (LJ_LIKELY(ctype_isnum(ct->info))) {
     if ((ct->info & CTF_FP)) {
       if (ct->size == sizeof(double))
         return IRT_NUM;
       else if (ct->size == sizeof(float))
         return IRT_FLOAT;
     } else {
       uint32_t b = lj_fls(ct->size);
       if (b <= 3)
         return IRT_I8 + 2*b + ((ct->info & CTF_UNSIGNED) ? 1 : 0);
     }
   } else if (ctype_isptr(ct->info)) {
     return (LJ_64 && ct->size == 8) ? IRT_P64 : IRT_P32;
   } else if (ctype_iscomplex(ct->info)) {
     if (ct->size == 2*sizeof(double))
       return IRT_NUM;
     else if (ct->size == 2*sizeof(float))
       return IRT_FLOAT;
   }
   return IRT_CDATA;
 }

 /* -- Optimized memory fill and copy ------------------------------------- */

 /* Maximum length and unroll of inlined copy/fill. */
 #define CREC_COPY_MAXUNROLL               16
 #define CREC_COPY_MAXLEN                 128

 #define CREC_FILL_MAXUNROLL               16

 /* Number of windowed registers used for optimized memory copy. */
 #if LJ_TARGET_X86
 #define CREC_COPY_REGWIN                  2
 #elif LJ_TARGET_PPC || LJ_TARGET_MIPS
 #define CREC_COPY_REGWIN                  8
 #else
 #define CREC_COPY_REGWIN                  4
 #endif

 /* List of memory offsets for copy/fill. */
 typedef struct CRecMemList {
   CTSize ofs;               /* Offset in bytes. */
```

```
  IRType tp;                /* Type of load/store. */
  TRef trofs;                /* TRef of interned offset. */
  TRef trval;                /* TRef of load value. */
} CRecMemList;

/* Generate copy list for element-wise struct copy. */
static MSize crec_copy_struct(CRecMemList *ml, CTState *cts, CType *ct)
{
  CTypeID fid = ct->sib;
  MSize mlp = 0;
  while (fid) {
    CType *df = ctype_get(cts, fid);
    fid = df->sib;
    if (ctype_isfield(df->info)) {
      CType *cct;
      IRType tp;
      if (!gcref(df->name)) continue;  /* Ignore unnamed fields. */
      cct = ctype_rawchild(cts, df);  /* Field type. */
      tp = crec_ct2irt(cts, cct);
      if (tp == IRT_CDATA) return 0;  /* NYI: aggregates. */
      if (mlp >= CREC_COPY_MAXUNROLL) return 0;
      ml[mlp].ofs = df->size;
      ml[mlp].tp = tp;
      mlp++;
      if (ctype_iscomplex(cct->info)) {
        if (mlp >= CREC_COPY_MAXUNROLL) return 0;
        ml[mlp].ofs = df->size + (cct->size >> 1);
        ml[mlp].tp = tp;
        mlp++;
      }
    } else if (!ctype_isconstval(df->info)) {
      /* NYI: bitfields and sub-structures. */
      return 0;
    }
  }
  return mlp;
}

/* Generate unrolled copy list, from highest to lowest step size/alignment. */
static MSize crec_copy_unroll(CRecMemList *ml, CTSize len, CTSize step,
                              IRType tp)
{
  CTSize ofs = 0;
  MSize mlp = 0;
  if (tp == IRT_CDATA) tp = IRT_U8 + 2*lj_fls(step);
  do {
    while (ofs + step <= len) {
      if (mlp >= CREC_COPY_MAXUNROLL) return 0;
      ml[mlp].ofs = ofs;
      ml[mlp].tp = tp;
      mlp++;
      ofs += step;
    }
    step >>= 1;
    tp -= 2;
  } while (ofs < len);
  return mlp;
}

/*
** Emit copy list with windowed loads/stores.
** LJ_TARGET_UNALIGNED: may emit unaligned loads/stores (not marked as such).
*/
static void crec_copy_emit(jit_State *J, CRecMemList *ml, MSize mlp,
                           TRef trdst, TRef trsrc)
{
  MSize i, j, rwin = 0;
  for (i = 0, j = 0; i < mlp; ) {
    TRef trofs = lj_ir_kintp(J, ml[i].ofs);
    TRef trsptr = emitir(IRT(IR_ADD, IRT_PTR), trsrc, trofs);
    ml[i].trval = emitir(IRT(IR_XLOAD, ml[i].tp), trsptr, 0);
    ml[i].trofs = trofs;
    i++;
    rwin += (LJ_SOFTFP && ml[i].tp == IRT_NUM) ? 2 : 1;
    if (rwin >= CREC_COPY_REGWIN || i >= mlp) {  /* Flush buffered stores. */
      rwin = 0;
```

```
      for ( ; j < i; j++) {
        TRef trdptr = emitir(IRT(IR_ADD, IRT_PTR), trdst, ml[j].trofs);
        emitir(IRT(IR_XSTORE, ml[j].tp), trdptr, ml[j].trval);
      }
    }
  }
}

/* Optimized memory copy. */
static void crec_copy(jit_State *J, TRef trdst, TRef trsrc, TRef trlen,
                      CType *ct)
{
  if (tref_isk(trlen)) {  /* Length must be constant. */
    CRecMemList ml[CREC_COPY_MAXUNROLL];
    MSize mlp = 0;
    CTSize step = 1, len = (CTSize)IR(tref_ref(trlen))->i;
    IRType tp = IRT_CDATA;
    int needxbar = 0;
    if (len == 0) return;  /* Shortcut. */
    if (len > CREC_COPY_MAXLEN) goto fallback;
    if (ct) {
      CTState *cts = ctype_ctsG(J2G(J));
      lua_assert(ctype_isarray(ct->info) || ctype_isstruct(ct->info));
      if (ctype_isarray(ct->info)) {
        CType *cct = ctype_rawchild(cts, ct);
        tp = crec_ct2irt(cts, cct);
        if (tp == IRT_CDATA) goto rawcopy;
        step = lj_ir_type_size[tp];
        lua_assert((len & (step-1)) == 0);
      } else if ((ct->info & CTF_UNION)) {
        step = (1u << ctype_align(ct->info));
        goto rawcopy;
      } else {
        mlp = crec_copy_struct(ml, cts, ct);
        goto emitcopy;
      }
    } else {
    rawcopy:
      needxbar = 1;
      if (LJ_TARGET_UNALIGNED || step >= CTSIZE_PTR)
        step = CTSIZE_PTR;
    }
    mlp = crec_copy_unroll(ml, len, step, tp);
  emitcopy:
    if (mlp) {
      crec_copy_emit(J, ml, mlp, trdst, trsrc);
      if (needxbar)
        emitir(IRT(IR_XBAR, IRT_NIL), 0, 0);
      return;
    }
  }
fallback:
  /* Call memcpy. Always needs a barrier to disable alias analysis. */
  lj_ir_call(J, IRCALL_memcpy, trdst, trsrc, trlen);
  emitir(IRT(IR_XBAR, IRT_NIL), 0, 0);
}

/* Generate unrolled fill list, from highest to lowest step size/alignment. */
static MSize crec_fill_unroll(CRecMemList *ml, CTSize len, CTSize step)
{
  CTSize ofs = 0;
  MSize mlp = 0;
  IRType tp = IRT_U8 + 2*lj_fls(step);
  do {
    while (ofs + step <= len) {
      if (mlp >= CREC_COPY_MAXUNROLL) return 0;
      ml[mlp].ofs = ofs;
      ml[mlp].tp = tp;
      mlp++;
      ofs += step;
    }
    step >>= 1;
    tp -= 2;
  } while (ofs < len);
  return mlp;
}
```

```
294
295 /*
296 ** Emit stores for fill list.
297 ** LJ_TARGET_UNALIGNED: may emit unaligned stores (not marked as such).
298 */
299 static void crec_fill_emit(jit_State *J, CRecMemList *ml, MSize mlp,
300                            TRef trdst, TRef trfill)
301 {
302   MSize i;
303   for (i = 0; i < mlp; i++) {
304     TRef trofs = lj_ir_kintp(J, ml[i].ofs);
305     TRef trdptr = emitir(IRT(IR_ADD, IRT_PTR), trdst, trofs);
306     emitir(IRT(IR_XSTORE, ml[i].tp), trdptr, trfill);
307   }
308 }
309
310 /* Optimized memory fill. */
311 static void crec_fill(jit_State *J, TRef trdst, TRef trlen, TRef trfill,
312                       CTSize step)
313 {
314   if (tref_isk(trlen)) {  /* Length must be constant. */
315     CRecMemList ml[CREC_FILL_MAXUNROLL];
316     MSize mlp;
317     CTSize len = (CTSize)IR(tref_ref(trlen))->i;
318     if (len == 0) return;  /* Shortcut. */
319     if (LJ_TARGET_UNALIGNED || step >= CTSIZE_PTR)
320       step = CTSIZE_PTR;
321     if (step * CREC_FILL_MAXUNROLL < len) goto fallback;
322     mlp = crec_fill_unroll(ml, len, step);
323     if (!mlp) goto fallback;
324     if (tref_isk(trfill) || ml[0].tp != IRT_U8)
325       trfill = emitconv(trfill, IRT_INT, IRT_U8, 0);
326     if (ml[0].tp != IRT_U8) {  /* Scatter U8 to U16/U32/U64. */
327       if (CTSIZE_PTR == 8 && ml[0].tp == IRT_U64) {
328         if (tref_isk(trfill))  /* Pointless on x64 with zero-extended regs. */
329           trfill = emitconv(trfill, IRT_U64, IRT_U32, 0);
330         trfill = emitir(IRT(IR_MUL, IRT_U64), trfill,
331                         lj_ir_kint64(J, U64x(01010101,01010101)));
332       } else {
333         trfill = emitir(IRTI(IR_MUL), trfill,
334                     lj_ir_kint(J, ml[0].tp == IRT_U16 ? 0x0101 : 0x01010101));
335       }
336     }
337     crec_fill_emit(J, ml, mlp, trdst, trfill);
338   } else {
339 fallback:
340     /* Call memset. Always needs a barrier to disable alias analysis. */
341     lj_ir_call(J, IRCALL_memset, trdst, trfill, trlen);  /* Note: arg order! */
342   }
343   emitir(IRT(IR_XBAR, IRT_NIL), 0, 0);
344 }
345
346 /* -- Convert C type to C type -------------------------------------------- */
347
348 /*
349 ** This code mirrors the code in lj_cconv.c. It performs the same steps
350 ** for the trace recorder that lj_cconv.c does for the interpreter.
351 **
352 ** One major difference is that we can get away with much fewer checks
353 ** here. E.g. checks for casts, constness or correct types can often be
354 ** omitted, even if they might fail. The interpreter subsequently throws
355 ** an error, which aborts the trace.
356 **
357 ** All operations are specialized to their C types, so the on-trace
358 ** outcome must be the same as the outcome in the interpreter. If the
359 ** interpreter doesn't throw an error, then the trace is correct, too.
360 ** Care must be taken not to generate invalid (temporary) IR or to
361 ** trigger asserts.
362 */
363
364 /* Determine whether a passed number or cdata number is non-zero. */
365 static int crec_isnonzero(CType *s, void *p)
366 {
367   if (p == (void *)0)
368     return 0;
369   if (p == (void *)1)
```

```
    return 1;
  if ((s->info & CTF_FP)) {
    if (s->size == sizeof(float))
      return (*(float *)p != 0);
    else
      return (*(double *)p != 0);
  } else {
    if (s->size == 1)
      return (*(uint8_t *)p != 0);
    else if (s->size == 2)
      return (*(uint16_t *)p != 0);
    else if (s->size == 4)
      return (*(uint32_t *)p != 0);
    else
      return (*(uint64_t *)p != 0);
  }
}

static TRef crec_ct_ct(jit_State *J, CType *d, CType *s, TRef dp, TRef sp,
                       void *svisnz)
{
  IRType dt = crec_ct2irt(ctype_ctsG(J2G(J)), d);
  IRType st = crec_ct2irt(ctype_ctsG(J2G(J)), s);
  CTSize dsize = d->size, ssize = s->size;
  CTInfo dinfo = d->info, sinfo = s->info;

  if (ctype_type(dinfo) > CT_MAYCONVERT || ctype_type(sinfo) > CT_MAYCONVERT)
    goto err_conv;

  /*
  ** Note: Unlike lj_cconv_ct_ct(), sp holds the _value_ of pointers and
  ** numbers up to 8 bytes. Otherwise sp holds a pointer.
  */

  switch (cconv_idx2(dinfo, sinfo)) {
  /* Destination is a bool. */
  case CCX(B, B):
    goto xstore;  /* Source operand is already normalized. */
  case CCX(B, I):
  case CCX(B, F):
    if (st != IRT_CDATA) {
      /* Specialize to the result of a comparison against 0. */
      TRef zero = (st == IRT_NUM  || st == IRT_FLOAT) ? lj_ir_knum(J, 0) :
                  (st == IRT_I64 || st == IRT_U64) ? lj_ir_kint64(J, 0) :
                  lj_ir_kint(J, 0);
      int isnz = crec_isnonzero(s, svisnz);
      emitir(IRTG(isnz ? IR_NE : IR_EQ, st), sp, zero);
      sp = lj_ir_kint(J, isnz);
      goto xstore;
    }
    goto err_nyi;

  /* Destination is an integer. */
  case CCX(I, B):
  case CCX(I, I):
  conv_I_I:
    if (dt == IRT_CDATA || st == IRT_CDATA) goto err_nyi;
    /* Extend 32 to 64 bit integer. */
    if (dsize == 8 && ssize < 8 && !(LJ_64 && (sinfo & CTF_UNSIGNED)))
      sp = emitconv(sp, dt, ssize < 4 ? IRT_INT : st,
                    (sinfo & CTF_UNSIGNED) ? 0 : IRCONV_SEXT);
    else if (dsize < 8 && ssize == 8)  /* Truncate from 64 bit integer. */
      sp = emitconv(sp, dsize < 4 ? IRT_INT : dt, st, 0);
    else if (st == IRT_INT)
      sp = lj_opt_narrow_toint(J, sp);
  xstore:
    if (dt == IRT_I64 || dt == IRT_U64) lj_needsplit(J);
    if (dp == 0) return sp;
    emitir(IRT(IR_XSTORE, dt), dp, sp);
    break;
  case CCX(I, C):
    sp = emitir(IRT(IR_XLOAD, st), sp, 0);  /* Load re. */
    /* fallthrough */
  case CCX(I, F):
    if (dt == IRT_CDATA || st == IRT_CDATA) goto err_nyi;
    sp = emitconv(sp, dsize < 4 ? IRT_INT : dt, st, IRCONV_ANY);
```

```
    goto xstore;
  case CCX(I, P):
  case CCX(I, A):
    sinfo = CTINFO(CT_NUM, CTF_UNSIGNED);
    ssize = CTSIZE_PTR;
    st = IRT_UINTP;
    if (((dsize ^ ssize) & 8) == 0) {  /* Must insert no-op type conversion. */
      sp = emitconv(sp, dsize < 4 ? IRT_INT : dt, IRT_PTR, 0);
      goto xstore;
    }
    goto conv_I_I;

  /* Destination is a floating-point number. */
  case CCX(F, B):
  case CCX(F, I):
  conv_F_I:
    if (dt == IRT_CDATA || st == IRT_CDATA) goto err_nyi;
    sp = emitconv(sp, dt, ssize < 4 ? IRT_INT : st, 0);
    goto xstore;
  case CCX(F, C):
    sp = emitir(IRT(IR_XLOAD, st), sp, 0);  /* Load re. */
    /* fallthrough */
  case CCX(F, F):
  conv_F_F:
    if (dt == IRT_CDATA || st == IRT_CDATA) goto err_nyi;
    if (dt != st) sp = emitconv(sp, dt, st, 0);
    goto xstore;

  /* Destination is a complex number. */
  case CCX(C, I):
  case CCX(C, F):
    {  /* Clear im. */
      TRef ptr = emitir(IRT(IR_ADD, IRT_PTR), dp, lj_ir_kintp(J, (dsize >> 1)));
      emitir(IRT(IR_XSTORE, dt), ptr, lj_ir_knum(J, 0));
    }
    /* Convert to re. */
    if ((sinfo & CTF_FP)) goto conv_F_F; else goto conv_F_I;

  case CCX(C, C):
    if (dt == IRT_CDATA || st == IRT_CDATA) goto err_nyi;
    {
      TRef re, im, ptr;
      re = emitir(IRT(IR_XLOAD, st), sp, 0);
      ptr = emitir(IRT(IR_ADD, IRT_PTR), sp, lj_ir_kintp(J, (ssize >> 1)));
      im = emitir(IRT(IR_XLOAD, st), ptr, 0);
      if (dt != st) {
        re = emitconv(re, dt, st, 0);
        im = emitconv(im, dt, st, 0);
      }
      emitir(IRT(IR_XSTORE, dt), dp, re);
      ptr = emitir(IRT(IR_ADD, IRT_PTR), dp, lj_ir_kintp(J, (dsize >> 1)));
      emitir(IRT(IR_XSTORE, dt), ptr, im);
    }
    break;

  /* Destination is a vector. */
  case CCX(V, I):
  case CCX(V, F):
  case CCX(V, C):
  case CCX(V, V):
    goto err_nyi;

  /* Destination is a pointer. */
  case CCX(P, P):
  case CCX(P, A):
  case CCX(P, S):
    /* There are only 32 bit pointers/addresses on 32 bit machines.
    ** Also ok on x64, since all 32 bit ops clear the upper part of the reg.
    */
    goto xstore;
  case CCX(P, I):
    if (st == IRT_CDATA) goto err_nyi;
    if (!LJ_64 && ssize == 8)  /* Truncate from 64 bit integer. */
      sp = emitconv(sp, IRT_U32, st, 0);
    goto xstore;
  case CCX(P, F):
```

```
    if (st == IRT_CDATA) goto err_nyi;
    /* The signed conversion is cheaper. x64 really has 47 bit pointers. */
    sp = emitconv(sp, (LJ_64 && dsize == 8) ? IRT_I64 : IRT_U32,
                  st, IRCONV_ANY);
    goto xstore;

  /* Destination is an array. */
  case CCX(A, A):
  /* Destination is a struct/union. */
  case CCX(S, S):
    if (dp == 0) goto err_conv;
    crec_copy(J, dp, sp, lj_ir_kint(J, dsize), d);
    break;

  default:
  err_conv:
  err_nyi:
    lj_trace_err(J, LJ_TRERR_NYICONV);
    break;
  }
  return 0;
}

/* -- Convert C type to TValue (load) ------------------------------------ */

static TRef crec_tv_ct(jit_State *J, CType *s, CTypeID sid, TRef sp)
{
  CTState *cts = ctype_ctsG(J2G(J));
  IRType t = crec_ct2irt(cts, s);
  CTInfo sinfo = s->info;
  if (ctype_isnum(sinfo)) {
    TRef tr;
    if (t == IRT_CDATA)
      goto err_nyi;  /* NYI: copyval of >64 bit integers. */
    tr = emitir(IRT(IR_XLOAD, t), sp, 0);
    if (t == IRT_FLOAT || t == IRT_U32) {  /* Keep uint32_t/float as numbers. */
      return emitconv(tr, IRT_NUM, t, 0);
    } else if (t == IRT_I64 || t == IRT_U64) {  /* Box 64 bit integer. */
      sp = tr;
      lj_needsplit(J);
    } else if ((sinfo & CTF_BOOL)) {
      /* Assume not equal to zero. Fixup and emit pending guard later. */
      lj_ir_set(J, IRTGI(IR_NE), tr, lj_ir_kint(J, 0));
      J->postproc = LJ_POST_FIXGUARD;
      return TREF_TRUE;
    } else {
      return tr;
    }
  } else if (ctype_isptr(sinfo) || ctype_isenum(sinfo)) {
    sp = emitir(IRT(IR_XLOAD, t), sp, 0);  /* Box pointers and enums. */
  } else if (ctype_isrefarray(sinfo) || ctype_isstruct(sinfo)) {
    cts->L = J->L;
    sid = lj_ctype_intern(cts, CTINFO_REF(sid), CTSIZE_PTR);  /* Create ref. */
  } else if (ctype_iscomplex(sinfo)) {  /* Unbox/box complex. */
    ptrdiff_t esz = (ptrdiff_t)(s->size >> 1);
    TRef ptr, tr1, tr2, dp;
    dp = emitir(IRTG(IR_CNEW, IRT_CDATA), lj_ir_kint(J, sid), TREF_NIL);
    tr1 = emitir(IRT(IR_XLOAD, t), sp, 0);
    ptr = emitir(IRT(IR_ADD, IRT_PTR), sp, lj_ir_kintp(J, esz));
    tr2 = emitir(IRT(IR_XLOAD, t), ptr, 0);
    ptr = emitir(IRT(IR_ADD, IRT_PTR), dp, lj_ir_kintp(J, sizeof(GCcdata)));
    emitir(IRT(IR_XSTORE, t), ptr, tr1);
    ptr = emitir(IRT(IR_ADD, IRT_PTR), dp, lj_ir_kintp(J, sizeof(GCcdata)+esz));
    emitir(IRT(IR_XSTORE, t), ptr, tr2);
    return dp;
  } else {
    /* NYI: copyval of vectors. */
  err_nyi:
    lj_trace_err(J, LJ_TRERR_NYICONV);
  }
  /* Box pointer, ref, enum or 64 bit integer. */
  return emitir(IRTG(IR_CNEWI, IRT_CDATA), lj_ir_kint(J, sid), sp);
}

/* -- Convert TValue to C type (store) ----------------------------------- */

```

```
static TRef crec_ct_tv(jit_State *J, CType *d, TRef dp, TRef sp, cTValue *sval)
{
  CTState *cts = ctype_ctsG(J2G(J));
  CTypeID sid = CTID_P_VOID;
  void *svisnz = 0;
  CType *s;
  if (LJ_LIKELY(tref_isinteger(sp))) {
    sid = CTID_INT32;
    svisnz = (void *)(intptr_t)(tvisint(sval)?(intV(sval)!=0):!tviszero(sval));
  } else if (tref_isnum(sp)) {
    sid = CTID_DOUBLE;
    svisnz = (void *)(intptr_t)(tvisint(sval)?(intV(sval)!=0):!tviszero(sval));
  } else if (tref_isbool(sp)) {
    sp = lj_ir_kint(J, tref_istrue(sp) ? 1 : 0);
    sid = CTID_BOOL;
  } else if (tref_isnil(sp)) {
    sp = lj_ir_kptr(J, NULL);
  } else if (tref_isudata(sp)) {
    GCudata *ud = udataV(sval);
    if (ud->udtype == UDTYPE_IO_FILE) {
      TRef tr = emitir(IRT(IR_FLOAD, IRT_U8), sp, IRFL_UDATA_UDTYPE);
      emitir(IRTGI(IR_EQ), tr, lj_ir_kint(J, UDTYPE_IO_FILE));
      sp = emitir(IRT(IR_FLOAD, IRT_PTR), sp, IRFL_UDATA_FILE);
    } else {
      sp = emitir(IRT(IR_ADD, IRT_PTR), sp, lj_ir_kintp(J, sizeof(GCudata)));
    }
  } else if (tref_isstr(sp)) {
    if (ctype_isenum(d->info)) {  /* Match string against enum constant. */
      GCstr *str = strV(sval);
      CTSize ofs;
      CType *cct = lj_ctype_getfield(cts, d, str, &ofs);
      /* Specialize to the name of the enum constant. */
      emitir(IRTG(IR_EQ, IRT_STR), sp, lj_ir_kstr(J, str));
      if (cct && ctype_isconstval(cct->info)) {
        lua_assert(ctype_child(cts, cct)->size == 4);
        svisnz = (void *)(intptr_t)(ofs != 0);
        sp = lj_ir_kint(J, (int32_t)ofs);
        sid = ctype_cid(cct->info);
      }  /* else: interpreter will throw. */
    } else if (ctype_isrefarray(d->info)) {  /* Copy string to array. */
      lj_trace_err(J, LJ_TRERR_BADTYPE);  /* NYI */
    } else {  /* Otherwise pass the string data as a const char[]. */
      /* Don't use STRREF. It folds with SNEW, which loses the trailing NUL. */
      sp = emitir(IRT(IR_ADD, IRT_PTR), sp, lj_ir_kintp(J, sizeof(GCstr)));
      sid = CTID_A_CCHAR;
    }
  } else if (tref_islightud(sp)) {
#if LJ_64
    sp = emitir(IRT(IR_BAND, IRT_P64), sp,
                lj_ir_kint64(J, U64x(00007fff,ffffffff)));
#endif
  } else {  /* NYI: tref_istab(sp). */
    IRType t;
    sid = argv2cdata(J, sp, sval)->ctypeid;
    s = ctype_raw(cts, sid);
    svisnz = cdataptr(cdataV(sval));
    if (ctype_isfunc(s->info)) {
      sid = lj_ctype_intern(cts, CTINFO(CT_PTR, CTALIGN_PTR|sid), CTSIZE_PTR);
      s = ctype_get(cts, sid);
      t = IRT_PTR;
    } else {
      t = crec_ct2irt(cts, s);
    }
    if (ctype_isptr(s->info)) {
      sp = emitir(IRT(IR_FLOAD, t), sp, IRFL_CDATA_PTR);
      if (ctype_isref(s->info)) {
        svisnz = *(void **)svisnz;
        s = ctype_rawchild(cts, s);
        if (ctype_isenum(s->info)) s = ctype_child(cts, s);
        t = crec_ct2irt(cts, s);
      } else {
        goto doconv;
      }
    } else if (t == IRT_I64 || t == IRT_U64) {
      sp = emitir(IRT(IR_FLOAD, t), sp, IRFL_CDATA_INT64);
      lj_needsplit(J);
```

```
      goto doconv;
    } else if (t == IRT_INT || t == IRT_U32) {
      if (ctype_isenum(s->info)) s = ctype_child(cts, s);
      sp = emitir(IRT(IR_FLOAD, t), sp, IRFL_CDATA_INT);
      goto doconv;
    } else {
      sp = emitir(IRT(IR_ADD, IRT_PTR), sp, lj_ir_kintp(J, sizeof(GCcdata)));
    }
    if (ctype_isnum(s->info) && t != IRT_CDATA)
      sp = emitir(IRT(IR_XLOAD, t), sp, 0);  /* Load number value. */
    goto doconv;
  }
  s = ctype_get(cts, sid);
doconv:
  if (ctype_isenum(d->info)) d = ctype_child(cts, d);
  return crec_ct_ct(J, d, s, dp, sp, svisnz);
}

/* -- C data metamethods ------------------------------------------------- */

/* This would be rather difficult in FOLD, so do it here:
** (base+k)+(idx*sz)+ofs ==> (base+idx*sz)+(ofs+k)
** (base+(idx+k)*sz)+ofs ==> (base+idx*sz)+(ofs+k*sz)
*/
static TRef crec_reassoc_ofs(jit_State *J, TRef tr, ptrdiff_t *ofsp, MSize sz)
{
  IRIns *ir = IR(tref_ref(tr));
  if (LJ_LIKELY(J->flags & JIT_F_OPT_FOLD) && irref_isk(ir->op2) &&
      (ir->o == IR_ADD || ir->o == IR_ADDOV || ir->o == IR_SUBOV)) {
    IRIns *irk = IR(ir->op2);
    ptrdiff_t k;
    if (LJ_64 && irk->o == IR_KINT64)
      k = (ptrdiff_t)ir_kint64(irk)->u64 * sz;
    else
      k = (ptrdiff_t)irk->i * sz;
    if (ir->o == IR_SUBOV) *ofsp -= k; else *ofsp += k;
    tr = ir->op1;  /* Not a TRef, but the caller doesn't care. */
  }
  return tr;
}

/* Record ctype __index/__newindex metamethods. */
static void crec_index_meta(jit_State *J, CTState *cts, CType *ct,
                            RecordFFData *rd)
{
  CTypeID id = ctype_typeid(cts, ct);
  CTValue *tv = lj_ctype_meta(cts, id, rd->data ? MM_newindex : MM_index);
  if (!tv)
    lj_trace_err(J, LJ_TRERR_BADTYPE);
  if (tvisfunc(tv)) {
    J->base[-1] = lj_ir_kfunc(J, funcV(tv)) | TREF_FRAME;
    rd->nres = -1;  /* Pending tailcall. */
  } else if (rd->data == 0 && tvistab(tv) && tref_isstr(J->base[1])) {
    /* Specialize to result of __index lookup. */
    CTValue *o = lj_tab_get(J->L, tabV(tv), &rd->argv[1]);
    J->base[0] = lj_record_constify(J, o);
    if (!J->base[0])
      lj_trace_err(J, LJ_TRERR_BADTYPE);
    /* Always specialize to the key. */
    emitir(IRTG(IR_EQ, IRT_STR), J->base[1], lj_ir_kstr(J, strV(&rd->argv[1])));
  } else {
    /* NYI: resolving of non-function metamethods. */
    /* NYI: non-string keys for __index table. */
    /* NYI: stores to __newindex table. */
    lj_trace_err(J, LJ_TRERR_BADTYPE);
  }
}

void LJ_FASTCALL recff_cdata_index(jit_State *J, RecordFFData *rd)
{
  TRef idx, ptr = J->base[0];
  ptrdiff_t ofs = sizeof(GCcdata);
  GCcdata *cd = argv2cdata(J, ptr, &rd->argv[0]);
  CTState *cts = ctype_ctsG(J2G(J));
  CType *ct = ctype_raw(cts, cd->ctypeid);
  CTypeID sid = 0;
```

```
750
751   /* Resolve pointer or reference for cdata object. */
752   if (ctype_isptr(ct->info)) {
753     IRType t = (LJ_64 && ct->size == 8) ? IRT_P64 : IRT_P32;
754     if (ctype_isref(ct->info)) ct = ctype_rawchild(cts, ct);
755     ptr = emitir(IRT(IR_FLOAD, t), ptr, IRFL_CDATA_PTR);
756     ofs = 0;
757     ptr = crec_reassoc_ofs(J, ptr, &ofs, 1);
758   }
759
760 again:
761   idx = J->base[1];
762   if (tref_isnumber(idx)) {
763     idx = lj_opt_narrow_cindex(J, idx);
764     if (ctype_ispointer(ct->info)) {
765       CTSize sz;
766   integer_key:
767       if ((ct->info & CTF_COMPLEX))
768         idx = emitir(IRT(IR_BAND, IRT_INTP), idx, lj_ir_kintp(J, 1));
769       sz = lj_ctype_size(cts, (sid = ctype_cid(ct->info)));
770       idx = crec_reassoc_ofs(J, idx, &ofs, sz);
771 #if LJ_TARGET_ARM || LJ_TARGET_PPC
772       /* Hoist base add to allow fusion of index/shift into operands. */
773       if (LJ_LIKELY(J->flags & JIT_F_OPT_LOOP) && ofs
774 #if LJ_TARGET_ARM
775           && (sz == 1 || sz == 4)
776 #endif
777           ) {
778         ptr = emitir(IRT(IR_ADD, IRT_PTR), ptr, lj_ir_kintp(J, ofs));
779         ofs = 0;
780       }
781 #endif
782       idx = emitir(IRT(IR_MUL, IRT_INTP), idx, lj_ir_kintp(J, sz));
783       ptr = emitir(IRT(IR_ADD, IRT_PTR), idx, ptr);
784     }
785   } else if (tref_iscdata(idx)) {
786     GCcdata *cdk = cdataV(&rd->argv[1]);
787     CType *ctk = ctype_raw(cts, cdk->ctypeid);
788     IRType t = crec_ct2irt(cts, ctk);
789     if (ctype_ispointer(ct->info) && t >= IRT_I8 && t <= IRT_U64) {
790       if (ctk->size == 8) {
791         idx = emitir(IRT(IR_FLOAD, t), idx, IRFL_CDATA_INT64);
792       } else if (ctk->size == 4) {
793         idx = emitir(IRT(IR_FLOAD, t), idx, IRFL_CDATA_INT);
794       } else {
795         idx = emitir(IRT(IR_ADD, IRT_PTR), idx,
796                      lj_ir_kintp(J, sizeof(GCcdata)));
797         idx = emitir(IRT(IR_XLOAD, t), idx, 0);
798       }
799       if (LJ_64 && ctk->size < sizeof(intptr_t) && !(ctk->info & CTF_UNSIGNED))
800         idx = emitconv(idx, IRT_INTP, IRT_INT, IRCONV_SEXT);
801       if (!LJ_64 && ctk->size > sizeof(intptr_t)) {
802         idx = emitconv(idx, IRT_INTP, t, 0);
803         lj_needsplit(J);
804       }
805       goto integer_key;
806     }
807   } else if (tref_isstr(idx)) {
808     GCstr *name = strV(&rd->argv[1]);
809     if (cd && cd->ctypeid == CTID_CTYPEID)
810       ct = ctype_raw(cts, crec_constructor(J, cd, ptr));
811     if (ctype_isstruct(ct->info)) {
812       CTSize fofs;
813       CType *fct;
814       fct = lj_ctype_getfield(cts, ct, name, &fofs);
815       if (fct) {
816         /* Always specialize to the field name. */
817         emitir(IRTG(IR_EQ, IRT_STR), idx, lj_ir_kstr(J, name));
818         if (ctype_isconstval(fct->info)) {
819           if (fct->size >= 0x80000000u &&
820               (ctype_child(cts, fct)->info & CTF_UNSIGNED)) {
821             J->base[0] = lj_ir_knum(J, (lua_Number)(uint32_t)fct->size);
822             return;
823           }
824           J->base[0] = lj_ir_kint(J, (int32_t)fct->size);
825           return;  /* Interpreter will throw for newindex. */
```

```
826         } else if (ctype_isbitfield(fct->info)) {
827           lj_trace_err(J, LJ_TRERR_NYICONV);
828         } else {
829           lua_assert(ctype_isfield(fct->info));
830           sid = ctype_cid(fct->info);
831         }
832         ofs += (ptrdiff_t)fofs;
833       }
834     } else if (ctype_iscomplex(ct->info)) {
835       if (name->len == 2 &&
836           ((strdata(name)[0] == 'r' && strdata(name)[1] == 'e') ||
837            (strdata(name)[0] == 'i' && strdata(name)[1] == 'm'))) {
838         /* Always specialize to the field name. */
839         emitir(IRTG(IR_EQ, IRT_STR), idx, lj_ir_kstr(J, name));
840         if (strdata(name)[0] == 'i') ofs += (ct->size >> 1);
841         sid = ctype_cid(ct->info);
842       }
843     }
844   }
845   if (!sid) {
846     if (ctype_isptr(ct->info)) {  /* Automatically perform '->'. */
847       CType *cct = ctype_rawchild(cts, ct);
848       if (ctype_isstruct(cct->info)) {
849         ct = cct;
850         cd = NULL;
851         if (tref_isstr(idx)) goto again;
852       }
853     }
854     crec_index_meta(J, cts, ct, rd);
855     return;
856   }
857
858   if (ofs)
859     ptr = emitir(IRT(IR_ADD, IRT_PTR), ptr, lj_ir_kintp(J, ofs));
860
861   /* Resolve reference for field. */
862   ct = ctype_get(cts, sid);
863   if (ctype_isref(ct->info)) {
864     ptr = emitir(IRT(IR_XLOAD, IRT_PTR), ptr, 0);
865     sid = ctype_cid(ct->info);
866     ct = ctype_get(cts, sid);
867   }
868
869   while (ctype_isattrib(ct->info))
870     ct = ctype_child(cts, ct);  /* Skip attributes. */
871
872   if (rd->data == 0) {  /* __index metamethod. */
873     J->base[0] = crec_tv_ct(J, ct, sid, ptr);
874   } else {  /* __newindex metamethod. */
875     rd->nres = 0;
876     J->needsnap = 1;
877     crec_ct_tv(J, ct, ptr, J->base[2], &rd->argv[2]);
878   }
879 }
880
881 /* Record setting a finalizer. */
882 static void crec_finalizer(jit_State *J, TRef trcd, TRef trfin, cTValue *fin)
883 {
884   if (tvisgcv(fin)) {
885     if (!trfin) trfin = lj_ir_kptr(J, gcval(fin));
886   } else if (tvisnil(fin)) {
887     trfin = lj_ir_kptr(J, NULL);
888   } else {
889     lj_trace_err(J, LJ_TRERR_BADTYPE);
890   }
891   lj_ir_call(J, IRCALL_lj_cdata_setfin, trcd,
892              trfin, lj_ir_kint(J, (int32_t)itype(fin)));
893   J->needsnap = 1;
894 }
895
896 /* Record cdata allocation. */
897 static void crec_alloc(jit_State *J, RecordFFData *rd, CTypeID id)
898 {
899   CTState *cts = ctype_ctsG(J2G(J));
900   CTSize sz;
901   CTInfo info = lj_ctype_info(cts, id, &sz);
```

```
  CType *d = ctype_raw(cts, id);
  TRef trcd, trid = lj_ir_kint(J, id);
  cTValue *fin;
  /* Use special instruction to box pointer or 32/64 bit integer. */
  if (ctype_isptr(info) || (ctype_isinteger(info) && (sz == 4 || sz == 8))) {
    TRef sp = J->base[1] ? crec_ct_tv(J, d, 0, J->base[1], &rd->argv[1]) :
              ctype_isptr(info) ? lj_ir_kptr(J, NULL) :
              sz == 4 ? lj_ir_kint(J, 0) :
              (lj_needsplit(J), lj_ir_kint64(J, 0));
    J->base[0] = emitir(IRTG(IR_CNEWI, IRT_CDATA), trid, sp);
    return;
  } else {
    TRef trsz = TREF_NIL;
    if ((info & CTF_VLA)) {  /* Calculate VLA/VLS size at runtime. */
      CTSize sz0, sz1;
      if (!J->base[1] || J->base[2])
        lj_trace_err(J, LJ_TRERR_NYICONV);  /* NYI: init VLA/VLS. */
      trsz = crec_ct_tv(J, ctype_get(cts, CTID_INT32), 0,
                        J->base[1], &rd->argv[1]);
      sz0 = lj_ctype_vlsize(cts, d, 0);
      sz1 = lj_ctype_vlsize(cts, d, 1);
      trsz = emitir(IRTGI(IR_MULOV), trsz, lj_ir_kint(J, (int32_t)(sz1-sz0)));
      trsz = emitir(IRTGI(IR_ADDOV), trsz, lj_ir_kint(J, (int32_t)sz0));
      J->base[1] = 0;  /* Simplify logic below. */
    } else if (ctype_align(info) > CT_MEMALIGN) {
      trsz = lj_ir_kint(J, sz);
    }
    trcd = emitir(IRTG(IR_CNEW, IRT_CDATA), trid, trsz);
    if (sz > 128 || (info & CTF_VLA)) {
      TRef dp;
      CTSize align;
    special:  /* Only handle bulk zero-fill for large/VLA/VLS types. */
      if (J->base[1])
        lj_trace_err(J, LJ_TRERR_NYICONV);  /* NYI: init large/VLA/VLS types. */
      dp = emitir(IRT(IR_ADD, IRT_PTR), trcd, lj_ir_kintp(J, sizeof(GCcdata)));
      if (trsz == TREF_NIL) trsz = lj_ir_kint(J, sz);
      align = ctype_align(info);
      if (align < CT_MEMALIGN) align = CT_MEMALIGN;
      crec_fill(J, dp, trsz, lj_ir_kint(J, 0), (1u << align));
    } else if (J->base[1] && !J->base[2] &&
        !lj_cconv_multi_init(cts, d, &rd->argv[1])) {
      goto single_init;
    } else if (ctype_isarray(d->info)) {
      CType *dc = ctype_rawchild(cts, d);  /* Array element type. */
      CTSize ofs, esize = dc->size;
      TRef sp = 0;
      TValue tv;
      TValue *sval = &tv;
      MSize i;
      tv.u64 = 0;
      if (!(ctype_isnum(dc->info) || ctype_isptr(dc->info)) ||
          esize * CREC_FILL_MAXUNROLL < sz)
        goto special;
      for (i = 1, ofs = 0; ofs < sz; ofs += esize) {
        TRef dp = emitir(IRT(IR_ADD, IRT_PTR), trcd,
                         lj_ir_kintp(J, ofs + sizeof(GCcdata)));
        if (J->base[i]) {
          sp = J->base[i];
          sval = &rd->argv[i];
          i++;
        } else if (i != 2) {
          sp = ctype_isnum(dc->info) ? lj_ir_kint(J, 0) : TREF_NIL;
        }
        crec_ct_tv(J, dc, dp, sp, sval);
      }
    } else if (ctype_isstruct(d->info)) {
      CTypeID fid = d->sib;
      MSize i = 1;
      while (fid) {
        CType *df = ctype_get(cts, fid);
        fid = df->sib;
        if (ctype_isfield(df->info)) {
          CType *dc;
          TRef sp, dp;
          TValue tv;
          TValue *sval = &tv;
```

```
978           setintV(&tv, 0);
979           if (!gcref(df->name)) continue;  /* Ignore unnamed fields. */
980           dc = ctype_rawchild(cts, df);  /* Field type. */
981           if (!(ctype_isnum(dc->info) || ctype_isptr(dc->info) ||
982                 ctype_isenum(dc->info)))
983             lj_trace_err(J, LJ_TRERR_NYICONV);  /* NYI: init aggregates. */
984           if (J->base[i]) {
985             sp = J->base[i];
986             sval = &rd->argv[i];
987             i++;
988           } else {
989             sp = ctype_isptr(dc->info) ? TREF_NIL : lj_ir_kint(J, 0);
990           }
991           dp = emitir(IRT(IR_ADD, IRT_PTR), trcd,
992                       lj_ir_kintp(J, df->size + sizeof(GCcdata)));
993           crec_ct_tv(J, dc, dp, sp, sval);
994         } else if (!ctype_isconstval(df->info)) {
995           /* NYI: init bitfields and sub-structures. */
996           lj_trace_err(J, LJ_TRERR_NYICONV);
997         }
998       }
999     } else {
1000       TRef dp;
1001     single_init:
1002       dp = emitir(IRT(IR_ADD, IRT_PTR), trcd, lj_ir_kintp(J, sizeof(GCcdata)));
1003       if (J->base[1]) {
1004         crec_ct_tv(J, d, dp, J->base[1], &rd->argv[1]);
1005       } else {
1006         TValue tv;
1007         tv.u64 = 0;
1008         crec_ct_tv(J, d, dp, lj_ir_kint(J, 0), &tv);
1009       }
1010     }
1011   }
1012   J->base[0] = trcd;
1013   /* Handle __gc metamethod. */
1014   fin = lj_ctype_meta(cts, id, MM_gc);
1015   if (fin)
1016     crec_finalizer(J, trcd, 0, fin);
1017 }
1018
1019 /* Record argument conversions. */
1020 static TRef crec_call_args(jit_State *J, RecordFFData *rd,
1021                            CTState *cts, CType *ct)
1022 {
1023   TRef args[CCI_NARGS_MAX];
1024   CTypeID fid;
1025   MSize i, n;
1026   TRef tr, *base;
1027   cTValue *o;
1028 #if LJ_TARGET_X86
1029 #if LJ_ABI_WIN
1030   TRef *arg0 = NULL, *arg1 = NULL;
1031 #endif
1032   int ngpr = 0;
1033   if (ctype_cconv(ct->info) == CTCC_THISCALL)
1034     ngpr = 1;
1035   else if (ctype_cconv(ct->info) == CTCC_FASTCALL)
1036     ngpr = 2;
1037 #endif
1038
1039   /* Skip initial attributes. */
1040   fid = ct->sib;
1041   while (fid) {
1042     CType *ctf = ctype_get(cts, fid);
1043     if (!ctype_isattrib(ctf->info)) break;
1044     fid = ctf->sib;
1045   }
1046   args[0] = TREF_NIL;
1047   for (n = 0, base = J->base+1, o = rd->argv+1; *base; n++, base++, o++) {
1048     CTypeID did;
1049     CType *d;
1050
1051     if (n >= CCI_NARGS_MAX)
1052       lj_trace_err(J, LJ_TRERR_NYICALL);
1053
```

```
    if (fid) {  /* Get argument type from field. */
      CType *ctf = ctype_get(cts, fid);
      fid = ctf->sib;
      lua_assert(ctype_isfield(ctf->info));
      did = ctype_cid(ctf->info);
    } else {
      if (!(ct->info & CTF_VARARG))
        lj_trace_err(J, LJ_TRERR_NYICALL);  /* Too many arguments. */
      did = lj_ccall_ctid_vararg(cts, o);  /* Infer vararg type. */
    }
    d = ctype_raw(cts, did);
    if (!(ctype_isnum(d->info) || ctype_isptr(d->info) ||
          ctype_isenum(d->info)))
      lj_trace_err(J, LJ_TRERR_NYICALL);
    tr = crec_ct_tv(J, d, 0, *base, o);
    if (ctype_isinteger_or_bool(d->info)) {
      if (d->size < 4) {
        if ((d->info & CTF_UNSIGNED))
          tr = emitconv(tr, IRT_INT, d->size==1 ? IRT_U8 : IRT_U16, 0);
        else
          tr = emitconv(tr, IRT_INT, d->size==1 ? IRT_I8 : IRT_I16,IRCONV_SEXT);
      }
    } else if (LJ_SOFTFP && ctype_isfp(d->info) && d->size > 4) {
      lj_needsplit(J);
    }
#if LJ_TARGET_X86
    /* 64 bit args must not end up in registers for fastcall/thiscall. */
#if LJ_ABI_WIN
    if (!ctype_isfp(d->info)) {
      /* Sigh, the Windows/x86 ABI allows reordering across 64 bit args. */
      if (tref_typerange(tr, IRT_I64, IRT_U64)) {
        if (ngpr) {
          arg0 = &args[n]; args[n++] = TREF_NIL; ngpr--;
          if (ngpr) {
            arg1 = &args[n]; args[n++] = TREF_NIL; ngpr--;
          }
        }
      } else {
        if (arg0) { *arg0 = tr; arg0 = NULL; n--; continue; }
        if (arg1) { *arg1 = tr; arg1 = NULL; n--; continue; }
        if (ngpr) ngpr--;
      }
    }
#else
    if (!ctype_isfp(d->info) && ngpr) {
      if (tref_typerange(tr, IRT_I64, IRT_U64)) {
        /* No reordering for other x86 ABIs. Simply add alignment args. */
        do { args[n++] = TREF_NIL; } while (--ngpr);
      } else {
        ngpr--;
      }
    }
#endif
#endif
    args[n] = tr;
  }
  tr = args[0];
  for (i = 1; i < n; i++)
    tr = emitir(IRT(IR_CARG, IRT_NIL), tr, args[i]);
  return tr;
}

/* Create a snapshot for the caller, simulating a 'false' return value. */
static void crec_snap_caller(jit_State *J)
{
  lua_State *L = J->L;
  TValue *base = L->base, *top = L->top;
  const BCIns *pc = J->pc;
  TRef ftr = J->base[-1];
  ptrdiff_t delta;
  if (!frame_islua(base-1) || J->framedepth <= 0)
    lj_trace_err(J, LJ_TRERR_NYICALL);
  J->pc = frame_pc(base-1); delta = 1+LJ_FR2+bc_a(J->pc[-1]);
  L->top = base; L->base = base - delta;
  J->base[-1] = TREF_FALSE;
  J->base -= delta; J->baseslot -= (BCReg)delta;
```

```
  J->maxslot = (BCReg)delta; J->framedepth--;
  lj_snap_add(J);
  L->base = base; L->top = top;
  J->framedepth++; J->maxslot = 1;
  J->base += delta; J->baseslot += (BCReg)delta;
  J->base[-1] = ftr; J->pc = pc;
}

/* Record function call. */
static int crec_call(jit_State *J, RecordFFData *rd, GCcdata *cd)
{
  CTState *cts = ctype_ctsG(J2G(J));
  CType *ct = ctype_raw(cts, cd->ctypeid);
  IRType tp = IRT_PTR;
  if (ctype_isptr(ct->info)) {
    tp = (LJ_64 && ct->size == 8) ? IRT_P64 : IRT_P32;
    ct = ctype_rawchild(cts, ct);
  }
  if (ctype_isfunc(ct->info)) {
    TRef func = emitir(IRT(IR_FLOAD, tp), J->base[0], IRFL_CDATA_PTR);
    CType *ctr = ctype_rawchild(cts, ct);
    IRType t = crec_ct2irt(cts, ctr);
    TRef tr;
    TValue tv;
    /* Check for blacklisted C functions that might call a callback. */
    setlightudV(&tv,
                cdata_getptr(cdataptr(cd), (LJ_64 && tp == IRT_P64) ? 8 : 4));
    if (tvistrue(lj_tab_get(J->L, cts->miscmap, &tv)))
      lj_trace_err(J, LJ_TRERR_BLACKL);
    if (ctype_isvoid(ctr->info)) {
      t = IRT_NIL;
      rd->nres = 0;
    } else if (!(ctype_isnum(ctr->info) || ctype_isptr(ctr->info) ||
                 ctype_isenum(ctr->info)) || t == IRT_CDATA) {
      lj_trace_err(J, LJ_TRERR_NYICALL);
    }
    if ((ct->info & CTF_VARARG)
#if LJ_TARGET_X86
        || ctype_cconv(ct->info) != CTCC_CDECL
#endif
        )
      func = emitir(IRT(IR_CARG, IRT_NIL), func,
                    lj_ir_kint(J, ctype_typeid(cts, ct)));
    tr = emitir(IRT(IR_CALLXS, t), crec_call_args(J, rd, cts, ct), func);
    if (ctype_isbool(ctr->info)) {
      if (frame_islua(J->L->base-1) && bc_b(frame_pc(J->L->base-1)[-1]) == 1) {
        /* Don't check result if ignored. */
        tr = TREF_NIL;
      } else {
        crec_snap_caller(J);
#if LJ_TARGET_X86ORX64
        /* Note: only the x86/x64 backend supports U8 and only for EQ(tr, 0). */
        lj_ir_set(J, IRTG(IR_NE, IRT_U8), tr, lj_ir_kint(J, 0));
#else
        lj_ir_set(J, IRTGI(IR_NE), tr, lj_ir_kint(J, 0));
#endif
        J->postproc = LJ_POST_FIXGUARDSNAP;
        tr = TREF_TRUE;
      }
    } else if (t == IRT_PTR || (LJ_64 && t == IRT_P32) ||
               t == IRT_I64 || t == IRT_U64 || ctype_isenum(ctr->info)) {
      TRef trid = lj_ir_kint(J, ctype_cid(ct->info));
      tr = emitir(IRTG(IR_CNEWI, IRT_CDATA), trid, tr);
      if (t == IRT_I64 || t == IRT_U64) lj_needsplit(J);
    } else if (t == IRT_FLOAT || t == IRT_U32) {
      tr = emitconv(tr, IRT_NUM, t, 0);
    } else if (t == IRT_I8 || t == IRT_I16) {
      tr = emitconv(tr, IRT_INT, t, IRCONV_SEXT);
    } else if (t == IRT_U8 || t == IRT_U16) {
      tr = emitconv(tr, IRT_INT, t, 0);
    }
    J->base[0] = tr;
    J->needsnap = 1;
    return 1;
  }
  return 0;
```

```
}

void LJ_FASTCALL recff_cdata_call(jit_State *J, RecordFFData *rd)
{
  CTState *cts = ctype_ctsG(J2G(J));
  GCcdata *cd = argv2cdata(J, J->base[0], &rd->argv[0]);
  CTypeID id = cd->ctypeid;
  CType *ct;
  CTValue *tv;
  MMS mm = MM_call;
  if (id == CTID_CTYPEID) {
    id = crec_constructor(J, cd, J->base[0]);
    mm = MM_new;
  } else if (crec_call(J, rd, cd)) {
    return;
  }
  /* Record ctype __call/__new metamethod. */
  ct = ctype_raw(cts, id);
  tv = lj_ctype_meta(cts, ctype_isptr(ct->info) ? ctype_cid(ct->info) : id, mm);
  if (tv) {
    if (tvisfunc(tv)) {
      J->base[-1] = lj_ir_kfunc(J, funcV(tv)) | TREF_FRAME;
      rd->nres = -1;  /* Pending tailcall. */
      return;
    }
  } else if (mm == MM_new) {
    crec_alloc(J, rd, id);
    return;
  }
  /* No metamethod or NYI: non-function metamethods. */
  lj_trace_err(J, LJ_TRERR_BADTYPE);
}

static TRef crec_arith_int64(jit_State *J, TRef *sp, CType **s, MMS mm)
{
  if (ctype_isnum(s[0]->info) && ctype_isnum(s[1]->info)) {
    IRType dt;
    CTypeID id;
    TRef tr;
    MSize i;
    IROp op;
    lj_needsplit(J);
    if (((s[0]->info & CTF_UNSIGNED) && s[0]->size == 8) ||
        ((s[1]->info & CTF_UNSIGNED) && s[1]->size == 8)) {
      dt = IRT_U64; id = CTID_UINT64;
    } else {
      dt = IRT_I64; id = CTID_INT64;
      if (mm < MM_add &&
          !((s[0]->info | s[1]->info) & CTF_FP) &&
          s[0]->size == 4 && s[1]->size == 4) {  /* Try to narrow comparison. */
        if (!((s[0]->info ^ s[1]->info) & CTF_UNSIGNED) ||
            (tref_isk(sp[1]) && IR(tref_ref(sp[1]))->i >= 0)) {
          dt = (s[0]->info & CTF_UNSIGNED) ? IRT_U32 : IRT_INT;
          goto comp;
        } else if (tref_isk(sp[0]) && IR(tref_ref(sp[0]))->i >= 0) {
          dt = (s[1]->info & CTF_UNSIGNED) ? IRT_U32 : IRT_INT;
          goto comp;
        }
      }
    }
    for (i = 0; i < 2; i++) {
      IRType st = tref_type(sp[i]);
      if (st == IRT_NUM || st == IRT_FLOAT)
        sp[i] = emitconv(sp[i], dt, st, IRCONV_ANY);
      else if (!(st == IRT_I64 || st == IRT_U64))
        sp[i] = emitconv(sp[i], dt, IRT_INT,
                         (s[i]->info & CTF_UNSIGNED) ? 0 : IRCONV_SEXT);
    }
    if (mm < MM_add) {
    comp:
      /* Assume true comparison. Fixup and emit pending guard later. */
      if (mm == MM_eq) {
        op = IR_EQ;
      } else {
        op = mm == MM_lt ? IR_LT : IR_LE;
        if (dt == IRT_U32 || dt == IRT_U64)
```

```
1282            op += (IR_ULT-IR_LT);
1283          }
1284          lj_ir_set(J, IRTG(op, dt), sp[0], sp[1]);
1285          J->postproc = LJ_POST_FIXGUARD;
1286          return TREF_TRUE;
1287        } else {
1288          tr = emitir(IRT(mm+(int)IR_ADD-(int)MM_add, dt), sp[0], sp[1]);
1289        }
1290        return emitir(IRTG(IR_CNEWI, IRT_CDATA), lj_ir_kint(J, id), tr);
1291      }
1292      return 0;
1293    }
1294
1295    static TRef crec_arith_ptr(jit_State *J, TRef *sp, CType **s, MMS mm)
1296    {
1297      CTState *cts = ctype_ctsG(J2G(J));
1298      CType *ctp = s[0];
1299      if (ctype_isptr(ctp->info) || ctype_isrefarray(ctp->info)) {
1300        if ((mm == MM_sub || mm == MM_eq || mm == MM_lt || mm == MM_le) &&
1301            (ctype_isptr(s[1]->info) || ctype_isrefarray(s[1]->info))) {
1302          if (mm == MM_sub) {  /* Pointer difference. */
1303            TRef tr;
1304            CTSize sz = lj_ctype_size(cts, ctype_cid(ctp->info));
1305            if (sz == 0 || (sz & (sz-1)) != 0)
1306              return 0;  /* NYI: integer division. */
1307            tr = emitir(IRT(IR_SUB, IRT_INTP), sp[0], sp[1]);
1308            tr = emitir(IRT(IR_BSAR, IRT_INTP), tr, lj_ir_kint(J, lj_fls(sz)));
1309    #if LJ_64
1310            tr = emitconv(tr, IRT_NUM, IRT_INTP, 0);
1311    #endif
1312            return tr;
1313          } else {  /* Pointer comparison (unsigned). */
1314            /* Assume true comparison. Fixup and emit pending guard later. */
1315            IROp op = mm == MM_eq ? IR_EQ : mm == MM_lt ? IR_ULT : IR_ULE;
1316            lj_ir_set(J, IRTG(op, IRT_PTR), sp[0], sp[1]);
1317            J->postproc = LJ_POST_FIXGUARD;
1318            return TREF_TRUE;
1319          }
1320        }
1321        if (!((mm == MM_add || mm == MM_sub) && ctype_isnum(s[1]->info)))
1322          return 0;
1323      } else if (mm == MM_add && ctype_isnum(ctp->info) &&
1324                 (ctype_isptr(s[1]->info) || ctype_isrefarray(s[1]->info))) {
1325        TRef tr = sp[0]; sp[0] = sp[1]; sp[1] = tr;  /* Swap pointer and index. */
1326        ctp = s[1];
1327      } else {
1328        return 0;
1329      }
1330      {
1331        TRef tr = sp[1];
1332        IRType t = tref_type(tr);
1333        CTSize sz = lj_ctype_size(cts, ctype_cid(ctp->info));
1334        CTypeID id;
1335    #if LJ_64
1336        if (t == IRT_NUM || t == IRT_FLOAT)
1337          tr = emitconv(tr, IRT_INTP, t, IRCONV_ANY);
1338        else if (!(t == IRT_I64 || t == IRT_U64))
1339          tr = emitconv(tr, IRT_INTP, IRT_INT,
1340                        ((t - IRT_I8) & 1) ? 0 : IRCONV_SEXT);
1341    #else
1342        if (!tref_typerange(sp[1], IRT_I8, IRT_U32)) {
1343          tr = emitconv(tr, IRT_INTP, t,
1344                        (t == IRT_NUM || t == IRT_FLOAT) ? IRCONV_ANY : 0);
1345        }
1346    #endif
1347        tr = emitir(IRT(IR_MUL, IRT_INTP), tr, lj_ir_kintp(J, sz));
1348        tr = emitir(IRT(mm+(int)IR_ADD-(int)MM_add, IRT_PTR), sp[0], tr);
1349        id = lj_ctype_intern(cts, CTINFO(CT_PTR, CTALIGN_PTR|ctype_cid(ctp->info)),
1350                             CTSIZE_PTR);
1351        return emitir(IRTG(IR_CNEWI, IRT_CDATA), lj_ir_kint(J, id), tr);
1352      }
1353    }
1354
1355    /* Record ctype arithmetic metamethods. */
1356    static TRef crec_arith_meta(jit_State *J, TRef *sp, CType **s, CTState *cts,
1357                                RecordFFData *rd)
```

```
{
  CTValue *tv = NULL;
  if (J->base[0]) {
    if (tviscdata(&rd->argv[0])) {
      CTypeID id = argv2cdata(J, J->base[0], &rd->argv[0])->ctypeid;
      CType *ct = ctype_raw(cts, id);
      if (ctype_isptr(ct->info)) id = ctype_cid(ct->info);
      tv = lj_ctype_meta(cts, id, (MMS)rd->data);
    }
    if (!tv && J->base[1] && tviscdata(&rd->argv[1])) {
      CTypeID id = argv2cdata(J, J->base[1], &rd->argv[1])->ctypeid;
      CType *ct = ctype_raw(cts, id);
      if (ctype_isptr(ct->info)) id = ctype_cid(ct->info);
      tv = lj_ctype_meta(cts, id, (MMS)rd->data);
    }
  }
  if (tv) {
    if (tvisfunc(tv)) {
      J->base[-1] = lj_ir_kfunc(J, funcV(tv)) | TREF_FRAME;
      rd->nres = -1;  /* Pending tailcall. */
      return 0;
    }  /* NYI: non-function metamethods. */
  } else if ((MMS)rd->data == MM_eq) {  /* Fallback cdata pointer comparison. */
    if (sp[0] && sp[1] && ctype_isnum(s[0]->info) == ctype_isnum(s[1]->info)) {
      /* Assume true comparison. Fixup and emit pending guard later. */
      lj_ir_set(J, IRTG(IR_EQ, IRT_PTR), sp[0], sp[1]);
      J->postproc = LJ_POST_FIXGUARD;
      return TREF_TRUE;
    } else {
      return TREF_FALSE;
    }
  }
  lj_trace_err(J, LJ_TRERR_BADTYPE);
  return 0;
}

void LJ_FASTCALL recff_cdata_arith(jit_State *J, RecordFFData *rd)
{
  CTState *cts = ctype_ctsG(J2G(J));
  TRef sp[2];
  CType *s[2];
  MSize i;
  for (i = 0; i < 2; i++) {
    TRef tr = J->base[i];
    CType *ct = ctype_get(cts, CTID_DOUBLE);
    if (!tr) {
      lj_trace_err(J, LJ_TRERR_BADTYPE);
    } else if (tref_iscdata(tr)) {
      CTypeID id = argv2cdata(J, tr, &rd->argv[i])->ctypeid;
      IRType t;
      ct = ctype_raw(cts, id);
      t = crec_ct2irt(cts, ct);
      if (ctype_isptr(ct->info)) {  /* Resolve pointer or reference. */
        tr = emitir(IRT(IR_FLOAD, t), tr, IRFL_CDATA_PTR);
        if (ctype_isref(ct->info)) {
          ct = ctype_rawchild(cts, ct);
          t = crec_ct2irt(cts, ct);
        }
      } else if (t == IRT_I64 || t == IRT_U64) {
        tr = emitir(IRT(IR_FLOAD, t), tr, IRFL_CDATA_INT64);
        lj_needsplit(J);
        goto ok;
      } else if (t == IRT_INT || t == IRT_U32) {
        tr = emitir(IRT(IR_FLOAD, t), tr, IRFL_CDATA_INT);
        if (ctype_isenum(ct->info)) ct = ctype_child(cts, ct);
        goto ok;
      } else if (ctype_isfunc(ct->info)) {
        tr = emitir(IRT(IR_FLOAD, IRT_PTR), tr, IRFL_CDATA_PTR);
        ct = ctype_get(cts,
          lj_ctype_intern(cts, CTINFO(CT_PTR, CTALIGN_PTR|id), CTSIZE_PTR));
        goto ok;
      } else {
        tr = emitir(IRT(IR_ADD, IRT_PTR), tr, lj_ir_kintp(J, sizeof(GCcdata)));
      }
      if (ctype_isenum(ct->info)) ct = ctype_child(cts, ct);
      if (ctype_isnum(ct->info)) {
```

```
        if (t == IRT_CDATA) {
          tr = 0;
        } else {
          if (t == IRT_I64 || t == IRT_U64) lj_needsplit(J);
          tr = emitir(IRT(IR_XLOAD, t), tr, 0);
        }
      }
    } else if (tref_isnil(tr)) {
      tr = lj_ir_kptr(J, NULL);
      ct = ctype_get(cts, CTID_P_VOID);
    } else if (tref_isinteger(tr)) {
      ct = ctype_get(cts, CTID_INT32);
    } else if (tref_isstr(tr)) {
      TRef tr2 = J->base[1-i];
      CTypeID id = argv2cdata(J, tr2, &rd->argv[1-i])->ctypeid;
      ct = ctype_raw(cts, id);
      if (ctype_isenum(ct->info)) {  /* Match string against enum constant. */
        GCstr *str = strV(&rd->argv[i]);
        CTSize ofs;
        CType *cct = lj_ctype_getfield(cts, ct, str, &ofs);
        if (cct && ctype_isconstval(cct->info)) {
          /* Specialize to the name of the enum constant. */
          emitir(IRTG(IR_EQ, IRT_STR), tr, lj_ir_kstr(J, str));
          ct = ctype_child(cts, cct);
          tr = lj_ir_kint(J, (int32_t)ofs);
        } else {  /* Interpreter will throw or return false. */
          ct = ctype_get(cts, CTID_P_VOID);
        }
      } else if (ctype_isptr(ct->info)) {
        tr = emitir(IRT(IR_ADD, IRT_PTR), tr, lj_ir_kintp(J, sizeof(GCstr)));
      } else {
        ct = ctype_get(cts, CTID_P_VOID);
      }
    } else if (!tref_isnum(tr)) {
      tr = 0;
      ct = ctype_get(cts, CTID_P_VOID);
    }
  ok:
    s[i] = ct;
    sp[i] = tr;
  }
  {
    TRef tr;
    if (!(tr = crec_arith_int64(J, sp, s, (MMS)rd->data)) &&
        !(tr = crec_arith_ptr(J, sp, s, (MMS)rd->data)) &&
        !(tr = crec_arith_meta(J, sp, s, cts, rd)))
      return;
    J->base[0] = tr;
    /* Fixup cdata comparisons, too. Avoids some cdata escapes. */
    if (J->postproc == LJ_POST_FIXGUARD && frame_iscont(J->L->base-1) &&
        !irt_isguard(J->guardemit)) {
      const BCIns *pc = frame_contpc(J->L->base-1) - 1;
      if (bc_op(*pc) <= BC_ISNEP) {
        J2G(J)->tmptv.u64 = (uint64_t)(uintptr_t)pc;
        J->postproc = LJ_POST_FIXCOMP;
      }
    }
  }
}

/* -- C library namespace metamethods ------------------------------------ */

void LJ_FASTCALL recff_clib_index(jit_State *J, RecordFFData *rd)
{
  CTState *cts = ctype_ctsG(J2G(J));
  if (tref_isudata(J->base[0]) && tref_isstr(J->base[1]) &&
      udataV(&rd->argv[0])->udtype == UDTYPE_FFI_CLIB) {
    CLibrary *cl = (CLibrary *)uddata(udataV(&rd->argv[0]));
    GCstr *name = strV(&rd->argv[1]);
    CType *ct;
    CTypeID id = lj_ctype_getname(cts, &ct, name, CLNS_INDEX);
    cTValue *tv = lj_tab_getstr(cl->cache, name);
    rd->nres = rd->data;
    if (id && tv && !tvisnil(tv)) {
      /* Specialize to the symbol name and make the result a constant. */
      emitir(IRTG(IR_EQ, IRT_STR), J->base[1], lj_ir_kstr(J, name));
```

```
      if (ctype_isconstval(ct->info)) {
        if (ct->size >= 0x80000000u &&
            (ctype_child(cts, ct)->info & CTF_UNSIGNED))
          J->base[0] = lj_ir_knum(J, (lua_Number)(uint32_t)ct->size);
        else
          J->base[0] = lj_ir_kint(J, (int32_t)ct->size);
      } else if (ctype_isextern(ct->info)) {
        CTypeID sid = ctype_cid(ct->info);
        void *sp = *(void **)cdataptr(cdataV(tv));
        TRef ptr;
        ct = ctype_raw(cts, sid);
        if (LJ_64 && !checkptr32(sp))
          ptr = lj_ir_kintp(J, (uintptr_t)sp);
        else
          ptr = lj_ir_kptr(J, sp);
        if (rd->data) {
          J->base[0] = crec_tv_ct(J, ct, sid, ptr);
        } else {
          J->needsnap = 1;
          crec_ct_tv(J, ct, ptr, J->base[2], &rd->argv[2]);
        }
      } else {
        J->base[0] = lj_ir_kgc(J, obj2gco(cdataV(tv)), IRT_CDATA);
      }
    } else {
      lj_trace_err(J, LJ_TRERR_NOCACHE);
    }
  }  /* else: interpreter will throw. */
}

/* -- FFI library functions ---------------------------------------------- */

static TRef crec_toint(jit_State *J, CTState *cts, TRef sp, TValue *sval)
{
  return crec_ct_tv(J, ctype_get(cts, CTID_INT32), 0, sp, sval);
}

void LJ_FASTCALL recff_ffi_new(jit_State *J, RecordFFData *rd)
{
  crec_alloc(J, rd, argv2ctype(J, J->base[0], &rd->argv[0]));
}

void LJ_FASTCALL recff_ffi_errno(jit_State *J, RecordFFData *rd)
{
  UNUSED(rd);
  if (J->base[0])
    lj_trace_err(J, LJ_TRERR_NYICALL);
  J->base[0] = lj_ir_call(J, IRCALL_lj_vm_errno);
}

void LJ_FASTCALL recff_ffi_string(jit_State *J, RecordFFData *rd)
{
  CTState *cts = ctype_ctsG(J2G(J));
  TRef tr = J->base[0];
  if (tr) {
    TRef trlen = J->base[1];
    if (!tref_isnil(trlen)) {
      trlen = crec_toint(J, cts, trlen, &rd->argv[1]);
      tr = crec_ct_tv(J, ctype_get(cts, CTID_P_CVOID), 0, tr, &rd->argv[0]);
    } else {
      tr = crec_ct_tv(J, ctype_get(cts, CTID_P_CCHAR), 0, tr, &rd->argv[0]);
      trlen = lj_ir_call(J, IRCALL_strlen, tr);
    }
    J->base[0] = emitir(IRT(IR_XSNEW, IRT_STR), tr, trlen);
  }  /* else: interpreter will throw. */
}

void LJ_FASTCALL recff_ffi_copy(jit_State *J, RecordFFData *rd)
{
  CTState *cts = ctype_ctsG(J2G(J));
  TRef trdst = J->base[0], trsrc = J->base[1], trlen = J->base[2];
  if (trdst && trsrc && (trlen || tref_isstr(trsrc))) {
    trdst = crec_ct_tv(J, ctype_get(cts, CTID_P_VOID), 0, trdst, &rd->argv[0]);
    trsrc = crec_ct_tv(J, ctype_get(cts, CTID_P_CVOID), 0, trsrc, &rd->argv[1]);
    if (trlen) {
      trlen = crec_toint(J, cts, trlen, &rd->argv[2]);
```

```
    } else {
      trlen = emitir(IRTI(IR_FLOAD), J->base[1], IRFL_STR_LEN);
      trlen = emitir(IRTI(IR_ADD), trlen, lj_ir_kint(J, 1));
    }
    rd->nres = 0;
    crec_copy(J, trdst, trsrc, trlen, NULL);
  }  /* else: interpreter will throw. */
}

void LJ_FASTCALL recff_ffi_fill(jit_State *J, RecordFFData *rd)
{
  CTState *cts = ctype_ctsG(J2G(J));
  TRef trdst = J->base[0], trlen = J->base[1], trfill = J->base[2];
  if (trdst && trlen) {
    CTSize step = 1;
    if (tviscdata(&rd->argv[0])) {  /* Get alignment of original destination. */
      CTSize sz;
      CType *ct = ctype_raw(cts, cdataV(&rd->argv[0])->ctypeid);
      if (ctype_isptr(ct->info))
        ct = ctype_rawchild(cts, ct);
      step = (1u<<ctype_align(lj_ctype_info(cts, ctype_typeid(cts, ct), &sz)));
    }
    trdst = crec_ct_tv(J, ctype_get(cts, CTID_P_VOID), 0, trdst, &rd->argv[0]);
    trlen = crec_toint(J, cts, trlen, &rd->argv[1]);
    if (trfill)
      trfill = crec_toint(J, cts, trfill, &rd->argv[2]);
    else
      trfill = lj_ir_kint(J, 0);
    rd->nres = 0;
    crec_fill(J, trdst, trlen, trfill, step);
  }  /* else: interpreter will throw. */
}

void LJ_FASTCALL recff_ffi_typeof(jit_State *J, RecordFFData *rd)
{
  if (tref_iscdata(J->base[0])) {
    TRef trid = lj_ir_kint(J, argv2ctype(J, J->base[0], &rd->argv[0]));
    J->base[0] = emitir(IRTG(IR_CNEWI, IRT_CDATA),
                        lj_ir_kint(J, CTID_CTYPEID), trid);
  } else {
    setfuncV(J->L, &J->errinfo, J->fn);
    lj_trace_err_info(J, LJ_TRERR_NYIFFU);
  }
}

void LJ_FASTCALL recff_ffi_istype(jit_State *J, RecordFFData *rd)
{
  argv2ctype(J, J->base[0], &rd->argv[0]);
  if (tref_iscdata(J->base[1])) {
    argv2ctype(J, J->base[1], &rd->argv[1]);
    J->postproc = LJ_POST_FIXBOOL;
    J->base[0] = TREF_TRUE;
  } else {
    J->base[0] = TREF_FALSE;
  }
}

void LJ_FASTCALL recff_ffi_abi(jit_State *J, RecordFFData *rd)
{
  if (tref_isstr(J->base[0])) {
    /* Specialize to the ABI string to make the boolean result a constant. */
    emitir(IRTG(IR_EQ, IRT_STR), J->base[0], lj_ir_kstr(J, strV(&rd->argv[0])));
    J->postproc = LJ_POST_FIXBOOL;
    J->base[0] = TREF_TRUE;
  } else {
    lj_trace_err(J, LJ_TRERR_BADTYPE);
  }
}

/* Record ffi.sizeof(), ffi.alignof(), ffi.offsetof(). */
void LJ_FASTCALL recff_ffi_xof(jit_State *J, RecordFFData *rd)
{
  CTypeID id = argv2ctype(J, J->base[0], &rd->argv[0]);
  if (rd->data == FF_ffi_sizeof) {
    CType *ct = lj_ctype_rawref(ctype_ctsG(J2G(J)), id);
    if (ctype_isvltype(ct->info))
```

```
1662        lj_trace_err(J, LJ_TRERR_BADTYPE);
1663    } else if (rd->data == FF_ffi_offsetof) {  /* Specialize to the field name. */
1664      if (!tref_isstr(J->base[1]))
1665        lj_trace_err(J, LJ_TRERR_BADTYPE);
1666      emitir(IRTG(IR_EQ, IRT_STR), J->base[1], lj_ir_kstr(J, strV(&rd->argv[1])));
1667      rd->nres = 3;  /* Just in case. */
1668    }
1669    J->postproc = LJ_POST_FIXCONST;
1670    J->base[0] = J->base[1] = J->base[2] = TREF_NIL;
1671  }
1672
1673  void LJ_FASTCALL recff_ffi_gc(jit_State *J, RecordFFData *rd)
1674  {
1675    argv2cdata(J, J->base[0], &rd->argv[0]);
1676    if (!J->base[1])
1677      lj_trace_err(J, LJ_TRERR_BADTYPE);
1678    crec_finalizer(J, J->base[0], J->base[1], &rd->argv[1]);
1679  }
1680
1681  /* -- 64 bit bit.* library functions ------------------------------------ */
1682
1683  /* Determine bit operation type from argument type. */
1684  static CTypeID crec_bit64_type(CTState *cts, cTValue *tv)
1685  {
1686    if (tviscdata(tv)) {
1687      CType *ct = lj_ctype_rawref(cts, cdataV(tv)->ctypeid);
1688      if (ctype_isenum(ct->info)) ct = ctype_child(cts, ct);
1689      if ((ct->info & (CTMASK_NUM|CTF_BOOL|CTF_FP|CTF_UNSIGNED)) ==
1690          CTINFO(CT_NUM, CTF_UNSIGNED) && ct->size == 8)
1691        return CTID_UINT64;  /* Use uint64_t, since it has the highest rank. */
1692      return CTID_INT64;  /* Otherwise use int64_t. */
1693    }
1694    return 0;  /* Use regular 32 bit ops. */
1695  }
1696
1697  void LJ_FASTCALL recff_bit64_tobit(jit_State *J, RecordFFData *rd)
1698  {
1699    CTState *cts = ctype_ctsG(J2G(J));
1700    TRef tr = crec_ct_tv(J, ctype_get(cts, CTID_INT64), 0,
1701                         J->base[0], &rd->argv[0]);
1702    if (!tref_isinteger(tr))
1703      tr = emitconv(tr, IRT_INT, tref_type(tr), 0);
1704    J->base[0] = tr;
1705  }
1706
1707  int LJ_FASTCALL recff_bit64_unary(jit_State *J, RecordFFData *rd)
1708  {
1709    CTState *cts = ctype_ctsG(J2G(J));
1710    CTypeID id = crec_bit64_type(cts, &rd->argv[0]);
1711    if (id) {
1712      TRef tr = crec_ct_tv(J, ctype_get(cts, id), 0, J->base[0], &rd->argv[0]);
1713      tr = emitir(IRT(rd->data, id-CTID_INT64+IRT_I64), tr, 0);
1714      J->base[0] = emitir(IRTG(IR_CNEWI, IRT_CDATA), lj_ir_kint(J, id), tr);
1715      return 1;
1716    }
1717    return 0;
1718  }
1719
1720  int LJ_FASTCALL recff_bit64_nary(jit_State *J, RecordFFData *rd)
1721  {
1722    CTState *cts = ctype_ctsG(J2G(J));
1723    CTypeID id = 0;
1724    MSize i;
1725    for (i = 0; J->base[i] != 0; i++) {
1726      CTypeID aid = crec_bit64_type(cts, &rd->argv[i]);
1727      if (id < aid) id = aid;  /* Determine highest type rank of all arguments. */
1728    }
1729    if (id) {
1730      CType *ct = ctype_get(cts, id);
1731      uint32_t ot = IRT(rd->data, id-CTID_INT64+IRT_I64);
1732      TRef tr = crec_ct_tv(J, ct, 0, J->base[0], &rd->argv[0]);
1733      for (i = 1; J->base[i] != 0; i++) {
1734        TRef tr2 = crec_ct_tv(J, ct, 0, J->base[i], &rd->argv[i]);
1735        tr = emitir(ot, tr, tr2);
1736      }
1737      J->base[0] = emitir(IRTG(IR_CNEWI, IRT_CDATA), lj_ir_kint(J, id), tr);
```

```
    return 1;
  }
  return 0;
}

int LJ_FASTCALL recff_bit64_shift(jit_State *J, RecordFFData *rd)
{
  CTState *cts = ctype_ctsG(J2G(J));
  CTypeID id;
  TRef tsh = 0;
  if (J->base[0] && tref_iscdata(J->base[1])) {
    tsh = crec_ct_tv(J, ctype_get(cts, CTID_INT64), 0,
                     J->base[1], &rd->argv[1]);
    if (!tref_isinteger(tsh))
      tsh = emitconv(tsh, IRT_INT, tref_type(tsh), 0);
    J->base[1] = tsh;
  }
  id = crec_bit64_type(cts, &rd->argv[0]);
  if (id) {
    TRef tr = crec_ct_tv(J, ctype_get(cts, id), 0, J->base[0], &rd->argv[0]);
    uint32_t op = rd->data;
    if (!tsh) tsh = lj_opt_narrow_tobit(J, J->base[1]);
    if (!(op < IR_BROL ? LJ_TARGET_MASKSHIFT : LJ_TARGET_MASKROT) &&
        !tref_isk(tsh))
      tsh = emitir(IRTI(IR_BAND), tsh, lj_ir_kint(J, 63));
#ifdef LJ_TARGET_UNIFYROT
      if (op == (LJ_TARGET_UNIFYROT == 1 ? IR_BROR : IR_BROL)) {
        op = LJ_TARGET_UNIFYROT == 1 ? IR_BROL : IR_BROR;
        tsh = emitir(IRTI(IR_NEG), tsh, tsh);
      }
#endif
    tr = emitir(IRT(op, id-CTID_INT64+IRT_I64), tr, tsh);
    J->base[0] = emitir(IRTG(IR_CNEWI, IRT_CDATA), lj_ir_kint(J, id), tr);
    return 1;
  }
  return 0;
}

TRef recff_bit64_tohex(jit_State *J, RecordFFData *rd, TRef hdr)
{
  CTState *cts = ctype_ctsG(J2G(J));
  CTypeID id = crec_bit64_type(cts, &rd->argv[0]);
  TRef tr, trsf = J->base[1];
  SFormat sf = (STRFMT_UINT|STRFMT_T_HEX);
  int32_t n;
  if (trsf) {
    CTypeID id2 = 0;
    n = (int32_t)lj_carith_check64(J->L, 2, &id2);
    if (id2)
      trsf = crec_ct_tv(J, ctype_get(cts, CTID_INT32), 0, trsf, &rd->argv[1]);
    else
      trsf = lj_opt_narrow_tobit(J, trsf);
    emitir(IRTGI(IR_EQ), trsf, lj_ir_kint(J, n));  /* Specialize to n. */
  } else {
    n = id ? 16 : 8;
  }
  if (n < 0) { n = -n; sf |= STRFMT_F_UPPER; }
  sf |= ((SFormat)((n+1)&255) << STRFMT_SH_PREC);
  if (id) {
    tr = crec_ct_tv(J, ctype_get(cts, id), 0, J->base[0], &rd->argv[0]);
    if (n < 16)
      tr = emitir(IRT(IR_BAND, IRT_U64), tr,
                  lj_ir_kint64(J, ((uint64_t)1 << 4*n)-1));
  } else {
    tr = lj_opt_narrow_tobit(J, J->base[0]);
    if (n < 8)
      tr = emitir(IRTI(IR_BAND), tr, lj_ir_kint(J, (int32_t)((1u << 4*n)-1)));
    tr = emitconv(tr, IRT_U64, IRT_INT, 0);  /* No sign-extension. */
    lj_needsplit(J);
  }
  return lj_ir_call(J, IRCALL_lj_strfmt_putfxint, hdr, lj_ir_kint(J, sf), tr);
}

/* -- Miscellaneous library functions ------------------------------------ */

void LJ_FASTCALL lj_crecord_tonumber(jit_State *J, RecordFFData *rd)
```

```
{
  CTState *cts = ctype_ctsG(J2G(J));
  CType *d, *ct = lj_ctype_rawref(cts, cdataV(&rd->argv[0])->ctypeid);
  if (ctype_isenum(ct->info)) ct = ctype_child(cts, ct);
  if (ctype_isnum(ct->info) || ctype_iscomplex(ct->info)) {
    if (ctype_isinteger_or_bool(ct->info) && ct->size <= 4 &&
        !(ct->size == 4 && (ct->info & CTF_UNSIGNED)))
      d = ctype_get(cts, CTID_INT32);
    else
      d = ctype_get(cts, CTID_DOUBLE);
    J->base[0] = crec_ct_tv(J, d, 0, J->base[0], &rd->argv[0]);
  } else {
    J->base[0] = TREF_NIL;
  }
}

#undef IR
#undef emitir
#undef emitconv

#endif
```

One Level Up Top Level

src/lj_ffdef.h - luajit-2.0-src

Macros defined

- FFDEF
- FF_NUM_ASMFUNC

Source code

```
1  /* This is a generated file. DO NOT EDIT! */
2
3  FFDEF(assert)
4  FFDEF(type)
5  FFDEF(next)
6  FFDEF(pairs)
7  FFDEF(ipairs_aux)
8  FFDEF(ipairs)
9  FFDEF(getmetatable)
10  FFDEF(setmetatable)
11  FFDEF(getfenv)
12  FFDEF(setfenv)
13  FFDEF(rawget)
14  FFDEF(rawset)
15  FFDEF(rawequal)
16  FFDEF(unpack)
17  FFDEF(select)
18  FFDEF(tonumber)
19  FFDEF(tostring)
20  FFDEF(error)
21  FFDEF(pcall)
22  FFDEF(xpcall)
23  FFDEF(loadfile)
24  FFDEF(load)
25  FFDEF(loadstring)
26  FFDEF(dofile)
27  FFDEF(gcinfo)
28  FFDEF(collectgarbage)
29  FFDEF(newproxy)
30  FFDEF(print)
31  FFDEF(coroutine_status)
32  FFDEF(coroutine_running)
33  FFDEF(coroutine_create)
34  FFDEF(coroutine_yield)
35  FFDEF(coroutine_resume)
36  FFDEF(coroutine_wrap_aux)
37  FFDEF(coroutine_wrap)
38  FFDEF(math_abs)
39  FFDEF(math_floor)
40  FFDEF(math_ceil)
41  FFDEF(math_sqrt)
42  FFDEF(math_log10)
43  FFDEF(math_exp)
44  FFDEF(math_sin)
45  FFDEF(math_cos)
46  FFDEF(math_tan)
47  FFDEF(math_asin)
48  FFDEF(math_acos)
49  FFDEF(math_atan)
50  FFDEF(math_sinh)
51  FFDEF(math_cosh)
52  FFDEF(math_tanh)
53  FFDEF(math_frexp)
54  FFDEF(math_modf)
55  FFDEF(math_log)
56  FFDEF(math_atan2)
57  FFDEF(math_pow)
58  FFDEF(math_fmod)
59  FFDEF(math_ldexp)
60  FFDEF(math_min)
```

```
FFDEF(math_max)
FFDEF(math_random)
FFDEF(math_randomseed)
FFDEF(bit_tobit)
FFDEF(bit_bnot)
FFDEF(bit_bswap)
FFDEF(bit_lshift)
FFDEF(bit_rshift)
FFDEF(bit_arshift)
FFDEF(bit_rol)
FFDEF(bit_ror)
FFDEF(bit_band)
FFDEF(bit_bor)
FFDEF(bit_bxor)
FFDEF(bit_tohex)
FFDEF(string_byte)
FFDEF(string_char)
FFDEF(string_sub)
FFDEF(string_rep)
FFDEF(string_reverse)
FFDEF(string_lower)
FFDEF(string_upper)
FFDEF(string_dump)
FFDEF(string_find)
FFDEF(string_match)
FFDEF(string_gmatch_aux)
FFDEF(string_gmatch)
FFDEF(string_gsub)
FFDEF(string_format)
FFDEF(table_maxn)
FFDEF(table_insert)
FFDEF(table_concat)
FFDEF(table_sort)
FFDEF(table_new)
FFDEF(table_clear)
FFDEF(io_method_close)
FFDEF(io_method_read)
FFDEF(io_method_write)
FFDEF(io_method_flush)
 FFDEF(io_method_seek)
 FFDEF(io_method_setvbuf)
 FFDEF(io_method_lines)
 FFDEF(io_method___gc)
 FFDEF(io_method___tostring)
 FFDEF(io_open)
 FFDEF(io_popen)
 FFDEF(io_tmpfile)
 FFDEF(io_close)
 FFDEF(io_read)
 FFDEF(io_write)
 FFDEF(io_flush)
 FFDEF(io_input)
 FFDEF(io_output)
 FFDEF(io_lines)
 FFDEF(io_type)
 FFDEF(os_execute)
 FFDEF(os_remove)
 FFDEF(os_rename)
 FFDEF(os_tmpname)
 FFDEF(os_getenv)
 FFDEF(os_exit)
 FFDEF(os_clock)
 FFDEF(os_date)
 FFDEF(os_time)
 FFDEF(os_difftime)
 FFDEF(os_setlocale)
 FFDEF(debug_getregistry)
 FFDEF(debug_getmetatable)
 FFDEF(debug_setmetatable)
 FFDEF(debug_getfenv)
 FFDEF(debug_setfenv)
 FFDEF(debug_getinfo)
 FFDEF(debug_getlocal)
 FFDEF(debug_setlocal)
 FFDEF(debug_getupvalue)
 FFDEF(debug_setupvalue)
```

```
FFDEF(debug_upvalueid)
FFDEF(debug_upvaluejoin)
FFDEF(debug_sethook)
FFDEF(debug_gethook)
FFDEF(debug_debug)
FFDEF(debug_traceback)
FFDEF(jit_on)
FFDEF(jit_off)
FFDEF(jit_flush)
FFDEF(jit_status)
FFDEF(jit_attach)
FFDEF(jit_util_funcinfo)
FFDEF(jit_util_funcbc)
FFDEF(jit_util_funck)
FFDEF(jit_util_funcuvname)
FFDEF(jit_util_traceinfo)
FFDEF(jit_util_traceir)
FFDEF(jit_util_tracek)
FFDEF(jit_util_tracesnap)
FFDEF(jit_util_tracemc)
FFDEF(jit_util_traceexitstub)
FFDEF(jit_util_ircalladdr)
FFDEF(jit_opt_start)
FFDEF(jit_profile_start)
FFDEF(jit_profile_stop)
FFDEF(jit_profile_dumpstack)
FFDEF(ffi_meta___index)
FFDEF(ffi_meta___newindex)
FFDEF(ffi_meta___eq)
FFDEF(ffi_meta___len)
FFDEF(ffi_meta___lt)
FFDEF(ffi_meta___le)
FFDEF(ffi_meta___concat)
FFDEF(ffi_meta___call)
FFDEF(ffi_meta___add)
FFDEF(ffi_meta___sub)
FFDEF(ffi_meta___mul)
FFDEF(ffi_meta___div)
FFDEF(ffi_meta___mod)
FFDEF(ffi_meta___pow)
FFDEF(ffi_meta___unm)
FFDEF(ffi_meta___tostring)
FFDEF(ffi_meta___pairs)
FFDEF(ffi_meta___ipairs)
FFDEF(ffi_clib___index)
FFDEF(ffi_clib___newindex)
FFDEF(ffi_clib___gc)
FFDEF(ffi_callback_free)
FFDEF(ffi_callback_set)
FFDEF(ffi_cdef)
FFDEF(ffi_new)
FFDEF(ffi_cast)
FFDEF(ffi_typeof)
FFDEF(ffi_typeinfo)
FFDEF(ffi_istype)
FFDEF(ffi_sizeof)
FFDEF(ffi_alignof)
FFDEF(ffi_offsetof)
FFDEF(ffi_errno)
FFDEF(ffi_string)
FFDEF(ffi_copy)
FFDEF(ffi_fill)
FFDEF(ffi_abi)
FFDEF(ffi_metatype)
FFDEF(ffi_gc)
FFDEF(ffi_load)

#undef FFDEF

#ifndef FF_NUM_ASMFUNC
#define FF_NUM_ASMFUNC 57
#endif

```

One Level Up Top Level

src/lj_libdef.h - luajit-2.0-src

Global variables defined

- lj_lib_cf_base
- lj_lib_cf_bit
- lj_lib_cf_coroutine
- lj_lib_cf_debug
- lj_lib_cf_ffi
- lj_lib_cf_ffi_callback
- lj_lib_cf_ffi_clib
- lj_lib_cf_ffi_meta
- lj_lib_cf_io
- lj_lib_cf_io_method
- lj_lib_cf_jit
- lj_lib_cf_jit_opt
- lj_lib_cf_jit_profile
- lj_lib_cf_jit_util
- lj_lib_cf_math
- lj_lib_cf_os
- lj_lib_cf_string
- lj_lib_cf_table
- lj_lib_init_base
- lj_lib_init_bit
- lj_lib_init_coroutine
- lj_lib_init_debug
- lj_lib_init_ffi
- lj_lib_init_ffi_callback
- lj_lib_init_ffi_clib
- lj_lib_init_ffi_meta
- lj_lib_init_io
- lj_lib_init_io_method
- lj_lib_init_jit
- lj_lib_init_jit_opt

- lj_lib_init_jit_profile
- lj_lib_init_jit_util
- lj_lib_init_math
- lj_lib_init_os
- lj_lib_init_string
- lj_lib_init_table

Macros defined

- LJLIB_MODULE_base
- LJLIB_MODULE_bit
- LJLIB_MODULE_coroutine
- LJLIB_MODULE_debug
- LJLIB_MODULE_ffi
- LJLIB_MODULE_ffi_callback
- LJLIB_MODULE_ffi_clib
- LJLIB_MODULE_ffi_meta
- LJLIB_MODULE_io
- LJLIB_MODULE_io_method
- LJLIB_MODULE_jit
- LJLIB_MODULE_jit_opt
- LJLIB_MODULE_jit_profile
- LJLIB_MODULE_jit_util
- LJLIB_MODULE_math
- LJLIB_MODULE_os
- LJLIB_MODULE_string
- LJLIB_MODULE_table

Source code

```
/* This is a generated file. DO NOT EDIT! */

#ifdef LJLIB_MODULE_base
#undef LJLIB_MODULE_base
static const lua_CFunction lj_lib_cf_base[] = {
  lj_ffh_assert,
  lj_ffh_next,
  lj_ffh_pairs,
  lj_ffh_ipairs_aux,
   lj_ffh_ipairs,
   lj_ffh_setmetatable,
   lj_cf_getfenv,
   lj_cf_setfenv,
   lj_ffh_rawget,
```

```
  lj_cf_rawset,
  lj_cf_rawequal,
  lj_cf_unpack,
  lj_cf_select,
  lj_ffh_tonumber,
  lj_ffh_tostring,
  lj_cf_error,
  lj_ffh_pcall,
  lj_cf_loadfile,
  lj_cf_load,
  lj_cf_loadstring,
  lj_cf_dofile,
  lj_cf_gcinfo,
  lj_cf_collectgarbage,
  lj_cf_newproxy,
  lj_cf_print
};
static const uint8_t lj_lib_init_base[] = {
2,0,28,70,97,115,115,101,114,116,195,110,105,108,199,98,111,111,108,101,97,
110,252,1,200,117,115,101,114,100,97,116,97,198,115,116,114,105,110,103,197,
117,112,118,97,108,198,116,104,114,101,97,100,197,112,114,111,116,111,200,102,
117,110,99,116,105,111,110,197,116,114,97,99,101,197,99,100,97,116,97,197,116,
97,98,108,101,252,9,198,110,117,109,98,101,114,132,116,121,112,101,68,110,101,
120,116,253,69,112,97,105,114,115,64,253,70,105,112,97,105,114,115,140,103,
101,116,109,101,116,97,116,97,98,108,101,76,115,101,116,109,101,116,97,116,
97,98,108,101,7,103,101,116,102,101,110,118,7,115,101,116,102,101,110,118,70,
114,97,119,103,101,116,6,114,97,119,115,101,116,8,114,97,119,101,113,117,97,
108,6,117,110,112,97,99,107,6,115,101,108,101,99,116,72,116,111,110,117,109,
98,101,114,72,116,111,115,116,114,105,110,103,5,101,114,114,111,114,69,112,
99,97,108,108,134,120,112,99,97,108,108,8,108,111,97,100,102,105,108,101,4,
108,111,97,100,10,108,111,97,100,115,116,114,105,110,103,6,100,111,102,105,
108,101,6,103,99,105,110,102,111,14,99,111,108,108,101,99,116,103,97,114,98,
97,103,101,252,2,8,110,101,119,112,114,111,120,121,200,116,111,115,116,114,
105,110,103,5,112,114,105,110,116,252,3,200,95,86,69,82,83,73,79,78,250,255
};
#endif

#ifdef LJLIB_MODULE_coroutine
#undef LJLIB_MODULE_coroutine
static const lua_CFunction lj_lib_cf_coroutine[] = {
  lj_cf_coroutine_status,
  lj_cf_coroutine_running,
  lj_cf_coroutine_create,
  lj_ffh_coroutine_yield,
  lj_ffh_coroutine_resume,
  lj_cf_coroutine_wrap
};
static const uint8_t lj_lib_init_coroutine[] = {
30,13,6,6,115,116,97,116,117,115,7,114,117,110,110,105,110,103,6,99,114,101,
97,116,101,69,121,105,101,108,100,70,114,101,115,117,109,101,254,4,119,114,
97,112,255
};
#endif

#ifdef LJLIB_MODULE_math
#undef LJLIB_MODULE_math
static const lua_CFunction lj_lib_cf_math[] = {
  lj_ffh_math_abs,
  lj_ffh_math_sqrt,
  lj_ffh_math_log,
  lj_ffh_math_atan2,
  lj_ffh_math_ldexp,
  lj_ffh_math_min,
  lj_cf_math_random,
  lj_cf_math_randomseed
};
static const uint8_t lj_lib_init_math[] = {
37,16,30,67,97,98,115,133,102,108,111,111,114,132,99,101,105,108,68,115,113,
114,116,133,108,111,103,49,48,131,101,120,112,131,115,105,110,131,99,111,115,
131,116,97,110,132,97,115,105,110,132,97,99,111,115,132,97,116,97,110,132,115,
105,110,104,132,99,111,115,104,132,116,97,110,104,133,102,114,101,120,112,132,
109,111,100,102,67,108,111,103,249,3,100,101,103,0,1,2,0,0,1,2,24,1,0,0,76,
1,2,0,241,135,158,166,3,220,203,178,130,4,249,3,114,97,100,0,1,2,0,0,1,2,24,
1,0,0,76,1,2,0,243,244,148,165,20,198,190,199,252,3,69,97,116,97,110,50,131,
112,111,119,132,102,109,111,100,69,108,100,101,120,112,67,109,105,110,131,109,
97,120,251,24,45,68,84,251,33,9,64,194,112,105,250,251,0,0,0,0,0,0,240,127,
```

```
196,104,117,103,101,250,252,2,6,114,97,110,100,111,109,252,2,10,114,97,110,
100,111,109,115,101,101,100,255
};
#endif

#ifdef LJLIB_MODULE_bit
#undef LJLIB_MODULE_bit
static const lua_CFunction lj_lib_cf_bit[] = {
  lj_ffh_bit_tobit,
   lj_ffh_bit_bnot,
   lj_ffh_bit_bswap,
   lj_ffh_bit_lshift,
   lj_ffh_bit_band,
   lj_cf_bit_tohex
};
static const uint8_t lj_lib_init_bit[] = {
63,40,12,69,116,111,98,105,116,68,98,110,111,116,69,98,115,119,97,112,70,108,
115,104,105,102,116,134,114,115,104,105,102,116,135,97,114,115,104,105,102,
116,131,114,111,108,131,114,111,114,68,98,97,110,100,131,98,111,114,132,98,
120,111,114,5,116,111,104,101,120,255
};
#endif

#ifdef LJLIB_MODULE_string
#undef LJLIB_MODULE_string
static const lua_CFunction lj_lib_cf_string[] = {
  lj_ffh_string_byte,
  lj_ffh_string_char,
  lj_ffh_string_sub,
  lj_cf_string_rep,
  lj_ffh_string_reverse,
  lj_cf_string_dump,
  lj_cf_string_find,
  lj_cf_string_match,
  lj_cf_string_gmatch,
  lj_cf_string_gsub,
  lj_cf_string_format
};
static const uint8_t lj_lib_init_string[] = {
75,51,14,249,3,108,101,110,0,1,2,0,0,0,3,16,0,5,0,21,1,0,0,76,1,2,0,68,98,121,
116,101,68,99,104,97,114,67,115,117,98,3,114,101,112,71,114,101,118,101,114,
115,101,133,108,111,119,101,114,133,117,112,112,101,114,4,100,117,109,112,4,
102,105,110,100,5,109,97,116,99,104,254,6,103,109,97,116,99,104,4,103,115,117,
98,6,102,111,114,109,97,116,255
};
#endif

#ifdef LJLIB_MODULE_table
#undef LJLIB_MODULE_table
static const lua_CFunction lj_lib_cf_table[] = {
  lj_cf_table_maxn,
  lj_cf_table_insert,
  lj_cf_table_concat,
  lj_cf_table_sort
};
static const uint8_t lj_lib_init_table[] = {
89,57,8,249,8,102,111,114,101,97,99,104,105,0,2,9,0,0,0,15,16,0,12,0,16,1,9,
0,41,2,1,0,21,3,0,0,41,4,1,0,77,2,8,128,18,6,1,0,18,7,5,0,59,8,5,0,66,6,3,2,
10,6,0,0,88,7,1,128,76,6,2,0,79,2,248,127,75,0,1,0,249,7,102,111,114,101,97,
99,104,0,2,10,0,0,0,16,16,0,12,0,16,1,9,0,43,2,0,0,18,3,0,0,41,4,0,0,88,5,7,
128,18,7,1,0,18,8,5,0,18,9,6,0,66,7,3,2,10,7,0,0,88,8,1,128,76,7,2,0,70,5,3,
3,82,5,247,127,75,0,1,0,249,4,103,101,116,110,0,1,2,0,0,0,3,16,0,12,0,21,1,
0,0,76,1,2,0,4,109,97,120,110,6,105,110,115,101,114,116,249,6,114,101,109,111,
118,101,0,2,10,0,0,0,2,30,16,0,12,0,21,2,0,0,11,1,0,0,88,3,7,128,8,2,0,0,88,3,
23,128,59,3,2,0,43,4,0,0,64,4,2,0,76,3,2,0,88,3,18,128,16,1,14,0,41,3,1,0,3,
3,1,0,88,3,14,128,3,1,2,0,88,3,12,128,59,3,1,0,22,4,1,1,18,5,2,0,41,6,1,0,77,
4,4,128,23,8,1,7,59,9,7,0,64,9,8,0,79,4,252,127,43,4,0,0,64,4,2,0,76,3,2,0,
75,0,1,0,0,2,6,99,111,110,99,97,116,4,115,111,114,116,254,254,255
};
#endif

#ifdef LJLIB_MODULE_io_method
#undef LJLIB_MODULE_io_method
static const lua_CFunction lj_lib_cf_io_method[] = {
  lj_cf_io_method_close,
  lj_cf_io_method_read,
```

```
  lj_cf_io_method_write,
  lj_cf_io_method_flush,
  lj_cf_io_method_seek,
  lj_cf_io_method_setvbuf,
  lj_cf_io_method_lines,
  lj_cf_io_method___gc,
  lj_cf_io_method___tostring
};
static const uint8_t lj_lib_init_io_method[] = {
95,57,10,5,99,108,111,115,101,4,114,101,97,100,5,119,114,105,116,101,5,102,
108,117,115,104,4,115,101,101,107,7,115,101,116,118,98,117,102,5,108,105,110,
101,115,4,95,95,103,99,10,95,95,116,111,115,116,114,105,110,103,252,1,199,95,
95,105,110,100,101,120,250,255
};
#endif

#ifdef LJLIB_MODULE_io
#undef LJLIB_MODULE_io
static const lua_CFunction lj_lib_cf_io[] = {
  lj_cf_io_open,
  lj_cf_io_popen,
  lj_cf_io_tmpfile,
  lj_cf_io_close,
  lj_cf_io_read,
  lj_cf_io_write,
  lj_cf_io_flush,
  lj_cf_io_input,
  lj_cf_io_output,
  lj_cf_io_lines,
  lj_cf_io_type
};
static const uint8_t lj_lib_init_io[] = {
104,57,12,252,2,192,250,4,111,112,101,110,5,112,111,112,101,110,7,116,109,112,
102,105,108,101,5,99,108,111,115,101,4,114,101,97,100,5,119,114,105,116,101,
5,102,108,117,115,104,5,105,110,112,117,116,6,111,117,116,112,117,116,5,108,
105,110,101,115,4,116,121,112,101,255
};
#endif

#ifdef LJLIB_MODULE_os
#undef LJLIB_MODULE_os
static const lua_CFunction lj_lib_cf_os[] = {
  lj_cf_os_execute,
  lj_cf_os_remove,
  lj_cf_os_rename,
  lj_cf_os_tmpname,
  lj_cf_os_getenv,
  lj_cf_os_exit,
  lj_cf_os_clock,
  lj_cf_os_date,
  lj_cf_os_time,
  lj_cf_os_difftime,
  lj_cf_os_setlocale
};
static const uint8_t lj_lib_init_os[] = {
115,57,11,7,101,120,101,99,117,116,101,6,114,101,109,111,118,101,6,114,101,
110,97,109,101,7,116,109,112,110,97,109,101,6,103,101,116,101,110,118,4,101,
120,105,116,5,99,108,111,99,107,4,100,97,116,101,4,116,105,109,101,8,100,105,
102,102,116,105,109,101,9,115,101,116,108,111,99,97,108,101,255
};
#endif

#ifdef LJLIB_MODULE_debug
#undef LJLIB_MODULE_debug
static const lua_CFunction lj_lib_cf_debug[] = {
  lj_cf_debug_getregistry,
  lj_cf_debug_getmetatable,
  lj_cf_debug_setmetatable,
  lj_cf_debug_getfenv,
  lj_cf_debug_setfenv,
  lj_cf_debug_getinfo,
  lj_cf_debug_getlocal,
  lj_cf_debug_setlocal,
  lj_cf_debug_getupvalue,
  lj_cf_debug_setupvalue,
  lj_cf_debug_upvalueid,
```

```
  lj_cf_debug_upvaluejoin,
  lj_cf_debug_sethook,
  lj_cf_debug_gethook,
  lj_cf_debug_debug,
  lj_cf_debug_traceback
};
static const uint8_t lj_lib_init_debug[] = {
126,57,16,11,103,101,116,114,101,103,105,115,116,114,121,12,103,101,116,109,
101,116,97,116,97,98,108,101,12,115,101,116,109,101,116,97,116,97,98,108,101,
7,103,101,116,102,101,110,118,7,115,101,116,102,101,110,118,7,103,101,116,105,
110,102,111,8,103,101,116,108,111,99,97,108,8,115,101,116,108,111,99,97,108,
10,103,101,116,117,112,118,97,108,117,101,10,115,101,116,117,112,118,97,108,
117,101,9,117,112,118,97,108,117,101,105,100,11,117,112,118,97,108,117,101,
106,111,105,110,7,115,101,116,104,111,111,107,7,103,101,116,104,111,111,107,
5,100,101,98,117,103,9,116,114,97,99,101,98,97,99,107,255
};
#endif

#ifdef LJLIB_MODULE_jit
#undef LJLIB_MODULE_jit
static const lua_CFunction lj_lib_cf_jit[] = {
  lj_cf_jit_on,
  lj_cf_jit_off,
  lj_cf_jit_flush,
  lj_cf_jit_status,
  lj_cf_jit_attach
};
static const uint8_t lj_lib_init_jit[] = {
142,57,9,2,111,110,3,111,102,102,5,102,108,117,115,104,6,115,116,97,116,117,
115,6,97,116,116,97,99,104,252,5,194,111,115,250,252,4,196,97,114,99,104,250,
252,3,203,118,101,114,115,105,111,110,95,110,117,109,250,252,2,199,118,101,
114,115,105,111,110,250,255
};
#endif

#ifdef LJLIB_MODULE_jit_util
#undef LJLIB_MODULE_jit_util
static const lua_CFunction lj_lib_cf_jit_util[] = {
  lj_cf_jit_util_funcinfo,
  lj_cf_jit_util_funcbc,
  lj_cf_jit_util_funck,
  lj_cf_jit_util_funcuvname,
  lj_cf_jit_util_traceinfo,
  lj_cf_jit_util_traceir,
  lj_cf_jit_util_tracek,
  lj_cf_jit_util_tracesnap,
  lj_cf_jit_util_tracemc,
  lj_cf_jit_util_traceexitstub,
  lj_cf_jit_util_ircalladdr
};
static const uint8_t lj_lib_init_jit_util[] = {
147,57,11,8,102,117,110,99,105,110,102,111,6,102,117,110,99,98,99,5,102,117,
110,99,107,10,102,117,110,99,117,118,110,97,109,101,9,116,114,97,99,101,105,
110,102,111,7,116,114,97,99,101,105,114,6,116,114,97,99,101,107,9,116,114,97,
99,101,115,110,97,112,7,116,114,97,99,101,109,99,13,116,114,97,99,101,101,120,
105,116,115,116,117,98,10,105,114,99,97,108,108,97,100,100,114,255
};
#endif

#ifdef LJLIB_MODULE_jit_opt
#undef LJLIB_MODULE_jit_opt
static const lua_CFunction lj_lib_cf_jit_opt[] = {
  lj_cf_jit_opt_start
};
static const uint8_t lj_lib_init_jit_opt[] = {
158,57,1,5,115,116,97,114,116,255
};
#endif

#ifdef LJLIB_MODULE_jit_profile
#undef LJLIB_MODULE_jit_profile
static const lua_CFunction lj_lib_cf_jit_profile[] = {
  lj_cf_jit_profile_start,
  lj_cf_jit_profile_stop,
  lj_cf_jit_profile_dumpstack
};
```

```
static const uint8_t lj_lib_init_jit_profile[] = {
159,57,3,5,115,116,97,114,116,4,115,116,111,112,9,100,117,109,112,115,116,97,
99,107,255
};
#endif

#ifdef LJLIB_MODULE_ffi_meta
#undef LJLIB_MODULE_ffi_meta
static const lua_CFunction lj_lib_cf_ffi_meta[] = {
  lj_cf_ffi_meta___index,
  lj_cf_ffi_meta___newindex,
  lj_cf_ffi_meta___eq,
  lj_cf_ffi_meta___len,
  lj_cf_ffi_meta___lt,
  lj_cf_ffi_meta___le,
  lj_cf_ffi_meta___concat,
  lj_cf_ffi_meta___call,
  lj_cf_ffi_meta___add,
  lj_cf_ffi_meta___sub,
  lj_cf_ffi_meta___mul,
  lj_cf_ffi_meta___div,
  lj_cf_ffi_meta___mod,
  lj_cf_ffi_meta___pow,
  lj_cf_ffi_meta___unm,
  lj_cf_ffi_meta___tostring,
  lj_cf_ffi_meta___pairs,
  lj_cf_ffi_meta___ipairs
};
static const uint8_t lj_lib_init_ffi_meta[] = {
162,57,19,7,95,95,105,110,100,101,120,10,95,95,110,101,119,105,110,100,101,
120,4,95,95,101,113,5,95,95,108,101,110,4,95,95,108,116,4,95,95,108,101,8,95,
95,99,111,110,99,97,116,6,95,95,99,97,108,108,5,95,95,97,100,100,5,95,95,115,
117,98,5,95,95,109,117,108,5,95,95,100,105,118,5,95,95,109,111,100,5,95,95,95,
112,111,119,5,95,95,117,110,109,10,95,95,116,111,115,116,114,105,110,103,7,
95,95,112,97,105,114,115,8,95,95,105,112,97,105,114,115,195,102,102,105,203,
95,95,109,101,116,97,116,97,98,108,101,250,255
};
#endif

#ifdef LJLIB_MODULE_ffi_clib
#undef LJLIB_MODULE_ffi_clib
static const lua_CFunction lj_lib_cf_ffi_clib[] = {
  lj_cf_ffi_clib___index,
  lj_cf_ffi_clib___newindex,
  lj_cf_ffi_clib___gc
};
static const uint8_t lj_lib_init_ffi_clib[] = {
180,57,3,7,95,95,105,110,100,101,120,10,95,95,110,101,119,105,110,100,101,120,
4,95,95,103,99,255
};
#endif

#ifdef LJLIB_MODULE_ffi_callback
#undef LJLIB_MODULE_ffi_callback
static const lua_CFunction lj_lib_cf_ffi_callback[] = {
  lj_cf_ffi_callback_free,
  lj_cf_ffi_callback_set
};
static const uint8_t lj_lib_init_ffi_callback[] = {
183,57,3,4,102,114,101,101,3,115,101,116,252,1,199,95,95,105,110,100,101,120,
250,255
};
#endif

#ifdef LJLIB_MODULE_ffi
#undef LJLIB_MODULE_ffi
static const lua_CFunction lj_lib_cf_ffi[] = {
  lj_cf_ffi_cdef,
  lj_cf_ffi_new,
  lj_cf_ffi_cast,
  lj_cf_ffi_typeof,
  lj_cf_ffi_typeinfo,
  lj_cf_ffi_istype,
  lj_cf_ffi_sizeof,
  lj_cf_ffi_alignof,
  lj_cf_ffi_offsetof,
```

```
  lj_cf_ffi_errno,
  lj_cf_ffi_string,
  lj_cf_ffi_copy,
  lj_cf_ffi_fill,
  lj_cf_ffi_abi,
  lj_cf_ffi_metatype,
  lj_cf_ffi_gc,
  lj_cf_ffi_load
};
static const uint8_t lj_lib_init_ffi[] = {
185,57,23,4,99,100,101,102,3,110,101,119,4,99,97,115,116,6,116,121,112,101,
111,102,8,116,121,112,101,105,110,102,111,6,105,115,116,121,112,101,6,115,105,
122,101,111,102,7,97,108,105,103,110,111,102,8,111,102,102,115,101,116,111,
102,5,101,114,114,110,111,6,115,116,114,105,110,103,4,99,111,112,121,4,102,
105,108,108,3,97,98,105,252,8,192,250,8,109,101,116,97,116,121,112,101,252,
7,192,250,2,103,99,252,5,192,250,4,108,111,97,100,252,4,193,67,250,252,3,194,
111,115,250,252,2,196,97,114,99,104,250,255
};
#endif

```

One Level Up Top Level

src/host/buildvm_fold.c - luajit-2.0-src

Global variables defined

- foldkeys
- funcidx
- lineno
- nkeys

Functions defined

- emit_fold
- foldrule
- makehash
- nexttoken
- printhash
- tryhash

Source code

```
1  /*
2  ** LuaJIT VM builder: IR folding hash table generator.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #include "buildvm.h"
7  #include "lj_obj.h"
8  #include "lj_ir.h"
9
10 /* Context for the folding hash table generator. */
11 static int lineno;
12 static int funcidx;
13 static uint32_t foldkeys[BUILD_MAX_FOLD];
14 static uint32_t nkeys;
15
16 /* Try to fill the hash table with keys using the hash parameters. */
17 static int tryhash(uint32_t *htab, uint32_t sz, uint32_t r, int dorol)
18 {
19   uint32_t i;
20   if (dorol && ((r & 31) == 0 || (r>>5) == 0))
21     return 0;  /* Avoid zero rotates. */
22   memset(htab, 0xff, (sz+1)*sizeof(uint32_t));
23   for (i = 0; i < nkeys; i++) {
24     uint32_t key = foldkeys[i];
25     uint32_t k = key & 0xffffff;
26     uint32_t h = (dorol ? lj_rol(lj_rol(k, r>>5) - k, r&31) :
27                           (((k << (r>>5)) - k) << (r&31))) % sz;
28     if (htab[h] != 0xffffffff) {  /* Collision on primary slot. */
29       if (htab[h+1] != 0xffffffff) {  /* Collision on secondary slot. */
30         /* Try to move the colliding key, if possible. */
31         if (h < sz-1 && htab[h+2] == 0xffffffff) {
32           uint32_t k2 = htab[h+1] & 0xffffff;
33           uint32_t h2 = (dorol ? lj_rol(lj_rol(k2, r>>5) - k2, r&31) :
34                                  (((k2 << (r>>5)) - k2) << (r&31))) % sz;
35           if (h2 != h+1) return 0;  /* Cannot resolve collision. */
36           htab[h+2] = htab[h+1];  /* Move colliding key to secondary slot. */
37         } else {
38           return 0;  /* Collision. */
```

```
        }
      }
      htab[h+1] = key;
    } else {
      htab[h] = key;
    }
  }
  return 1;  /* Success, all keys could be stored. */
}

/* Print the generated hash table. */
static void printhash(BuildCtx *ctx, uint32_t *htab, uint32_t sz)
{
  uint32_t i;
  fprintf(ctx->fp, "static const uint32_t fold_hash[%d] = {\n0x%08x",
          sz+1, htab[0]);
  for (i = 1; i < sz+1; i++)
    fprintf(ctx->fp, ",\n0x%08x", htab[i]);
  fprintf(ctx->fp, "\n};\n\n");
}

/* Exhaustive search for the shortest semi-perfect hash table. */
static void makehash(BuildCtx *ctx)
{
  uint32_t htab[BUILD_MAX_FOLD*2+1];
  uint32_t sz, r;
  /* Search for the smallest hash table with an odd size. */
  for (sz = (nkeys|1); sz < BUILD_MAX_FOLD*2; sz += 2) {
    /* First try all shift hash combinations. */
    for (r = 0; r < 32*32; r++) {
      if (tryhash(htab, sz, r, 0)) {
        printhash(ctx, htab, sz);
        fprintf(ctx->fp,
                "#define fold_hashkey(k)\t(((((k)<<%u)-(k))<<%u)%%%u)\n\n",
                r>>5, r&31, sz);
        return;
      }
    }
    /* Then try all rotate hash combinations. */
    for (r = 0; r < 32*32; r++) {
      if (tryhash(htab, sz, r, 1)) {
        printhash(ctx, htab, sz);
        fprintf(ctx->fp,
          "#define fold_hashkey(k)\t(lj_rol(lj_rol((k),%u)-(k),%u)%%%u)\n\n",
                r>>5, r&31, sz);
        return;
      }
    }
  }
  fprintf(stderr, "Error: search for perfect hash failed\n");
  exit(1);
}

/* Parse one token of a fold rule. */
static uint32_t nexttoken(char **pp, int allowlit, int allowany)
{
  char *p = *pp;
  if (p) {
    uint32_t i;
    char *q = strchr(p, ' ');
    if (q) *q++ = '\0';
     *pp = q;
     if (allowlit && !strncmp(p, "IRFPM_", 6)) {
       for (i = 0; irfpm_names[i]; i++)
         if (!strcmp(irfpm_names[i], p+6))
           return i;
     } else if (allowlit && !strncmp(p, "IRFL_", 5)) {
       for (i = 0; irfield_names[i]; i++)
         if (!strcmp(irfield_names[i], p+5))
           return i;
     } else if (allowlit && !strncmp(p, "IRCALL_", 7)) {
       for (i = 0; ircall_names[i]; i++)
         if (!strcmp(ircall_names[i], p+7))
           return i;
     } else if (allowlit && !strncmp(p, "IRCONV_", 7)) {
       for (i = 0; irt_names[i]; i++) {
```

```
        const char *r = strchr(p+7, '_');
        if (r && !strncmp(irt_names[i], p+7, r-(p+7))) {
          uint32_t j;
          for (j = 0; irt_names[j]; j++)
            if (!strcmp(irt_names[j], r+1))
              return (i << 5) + j;
        }
      }
    } else if (allowlit && *p >= '0' && *p <= '9') {
      for (i = 0; *p >= '0' && *p <= '9'; p++)
        i = i*10 + (*p - '0');
      if (*p == '\0')
        return i;
    } else if (allowany && !strcmp("any", p)) {
      return allowany;
    } else {
      for (i = 0; ir_names[i]; i++)
        if (!strcmp(ir_names[i], p))
          return i;
    }
    fprintf(stderr, "Error: bad fold definition token \"%s\" at line %d\n", p, lineno);
    exit(1);
  }
  return 0;
}

/* Parse a fold rule. */
static void foldrule(char *p)
{
  uint32_t op = nexttoken(&p, 0, 0);
  uint32_t left = nexttoken(&p, 0, 0x7f);
  uint32_t right = nexttoken(&p, 1, 0x3ff);
  uint32_t key = (funcidx << 24) | (op << 17) | (left << 10) | right;
  uint32_t i;
  if (nkeys >= BUILD_MAX_FOLD) {
    fprintf(stderr, "Error: too many fold rules, increase BUILD_MAX_FOLD.\n");
    exit(1);
  }
  /* Simple insertion sort to detect duplicates. */
  for (i = nkeys; i > 0; i--) {
    if ((foldkeys[i-1]&0xffffff) < (key & 0xffffff))
      break;
    if ((foldkeys[i-1]&0xffffff) == (key & 0xffffff)) {
      fprintf(stderr, "Error: duplicate fold definition at line %d\n", lineno);
      exit(1);
    }
    foldkeys[i] = foldkeys[i-1];
  }
  foldkeys[i] = key;
  nkeys++;
}

/* Emit C source code for IR folding hash table. */
void emit_fold(BuildCtx *ctx)
{
  char buf[256];  /* We don't care about analyzing lines longer than that. */
  const char *fname = ctx->args[0];
  FILE *fp;

  if (fname == NULL) {
    fprintf(stderr, "Error: missing input filename\n");
    exit(1);
  }

  if (fname[0] == '-' && fname[1] == '\0') {
    fp = stdin;
  } else {
    fp = fopen(fname, "r");
    if (!fp) {
      fprintf(stderr, "Error: cannot open input file '%s': %s\n",
              fname, strerror(errno));
      exit(1);
    }
  }

  fprintf(ctx->fp, "/* This is a generated file. DO NOT EDIT! */\n\n");
```

```
  fprintf(ctx->fp, "static const FoldFunc fold_func[] = {\n");

  lineno = 0;
  funcidx = 0;
  nkeys = 0;
  while (fgets(buf, sizeof(buf), fp) != NULL) {
    lineno++;
    /* The prefix must be at the start of a line, otherwise it's ignored. */
    if (!strncmp(buf, FOLDDEF_PREFIX, sizeof(FOLDDEF_PREFIX)-1)) {
      char *p = buf+sizeof(FOLDDEF_PREFIX)-1;
      char *q = strchr(p, ')');
      if (p[0] == '(' && q) {
        p++;
        *q = '\0';
        foldrule(p);
      } else if ((p[0] == 'F' || p[0] == 'X') && p[1] == '(' && q) {
        p += 2;
        *q = '\0';
        if (funcidx)
          fprintf(ctx->fp, ",\n");
        if (p[-2] == 'X')
          fprintf(ctx->fp, "  %s", p);
        else
          fprintf(ctx->fp, "  fold_%s", p);
        funcidx++;
      } else {
        buf[strlen(buf)-1] = '\0';
        fprintf(stderr, "Error: unknown fold definition tag %s%s at line %d\n",
                FOLDDEF_PREFIX, p, lineno);
        exit(1);
      }
    }
  }
  fclose(fp);
  fprintf(ctx->fp, "\n};\n\n");

  makehash(ctx);
}

```

One Level Up Top Level

src/lj_folddef.h - luajit-2.0-src

Global variables defined

- fold_func
- fold_hash

Macros defined

- fold_hashkey

Source code

```
1  /* This is a generated file. DO NOT EDIT! */
2
3  static const FoldFunc fold_func[] = {
4    fold_kfold_numarith,
5    fold_kfold_ldexp,
6    fold_kfold_fpmath,
7    fold_kfold_numpow,
8    fold_kfold_numcomp,
9    fold_kfold_intarith,
10    fold_kfold_intovarith,
11    fold_kfold_bnot,
12    fold_kfold_bswap,
13    fold_kfold_intcomp,
14    fold_kfold_intcomp0,
15    fold_kfold_int64arith,
16    fold_kfold_int64arith2,
17    fold_kfold_int64shift,
18    fold_kfold_bnot64,
19    fold_kfold_bswap64,
20    fold_kfold_int64comp,
21    fold_kfold_int64comp0,
22    fold_kfold_snew_kptr,
23    fold_kfold_snew_empty,
24    fold_kfold_strref,
25    fold_kfold_strref_snew,
26    fold_kfold_strcmp,
27    fold_bufput_append,
28    fold_bufput_kgc,
29    fold_bufstr_kfold_cse,
30    fold_bufput_kfold_op,
31    fold_bufput_kfold_rep,
32    fold_bufput_kfold_fmt,
33    fold_kfold_add_kgc,
34    fold_kfold_add_kptr,
35    fold_kfold_add_kright,
36    fold_kfold_tobit,
37    fold_kfold_conv_kint_num,
38    fold_kfold_conv_kintu32_num,
39    fold_kfold_conv_kint_ext,
40    fold_kfold_conv_kint_i64,
41    fold_kfold_conv_kint64_num_i64,
42    fold_kfold_conv_kint64_num_u64,
43    fold_kfold_conv_kint64_int_i64,
44    fold_kfold_conv_knum_int_num,
45    fold_kfold_conv_knum_u32_num,
46    fold_kfold_conv_knum_i64_num,
47    fold_kfold_conv_knum_u64_num,
48    fold_kfold_tostr_knum,
49    fold_kfold_tostr_kint,
50    fold_kfold_strto,
51    lj_opt_cse,
52    fold_kfold_kref,
53    fold_shortcut_round,
54    fold_shortcut_left,
```

```
  fold_shortcut_dropleft,
  fold_shortcut_leftleft,
  fold_simplify_numadd_negx,
  fold_simplify_numadd_xneg,
  fold_simplify_numsub_k,
  fold_simplify_numsub_negk,
  fold_simplify_numsub_xneg,
  fold_simplify_nummuldiv_k,
  fold_simplify_nummuldiv_negk,
  fold_simplify_nummuldiv_negneg,
  fold_simplify_numpow_xk,
  fold_simplify_numpow_kx,
  fold_shortcut_conv_num_int,
  fold_simplify_conv_int_num,
  fold_simplify_conv_i64_num,
  fold_simplify_conv_int_i64,
  fold_simplify_conv_flt_num,
  fold_simplify_tobit_conv,
  fold_simplify_floor_conv,
  fold_simplify_conv_sext,
  fold_simplify_conv_narrow,
  fold_cse_conv,
  fold_narrow_convert,
  fold_simplify_intadd_k,
  fold_simplify_intmul_k,
  fold_simplify_intsub_k,
  fold_simplify_intsub_kleft,
  fold_simplify_intadd_k64,
  fold_simplify_intsub_k64,
  fold_simplify_intmul_k32,
  fold_simplify_intmul_k64,
  fold_simplify_intmod_k,
  fold_simplify_intmod_kleft,
  fold_simplify_intsub,
  fold_simplify_intsubadd_leftcancel,
  fold_simplify_intsubsub_leftcancel,
  fold_simplify_intsubsub_rightcancel,
  fold_simplify_intsubadd_rightcancel,
  fold_simplify_intsubaddadd_cancel,
  fold_simplify_band_k,
  fold_simplify_bor_k,
  fold_simplify_bxor_k,
  fold_simplify_shift_ik,
  fold_simplify_shift_andk,
  fold_simplify_shift1_ki,
   fold_simplify_shift2_ki,
   fold_simplify_shiftk_andk,
   fold_simplify_andk_shiftk,
   fold_reassoc_intarith_k,
   fold_reassoc_intarith_k64,
   fold_reassoc_dup,
   fold_reassoc_bxor,
   fold_reassoc_shift,
   fold_reassoc_minmax_k,
   fold_reassoc_minmax_left,
   fold_reassoc_minmax_right,
   fold_abc_fwd,
   fold_abc_k,
   fold_abc_invar,
   fold_comm_swap,
   fold_comm_equal,
   fold_comm_comp,
   fold_comm_dup,
   fold_comm_bxor,
   fold_merge_eqne_snew_kgc,
   lj_opt_fwd_aload,
   fold_kfold_hload_kkptr,
   lj_opt_fwd_hload,
   lj_opt_fwd_uload,
   lj_opt_fwd_tab_len,
   fold_cse_uref,
   lj_opt_fwd_hrefk,
   fold_fwd_href_tnew,
   fold_fwd_href_tdup,
   fold_fload_tab_tnew_asize,
   fold_fload_tab_tnew_hmask,
```

```
  fold_fload_tab_tdup_asize,
  fold_fload_tab_tdup_hmask,
  fold_fload_tab_ah,
  fold_fload_str_len_kgc,
  fold_fload_str_len_snew,
  fold_fload_str_len_tostr,
  fold_fload_cdata_typeid_kgc,
  fold_fload_cdata_int64_kgc,
  fold_fload_cdata_typeid_cnew,
  fold_fload_cdata_ptr_int64_cnew,
  lj_opt_cse,
  lj_opt_fwd_fload,
  fold_fwd_sload,
  fold_xload_kptr,
  lj_opt_fwd_xload,
  fold_barrier_tab,
  fold_barrier_tnew_tdup,
  fold_prof,
  lj_opt_dse_ahstore,
  lj_opt_dse_ustore,
  lj_opt_dse_fstore,
  lj_opt_dse_xstore,
  lj_ir_emit
};

static const uint32_t fold_hash[978] = {
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x0b54741d,
0x47b4aa76,
0xffffffff,
0x0c58741d,
0x898bfc11,
0xffffffff,
0x0c5c741d,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x47b4aa96,
0xffffffff,
0x035c7017,
0xffffffff,
0x1cc18c0c,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x316ad401,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x5f485fff,
0xffffffff,
0xffffffff,
0x604c5fff,
0xffffffff,
0xffffffff,
```

```
0x60505fff,
0x6352a417,
0x64468c1d,
0x4d545fff,
0xffffffff,
0xffffffff,
0x2cb873ff,
0xffffffff,
0xffffffff,
0x0402701c,
0x868a6010,
0xffffffff,
0x0406701c,
0xffffffff,
0xffffffff,
0x040a701c,
0xffffffff,
0xffffffff,
0x040e701c,
0x818bfc07,
0xffffffff,
0x0412701c,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x1000741d,
0x3b56bc1c,
0xffffffff,
0x1004741d,
0xffffffff,
0xffffffff,
0x1008741d,
0x129c6817,
0xffffffff,
0x100c741d,
0x888b4c0f,
0xffffffff,
0x3010741d,
0x139dfc17,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x3fb569d3,
0xffffffff,
0xffffffff,
0xffffffff,
0x5f4a77ff,
0xffffffff,
0x5e49fc21,
0x604e77ff,
0xffffffff,
0x5e4dfc21,
0xffffffff,
0xffffffff,
0x5e51fc21,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x1cc18c0d,
0xffffffff,
0x4a6dfc17,
0xffffffff,
0x6750a017,
0x6444881d,
0x4b71fc17,
0xffffffff,
0xffffffff,
```

```
0xffffffff,
0xffffffff,
0xffffffff,
0x3013fc1b,
0xffffffff,
0x316ad402,
0xffffffff,
0xffffffff,
0x61488417,
0x7c774418,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x61508417,
0x3a57fc1c,
0x5654abff,
0xffffffff,
0xffffffff,
0xffffffff,
0x95c1ffff,
0xffffffff,
0xffffffff,
0x7d8b4008,
0x95c5ffff,
0xffffffff,
0x868a6011,
0xffffffff,
0x17ab5056,
0x5b45fc17,
0xffffffff,
0xffffffff,
0x5d49fc17,
0xffffffff,
0xffffffff,
0x5d4dfc17,
0x818bfc08,
0xffffffff,
0x5d51fc17,
0xffffffff,
0xffffffff,
0x4c55fc17,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x3d5dfc17,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x888b4c10,
0x6868d017,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x3e5c73ff,
0xffffffff,
0xffffffff,
0x95a1ffff,
0xffffffff,
0xffffffff,
0x95a5ffff,
0x674e9c17,
0x6442841d,
0x95a9ffff,
0xffffffff,
0xffffffff,
0x19adffff,
0xffffffff,
0xffffffff,
0x8eb1ffff,
0xffffffff,
```

```
0xffffffff,
0x48b5ffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x14806017,
0x1cc18c0e,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x5a43fc1d,
0xffffffff,
0xffffffff,
0x5c47fc1d,
0xffffffff,
0xffffffff,
0xffffffff,
0x1e52681d,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x4e53fc1d,
0xffffffff,
0xffffffff,
0x5157fc1d,
0x1e526417,
0x8c8c6bff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x8d8dffff,
0xffffffff,
0x7e8b4009,
0x5954a429,
0x8991ffff,
0xffffffff,
0x9195ffff,
0xffffffff,
0xffffffff,
0x9399ffff,
0x49b4aa6e,
0xffffffff,
0x6c15fc17,
0x6866cc17,
0x818bfc09,
0x23b45e6f,
0x44b7681c,
0x0f407400,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x674c9817,
0xffffffff,
0xffffffff,
0xffffffff,
0x073e5c00,
0xffffffff,
0xffffffff,
0xffffffff,
0x888b4c11,
0xffffffff,
0xffffffff,
0xffffffff,
0x49b4aaae,
0xffffffff,
0xffffffff,
```

```
0x7169ffff,
0xffffffff,
0xffffffff,
0x6e6dffff,
0xffffffff,
0xffffffff,
0x6e71ffff,
0xffffffff,
0xffffffff,
0x7a75ffff,
0x28b4726e,
0xffffffff,
0x9579ffff,
0xffffffff,
0x95b20000,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x29b4728e,
0x1cc18c0f,
0xffffffff,
0xffffffff,
0xffffffff,
0x25b475d5,
0xffffffff,
0x110bfc1d,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x2ab472ae,
0xffffffff,
0xffffffff,
0xffffffff,
0x7145ffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x2bb472ce,
0x05665c17,
0x62429417,
0x5455ffff,
0xffffffff,
0xffffffff,
0x47b4ae75,
0x066e5c17,
0x30126fff,
0x674a9417,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x6a67fc34,
0xffffffff,
0x6568d3ff,
0xffffffff,
0x42b56a75,
0x23b45e70,
0x47b4ae95,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x8faf4000,
0xffffffff,
0xffffffff,
```

```
0x95a3fc00,
0xffffffff,
0xffffffff,
0x42b56a95,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x3c58bc2f,
0x15813bff,
0x8eaffc00,
0xffffffff,
0x05425c17,
0xffffffff,
0xffffffff,
0x05465c17,
0x1d52601d,
0x27b47675,
0x054a5c17,
0xffffffff,
0xffffffff,
0x054e5c17,
0xffffffff,
0xffffffff,
0x05525c17,
0xffffffff,
0xffffffff,
0x05565c17,
0xffffffff,
0x27b47695,
0x055a5c17,
0xffffffff,
0xffffffff,
0x055e5c17,
0x49b6a41c,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x75866800,
0xffffffff,
0xffffffff,
0x26b475d6,
0x7001ffff,
0x7687fc00,
0xffffffff,
0x7005ffff,
0x898bfc00,
0xffffffff,
0x7009ffff,
0x67489017,
0xffffffff,
0x700dffff,
0xffffffff,
0xffffffff,
0x6f11ffff,
0xffffffff,
0xffffffff,
0x6d15ffff,
0x6566cfff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x0060701c,
0x3955fc2f,
0xffffffff,
0x0064701c,
0x0d487417,
0x47b4ae76,
0x2f13141b,
0x0068701c,
0x0d4c7417,
0xffffffff,
```

```
0x0d507417,
0xffffffff,
0xffffffff,
0x797c63ff,
0xffffffff,
0xffffffff,
0x42b56a76,
0x23b45e71,
0x47b4ae96,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x42b56a96,
0x09025c17,
0xffffffff,
0xffffffff,
0x09065c17,
0xffffffff,
0xffffffff,
0x090a5c17,
0xffffffff,
0xffffffff,
0x090e5c17,
0x6456ac1d,
0xffffffff,
0x30125c17,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x73113818,
0xffffffff,
0xffffffff,
0x63468c17,
0x0054701c,
0x47b4a675,
0xffffffff,
0x30106018,
0x0058701c,
0xffffffff,
0x0b42741d,
0xffffffff,
0xffffffff,
0x0b46741d,
0xffffffff,
0xffffffff,
0xffffffff,
0x47b4a695,
0xffffffff,
0xffffffff,
0xffffffff,
0x898bfc01,
0x0b52741d,
0xffffffff,
0x848b7000,
0x0b56741d,
0xffffffff,
0xffffffff,
0x0c5a741d,
0x7c77441c,
0xffffffff,
0xffffffff,
0xffffffff,
```

```
0xffffffff,
0xffffffff,
0x3360bc1c,
0xffffffff,
0xffffffff,
0xffffffff,
0x2eba6000,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x34408000,
0xffffffff,
0x23b45e72,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x5f4a5fff,
0x838b3800,
0xffffffff,
0x604e5fff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x828a6000,
0xffffffff,
0xffffffff,
0xffffffff,
0x0400701c,
0x535a5fff,
0x63448817,
0x0404701c,
0xffffffff,
0xffffffff,
0x0408701c,
0xffffffff,
0xffffffff,
0x040c701c,
0x878b480e,
0x21b45dd3,
0x0410701c,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x3854bc1c,
0x7f8b4408,
0x1002741d,
0x3b58bc1c,
0x47b4a676,
0x1006741d,
0xffffffff,
0xffffffff,
0x100a741d,
0xffffffff,
0xffffffff,
0x100e741d,
0xffffffff,
0xffffffff,
0x3012741d,
0xffffffff,
0xffffffff,
```

```
0x47b4a696,
0xffffffff,
0xffffffff,
0xffffffff,
0x5f4877ff,
0xffffffff,
0xffffffff,
0x604c77ff,
0xffffffff,
0x5e4bfc21,
0x605077ff,
0xffffffff,
0x5e4ffc21,
0x4d5477ff,
0xffffffff,
0xffffffff,
0xffffffff,
0x43b569ae,
0xffffffff,
0xffffffff,
0x6452a41d,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x4a6ffc17,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x3011fc1b,
0xffffffff,
0x63428417,
0x026a73ff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x614a8417,
0xffffffff,
0xffffffff,
0x614e8417,
0x3755fc1c,
0x78c1fc1e,
0x3a59fc1c,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x95c3ffff,
0xffffffff,
0x66468fff,
0xffffffff,
0x6868d01c,
0xffffffff,
0x5a43fc17,
0xffffffff,
0xffffffff,
0x5c47fc17,
0xffffffff,
0x24b45eb3,
0x5d4bfc17,
0x22b45dd4,
0x1e526817,
0x5d4ffc17,
0xffffffff,
0xffffffff,
0x4a53fc17,
0xffffffff,
0xffffffff,
0x5057fc17,
```

```
0x808b4409,
0xffffffff,
0x525bfc17,
0x24b45ed3,
0xffffffff,
0xffffffff,
0x343e7c00,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x1ac18c13,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x40b56a6e,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x456b6800,
0x18abffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x40b56a8e,
0x08405c00,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x95bfffff,
0xffffffff,
0xffffffff,
0x5554a7ff,
0x41b56aae,
0xffffffff,
0xffffffff,
0xffffffff,
0x5b45fc1d,
0xffffffff,
0x65448bff,
0xffffffff,
0x6866cc1c,
0xffffffff,
0x898bfc0e,
0xffffffff,
0xffffffff,
0x41b56ace,
0xffffffff,
0xffffffff,
0x4f55fc1d,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x8a8bffff,
0x1ac18c09,
0xffffffff,
0x8b8fffff,
0xffffffff,
0xffffffff,
0x0a0bfc17,
0x9193ffff,
```

```
0xffffffff,
0x9297ffff,
0xffffffff,
0xffffffff,
0x6a69fc33,
0x949bffff,
0x24b45eb4,
0x959fffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x1f53fc18,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x24b45ed4,
0xffffffff,
0x3552bfff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x1ac18c14,
0xffffffff,
0x345ebfff,
0xffffffff,
0x7167ffff,
0x7b7743ff,
0x8faf4400,
0xffffffff,
0xffffffff,
0x898bfc04,
0x546fffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x456b6801,
0x8177ffff,
0xffffffff,
0xffffffff,
0x654287ff,
0x8e7bffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x1e52641d,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x1d526017,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x49b6a81c,
0x7143ffff,
0x898bfc0f,
0xffffffff,
0x7247ffff,
0xffffffff,
0x0e3e7400,
```

```
0x49b4a66e,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x6e53ffff,
0x05685c17,
0x1cc18c0a,
0x6e57ffff,
0x066c5c17,
0x30106fff,
0xffffffff,
0x06705c17,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x6966d3ff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x49b4a6ae,
0xffffffff,
0x1f53fc19,
0xffffffff,
0xffffffff,
0x20b6701c,
0xffffffff,
 0xffffffff,
 0xffffffff,
 0xffffffff,
 0x5855fc29,
 0xffffffff,
 0xffffffff,
 0x3c56bc2f,
 0xffffffff,
 0xffffffff,
 0x902bffff,
 0x858a600e,
 0x1ac18c15,
 0xffffffff,
 0x05445c17,
 0xffffffff,
 0xffffffff,
 0x05485c17,
 0xffffffff,
 0xffffffff,
 0x054c5c17,
 0xffffffff,
 0xffffffff,
 0x05505c17,
 0xffffffff,
 0xffffffff,
 0x05545c17,
 0x456b6802,
 0xffffffff,
 0xffffffff,
 0xffffffff,
 0xffffffff,
 0xffffffff,
 0xffffffff,
 0xffffffff,
 0xffffffff,
 0xffffffff,
 0xffffffff,
 0xffffffff,
 0xffffffff,
 0xffffffff,
 0xffffffff,
 0x6356ac17,
```

```
0x7485fc00,
0xffffffff,
0x7003ffff,
0x62429017,
0x7789fc00,
0x7007ffff,
0xffffffff,
0x47b4aa75,
0x700bffff,
0xffffffff,
0xffffffff,
0x700fffff,
0x898bfc10,
0xffffffff,
0x6f13ffff,
0xffffffff,
0xffffffff,
0x9517ffff,
0x6968cfff,
0xffffffff,
0x47b4aa95,
0x2db85fff,
0x3653fc2f,
0xffffffff,
0x0062701c,
0x1cc18c0b,
0xffffffff,
0x2f11141b,
0x0066701c,
0x0d4a7417,
0xffffffff,
0x0d4e7417,
0xffffffff,
0xffffffff,
0x797a63ff,
0xffffffff,
0xffffffff,
0x46b5feb3,
0x3260c01c,
0x316ad400,
0xffffffff,
0x7c774416,
0xffffffff,
0xffffffff,
0x6b15fc29,
0x1f53fc1a,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0x46b5fed3,
0x09005c17,
0x5755fc2a,
0x16bd8c00,
0x09045c17,
0x01647017,
0xffffffff,
0x09085c17,
0xffffffff,
0x868a600f,
0x1bc18c16,
0x090c5c17,
0xffffffff,
0x30105c17,
0xffffffff,
0xffffffff,
0x09145c17,
0xffffffff,
0x818bfc06,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
0xffffffff,
```

```
0xffffffff,
0x73133818,
0x0052701c,
0xffffffff,
0xffffffff,
0x0056701c,
0xffffffff,
0xffffffff,
0x30126018,
0xffffffff,
0x878b4c0e,
0x0b44741d,
0x005e701c,
0xffffffff,
0xffffffff,
0xffffffff
};

#define fold_hashkey(k)         (lj_rol(lj_rol((k),10)-(k),1)%977)

```

One Level Up Top Level

src/lj_gdbjit.h - luajit-2.0-src

Macros defined

- _LJ_GDBJIT_H
- lj_gdbjit_addtrace
- lj_gdbjit_deltrace

Source code

```
1  /*
2  ** Client for the GDB JIT API.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_GDBJIT_H
7  #define _LJ_GDBJIT_H
8
9  #include "lj_obj.h"
10 #include "lj_jit.h"
11
12 #if LJ_HASJIT && defined(LUAJIT_USE_GDBJIT)
13
14 LJ_FUNC void lj_gdbjit_addtrace(jit_State *J, GCtrace *T);
15 LJ_FUNC void lj_gdbjit_deltrace(jit_State *J, GCtrace *T);
16
17 #else
18 #define lj_gdbjit_addtrace(J, T)        UNUSED(T)
19 #define lj_gdbjit_deltrace(J, T)        UNUSED(T)
20 #endif
21
22 #endif
```

src/host/buildvm_asm.c - luajit-2.0-src

Global variables defined

- jccnames

Functions defined

- emit_asm
- emit_asm_align
- emit_asm_bytes
- emit_asm_label
- emit_asm_reloc
- emit_asm_reloc_text
- emit_asm_wordreloc
- emit_asm_words

Macros defined

- ELFASM_PX
- ELFASM_PX

Source code

```
1  /*
2  ** LuaJIT VM builder: Assembler source code emitter.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #include "buildvm.h"
7  #include "lj_bc.h"
8
9  /* ------------------------------------------------------------------------ */
10
11  #if LJ_TARGET_X86ORX64
12  /* Emit bytes piecewise as assembler text. */
13  static void emit_asm_bytes(BuildCtx *ctx, uint8_t *p, int n)
14  {
15    int i;
16    for (i = 0; i < n; i++) {
17      if ((i & 15) == 0)
18        fprintf(ctx->fp, "\t.byte %d", p[i]);
19      else
20        fprintf(ctx->fp, ",%d", p[i]);
21      if ((i & 15) == 15) putc('\n', ctx->fp);
22    }
23    if ((n & 15) != 0) putc('\n', ctx->fp);
24  }
25
26  /* Emit relocation */
27  static void emit_asm_reloc(BuildCtx *ctx, int type, const char *sym)
28  {
29    switch (ctx->mode) {
30    case BUILD_elfasm:
31      if (type)
32        fprintf(ctx->fp, "\t.long %s-.-4\n", sym);
33      else
```

```
      fprintf(ctx->fp, "\t.long %s\n", sym);
    break;
  case BUILD_coffasm:
    fprintf(ctx->fp, "\t.def %s; .scl 3; .type 32; .endef\n", sym);
    if (type)
      fprintf(ctx->fp, "\t.long %s-.-4\n", sym);
    else
      fprintf(ctx->fp, "\t.long %s\n", sym);
    break;
  default:  /* BUILD_machasm for relative relocations handled below. */
    fprintf(ctx->fp, "\t.long %s\n", sym);
    break;
  }
}

static const char *const jccnames[] = {
  "jo", "jno", "jb", "jnb", "jz", "jnz", "jbe", "ja",
  "js", "jns", "jpe", "jpo", "jl", "jge", "jle", "jg"
};

/* Emit x86/x64 text relocations. */
static void emit_asm_reloc_text(BuildCtx *ctx, uint8_t *cp, int n,
                                const char *sym)
{
  const char *opname = NULL;
  if (--n < 0) goto err;
  if (cp[n] == 0xe8) {
    opname = "call";
  } else if (cp[n] == 0xe9) {
    opname = "jmp";
  } else if (cp[n] >= 0x80 && cp[n] <= 0x8f && n > 0 && cp[n-1] == 0x0f) {
    opname = jccnames[cp[n]-0x80];
    n--;
  } else {
err:
    fprintf(stderr, "Error: unsupported opcode for %s symbol relocation.\n",
            sym);
    exit(1);
  }
  emit_asm_bytes(ctx, cp, n);
  if (strncmp(sym+(*sym == '_'), LABEL_PREFIX, sizeof(LABEL_PREFIX)-1)) {
    /* Various fixups for external symbols outside of our binary. */
    if (ctx->mode == BUILD_elfasm) {
      if (LJ_32)
        fprintf(ctx->fp, "#if __PIC__\n\t%s lj_wrap_%s\n#else\n", opname, sym);
      fprintf(ctx->fp, "\t%s %s@PLT\n", opname, sym);
      if (LJ_32)
        fprintf(ctx->fp, "#endif\n");
      return;
    } else if (LJ_32 && ctx->mode == BUILD_machasm) {
      fprintf(ctx->fp, "\t%s L%s$stub\n", opname, sym);
      return;
    }
  }
  fprintf(ctx->fp, "\t%s %s\n", opname, sym);
}
#else
/* Emit words piecewise as assembler text. */
static void emit_asm_words(BuildCtx *ctx, uint8_t *p, int n)
{
  int i;
  for (i = 0; i < n; i += 4) {
    if ((i & 15) == 0)
      fprintf(ctx->fp, "\t.long 0x%08x", *(uint32_t *)(p+i));
    else
      fprintf(ctx->fp, ",0x%08x", *(uint32_t *)(p+i));
    if ((i & 15) == 12) putc('\n', ctx->fp);
  }
  if ((n & 15) != 0) putc('\n', ctx->fp);
}

/* Emit relocation as part of an instruction. */
static void emit_asm_wordreloc(BuildCtx *ctx, uint8_t *p, int n,
                               const char *sym)
{
  uint32_t ins;
```

```
  emit_asm_words(ctx, p, n-4);
  ins = *(uint32_t *)(p+n-4);
#if LJ_TARGET_ARM
  if ((ins & 0xff000000u) == 0xfa000000u) {
    fprintf(ctx->fp, "\tblx %s\n", sym);
  } else if ((ins & 0x0e000000u) == 0x0a000000u) {
    fprintf(ctx->fp, "\t%s%.2s %s\n", (ins & 0x01000000u) ? "bl" : "b",
            &"eqnecsccmiplvsvchilsgeltgtle"[2*(ins >> 28)], sym);
  } else {
    fprintf(stderr,
            "Error: unsupported opcode %08x for %s symbol relocation.\n",
            ins, sym);
    exit(1);
  }
#elif LJ_TARGET_ARM64
  if ((ins >> 26) == 0x25u) {
    fprintf(ctx->fp, "\tbl %s\n", sym);
  } else {
    fprintf(stderr,
            "Error: unsupported opcode %08x for %s symbol relocation.\n",
            ins, sym);
    exit(1);
  }
#elif LJ_TARGET_PPC
#if LJ_TARGET_PS3
#define TOCPREFIX "."
#else
#define TOCPREFIX ""
#endif
  if ((ins >> 26) == 16) {
    fprintf(ctx->fp, "\t%s %d, %d, " TOCPREFIX "%s\n",
            (ins & 1) ? "bcl" : "bc", (ins >> 21) & 31, (ins >> 16) & 31, sym);
  } else if ((ins >> 26) == 18) {
#if LJ_ARCH_PPC64
    const char *suffix = strchr(sym, '@');
    if (suffix && suffix[1] == 'h') {
      fprintf(ctx->fp, "\taddis 11, 2, %s\n", sym);
    } else if (suffix && suffix[1] == 'l') {
      fprintf(ctx->fp, "\tld 12, %s\n", sym);
    } else
#endif
    fprintf(ctx->fp, "\t%s " TOCPREFIX "%s\n", (ins & 1) ? "bl" : "b", sym);
  } else {
    fprintf(stderr,
            "Error: unsupported opcode %08x for %s symbol relocation.\n",
            ins, sym);
    exit(1);
  }
#elif LJ_TARGET_MIPS
  fprintf(stderr,
          "Error: unsupported opcode %08x for %s symbol relocation.\n",
          ins, sym);
  exit(1);
#else
#error "missing relocation support for this architecture"
#endif
}
#endif

#if LJ_TARGET_ARM
#define ELFASM_PX       "%%"
#else
#define ELFASM_PX       "@"
#endif

/* Emit an assembler label. */
static void emit_asm_label(BuildCtx *ctx, const char *name, int size, int isfunc)
{
  switch (ctx->mode) {
  case BUILD_elfasm:
#if LJ_TARGET_PS3
    if (!strncmp(name, "lj_vm_", 6) &&
        strcmp(name, ctx->beginsym) &&
        !strstr(name, "hook")) {
      fprintf(ctx->fp,
        "\n\t.globl %s\n"
```

```
        "\t.section \".opd\",\"aw\"\n"
        "%s:\n"
        "\t.long .%s,.TOC.@tocbase32\n"
        "\t.size %s,8\n"
        "\t.previous\n"
        "\t.globl .%s\n"
        "\t.hidden .%s\n"
        "\t.type .%s, " ELFASM_PX "function\n"
        "\t.size .%s, %d\n"
        ".%s:\n",
        name, name, name, name, name, name, name, name, size, name);
      break;
    }
#endif
    fprintf(ctx->fp,
      "\n\t.globl %s\n"
      "\t.hidden %s\n"
      "\t.type %s, " ELFASM_PX "%s\n"
      "\t.size %s, %d\n"
      "%s:\n",
      name, name, name, isfunc ? "function" : "object", name, size, name);
    break;
  case BUILD_coffasm:
    fprintf(ctx->fp, "\n\t.globl %s\n", name);
    if (isfunc)
      fprintf(ctx->fp, "\t.def %s; .scl 3; .type 32; .endef\n", name);
    fprintf(ctx->fp, "%s:\n", name);
    break;
  case BUILD_machasm:
    fprintf(ctx->fp,
      "\n\t.private_extern %s\n"
      "%s:\n", name, name);
    break;
  default:
    break;
  }
}

/* Emit alignment. */
static void emit_asm_align(BuildCtx *ctx, int bits)
{
  switch (ctx->mode) {
  case BUILD_elfasm:
  case BUILD_coffasm:
    fprintf(ctx->fp, "\t.p2align %d\n", bits);
    break;
  case BUILD_machasm:
    fprintf(ctx->fp, "\t.align %d\n", bits);
    break;
  default:
    break;
  }
}

/* ---------------------------------------------------------------------- */

/* Emit assembler source code. */
void emit_asm(BuildCtx *ctx)
{
  int i, rel;

  fprintf(ctx->fp, "\t.file \"buildvm_%s.dasc\"\n", ctx->dasm_arch);
#if LJ_ARCH_PPC64
  fprintf(ctx->fp, "\t.abiversion 2\n");
#endif
  fprintf(ctx->fp, "\t.text\n");
  emit_asm_align(ctx, 4);

#if LJ_TARGET_PS3
  emit_asm_label(ctx, ctx->beginsym, ctx->codesz, 0);
#else
  emit_asm_label(ctx, ctx->beginsym, 0, 0);
#endif
  if (ctx->mode != BUILD_machasm)
    fprintf(ctx->fp, ".Lbegin:\n");

```

```
#if LJ_TARGET_ARM && defined(__GNUC__) && !LJ_NO_UNWIND
  /* This should really be moved into buildvm_arm.dasc. */
  fprintf(ctx->fp,
          ".fnstart\n"
          ".save {r4, r5, r6, r7, r8, r9, r10, r11, lr}\n"
          ".pad #28\n");
#endif
#if LJ_TARGET_MIPS
  fprintf(ctx->fp, ".set nomips16\n.abicalls\n.set noreorder\n.set nomacro\n");
#endif

  for (i = rel = 0; i < ctx->nsym; i++) {
    int32_t ofs = ctx->sym[i].ofs;
    int32_t next = ctx->sym[i+1].ofs;
#if LJ_TARGET_ARM && defined(__GNUC__) && !LJ_NO_UNWIND && LJ_HASFFI
    if (!strcmp(ctx->sym[i].name, "lj_vm_ffi_call"))
      fprintf(ctx->fp,
              ".globl lj_err_unwind_arm\n"
              ".personality lj_err_unwind_arm\n"
              ".fnend\n"
              ".fnstart\n"
              ".save {r4, r5, r11, lr}\n"
              ".setfp r11, sp\n");
#endif
    emit_asm_label(ctx, ctx->sym[i].name, next - ofs, 1);
    while (rel < ctx->nreloc && ctx->reloc[rel].ofs <= next) {
      BuildReloc *r = &ctx->reloc[rel];
      int n = r->ofs - ofs;
#if LJ_TARGET_X86ORX64
      if (r->type != 0 &&
          (ctx->mode == BUILD_elfasm || ctx->mode == BUILD_machasm)) {
        emit_asm_reloc_text(ctx, ctx->code+ofs, n, ctx->relocsym[r->sym]);
      } else {
        emit_asm_bytes(ctx, ctx->code+ofs, n);
        emit_asm_reloc(ctx, r->type, ctx->relocsym[r->sym]);
      }
      ofs += n+4;
#else
      emit_asm_wordreloc(ctx, ctx->code+ofs, n, ctx->relocsym[r->sym]);
      ofs += n;
#endif
      rel++;
    }
#if LJ_TARGET_X86ORX64
    emit_asm_bytes(ctx, ctx->code+ofs, next-ofs);
#else
    emit_asm_words(ctx, ctx->code+ofs, next-ofs);
#endif
  }

#if LJ_TARGET_ARM && defined(__GNUC__) && !LJ_NO_UNWIND
  fprintf(ctx->fp,
#if !LJ_HASFFI
          ".globl lj_err_unwind_arm\n"
          ".personality lj_err_unwind_arm\n"
#endif
          ".fnend\n");
#endif

  fprintf(ctx->fp, "\n");
  switch (ctx->mode) {
  case BUILD_elfasm:
#if !(LJ_TARGET_PS3 || LJ_TARGET_PSVITA)
    fprintf(ctx->fp, "\t.section .note.GNU-stack,\"\"," ELFASM_PX "progbits\n");
#endif
#if LJ_TARGET_PPC && !LJ_TARGET_PS3
    /* Hard-float ABI. */
    fprintf(ctx->fp, "\t.gnu_attribute 4, 1\n");
#endif
    /* fallthrough */
  case BUILD_coffasm:
    fprintf(ctx->fp, "\t.ident \"%s\"\n", ctx->dasm_ident);
    break;
  case BUILD_machasm:
    fprintf(ctx->fp,
      "\t.cstring\n"
```

```
338       "\t.ascii \"%s\\0\"\n", ctx->dasm_ident);
339     break;
340   default:
341     break;
342   }
343   fprintf(ctx->fp, "\n");
344 }
345
```

One Level Up Top Level

src/host/buildvm_peobj.c - luajit-2.0-src

Global variables defined

- strtab
- strtabofs

Data types defined

- PEheader
- PEheader
- PEreloc
- PEreloc
- PEsection
- PEsection
- PEsym
- PEsym
- PEsymaux
- PEsymaux

Functions defined

- emit_peobj
- emit_peobj
- emit_peobj_sym
- emit_peobj_sym_sect

Macros defined

- PEOBJ_ARCH_TARGET
- PEOBJ_ARCH_TARGET
- PEOBJ_ARCH_TARGET
- PEOBJ_RELOC_ADDR32NB
- PEOBJ_RELOC_DIR32
- PEOBJ_RELOC_DIR32
- PEOBJ_RELOC_DIR32
- PEOBJ_RELOC_OFS
- PEOBJ_RELOC_OFS

- PEOBJ_RELOC_OFS
- PEOBJ_RELOC_REL32
- PEOBJ_RELOC_REL32
- PEOBJ_RELOC_REL32
- PEOBJ_RELOC_SIZE
- PEOBJ_SCL_EXTERN
- PEOBJ_SCL_STATIC
- PEOBJ_SYM_SIZE
- PEOBJ_TEXT_FLAGS
- PEOBJ_TEXT_FLAGS
- PEOBJ_TEXT_FLAGS
- PEOBJ_TYPE_FUNC
- PEOBJ_TYPE_NULL

Source code

```
/*
** LuaJIT VM builder: PE object emitter.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
**
** Only used for building on Windows, since we cannot assume the presence
** of a suitable assembler. The host and target byte order must match.
*/

#include "buildvm.h"
#include "lj_bc.h"

#if LJ_TARGET_X86ORX64 || LJ_TARGET_PPC

/* Context for PE object emitter. */
static char *strtab;
static size_t strtabofs;

/* -- PE object definitions ---------------------------------------------- */

/* PE header. */
typedef struct PEheader {
  uint16_t arch;
  uint16_t nsects;
  uint32_t time;
  uint32_t symtabofs;
  uint32_t nsyms;
  uint16_t opthdrsz;
  uint16_t flags;
} PEheader;

/* PE section. */
typedef struct PEsection {
  char name[8];
  uint32_t vsize;
  uint32_t vaddr;
  uint32_t size;
  uint32_t ofs;
  uint32_t relocofs;
  uint32_t lineofs;
  uint16_t nreloc;
  uint16_t nline;
  uint32_t flags;
} PEsection;
```

```
44
45  /* PE relocation. */
46  typedef struct PEreloc {
47    uint32_t vaddr;
48    uint32_t symidx;
49    uint16_t type;
50  } PEreloc;
51
52  /* Cannot use sizeof, because it pads up to the max. alignment. */
53  #define PEOBJ_RELOC_SIZE        (4+4+2)
54
55  /* PE symbol table entry. */
56  typedef struct PEsym {
57    union {
58      char name[8];
59      uint32_t nameref[2];
60    } n;
61    uint32_t value;
62    int16_t sect;
63    uint16_t type;
64    uint8_t scl;
65    uint8_t naux;
66  } PEsym;
67
68  /* PE symbol table auxiliary entry for a section. */
69  typedef struct PEsymaux {
70    uint32_t size;
71    uint16_t nreloc;
72    uint16_t nline;
73    uint32_t cksum;
74    uint16_t assoc;
75    uint8_t comdatsel;
76    uint8_t unused[3];
77  } PEsymaux;
78
79  /* Cannot use sizeof, because it pads up to the max. alignment. */
80  #define PEOBJ_SYM_SIZE        (8+4+2+2+1+1)
81
82  /* PE object CPU specific defines. */
83  #if LJ_TARGET_X86
84  #define PEOBJ_ARCH_TARGET        0x014c
85  #define PEOBJ_RELOC_REL32        0x14  /* MS: REL32, GNU: DISP32. */
86  #define PEOBJ_RELOC_DIR32        0x06
87  #define PEOBJ_RELOC_OFS             0
88  #define PEOBJ_TEXT_FLAGS        0x60500020  /* 60=r+x, 50=align16, 20=code. */
89  #elif LJ_TARGET_X64
90  #define PEOBJ_ARCH_TARGET        0x8664
91  #define PEOBJ_RELOC_REL32        0x04  /* MS: REL32, GNU: DISP32. */
92  #define PEOBJ_RELOC_DIR32        0x02
93  #define PEOBJ_RELOC_ADDR32NB        0x03
94  #define PEOBJ_RELOC_OFS             0
95  #define PEOBJ_TEXT_FLAGS        0x60500020  /* 60=r+x, 50=align16, 20=code. */
96  #elif LJ_TARGET_PPC
97  #define PEOBJ_ARCH_TARGET        0x01f2
98  #define PEOBJ_RELOC_REL32        0x06
99  #define PEOBJ_RELOC_DIR32        0x02
100 #define PEOBJ_RELOC_OFS             (-4)
101 #define PEOBJ_TEXT_FLAGS        0x60400020  /* 60=r+x, 40=align8, 20=code. */
102 #endif
103
104 /* Section numbers (0-based). */
105 enum {
106   PEOBJ_SECT_ABS = -2,
107   PEOBJ_SECT_UNDEF = -1,
108   PEOBJ_SECT_TEXT,
109 #if LJ_TARGET_X64
110   PEOBJ_SECT_PDATA,
111   PEOBJ_SECT_XDATA,
112 #endif
113   PEOBJ_SECT_RDATA_Z,
114   PEOBJ_NSECTIONS
115 };
116
117 /* Symbol types. */
118 #define PEOBJ_TYPE_NULL             0
119 #define PEOBJ_TYPE_FUNC             0x20
```

```c

/* Symbol storage class. */
#define PEOBJ_SCL_EXTERN        2
#define PEOBJ_SCL_STATIC        3

/* -- PE object emitter ---------------------------------------------- */

/* Emit PE object symbol. */
static void emit_peobj_sym(BuildCtx *ctx, const char *name, uint32_t value,
                           int sect, int type, int scl)
{
  PEsym sym;
  size_t len = strlen(name);
  if (!strtab) {  /* Pass 1: only calculate string table length. */
    if (len > 8) strtabofs += len+1;
    return;
  }
  if (len <= 8) {
    memcpy(sym.n.name, name, len);
    memset(sym.n.name+len, 0, 8-len);
  } else {
    sym.n.nameref[0] = 0;
    sym.n.nameref[1] = (uint32_t)strtabofs;
    memcpy(strtab + strtabofs, name, len);
    strtab[strtabofs+len] = 0;
    strtabofs += len+1;
  }
  sym.value = value;
  sym.sect = (int16_t)(sect+1);  /* 1-based section number. */
  sym.type = (uint16_t)type;
  sym.scl = (uint8_t)scl;
  sym.naux = 0;
  owrite(ctx, &sym, PEOBJ_SYM_SIZE);
}

/* Emit PE object section symbol. */
static void emit_peobj_sym_sect(BuildCtx *ctx, PEsection *pesect, int sect)
{
  PEsym sym;
  PEsymaux aux;
  if (!strtab) return;  /* Pass 1: no output. */
  memcpy(sym.n.name, pesect[sect].name, 8);
  sym.value = 0;
  sym.sect = (int16_t)(sect+1);  /* 1-based section number. */
  sym.type = PEOBJ_TYPE_NULL;
  sym.scl = PEOBJ_SCL_STATIC;
  sym.naux = 1;
  owrite(ctx, &sym, PEOBJ_SYM_SIZE);
  memset(&aux, 0, sizeof(PEsymaux));
  aux.size = pesect[sect].size;
  aux.nreloc = pesect[sect].nreloc;
  owrite(ctx, &aux, PEOBJ_SYM_SIZE);
}

/* Emit Windows PE object file. */
void emit_peobj(BuildCtx *ctx)
{
  PEheader pehdr;
  PEsection pesect[PEOBJ_NSECTIONS];
  uint32_t sofs;
  int i, nrsym;
  union { uint8_t b; uint32_t u; } host_endian;

  sofs = sizeof(PEheader) + PEOBJ_NSECTIONS*sizeof(PEsection);

  /* Fill in PE sections. */
  memset(&pesect, 0, PEOBJ_NSECTIONS*sizeof(PEsection));
  memcpy(pesect[PEOBJ_SECT_TEXT].name, ".text", sizeof(".text")-1);
  pesect[PEOBJ_SECT_TEXT].ofs = sofs;
  sofs += (pesect[PEOBJ_SECT_TEXT].size = (uint32_t)ctx->codesz);
  pesect[PEOBJ_SECT_TEXT].relocofs = sofs;
  sofs += (pesect[PEOBJ_SECT_TEXT].nreloc = (uint16_t)ctx->nreloc) * PEOBJ_RELOC_SIZE;
  /* Flags: 60 = read+execute, 50 = align16, 20 = code. */
  pesect[PEOBJ_SECT_TEXT].flags = PEOBJ_TEXT_FLAGS;

#if LJ_TARGET_X64
```

```
  memcpy(pesect[PEOBJ_SECT_PDATA].name, ".pdata", sizeof(".pdata")-1);
  pesect[PEOBJ_SECT_PDATA].ofs = sofs;
  sofs += (pesect[PEOBJ_SECT_PDATA].size = 6*4);
  pesect[PEOBJ_SECT_PDATA].relocofs = sofs;
  sofs += (pesect[PEOBJ_SECT_PDATA].nreloc = 6) * PEOBJ_RELOC_SIZE;
  /* Flags: 40 = read, 30 = align4, 40 = initialized data. */
  pesect[PEOBJ_SECT_PDATA].flags = 0x40300040;

  memcpy(pesect[PEOBJ_SECT_XDATA].name, ".xdata", sizeof(".xdata")-1);
  pesect[PEOBJ_SECT_XDATA].ofs = sofs;
  sofs += (pesect[PEOBJ_SECT_XDATA].size = 8*2+4+6*2);  /* See below. */
  pesect[PEOBJ_SECT_XDATA].relocofs = sofs;
  sofs += (pesect[PEOBJ_SECT_XDATA].nreloc = 1) * PEOBJ_RELOC_SIZE;
  /* Flags: 40 = read, 30 = align4, 40 = initialized data. */
  pesect[PEOBJ_SECT_XDATA].flags = 0x40300040;
#endif

  memcpy(pesect[PEOBJ_SECT_RDATA_Z].name, ".rdata$Z", sizeof(".rdata$Z")-1);
  pesect[PEOBJ_SECT_RDATA_Z].ofs = sofs;
  sofs += (pesect[PEOBJ_SECT_RDATA_Z].size = (uint32_t)strlen(ctx->dasm_ident)+1);
  /* Flags: 40 = read, 30 = align4, 40 = initialized data. */
  pesect[PEOBJ_SECT_RDATA_Z].flags = 0x40300040;

  /* Fill in PE header. */
  pehdr.arch = PEOBJ_ARCH_TARGET;
  pehdr.nsects = PEOBJ_NSECTIONS;
  pehdr.time = 0;  /* Timestamp is optional. */
  pehdr.symtabofs = sofs;
  pehdr.opthdrsz = 0;
  pehdr.flags = 0;

  /* Compute the size of the symbol table:
  ** @feat.00 + nsections*2
  ** + asm_start + nsym
  ** + nrsym
  */
  nrsym = ctx->nrelocsym;
  pehdr.nsyms = 1+PEOBJ_NSECTIONS*2 + 1+ctx->nsym + nrsym;
#if LJ_TARGET_X64
  pehdr.nsyms += 1;  /* Symbol for lj_err_unwind_win64. */
#endif

  /* Write PE object header and all sections. */
  owrite(ctx, &pehdr, sizeof(PEheader));
  owrite(ctx, &pesect, sizeof(PEsection)*PEOBJ_NSECTIONS);

  /* Write .text section. */
  host_endian.u = 1;
  if (host_endian.b != LJ_ENDIAN_SELECT(1, 0)) {
#if LJ_TARGET_PPC
    uint32_t *p = (uint32_t *)ctx->code;
    int n = (int)(ctx->codesz >> 2);
    for (i = 0; i < n; i++, p++)
      *p = lj_bswap(*p);  /* Byteswap .text section. */
#else
    fprintf(stderr, "Error: different byte order for host and target\n");
    exit(1);
#endif
  }
  owrite(ctx, ctx->code, ctx->codesz);
  for (i = 0; i < ctx->nreloc; i++) {
    PEreloc reloc;
    reloc.vaddr = (uint32_t)ctx->reloc[i].ofs + PEOBJ_RELOC_OFS;
    reloc.symidx = 1+2+ctx->reloc[i].sym;  /* Reloc syms are after .text sym. */
    reloc.type = ctx->reloc[i].type ? PEOBJ_RELOC_REL32 : PEOBJ_RELOC_DIR32;
    owrite(ctx, &reloc, PEOBJ_RELOC_SIZE);
  }

#if LJ_TARGET_X64
  { /* Write .pdata section. */
    uint32_t fcofs = (uint32_t)ctx->sym[ctx->nsym-1].ofs;
    uint32_t pdata[3];  /* Start of .text, end of .text and .xdata. */
    PEreloc reloc;
    pdata[0] = 0; pdata[1] = fcofs; pdata[2] = 0;
    owrite(ctx, &pdata, sizeof(pdata));
    pdata[0] = fcofs; pdata[1] = (uint32_t)ctx->codesz; pdata[2] = 20;
```

```
    owrite(ctx, &pdata, sizeof(pdata));
    reloc.vaddr = 0; reloc.symidx = 1+2+nrsym+2+2+1;
    reloc.type = PEOBJ_RELOC_ADDR32NB;
    owrite(ctx, &reloc, PEOBJ_RELOC_SIZE);
    reloc.vaddr = 4; reloc.symidx = 1+2+nrsym+2+2+1;
    reloc.type = PEOBJ_RELOC_ADDR32NB;
    owrite(ctx, &reloc, PEOBJ_RELOC_SIZE);
    reloc.vaddr = 8; reloc.symidx = 1+2+nrsym+2;
    reloc.type = PEOBJ_RELOC_ADDR32NB;
    owrite(ctx, &reloc, PEOBJ_RELOC_SIZE);
    reloc.vaddr = 12; reloc.symidx = 1+2+nrsym+2+2+1;
    reloc.type = PEOBJ_RELOC_ADDR32NB;
    owrite(ctx, &reloc, PEOBJ_RELOC_SIZE);
    reloc.vaddr = 16; reloc.symidx = 1+2+nrsym+2+2+1;
    reloc.type = PEOBJ_RELOC_ADDR32NB;
    owrite(ctx, &reloc, PEOBJ_RELOC_SIZE);
    reloc.vaddr = 20; reloc.symidx = 1+2+nrsym+2;
    reloc.type = PEOBJ_RELOC_ADDR32NB;
    owrite(ctx, &reloc, PEOBJ_RELOC_SIZE);
  }
  { /* Write .xdata section. */
    uint16_t xdata[8+2+6];
    PEreloc reloc;
    xdata[0] = 0x01|0x08|0x10;  /* Ver. 1, uhandler/ehandler, prolog size 0. */
    xdata[1] = 0x0005;  /* Number of unwind codes, no frame pointer. */
    xdata[2] = 0x4200;  /* Stack offset 4*8+8 = aword*5. */
    xdata[3] = 0x3000;  /* Push rbx. */
    xdata[4] = 0x6000;  /* Push rsi. */
    xdata[5] = 0x7000;  /* Push rdi. */
    xdata[6] = 0x5000;  /* Push rbp. */
    xdata[7] = 0;  /* Alignment. */
    xdata[8] = xdata[9] = 0;  /* Relocated address of exception handler. */
    xdata[10] = 0x01;  /* Ver. 1, no handler, prolog size 0. */
    xdata[11] = 0x1504;  /* Number of unwind codes, fp = rbp, fpofs = 16. */
    xdata[12] = 0x0300;  /* set_fpreg. */
    xdata[13] = 0x0200;  /* stack offset 0*8+8 = aword*1. */
    xdata[14] = 0x3000;  /* Push rbx. */
    xdata[15] = 0x5000;  /* Push rbp. */
    owrite(ctx, &xdata, sizeof(xdata));
    reloc.vaddr = 2*8; reloc.symidx = 1+2+nrsym+2+2;
    reloc.type = PEOBJ_RELOC_ADDR32NB;
    owrite(ctx, &reloc, PEOBJ_RELOC_SIZE);
  }
#endif

  /* Write .rdata$Z section. */
  owrite(ctx, ctx->dasm_ident, strlen(ctx->dasm_ident)+1);

  /* Write symbol table. */
  strtab = NULL;  /* 1st pass: collect string sizes. */
  for (;;) {
    strtabofs = 4;
    /* Mark as SafeSEH compliant. */
    emit_peobj_sym(ctx, "@feat.00", 1,
                   PEOBJ_SECT_ABS, PEOBJ_TYPE_NULL, PEOBJ_SCL_STATIC);

    emit_peobj_sym_sect(ctx, pesect, PEOBJ_SECT_TEXT);
    for (i = 0; i < nrsym; i++)
      emit_peobj_sym(ctx, ctx->relocsym[i], 0,
                     PEOBJ_SECT_UNDEF, PEOBJ_TYPE_FUNC, PEOBJ_SCL_EXTERN);

#if LJ_TARGET_X64
    emit_peobj_sym_sect(ctx, pesect, PEOBJ_SECT_PDATA);
    emit_peobj_sym_sect(ctx, pesect, PEOBJ_SECT_XDATA);
    emit_peobj_sym(ctx, "lj_err_unwind_win64", 0,
                   PEOBJ_SECT_UNDEF, PEOBJ_TYPE_FUNC, PEOBJ_SCL_EXTERN);
#endif

    emit_peobj_sym(ctx, ctx->beginsym, 0,
                   PEOBJ_SECT_TEXT, PEOBJ_TYPE_NULL, PEOBJ_SCL_EXTERN);
    for (i = 0; i < ctx->nsym; i++)
      emit_peobj_sym(ctx, ctx->sym[i].name, (uint32_t)ctx->sym[i].ofs,
                     PEOBJ_SECT_TEXT, PEOBJ_TYPE_FUNC, PEOBJ_SCL_EXTERN);

    emit_peobj_sym_sect(ctx, pesect, PEOBJ_SECT_RDATA_Z);

```

```
348     if (strtab)
349       break;
350     /* 2nd pass: alloc strtab, write syms and copy strings. */
351     strtab = (char *)malloc(strtabofs);
352     *(uint32_t *)strtab = (uint32_t)strtabofs;
353   }
354
355   /* Write string table. */
356   owrite(ctx, strtab, strtabofs);
357 }
358
359 #else
360
361 void emit_peobj(BuildCtx *ctx)
362 {
363   UNUSED(ctx);
364   fprintf(stderr, "Error: no PE object support for this target\n");
365   exit(1);
366 }
367
368 #endif
```

One Level Up Top Level

src/host/genlibbc.lua - luajit-2.0-src

Functions defined

Source code

```
1  ------------------------------------------------------------------------------
2  -- Lua script to dump the bytecode of the library functions written in Lua.
3  -- The resulting 'buildvm_libbc.h' is used for the build process of LuaJIT.
4  ------------------------------------------------------------------------------
5  -- Copyright (C) 2005-2015 Mike Pall. All rights reserved.
6  -- Released under the MIT license. See Copyright Notice in luajit.h
7  ------------------------------------------------------------------------------
8
9  local ffi = require("ffi")
10 local bit = require("bit")
11 local vmdef = require("jit.vmdef")
12 local bcnames = vmdef.bcnames
13
14 local format = string.format
15
16 local isbe = (string.byte(string.dump(function() end), 5) % 2 == 1)
17
18 local function usage(arg)
19   io.stderr:write("Usage: ", arg and arg[0] or "genlibbc",
20                   " [-o buildvm_libbc.h] lib_*.c\n")
21   os.exit(1)
22 end
23
24 local function parse_arg(arg)
25   local outfile = "-"
26   if not (arg and arg[1]) then
27     usage(arg)
28   end
29   if arg[1] == "-o" then
30     outfile = arg[2]
31     if not outfile then usage(arg) end
32     table.remove(arg, 1)
33     table.remove(arg, 1)
34   end
35   return outfile
```

```lua
end

local function read_files(names)
  local src = ""
  for _,name in ipairs(names) do
    local fp = assert(io.open(name))
    src = src .. fp:read("*a")
    fp:close()
  end
  return src
end

local function transform_lua(code)
  local fixup = {}
  local n = -30000
  code = string.gsub(code, "CHECK_(%w*)%((.-)%)", function(tp, var)
    n = n + 1
    fixup[n] = { "CHECK", tp }
    return format("%s=%d", var, n)
  end)
  code = string.gsub(code, "PAIRS%((.-)%)", function(var)
    fixup.PAIRS = true
    return format("nil, %s, 0", var)
  end)
  return "return "..code, fixup
end

local function read_uleb128(p)
  local v = p[0]; p = p + 1
  if v >= 128 then
    local sh = 7; v = v - 128
    repeat
      local r = p[0]
      v = v + bit.lshift(bit.band(r, 127), sh)
      sh = sh + 7
      p = p + 1
    until r < 128
  end
  return p, v
end

-- ORDER LJ_T
local name2itype = {
  str = 5, func = 9, tab = 12, int = 14, num = 15
}

local BC = {}
for i=0,#bcnames/6-1 do
  BC[string.gsub(string.sub(bcnames, i*6+1, i*6+6), " ", "")] = i
end
local xop, xra = isbe and 3 or 0, isbe and 2 or 1
local xrc, xrb = isbe and 1 or 2, isbe and 0 or 3

local function fixup_dump(dump, fixup)
  local buf = ffi.new("uint8_t[?]", #dump+1, dump)
  local p = buf+5
  local n, sizebc
  p, n = read_uleb128(p)
  local start = p
  p = p + 4
  p = read_uleb128(p)
  p = read_uleb128(p)
  p, sizebc = read_uleb128(p)
  local rawtab = {}
   for i=0,sizebc-1 do
     local op = p[xop]
     if op == BC.KSHORT then
       local rd = p[xrc] + 256*p[xrb]
       rd = bit.arshift(bit.lshift(rd, 16), 16)
       local f = fixup[rd]
       if f then
         if f[1] == "CHECK" then
           local tp = f[2]
           if tp == "tab" then rawtab[p[xra]] = true end
           p[xop] = tp == "num" and BC.ISNUM or BC.ISTYPE
           p[xrb] = 0
```

```
          p[xrc] = name2itype[tp]
        else
          error("unhandled fixup type: "..f[1])
        end
      end
    elseif op == BC.TGETV then
      if rawtab[p[xrb]] then
        p[xop] = BC.TGETR
      end
    elseif op == BC.TSETV then
      if rawtab[p[xrb]] then
        p[xop] = BC.TSETR
      end
    elseif op == BC.ITERC then
      if fixup.PAIRS then
        p[xop] = BC.ITERN
      end
    end
    p = p + 4
  end
  return ffi.string(start, n)
end

local function find_defs(src)
  local defs = {}
  for name, code in string.gmatch(src, "LJLIB_LUA%(([^)]*)%)%s*/%*(.-)%*/") do
    local env = {}
    local tcode, fixup = transform_lua(code)
    local func = assert(load(tcode, "", nil, env))()
    defs[name] = fixup_dump(string.dump(func, true), fixup)
    defs[#defs+1] = name
  end
  return defs
end

local function gen_header(defs)
  local t = {}
  local function w(x) t[#t+1] = x end
  w("/* This is a generated file. DO NOT EDIT! */\n\n")
  w("static const int libbc_endian = ") w(isbe and 1 or 0) w(";\n\n")
  local s = ""
  for _,name in ipairs(defs) do
    s = s .. defs[name]
  end
  w("static const uint8_t libbc_code[] = {\n")
  local n = 0
  for i=1,#s do
    local x = string.byte(s, i)
    w(x); w(",")
    n = n + (x < 10 and 2 or (x < 100 and 3 or 4))
    if n >= 75 then n = 0; w("\n") end
  end
  w("0\n};\n\n")
  w("static const struct { const char *name; int ofs; } libbc_map[] = {\n")
  local m = 0
  for _,name in ipairs(defs) do
    w('{"'); w(name); w('",'); w(m) w('},\n')
    m = m + #defs[name]
  end
  w("{NULL,"); w(m); w("}\n};\n\n")
  return table.concat(t)
end

local function write_file(name, data)
  if name == "-" then
    assert(io.write(data))
    assert(io.flush())
  else
    local fp = io.open(name)
    if fp then
      local old = fp:read("*a")
      fp:close()
      if data == old then return end
    end
    fp = assert(io.open(name, "w"))
    assert(fp:write(data))
```

```
    assert(fp:close())
  end
end

local outfile = parse_arg(arg)
local src = read_files(arg)
local defs = find_defs(src)
local hdr = gen_header(defs)
write_file(outfile, hdr)

```

One Level Up Top Level

src/jit/bcsave.lua - luajit-2.0-src

Functions defined

Source code

```
1  ----------------------------------------------------------------------------
2  -- LuaJIT module to save/list bytecode.
3  --
4  -- Copyright (C) 2005-2015 Mike Pall. All rights reserved.
5  -- Released under the MIT license. See Copyright Notice in luajit.h
6  ----------------------------------------------------------------------------
7  --
8  -- This module saves or lists the bytecode for an input file.
```

```
-- It's run by the -b command line option.
--
------------------------------------------------------------------------------

local jit = require("jit")
assert(jit.version_num == 20100, "LuaJIT core/library version mismatch")
local bit = require("bit")

-- Symbol name prefix for LuaJIT bytecode.
local LJBC_PREFIX = "luaJIT_BC_"

------------------------------------------------------------------------------

local function usage()
  io.stderr:write[[
Save LuaJIT bytecode: luajit -b[options] input output
  -l        Only list bytecode.
  -s        Strip debug info (default).
  -g        Keep debug info.
  -n name   Set module name (default: auto-detect from input name).
  -t type   Set output file type (default: auto-detect from output name).
  -a arch   Override architecture for object files (default: native).
  -o os     Override OS for object files (default: native).
  -e chunk  Use chunk string as input.
  --        Stop handling options.
  -         Use stdin as input and/or stdout as output.

File types: c h obj o raw (default)
]]
  os.exit(1)
end

local function check(ok, ...)
  if ok then return ok, ... end
  io.stderr:write("luajit: ", ...)
  io.stderr:write("\n")
  os.exit(1)
end

local function readfile(input)
  if type(input) == "function" then return input end
  if input == "-" then input = nil end
  return check(loadfile(input))
end

local function savefile(name, mode)
  if name == "-" then return io.stdout end
  return check(io.open(name, mode))
end

------------------------------------------------------------------------------

local map_type = {
  raw = "raw", c = "c", h = "h", o = "obj", obj = "obj",
}

local map_arch = {
  x86 = true, x64 = true, arm = true, ppc = true,
  mips = true, mipsel = true,
}

local map_os = {
  linux = true, windows = true, osx = true, freebsd = true, netbsd = true,
  openbsd = true, dragonfly = true, solaris = true,
}

local function checkarg(str, map, err)
  str = string.lower(str)
  local s = check(map[str], "unknown ", err)
  return s == true and str or s
end

local function detecttype(str)
  local ext = string.match(string.lower(str), "%.(%a+)$")
  return map_type[ext] or "raw"
end
```

```
85
86  local function checkmodname(str)
87    check(string.match(str, "^[%w_.%-]+$"), "bad module name")
88    return string.gsub(str, "[%.%-]", "_")
89  end
90
91  local function detectmodname(str)
92    if type(str) == "string" then
93      local tail = string.match(str, "[^/\\]+$")
94      if tail then str = tail end
95      local head = string.match(str, "^(.*)%.[^.]*$")
96      if head then str = head end
97      str = string.match(str, "^[%w_.%-]+")
98    else
99      str = nil
100   end
101   check(str, "cannot derive module name, use -n name")
102   return string.gsub(str, "[%.%-]", "_")
103 end
104
105 ------------------------------------------------------------------------------
106
107 local function bcsave_tail(fp, output, s)
108   local ok, err = fp:write(s)
109   if ok and output ~= "-" then ok, err = fp:close() end
110   check(ok, "cannot write ", output, ": ", err)
111 end
112
113 local function bcsave_raw(output, s)
114   local fp = savefile(output, "wb")
115   bcsave_tail(fp, output, s)
116 end
117
118 local function bcsave_c(ctx, output, s)
119   local fp = savefile(output, "w")
120   if ctx.type == "c" then
121     fp:write(string.format([[
122 #ifdef _cplusplus
123 extern "C"
124 #endif
125 #ifdef _WIN32
126 __declspec(dllexport)
127 #endif
128 const char %s%s[] = {
129 ]], LJBC_PREFIX, ctx.modname))
130   else
131     fp:write(string.format([[
132 #define %s%s_SIZE %d
133 static const char %s%s[] = {
134 ]], LJBC_PREFIX, ctx.modname, #s, LJBC_PREFIX, ctx.modname))
135   end
136   local t, n, m = {}, 0, 0
137   for i=1,#s do
138     local b = tostring(string.byte(s, i))
139     m = m + #b + 1
140     if m > 78 then
141       fp:write(table.concat(t, ",", 1, n), ",\n")
142       n, m = 0, #b + 1
143     end
144     n = n + 1
145     t[n] = b
146   end
147   bcsave_tail(fp, output, table.concat(t, ",", 1, n).."\n};\n")
148 end
149
150 local function bcsave_elfobj(ctx, output, s, ffi)
151   ffi.cdef[[
152 typedef struct {
153   uint8_t emagic[4], eclass, eendian, eversion, eosabi, eabiversion, epad[7];
154   uint16_t type, machine;
155   uint32_t version;
156   uint32_t entry, phofs, shofs;
157   uint32_t flags;
158   uint16_t ehsize, phentsize, phnum, shentsize, shnum, shstridx;
159 } ELF32header;
160 typedef struct {
```

```
  uint8_t emagic[4], eclass, eendian, eversion, eosabi, eabiversion, epad[7];
  uint16_t type, machine;
  uint32_t version;
  uint64_t entry, phofs, shofs;
  uint32_t flags;
  uint16_t ehsize, phentsize, phnum, shentsize, shnum, shstridx;
} ELF64header;
typedef struct {
  uint32_t name, type, flags, addr, ofs, size, link, info, align, entsize;
} ELF32sectheader;
typedef struct {
  uint32_t name, type;
  uint64_t flags, addr, ofs, size;
  uint32_t link, info;
  uint64_t align, entsize;
} ELF64sectheader;
typedef struct {
  uint32_t name, value, size;
  uint8_t info, other;
  uint16_t sectidx;
} ELF32symbol;
typedef struct {
  uint32_t name;
  uint8_t info, other;
  uint16_t sectidx;
  uint64_t value, size;
} ELF64symbol;
typedef struct {
  ELF32header hdr;
  ELF32sectheader sect[6];
  ELF32symbol sym[2];
  uint8_t space[4096];
} ELF32obj;
typedef struct {
  ELF64header hdr;
  ELF64sectheader sect[6];
  ELF64symbol sym[2];
  uint8_t space[4096];
} ELF64obj;
]]
  local symname = LJBC_PREFIX..ctx.modname
  local is64, isbe = false, false
  if ctx.arch == "x64" then
    is64 = true
  elseif ctx.arch == "ppc" or ctx.arch == "mips" then
    isbe = true
  end

  -- Handle different host/target endianess.
  local function f32(x) return x end
  local f16, fofs = f32, f32
  if ffi.abi("be") ~= isbe then
    f32 = bit.bswap
    function f16(x) return bit.rshift(bit.bswap(x), 16) end
    if is64 then
      local two32 = ffi.cast("int64_t", 2^32)
      function fofs(x) return bit.bswap(x)*two32 end
    else
      fofs = f32
    end
  end

  -- Create ELF object and fill in header.
  local o = ffi.new(is64 and "ELF64obj" or "ELF32obj")
  local hdr = o.hdr
  if ctx.os == "bsd" or ctx.os == "other" then -- Determine native hdr.eosabi.
    local bf = assert(io.open("/bin/ls", "rb"))
    local bs = bf:read(9)
    bf:close()
    ffi.copy(o, bs, 9)
    check(hdr.emagic[0] == 127, "no support for writing native object files")
  else
    hdr.emagic = "\127ELF"
    hdr.eosabi = ({ freebsd=9, netbsd=2, openbsd=12, solaris=6 })[ctx.os] or 0
  end
  hdr.eclass = is64 and 2 or 1
```

```
  hdr.eendian = isbe and 2 or 1
  hdr.eversion = 1
  hdr.type = f16(1)
  hdr.machine = f16(({ x86=3, x64=62, arm=40, ppc=20, mips=8, mipsel=8 })[ctx.arch])
  if ctx.arch == "mips" or ctx.arch == "mipsel" then
    hdr.flags = 0x50001006
  end
  hdr.version = f32(1)
  hdr.shofs = fofs(ffi.offsetof(o, "sect"))
  hdr.ehsize = f16(ffi.sizeof(hdr))
  hdr.shentsize = f16(ffi.sizeof(o.sect[0]))
  hdr.shnum = f16(6)
  hdr.shstridx = f16(2)

  -- Fill in sections and symbols.
  local sofs, ofs = ffi.offsetof(o, "space"), 1
  for i,name in ipairs{
      ".symtab", ".shstrtab", ".strtab", ".rodata", ".note.GNU-stack",
    } do
    local sect = o.sect[i]
    sect.align = fofs(1)
    sect.name = f32(ofs)
    ffi.copy(o.space+ofs, name)
    ofs = ofs + #name+1
  end
  o.sect[1].type = f32(2) -- .symtab
  o.sect[1].link = f32(3)
  o.sect[1].info = f32(1)
  o.sect[1].align = fofs(8)
  o.sect[1].ofs = fofs(ffi.offsetof(o, "sym"))
  o.sect[1].entsize = fofs(ffi.sizeof(o.sym[0]))
  o.sect[1].size = fofs(ffi.sizeof(o.sym))
  o.sym[1].name = f32(1)
  o.sym[1].sectidx = f16(4)
  o.sym[1].size = fofs(#s)
  o.sym[1].info = 17
  o.sect[2].type = f32(3) -- .shstrtab
  o.sect[2].ofs = fofs(sofs)
  o.sect[2].size = fofs(ofs)
  o.sect[3].type = f32(3) -- .strtab
  o.sect[3].ofs = fofs(sofs + ofs)
  o.sect[3].size = fofs(#symname+1)
  ffi.copy(o.space+ofs+1, symname)
  ofs = ofs + #symname + 2
  o.sect[4].type = f32(1) -- .rodata
  o.sect[4].flags = fofs(2)
  o.sect[4].ofs = fofs(sofs + ofs)
  o.sect[4].size = fofs(#s)
  o.sect[5].type = f32(1) -- .note.GNU-stack
  o.sect[5].ofs = fofs(sofs + ofs + #s)

  -- Write ELF object file.
  local fp = savefile(output, "wb")
  fp:write(ffi.string(o, ffi.sizeof(o)-4096+ofs))
  bcsave_tail(fp, output, s)
end

local function bcsave_peobj(ctx, output, s, ffi)
  ffi.cdef[[
typedef struct {
  uint16_t arch, nsects;
  uint32_t time, symtabofs, nsyms;
  uint16_t opthdrsz, flags;
} PEheader;
typedef struct {
  char name[8];
  uint32_t vsize, vaddr, size, ofs, relocofs, lineofs;
  uint16_t nreloc, nline;
  uint32_t flags;
} PEsection;
typedef struct __attribute((packed)) {
  union {
    char name[8];
    uint32_t nameref[2];
  };
  uint32_t value;
```

```
  int16_t sect;
  uint16_t type;
  uint8_t scl, naux;
} PEsym;
typedef struct __attribute((packed)) {
  uint32_t size;
  uint16_t nreloc, nline;
  uint32_t cksum;
  uint16_t assoc;
  uint8_t comdatsel, unused[3];
} PEsymaux;
typedef struct {
  PEheader hdr;
  PEsection sect[2];
  // Must be an even number of symbol structs.
  PEsym sym0;
  PEsymaux sym0aux;
  PEsym sym1;
  PEsymaux sym1aux;
  PEsym sym2;
  PEsym sym3;
  uint32_t strtabsize;
  uint8_t space[4096];
} PEobj;
]]
  local symname = LJBC_PREFIX..ctx.modname
  local is64 = false
  if ctx.arch == "x86" then
    symname = "_"..symname
  elseif ctx.arch == "x64" then
    is64 = true
  end
  local symexport = "   /EXPORT:"..symname..",DATA "

  -- The file format is always little-endian. Swap if the host is big-endian.
  local function f32(x) return x end
  local f16 = f32
  if ffi.abi("be") then
    f32 = bit.bswap
    function f16(x) return bit.rshift(bit.bswap(x), 16) end
  end

  -- Create PE object and fill in header.
  local o = ffi.new("PEobj")
  local hdr = o.hdr
  hdr.arch = f16(({ x86=0x14c, x64=0x8664, arm=0x1c0, ppc=0x1f2, mips=0x366, mipsel=0x366 })[ctx.arch])
  hdr.nsects = f16(2)
  hdr.symtabofs = f32(ffi.offsetof(o, "sym0"))
  hdr.nsyms = f32(6)

  -- Fill in sections and symbols.
  o.sect[0].name = ".drectve"
  o.sect[0].size = f32(#symexport)
  o.sect[0].flags = f32(0x00100a00)
  o.sym0.sect = f16(1)
  o.sym0.scl = 3
  o.sym0.name = ".drectve"
  o.sym0.naux = 1
  o.sym0aux.size = f32(#symexport)
  o.sect[1].name = ".rdata"
  o.sect[1].size = f32(#s)
  o.sect[1].flags = f32(0x40300040)
  o.sym1.sect = f16(2)
  o.sym1.scl = 3
  o.sym1.name = ".rdata"
  o.sym1.naux = 1
  o.sym1aux.size = f32(#s)
  o.sym2.sect = f16(2)
  o.sym2.scl = 2
  o.sym2.nameref[1] = f32(4)
  o.sym3.sect = f16(-1)
  o.sym3.scl = 2
  o.sym3.value = f32(1)
  o.sym3.name = "@feat.00" -- Mark as SafeSEH compliant.
  ffi.copy(o.space, symname)
  local ofs = #symname + 1
```

```
  o.strtabsize = f32(ofs + 4)
  o.sect[0].ofs = f32(ffi.offsetof(o, "space") + ofs)
  ffi.copy(o.space + ofs, symexport)
  ofs = ofs + #symexport
  o.sect[1].ofs = f32(ffi.offsetof(o, "space") + ofs)

  -- Write PE object file.
  local fp = savefile(output, "wb")
  fp:write(ffi.string(o, ffi.sizeof(o)-4096+ofs))
  bcsave_tail(fp, output, s)
end

local function bcsave_machobj(ctx, output, s, ffi)
  ffi.cdef[[
typedef struct
{
  uint32_t magic, cputype, cpusubtype, filetype, ncmds, sizeofcmds, flags;
} mach_header;
typedef struct
{
  mach_header; uint32_t reserved;
} mach_header_64;
typedef struct {
  uint32_t cmd, cmdsize;
  char segname[16];
  uint32_t vmaddr, vmsize, fileoff, filesize;
  uint32_t maxprot, initprot, nsects, flags;
} mach_segment_command;
typedef struct {
  uint32_t cmd, cmdsize;
  char segname[16];
  uint64_t vmaddr, vmsize, fileoff, filesize;
  uint32_t maxprot, initprot, nsects, flags;
} mach_segment_command_64;
typedef struct {
  char sectname[16], segname[16];
  uint32_t addr, size;
  uint32_t offset, align, reloff, nreloc, flags;
  uint32_t reserved1, reserved2;
} mach_section;
typedef struct {
  char sectname[16], segname[16];
  uint64_t addr, size;
  uint32_t offset, align, reloff, nreloc, flags;
  uint32_t reserved1, reserved2, reserved3;
} mach_section_64;
typedef struct {
  uint32_t cmd, cmdsize, symoff, nsyms, stroff, strsize;
} mach_symtab_command;
typedef struct {
  int32_t strx;
  uint8_t type, sect;
  int16_t desc;
  uint32_t value;
} mach_nlist;
typedef struct {
  uint32_t strx;
  uint8_t type, sect;
  uint16_t desc;
  uint64_t value;
} mach_nlist_64;
typedef struct
{
  uint32_t magic, nfat_arch;
} mach_fat_header;
typedef struct
{
  uint32_t cputype, cpusubtype, offset, size, align;
} mach_fat_arch;
typedef struct {
  struct {
    mach_header hdr;
    mach_segment_command seg;
    mach_section sec;
    mach_symtab_command sym;
  } arch[1];
```

```
  mach_nlist sym_entry;
  uint8_t space[4096];
} mach_obj;
typedef struct {
  struct {
    mach_header_64 hdr;
    mach_segment_command_64 seg;
    mach_section_64 sec;
    mach_symtab_command sym;
  } arch[1];
  mach_nlist_64 sym_entry;
  uint8_t space[4096];
} mach_obj_64;
typedef struct {
  mach_fat_header fat;
  mach_fat_arch fat_arch[4];
  struct {
    mach_header hdr;
    mach_segment_command seg;
    mach_section sec;
    mach_symtab_command sym;
  } arch[4];
  mach_nlist sym_entry;
  uint8_t space[4096];
} mach_fat_obj;
]]
  local symname = '_'..LJBC_PREFIX..ctx.modname
  local isfat, is64, align, mobj = false, false, 4, "mach_obj"
  if ctx.arch == "x64" then
    is64, align, mobj = true, 8, "mach_obj_64"
  elseif ctx.arch == "arm" then
    isfat, mobj = true, "mach_fat_obj"
  else
    check(ctx.arch == "x86", "unsupported architecture for OSX")
  end
  local function aligned(v, a) return bit.band(v+a-1, -a) end
  local be32 = bit.bswap -- Mach-O FAT is BE, supported archs are LE.

  -- Create Mach-O object and fill in header.
  local o = ffi.new(mobj)
  local mach_size = aligned(ffi.offsetof(o, "space")+#symname+2, align)
  local cputype = ({ x86={7}, x64={0x01000007}, arm={7,12,12,12} })[ctx.arch]
  local cpusubtype = ({ x86={3}, x64={3}, arm={3,6,9,11} })[ctx.arch]
  if isfat then
    o.fat.magic = be32(0xcafebabe)
    o.fat.nfat_arch = be32(#cpusubtype)
  end

  -- Fill in sections and symbols.
  for i=0,#cpusubtype-1 do
    local ofs = 0
    if isfat then
      local a = o.fat_arch[i]
      a.cputype = be32(cputype[i+1])
      a.cpusubtype = be32(cpusubtype[i+1])
      -- Subsequent slices overlap each other to share data.
      ofs = ffi.offsetof(o, "arch") + i*ffi.sizeof(o.arch[0])
      a.offset = be32(ofs)
      a.size = be32(mach_size-ofs+#s)
    end
    local a = o.arch[i]
    a.hdr.magic = is64 and 0xfeedfacf or 0xfeedface
    a.hdr.cputype = cputype[i+1]
    a.hdr.cpusubtype = cpusubtype[i+1]
    a.hdr.filetype = 1
    a.hdr.ncmds = 2
    a.hdr.sizeofcmds = ffi.sizeof(a.seg)+ffi.sizeof(a.sec)+ffi.sizeof(a.sym)
    a.seg.cmd = is64 and 0x19 or 0x1
    a.seg.cmdsize = ffi.sizeof(a.seg)+ffi.sizeof(a.sec)
    a.seg.vmsize = #s
    a.seg.fileoff = mach_size-ofs
    a.seg.filesize = #s
    a.seg.maxprot = 1
    a.seg.initprot = 1
    a.seg.nsects = 1
    ffi.copy(a.sec.sectname, "__data")
```

```
    ffi.copy(a.sec.segname, "__DATA")
    a.sec.size = #s
    a.sec.offset = mach_size-ofs
    a.sym.cmd = 2
    a.sym.cmdsize = ffi.sizeof(a.sym)
    a.sym.symoff = ffi.offsetof(o, "sym_entry")-ofs
    a.sym.nsyms = 1
    a.sym.stroff = ffi.offsetof(o, "sym_entry")+ffi.sizeof(o.sym_entry)-ofs
    a.sym.strsize = aligned(#symname+2, align)
  end
  o.sym_entry.type = 0xf
  o.sym_entry.sect = 1
  o.sym_entry.strx = 1
  ffi.copy(o.space+1, symname)

  -- Write Macho-O object file.
  local fp = savefile(output, "wb")
  fp:write(ffi.string(o, mach_size))
  bcsave_tail(fp, output, s)
end

local function bcsave_obj(ctx, output, s)
  local ok, ffi = pcall(require, "ffi")
  check(ok, "FFI library required to write this file type")
  if ctx.os == "windows" then
    return bcsave_peobj(ctx, output, s, ffi)
  elseif ctx.os == "osx" then
    return bcsave_machobj(ctx, output, s, ffi)
  else
    return bcsave_elfobj(ctx, output, s, ffi)
  end
end

------------------------------------------------------------------------------

local function bclist(input, output)
  local f = readfile(input)
  require("jit.bc").dump(f, savefile(output, "w"), true)
end

local function bcsave(ctx, input, output)
  local f = readfile(input)
  local s = string.dump(f, ctx.strip)
  local t = ctx.type
  if not t then
    t = detecttype(output)
    ctx.type = t
  end
  if t == "raw" then
    bcsave_raw(output, s)
  else
    if not ctx.modname then ctx.modname = detectmodname(input) end
    if t == "obj" then
      bcsave_obj(ctx, output, s)
    else
      bcsave_c(ctx, output, s)
    end
  end
end

local function docmd(...)
  local arg = {...}
  local n = 1
  local list = false
  local ctx = {
    strip = true, arch = jit.arch, os = string.lower(jit.os),
    type = false, modname = false,
  }
  while n <= #arg do
    local a = arg[n]
    if type(a) == "string" and string.sub(a, 1, 1) == "-" and a ~= "-" then
      table.remove(arg, n)
      if a == "--" then break end
      for m=2,#a do
        local opt = string.sub(a, m, m)
        if opt == "l" then
```

```
          list = true
        elseif opt == "s" then
          ctx.strip = true
        elseif opt == "g" then
          ctx.strip = false
        else
          if arg[n] == nil or m ~= #a then usage() end
          if opt == "e" then
            if n ~= 1 then usage() end
            arg[1] = check(loadstring(arg[1]))
          elseif opt == "n" then
            ctx.modname = checkmodname(table.remove(arg, n))
          elseif opt == "t" then
            ctx.type = checkarg(table.remove(arg, n), map_type, "file type")
          elseif opt == "a" then
            ctx.arch = checkarg(table.remove(arg, n), map_arch, "architecture")
          elseif opt == "o" then
            ctx.os = checkarg(table.remove(arg, n), map_os, "OS name")
          else
            usage()
          end
        end
      end
    else
      n = n + 1
    end
  end
  if list then
    if #arg == 0 or #arg > 2 then usage() end
    bclist(arg[1], arg[2] or "-")
  else
    if #arg ~= 2 then usage() end
    bcsave(ctx, arg[1], arg[2])
  end
end

------------------------------------------------------------------------------

-- Public module functions.
return {
  start = docmd -- Process -b command line option.
}

```

One Level Up Top Level

src/jit/ - luajit-2.0-src

- bc.lua
- bcsave.lua
- dis_arm.lua
- dis_mips.lua
- dis_mipsel.lua
- dis_ppc.lua
- dis_x64.lua
- dis_x86.lua
- dump.lua
- p.lua
- v.lua
- vmdef.lua
- zone.lua

src/jit/bc.lua - luajit-2.0-src

Functions defined

- bcdump
- bcline
- bclistoff
- bcliston
- bctargets
- ctlsub
- h_list

Source code

```
1  ----------------------------------------------------------------------------
2  -- LuaJIT bytecode listing module.
3  --
4  -- Copyright (C) 2005-2015 Mike Pall. All rights reserved.
5  -- Released under the MIT license. See Copyright Notice in luajit.h
6  ----------------------------------------------------------------------------
7  --
8  -- This module lists the bytecode of a Lua function. If it's loaded by -jbc
9  -- it hooks into the parser and lists all functions of a chunk as they
10 -- are parsed.
11 --
12 -- Example usage:
13 --
14 --   luajit -jbc -e 'local x=0; for i=1,1e6 do x=x+i end; print(x)'
15 --   luajit -jbc=- foo.lua
16 --   luajit -jbc=foo.list foo.lua
17 --
18 -- Default output is to stderr. To redirect the output to a file, pass a
19 -- filename as an argument (use '-' for stdout) or set the environment
20 -- variable LUAJIT_LISTFILE. The file is overwritten every time the module
21 -- is started.
22 --
23 -- This module can also be used programmatically:
24 --
25 --   local bc = require("jit.bc")
26 --
27 --   local function foo() print("hello") end
28 --
29 --   bc.dump(foo)           --> -- BYTECODE -- [...]
30 --   print(bc.line(foo, 2)) --> 0002    KSTR     1   1      ; "hello"
31 --
32 --   local out = {
33 --     -- Do something with each line:
34 --     write = function(t, ...) io.write(...) end,
35 --     close = function(t) end,
36 --     flush = function(t) end,
37 --   }
38 --   bc.dump(foo, out)
39 --
40 ----------------------------------------------------------------------------
41
42 -- Cache some library functions and objects.
43 local jit = require("jit")
44 assert(jit.version_num == 20100, "LuaJIT core/library version mismatch")
45 local jutil = require("jit.util")
46 local vmdef = require("jit.vmdef")
47 local bit = require("bit")
48 local sub, gsub, format = string.sub, string.gsub, string.format
49 local byte, band, shr = string.byte, bit.band, bit.rshift
```

```lua
local funcinfo, funcbc, funck = jutil.funcinfo, jutil.funcbc, jutil.funck
local funcuvname = jutil.funcuvname
local bcnames = vmdef.bcnames
local stdout, stderr = io.stdout, io.stderr

------------------------------------------------------------------------------

local function ctlsub(c)
  if c == "\n" then return "\\n"
  elseif c == "\r" then return "\\r"
  elseif c == "\t" then return "\\t"
  else return format("\\%03d", byte(c))
  end
end

-- Return one bytecode line.
local function bcline(func, pc, prefix)
  local ins, m = funcbc(func, pc)
  if not ins then return end
  local ma, mb, mc = band(m, 7), band(m, 15*8), band(m, 15*128)
  local a = band(shr(ins, 8), 0xff)
  local oidx = 6*band(ins, 0xff)
  local op = sub(bcnames, oidx+1, oidx+6)
  local s = format("%04d %s %-6s %3s ",
    pc, prefix or "  ", op, ma == 0 and "" or a)
  local d = shr(ins, 16)
  if mc == 13*128 then -- BCMjump
    return format("%s=> %04d\n", s, pc+d-0x7fff)
  end
  if mb ~= 0 then
    d = band(d, 0xff)
  elseif mc == 0 then
    return s.."\n"
  end
  local kc
  if mc == 10*128 then -- BCMstr
    kc = funck(func, -d-1)
    kc = format(#kc > 40 and '"%.40s"~' or '"%s"', gsub(kc, "%c", ctlsub))
  elseif mc == 9*128 then -- BCMnum
    kc = funck(func, d)
    if op == "TSETM " then kc = kc - 2^52 end
  elseif mc == 12*128 then -- BCMfunc
    local fi = funcinfo(funck(func, -d-1))
    if fi.ffid then
      kc = vmdef.ffnames[fi.ffid]
    else
      kc = fi.loc
    end
  elseif mc == 5*128 then -- BCMuv
    kc = funcuvname(func, d)
   end
   if ma == 5 then -- BCMuv
     local ka = funcuvname(func, a)
     if kc then kc = ka.." ; "..kc else kc = ka end
   end
   if mb ~= 0 then
     local b = shr(ins, 24)
     if kc then return format("%s%3d %3d  ; %s\n", s, b, d, kc) end
     return format("%s%3d %3d\n", s, b, d)
   end
   if kc then return format("%s%3d      ; %s\n", s, d, kc) end
   if mc == 7*128 and d > 32767 then d = d - 65536 end -- BCMlits
   return format("%s%3d\n", s, d)
 end

 -- Collect branch targets of a function.
 local function bctargets(func)
   local target = {}
   for pc=1,1000000000 do
     local ins, m = funcbc(func, pc)
     if not ins then break end
     if band(m, 15*128) == 13*128 then target[pc+shr(ins, 16)-0x7fff] = true end
   end
   return target
 end

```

```
126 -- Dump bytecode instructions of a function.
127 local function bcdump(func, out, all)
128   if not out then out = stdout end
129   local fi = funcinfo(func)
130   if all and fi.children then
131     for n=-1,-1000000000,-1 do
132       local k = funck(func, n)
133       if not k then break end
134       if type(k) == "proto" then bcdump(k, out, true) end
135     end
136   end
137   out:write(format("-- BYTECODE -- %s-%d\n", fi.loc, fi.lastlinedefined))
138   local target = bctargets(func)
139   for pc=1,1000000000 do
140     local s = bcline(func, pc, target[pc] and "=>")
141     if not s then break end
142     out:write(s)
143   end
144   out:write("\n")
145   out:flush()
146 end
147
148 ------------------------------------------------------------------------------
149
150 -- Active flag and output file handle.
151 local active, out
152
153 -- List handler.
154 local function h_list(func)
155   return bcdump(func, out)
156 end
157
158 -- Detach list handler.
159 local function bclistoff()
160   if active then
161     active = false
162     jit.attach(h_list)
163     if out and out ~= stdout and out ~= stderr then out:close() end
164     out = nil
165   end
166 end
167
168 -- Open the output file and attach list handler.
169 local function bcliston(outfile)
170   if active then bclistoff() end
171   if not outfile then outfile = os.getenv("LUAJIT_LISTFILE") end
172   if outfile then
173     out = outfile == "-" and stdout or assert(io.open(outfile, "w"))
174   else
175     out = stderr
176   end
177   jit.attach(h_list, "bc")
178   active = true
179 end
180
181 -- Public module functions.
182 return {
183   line = bcline,
184   dump = bcdump,
185   targets = bctargets,
186   on = bcliston,
187   off = bclistoff,
188   start = bcliston -- For -j command line option.
189 }
190
```

One Level Up Top Level

src/jit/zone.lua - luajit-2.0-src

Functions defined

- __call
- flush
- get

Source code

```
1  ----------------------------------------------------------------------------
2  -- LuaJIT profiler zones.
3  --
4  -- Copyright (C) 2005-2015 Mike Pall. All rights reserved.
5  -- Released under the MIT license. See Copyright Notice in luajit.h
6  ----------------------------------------------------------------------------
7  --
8  -- This module implements a simple hierarchical zone model.
9  --
10 -- Example usage:
11 --
12 --   local zone = require("jit.zone")
13 --   zone("AI")
14 --   ...
15 --     zone("A*")
16 --     ...
17 --     print(zone:get()) --> "A*"
18 --     ...
19 --     zone()
20 --   ...
21 --   print(zone:get()) --> "AI"
22 --   ...
23 --   zone()
24 --
25 ----------------------------------------------------------------------------
26
27 local remove = table.remove
28
29 return setmetatable({
30   flush = function(t)
31     for i=#t,1,-1 do t[i] = nil end
32   end,
33   get = function(t)
34     return t[#t]
35   end
36 }, {
37   __call = function(t, zone)
38     if zone then
39       t[#t+1] = zone
40     else
41       return (assert(remove(t), "empty zone stack"))
42     end
43   end
44 })
45
```

src/jit/p.lua - luajit-2.0-src

Functions defined

- key_stack
- mode
- mode
- mode
- prof_annotate
- prof_cb
- prof_finish
- prof_start
- prof_top
- start

Source code

```
1  ----------------------------------------------------------------------------
2  -- LuaJIT profiler.
3  --
4  -- Copyright (C) 2005-2015 Mike Pall. All rights reserved.
5  -- Released under the MIT license. See Copyright Notice in luajit.h
6  ----------------------------------------------------------------------------
7  --
8  -- This module is a simple command line interface to the built-in
9  -- low-overhead profiler of LuaJIT.
10 --
11 -- The lower-level API of the profiler is accessible via the "jit.profile"
12 -- module or the luaJIT_profile_* C API.
13 --
14 -- Example usage:
15 --
16 --   luajit -jp myapp.lua
17 --   luajit -jp=s myapp.lua
18 --   luajit -jp=-s myapp.lua
19 --   luajit -jp=vl myapp.lua
20 --   luajit -jp=G,profile.txt myapp.lua
21 --
22 -- The following dump features are available:
23 --
24 --   f  Stack dump: function name, otherwise module:line. Default mode.
25 --   F  Stack dump: ditto, but always prepend module.
26 --   l  Stack dump: module:line.
27 --   <number> stack dump depth (callee < caller). Default: 1.
28 --   -<number> Inverse stack dump depth (caller > callee).
29 --   s  Split stack dump after first stack level. Implies abs(depth) >= 2.
30 --   p  Show full path for module names.
31 --   v  Show VM states. Can be combined with stack dumps, e.g. vf or fv.
32 --   z  Show zones. Can be combined with stack dumps, e.g. zf or fz.
33 --   r  Show raw sample counts. Default: show percentages.
34 --   a  Annotate excerpts from source code files.
35 --   A  Annotate complete source code files.
36 --   G  Produce raw output suitable for graphical tools (e.g. flame graphs).
37 --   m<number> Minimum sample percentage to be shown. Default: 3.
38 --   i<number> Sampling interval in milliseconds. Default: 10.
39 --
40 ----------------------------------------------------------------------------
41
42 -- Cache some library functions and objects.
```

```lua
local jit = require("jit")
assert(jit.version_num == 20100, "LuaJIT core/library version mismatch")
local profile = require("jit.profile")
local vmdef = require("jit.vmdef")
local math = math
local pairs, ipairs, tonumber, floor = pairs, ipairs, tonumber, math.floor
local sort, format = table.sort, string.format
local stdout = io.stdout
local zone -- Load jit.zone module on demand.

-- Output file handle.
local out

------------------------------------------------------------------------------

local prof_ud
local prof_states, prof_split, prof_min, prof_raw, prof_fmt, prof_depth
local prof_ann, prof_count1, prof_count2, prof_samples

local map_vmmode = {
  N = "Compiled",
  I = "Interpreted",
  C = "C code",
  G = "Garbage Collector",
  J = "JIT Compiler",
}

-- Profiler callback.
local function prof_cb(th, samples, vmmode)
  prof_samples = prof_samples + samples
  local key_stack, key_stack2, key_state
  -- Collect keys for sample.
  if prof_states then
    if prof_states == "v" then
      key_state = map_vmmode[vmmode] or vmmode
    else
      key_state = zone:get() or "(none)"
    end
  end
  if prof_fmt then
    key_stack = profile.dumpstack(th, prof_fmt, prof_depth)
    key_stack = key_stack:gsub("%[builtin#(%d+)%]", function(x)
      return vmdef.ffnames[tonumber(x)]
    end)
    if prof_split == 2 then
      local k1, k2 = key_stack:match("(.-) [<>] (.*)")
      if k2 then key_stack, key_stack2 = k1, k2 end
    elseif prof_split == 3 then
      key_stack2 = profile.dumpstack(th, "l", 1)
    end
  end
  -- Order keys.
  local k1, k2
  if prof_split == 1 then
    if key_state then
      k1 = key_state
      if key_stack then k2 = key_stack end
    end
  elseif key_stack then
    k1 = key_stack
    if key_stack2 then k2 = key_stack2 elseif key_state then k2 = key_state end
  end
  -- Coalesce samples in one or two levels.
  if k1 then
    local t1 = prof_count1
    t1[k1] = (t1[k1] or 0) + samples
    if k2 then
      local t2 = prof_count2
      local t3 = t2[k1]
      if not t3 then t3 = {}; t2[k1] = t3 end
      t3[k2] = (t3[k2] or 0) + samples
    end
  end
end

------------------------------------------------------------------------------
```

```lua

-- Show top N list.
local function prof_top(count1, count2, samples, indent)
  local t, n = {}, 0
  for k, v in pairs(count1) do
    n = n + 1
    t[n] = k
  end
  sort(t, function(a, b) return count1[a] > count1[b] end)
  for i=1,n do
    local k = t[i]
    local v = count1[k]
    local pct = floor(v*100/samples + 0.5)
    if pct < prof_min then break end
    if not prof_raw then
      out:write(format("%s%2d%%  %s\n", indent, pct, k))
    elseif prof_raw == "r" then
      out:write(format("%s%5d  %s\n", indent, v, k))
    else
      out:write(format("%s %d\n", k, v))
    end
    if count2 then
      local r = count2[k]
      if r then
        prof_top(r, nil, v, (prof_split == 3 or prof_split == 1) and "  -- " or
                            (prof_depth < 0 and "  -> " or "  <- "))
      end
    end
  end
end

-- Annotate source code
local function prof_annotate(count1, samples)
  local files = {}
  local ms = 0
  for k, v in pairs(count1) do
    local pct = floor(v*100/samples + 0.5)
    ms = math.max(ms, v)
    if pct >= prof_min then
      local file, line = k:match("^(.*):(%d+)$")
      local fl = files[file]
      if not fl then fl = {}; files[file] = fl; files[#files+1] = file end
      line = tonumber(line)
      fl[line] = prof_raw and v or pct
    end
  end
  sort(files)
  local fmtv, fmtn = " %3d%% | %s\n", "      | %s\n"
  if prof_raw then
    local n = math.max(5, math.ceil(math.log10(ms)))
    fmtv = "%"..n.."d | %s\n"
    fmtn = (" "):rep(n).." | %s\n"
  end
  local ann = prof_ann
  for _, file in ipairs(files) do
    local f0 = file:byte()
    if f0 == 40 or f0 == 91 then
      out:write(format("\n====== %s ======\n[Cannot annotate non-file]\n", file))
      break
    end
    local fp, err = io.open(file)
    if not fp then
      out:write(format("====== ERROR: %s: %s\n", file, err))
      break
    end
    out:write(format("\n====== %s ======\n", file))
    local fl = files[file]
    local n, show = 1, false
    if ann ~= 0 then
      for i=1,ann do
        if fl[i] then show = true; out:write("@@ 1 @@\n"); break end
      end
    end
    for line in fp:lines() do
      if line:byte() == 27 then
        out:write("[Cannot annotate bytecode file]\n")
```

```
        break
      end
      local v = fl[n]
      if ann ~= 0 then
        local v2 = fl[n+ann]
        if show then
          if v2 then show = n+ann elseif v then show = n
          elseif show+ann < n then show = false end
        elseif v2 then
          show = n+ann
          out:write(format("@@ %d @@\n", n))
        end
        if not show then goto next end
      end
      if v then
        out:write(format(fmtv, v, line))
      else
        out:write(format(fmtn, line))
      end
    ::next::
      n = n + 1
    end
    fp:close()
  end
end

------------------------------------------------------------------------------

-- Finish profiling and dump result.
local function prof_finish()
  if prof_ud then
    profile.stop()
    local samples = prof_samples
    if samples == 0 then
      if prof_raw ~= true then out:write("[No samples collected]\n") end
      return
    end
    if prof_ann then
      prof_annotate(prof_count1, samples)
    else
      prof_top(prof_count1, prof_count2, samples, "")
    end
    prof_count1 = nil
    prof_count2 = nil
    prof_ud = nil
  end
end

-- Start profiling.
local function prof_start(mode)
  local interval = ""
  mode = mode:gsub("i%d*", function(s) interval = s; return "" end)
  prof_min = 3
  mode = mode:gsub("m(%d+)", function(s) prof_min = tonumber(s); return "" end)
  prof_depth = 1
  mode = mode:gsub("%-?%d+", function(s) prof_depth = tonumber(s); return "" end)
  local m = {}
  for c in mode:gmatch(".") do m[c] = c end
  prof_states = m.z or m.v
  if prof_states == "z" then zone = require("jit.zone") end
  local scope = m.l or m.f or m.F or (prof_states and "" or "f")
  local flags = (m.p or "")
  prof_raw = m.r
  if m.s then
    prof_split = 2
    if prof_depth == -1 or m["-"] then prof_depth = -2
    elseif prof_depth == 1 then prof_depth = 2 end
  elseif mode:find("[fF].*l") then
    scope = "l"
    prof_split = 3
  else
    prof_split = (scope == "" or mode:find("[zv].*[lfF]")) and 1 or 0
  end
  prof_ann = m.A and 0 or (m.a and 3)
  if prof_ann then
    scope = "l"
```

```
    prof_fmt = "pl"
    prof_split = 0
    prof_depth = 1
  elseif m.G and scope ~= "" then
    prof_fmt = flags..scope.."Z;"
    prof_depth = -100
    prof_raw = true
    prof_min = 0
  elseif scope == "" then
    prof_fmt = false
  else
    local sc = prof_split == 3 and m.f or m.F or scope
    prof_fmt = flags..sc..(prof_depth >= 0 and "Z < " or "Z > ")
  end
  prof_count1 = {}
  prof_count2 = {}
  prof_samples = 0
  profile.start(scope:lower()..interval, prof_cb)
  prof_ud = newproxy(true)
  getmetatable(prof_ud).__gc = prof_finish
end

------------------------------------------------------------------------------

local function start(mode, outfile)
  if not outfile then outfile = os.getenv("LUAJIT_PROFILEFILE") end
  if outfile then
    out = outfile == "-" and stdout or assert(io.open(outfile, "w"))
  else
    out = stdout
  end
  prof_start(mode or "f")
end

-- Public module functions.
return {
  start = start, -- For -j command line option.
  stop = prof_finish
}
```

One Level Up Top Level

src/jit/dis_x86.lua - luajit-2.0-src

Functions defined

Source code

```
1  ------------------------------------------------------------------------------
2  -- LuaJIT x86/x64 disassembler module.
3  --
4  -- Copyright (C) 2005-2015 Mike Pall. All rights reserved.
```

```
-- Released under the MIT license. See Copyright Notice in luajit.h
----------------------------------------------------------------------------
-- This is a helper module used by the LuaJIT machine code dumper module.
--
-- Sending small code snippets to an external disassembler and mixing the
-- output with our own stuff was too fragile. So I had to bite the bullet
-- and write yet another x86 disassembler. Oh well ...
--
-- The output format is very similar to what ndisasm generates. But it has
-- been developed independently by looking at the opcode tables from the
-- Intel and AMD manuals. The supported instruction set is quite extensive
-- and reflects what a current generation Intel or AMD CPU implements in
-- 32 bit and 64 bit mode. Yes, this includes MMX, SSE, SSE2, SSE3, SSSE3,
-- SSE4.1, SSE4.2, SSE4a and even privileged and hypervisor (VMX/SVM)
-- instructions.
--
-- Notes:
-- * The (useless) a16 prefix, 3DNow and pre-586 opcodes are unsupported.
-- * No attempt at optimization has been made -- it's fast enough for my needs.
-- * The public API may change when more architectures are added.
----------------------------------------------------------------------------

local type = type
local sub, byte, format = string.sub, string.byte, string.format
local match, gmatch, gsub = string.match, string.gmatch, string.gsub
local lower, rep = string.lower, string.rep
local bit = require("bit")
local tohex = bit.tohex

-- Map for 1st opcode byte in 32 bit mode. Ugly? Well ... read on.
local map_opc1_32 = {
--0x
[0]="addBmr","addVmr","addBrm","addVrm","addBai","addVai","push es","pop es",
"orBmr","orVmr","orBrm","orVrm","orBai","orVai","push cs","opc2*",
--1x
"adcBmr","adcVmr","adcBrm","adcVrm","adcBai","adcVai","push ss","pop ss",
"sbbBmr","sbbVmr","sbbBrm","sbbVrm","sbbBai","sbbVai","push ds","pop ds",
--2x
"andBmr","andVmr","andBrm","andVrm","andBai","andVai","es:seg","daa",
"subBmr","subVmr","subBrm","subVrm","subBai","subVai","cs:seg","das",
--3x
"xorBmr","xorVmr","xorBrm","xorVrm","xorBai","xorVai","ss:seg","aaa",
"cmpBmr","cmpVmr","cmpBrm","cmpVrm","cmpBai","cmpVai","ds:seg","aas",
--4x
"incVR","incVR","incVR","incVR","incVR","incVR","incVR","incVR",
"decVR","decVR","decVR","decVR","decVR","decVR","decVR","decVR",
--5x
"pushUR","pushUR","pushUR","pushUR","pushUR","pushUR","pushUR","pushUR",
"popUR","popUR","popUR","popUR","popUR","popUR","popUR","popUR",
--6x
"sz*pushaw,pusha","sz*popaw,popa","boundVrm","arplWmr",
"fs:seg","gs:seg","o16:","a16",
"pushUi","imulVrmi","pushBs","imulVrms",
"insb","insVS","outsb","outsVS",
--7x
"joBj","jnoBj","jbBj","jnbBj","jzBj","jnzBj","jbeBj","jaBj",
"jsBj","jnsBj","jpeBj","jpoBj","jlBj","jgeBj","jleBj","jgBj",
--8x
"arith!Bmi","arith!Vmi","arith!Bmi","arith!Vms",
"testBmr","testVmr","xchgBrm","xchgVrm",
"movBmr","movVmr","movBrm","movVrm",
"movVmg","leaVrm","movWgm","popUm",
--9x
"nop*xchgVaR|pause|xchgWaR|repne nop","xchgVaR","xchgVaR","xchgVaR",
"xchgVaR","xchgVaR","xchgVaR","xchgVaR",
"sz*cbw,cwde,cdqe","sz*cwd,cdq,cqo","call farViw","wait",
"sz*pushfw,pushf","sz*popfw,popf","sahf","lahf",
--Ax
"movBao","movVao","movBoa","movVoa",
"movsb","movsVS","cmpsb","cmpsVS",
"testBai","testVai","stosb","stosVS",
"lodsb","lodsVS","scasb","scasVS",
--Bx
"movBRi","movBRi","movBRi","movBRi","movBRi","movBRi","movBRi","movBRi",
"movVRI","movVRI","movVRI","movVRI","movVRI","movVRI","movVRI","movVRI",
--Cx
```

```
"shift!Bmu","shift!Vmu","retBw","ret","$lesVrm","$ldsVrm","movBmi","movVmi",
"enterBwu","leave","retfBw","retf","int3","intBu","into","iretVS",
--Dx
"shift!Bm1","shift!Vm1","shift!Bmc","shift!Vmc","aamBu","aadBu","salc","xlatb",
"fp*0","fp*1","fp*2","fp*3","fp*4","fp*5","fp*6","fp*7",
--Ex
"loopneBj","loopeBj","loopBj","sz*jcxzBj,jecxzBj,jrcxzBj",
"inBau","inVau","outBua","outVua",
"callVj","jmpVj","jmp farViw","jmpBj","inBad","inVad","outBda","outVda",
--Fx
"lock:","int1","repne:rep","rep:","hlt","cmc","testb!Bm","testv!Vm",
"clc","stc","cli","sti","cld","std","incb!Bm","incd!Vm",
}
assert(#map_opc1_32 == 255)

-- Map for 1st opcode byte in 64 bit mode (overrides only).
local map_opc1_64 = setmetatable({
  [0x06]=false, [0x07]=false, [0x0e]=false,
  [0x16]=false, [0x17]=false, [0x1e]=false, [0x1f]=false,
   [0x27]=false, [0x2f]=false, [0x37]=false, [0x3f]=false,
   [0x60]=false, [0x61]=false, [0x62]=false, [0x63]="movsxdVrDmt", [0x67]="a32:",
   [0x40]="rex*",   [0x41]="rex*b",   [0x42]="rex*x",   [0x43]="rex*xb",
   [0x44]="rex*r",  [0x45]="rex*rb",  [0x46]="rex*rx",  [0x47]="rex*rxb",
   [0x48]="rex*w",  [0x49]="rex*wb",  [0x4a]="rex*wx",  [0x4b]="rex*wxb",
   [0x4c]="rex*wr", [0x4d]="rex*wrb", [0x4e]="rex*wrx", [0x4f]="rex*wrxb",
   [0x82]=false, [0x9a]=false, [0xc4]=false, [0xc5]=false, [0xce]=false,
   [0xd4]=false, [0xd5]=false, [0xd6]=false, [0xea]=false,
}, { __index = map_opc1_32 })

-- Map for 2nd opcode byte (0F xx). True CISC hell. Hey, I told you.
-- Prefix dependent MMX/SSE opcodes: (none)|rep|o16|repne, -|F3|66|F2
local map_opc2 = {
--0x
[0]="sldt!Dmp","sgdt!Ump","larVrm","lslVrm",nil,"syscall","clts","sysret",
"invd","wbinvd",nil,"ud1",nil,"$prefetch!Bm","femms","3dnowMrmu",
--1x
"movupsXrm|movssXrm|movupdXrm|movsdXrm",
"movupsXmr|movssXmr|movupdXmr|movsdXmr",
"movhlpsXrm$movlpsXrm|movsldupXrm|movlpdXrm|movddupXrm",
"movlpsXmr||movlpdXmr",
"unpcklpsXrm||unpcklpdXrm",
"unpckhpsXrm||unpckhpdXrm",
"movlhpsXrm$movhpsXrm|movshdupXrm|movhpdXrm",
"movhpsXmr||movhpdXmr",
"$prefetcht!Bm","hintnopVm","hintnopVm","hintnopVm",
"hintnopVm","hintnopVm","hintnopVm","hintnopVm",
--2x
"movUmx$","movUmy$","movUxm$","movUym$","movUmz$",nil,"movUzm$",nil,
"movapsXrm||movapdXrm",
"movapsXmr||movapdXmr",
"cvtpi2psXrMm|cvtsi2ssXrVmt|cvtpi2pdXrMm|cvtsi2sdXrVmt",
"movntpsXmr|movntssXmr|movntpdXmr|movntsdXmr",
"cvttps2piMrXm|cvttss2siVrXm|cvttpd2piMrXm|cvttsd2siVrXm",
"cvtps2piMrXm|cvtss2siVrXm|cvtpd2piMrXm|cvtsd2siVrXm",
"ucomissXrm||ucomisdXrm",
"comissXrm||comisdXrm",
--3x
"wrmsr","rdtsc","rdmsr","rdpmc","sysenter","sysexit",nil,"getsec",
"opc3*38",nil,"opc3*3a",nil,nil,nil,nil,nil,
--4x
"cmovoVrm","cmovnoVrm","cmovbVrm","cmovnbVrm",
"cmovzVrm","cmovnzVrm","cmovbeVrm","cmovaVrm",
"cmovsVrm","cmovnsVrm","cmovpeVrm","cmovpoVrm",
"cmovlVrm","cmovgeVrm","cmovleVrm","cmovgVrm",
--5x
"movmskpsVrXm$||movmskpdVrXm$","sqrtpsXrm|sqrtssXrm|sqrtpdXrm|sqrtsdXrm",
"rsqrtpsXrm|rsqrtssXrm","rcppsXrm|rcpssXrm",
"andpsXrm||andpdXrm","andnpsXrm||andnpdXrm",
"orpsXrm||orpdXrm","xorpsXrm||xorpdXrm",
"addpsXrm|addssXrm|addpdXrm|addsdXrm","mulpsXrm|mulssXrm|mulpdXrm|mulsdXrm",
"cvtps2pdXrm|cvtss2sdXrm|cvtpd2psXrm|cvtsd2ssXrm",
"cvtdq2psXrm|cvttps2dqXrm|cvtps2dqXrm",
"subpsXrm|subssXrm|subpdXrm|subsdXrm","minpsXrm|minssXrm|minpdXrm|minsdXrm",
"divpsXrm|divssXrm|divpdXrm|divsdXrm","maxpsXrm|maxssXrm|maxpdXrm|maxsdXrm",
--6x
"punpcklbwPrm","punpcklwdPrm","punpckldqPrm","packsswbPrm",
```

```
"pcmpgtbPrm","pcmpgtwPrm","pcmpgtdPrm","packuswbPrm",
"punpckhbwPrm","punpckhwdPrm","punpckhdqPrm","packssdwPrm",
"||punpcklqdqXrm","||punpckhqdqXrm",
"movPrVSm","movqMrm|movdquXrm|movdqaXrm",
--7x
"pshufwMrmu|pshufhwXrmu|pshufdXrmu|pshuflwXrmu","pshiftw!Pmu",
"pshiftd!Pmu","pshiftq!Mmu||pshiftdq!Xmu",
"pcmpeqbPrm","pcmpeqwPrm","pcmpeqdPrm","emms|",
"vmreadUmr||extrqXmuu$|insertqXrmuu$","vmwriteUrm||extrqXrm$|insertqXrm$",
nil,nil,
"||haddpdXrm|haddpsXrm","||hsubpdXrm|hsubpsXrm",
"movVSmMr|movqXrm|movVSmXr","movqMmr|movdquXmr|movdqaXmr",
--8x
"joVj","jnoVj","jbVj","jnbVj","jzVj","jnzVj","jbeVj","jaVj",
"jsVj","jnsVj","jpeVj","jpoVj","jlVj","jgeVj","jleVj","jgVj",
--9x
"setoBm","setnoBm","setbBm","setnbBm","setzBm","setnzBm","setbeBm","setaBm",
"setsBm","setnsBm","setpeBm","setpoBm","setlBm","setgeBm","setleBm","setgBm",
--Ax
"push fs","pop fs","cpuid","btVmr","shldVmru","shldVmrc",nil,nil,
"push gs","pop gs","rsm","btsVmr","shrdVmru","shrdVmrc","fxsave!Dmp","imulVrm",
--Bx
"cmpxchgBmr","cmpxchgVmr","$lssVrm","btrVmr",
"$lfsVrm","$lgsVrm","movzxVrBmt","movzxVrWmt",
"|popcntVrm","ud2Dp","bt!Vmu","btcVmr",
"bsfVrm","bsrVrm|lzcntVrm|bsrWrm","movsxVrBmt","movsxVrWmt",
--Cx
"xaddBmr","xaddVmr",
"cmppsXrmu|cmpssXrmu|cmppdXrmu|cmpsdXrmu","$movntiVmr|",
"pinsrwPrWmu","pextrwDrPmu",
"shufpsXrmu||shufpdXrmu","$cmpxchg!Qmp",
"bswapVR","bswapVR","bswapVR","bswapVR","bswapVR","bswapVR","bswapVR","bswapVR",
--Dx
"||addsubpdXrm|addsubpsXrm","psrlwPrm","psrldPrm","psrlqPrm",
"paddqPrm","pmullwPrm",
"|movq2dqXrMm|movqXmr|movdq2qMrXm$","pmovmskbVrMm||pmovmskbVrXm",
"psubusbPrm","psubuswPrm","pminubPrm","pandPrm",
"paddusbPrm","padduswPrm","pmaxubPrm","pandnPrm",
--Ex
"pavgbPrm","psrawPrm","psradPrm","pavgwPrm",
"pmulhuwPrm","pmulhwPrm",
"|cvtdq2pdXrm|cvttpd2dqXrm|cvtpd2dqXrm","$movntqMmr||$movntdqXmr",
"psubsbPrm","psubswPrm","pminswPrm","porPrm",
"paddsbPrm","paddswPrm","pmaxswPrm","pxorPrm",
--Fx
"|||lddquXrm","psllwPrm","pslldPrm","psllqPrm",
"pmuludqPrm","pmaddwdPrm","psadbwPrm","maskmovqMrm||maskmovdquXrm$",
"psubbPrm","psubwPrm","psubdPrm","psubqPrm",
"paddbPrm","paddwPrm","padddPrm","ud",
}
assert(map_opc2[255] == "ud")

-- Map for three-byte opcodes. Can't wait for their next invention.
local map_opc3 = {
["38"] = { -- [66] 0f 38 xx
--0x
[0]="pshufbPrm","phaddwPrm","phadddPrm","phaddswPrm",
"pmaddubswPrm","phsubwPrm","phsubdPrm","phsubswPrm",
"psignbPrm","psignwPrm","psigndPrm","pmulhrswPrm",
nil,nil,nil,nil,
--1x
"||pblendvbXrma",nil,nil,nil,
"||blendvpsXrma","||blendvpdXrma",nil,"||ptestXrm",
nil,nil,nil,nil,
"pabsbPrm","pabswPrm","pabsdPrm",nil,
--2x
"||pmovsxbwXrm","||pmovsxbdXrm","||pmovsxbqXrm","||pmovsxwdXrm",
"||pmovsxwqXrm","||pmovsxdqXrm",nil,nil,
"||pmuldqXrm","||pcmpeqqXrm","||$movntdqaXrm","||packusdwXrm",
nil,nil,nil,nil,
--3x
"||pmovzxbwXrm","||pmovzxbdXrm","||pmovzxbqXrm","||pmovzxwdXrm",
"||pmovzxwqXrm","||pmovzxdqXrm",nil,"||pcmpgtqXrm",
"||pminsbXrm","||pminsdXrm","||pminuwXrm","||pminudXrm",
"||pmaxsbXrm","||pmaxsdXrm","||pmaxuwXrm","||pmaxudXrm",
--4x
```

```
"||pmulddXrm","||phminposuwXrm",
--Fx
[0xf0] = "|||crc32TrBmt",[0xf1] = "|||crc32TrVmt",
},

["3a"] = { -- [66] 0f 3a xx
--0x
[0x00]=nil,nil,nil,nil,nil,nil,nil,nil,
"||roundpsXrmu","||roundpdXrmu","||roundssXrmu","||roundsdXrmu",
"||blendpsXrmu","||blendpdXrmu","||pblendwXrmu","palignrPrmu",
--1x
nil,nil,nil,nil,
"||pextrbVmXru","||pextrwVmXru","||pextrVmSXru","||extractpsVmXru",
nil,nil,nil,nil,nil,nil,nil,nil,
--2x
"||pinsrbXrVmu","||insertpsXrmu","||pinsrXrVmuS",nil,
--4x
[0x40] = "||dppsXrmu",
[0x41] = "||dppdXrmu",
[0x42] = "||mpsadbwXrmu",
--6x
[0x60] = "||pcmpestrmXrmu",[0x61] = "||pcmpestriXrmu",
[0x62] = "||pcmpistrmXrmu",[0x63] = "||pcmpistriXrmu",
},
}

-- Map for VMX/SVM opcodes 0F 01 C0-FF (sgdt group with register operands).
local map_opcvm = {
[0xc1]="vmcall",[0xc2]="vmlaunch",[0xc3]="vmresume",[0xc4]="vmxoff",
[0xc8]="monitor",[0xc9]="mwait",
[0xd8]="vmrun",[0xd9]="vmmcall",[0xda]="vmload",[0xdb]="vmsave",
[0xdc]="stgi",[0xdd]="clgi",[0xde]="skinit",[0xdf]="invlpga",
[0xf8]="swapgs",[0xf9]="rdtscp",
}

-- Map for FP opcodes. And you thought stack machines are simple?
local map_opcfp = {
-- D8-DF 00-BF: opcodes with a memory operand.
-- D8
[0]="faddFm","fmulFm","fcomFm","fcompFm","fsubFm","fsubrFm","fdivFm","fdivrFm",
"fldFm",nil,"fstFm","fstpFm","fldenvVm","fldcwWm","fnstenvVm","fnstcwWm",
-- DA
"fiaddDm","fimulDm","ficomDm","ficompDm",
"fisubDm","fisubrDm","fidivDm","fidivrDm",
-- DB
"fildDm","fisttpDm","fistDm","fistpDm",nil,"fld twordFmp",nil,"fstp twordFmp",
-- DC
"faddGm","fmulGm","fcomGm","fcompGm","fsubGm","fsubrGm","fdivGm","fdivrGm",
-- DD
"fldGm","fisttpQm","fstGm","fstpGm","frstorDmp",nil,"fnsaveDmp","fnstswWm",
-- DE
"fiaddWm","fimulWm","ficomWm","ficompWm",
"fisubWm","fisubrWm","fidivWm","fidivrWm",
-- DF
"fildWm","fisttpWm","fistWm","fistpWm",
"fbld twordFmp","fildQm","fbstp twordFmp","fistpQm",
-- xx C0-FF: opcodes with a pseudo-register operand.
-- D8
"faddFf","fmulFf","fcomFf","fcompFf","fsubFf","fsubrFf","fdivFf","fdivrFf",
-- D9
"fldFf","fxchFf",{"fnop"},nil,
{"fchs","fabs",nil,nil,"ftst","fxam"},
{"fld1","fldl2t","fldl2e","fldpi","fldlg2","fldln2","fldz"},
{"f2xm1","fyl2x","fptan","fpatan","fxtract","fprem1","fdecstp","fincstp"},
{"fprem","fyl2xp1","fsqrt","fsincos","frndint","fscale","fsin","fcos"},
-- DA
"fcmovbFf","fcmoveFf","fcmovbeFf","fcmovuFf",nil,{nil,"fucompp"},nil,nil,
-- DB
"fcmovnbFf","fcmovneFf","fcmovnbeFf","fcmovnuFf",
{nil,nil,"fnclex","fninit"},"fucomiFf","fcomiFf",nil,
-- DC
"fadd toFf","fmul toFf",nil,nil,
"fsub toFf","fsubr toFf","fdivr toFf","fdiv toFf",
-- DD
"ffreeFf",nil,"fstFf","fstpFf","fucomFf","fucompFf",nil,nil,
-- DE
```

```
"faddpFf","fmulpFf",nil,{nil,"fcompp"},
"fsubrpFf","fsubpFf","fdivrpFf","fdivpFf",
-- DF
nil,nil,nil,nil,{"fnstsw ax"},"fucomipFf","fcomipFf",nil,
}
assert(map_opcfp[126] == "fcomipFf")

-- Map for opcode groups. The subkey is sp from the ModRM byte.
local map_opcgroup = {
  arith = { "add", "or", "adc", "sbb", "and", "sub", "xor", "cmp" },
  shift = { "rol", "ror", "rcl", "rcr", "shl", "shr", "sal", "sar" },
  testb = { "testBmi", "testBmi", "not", "neg", "mul", "imul", "div", "idiv" },
  testv = { "testVmi", "testVmi", "not", "neg", "mul", "imul", "div", "idiv" },
  incb = { "inc", "dec" },
  incd = { "inc", "dec", "callUmp", "$call farDmp",
           "jmpUmp", "$jmp farDmp", "pushUm" },
  sldt = { "sldt", "str", "lldt", "ltr", "verr", "verw" },
  sgdt = { "vm*$sgdt", "vm*$sidt", "$lgdt", "vm*$lidt",
           "smsw", nil, "lmsw", "vm*$invlpg" },
  bt = { nil, nil, nil, nil, "bt", "bts", "btr", "btc" },
  cmpxchg = { nil, "sz*,cmpxchg8bQmp,cmpxchg16bXmp", nil, nil,
              nil, nil, "vmptrld|vmxon|vmclear", "vmptrst" },
  pshiftw = { nil, nil, "psrlw", nil, "psraw", nil, "psllw" },
  pshiftd = { nil, nil, "psrld", nil, "psrad", nil, "pslld" },
  pshiftq = { nil, nil, "psrlq", nil, nil, nil, "psllq" },
  pshiftdq = { nil, nil, "psrlq", "psrldq", nil, nil, "psllq", "pslldq" },
  fxsave = { "$fxsave", "$fxrstor", "$ldmxcsr", "$stmxcsr",
             nil, "lfenceDp$", "mfenceDp$", "sfenceDp$clflush" },
  prefetch = { "prefetch", "prefetchw" },
  prefetcht = { "prefetchnta", "prefetcht0", "prefetcht1", "prefetcht2" },
}

------------------------------------------------------------------------------

-- Maps for register names.
local map_regs = {
  B = { "al", "cl", "dl", "bl", "ah", "ch", "dh", "bh",
        "r8b", "r9b", "r10b", "r11b", "r12b", "r13b", "r14b", "r15b" },
  B64 = { "al", "cl", "dl", "bl", "spl", "bpl", "sil", "dil",
          "r8b", "r9b", "r10b", "r11b", "r12b", "r13b", "r14b", "r15b" },
  W = { "ax", "cx", "dx", "bx", "sp", "bp", "si", "di",
        "r8w", "r9w", "r10w", "r11w", "r12w", "r13w", "r14w", "r15w" },
  D = { "eax", "ecx", "edx", "ebx", "esp", "ebp", "esi", "edi",
        "r8d", "r9d", "r10d", "r11d", "r12d", "r13d", "r14d", "r15d" },
  Q = { "rax", "rcx", "rdx", "rbx", "rsp", "rbp", "rsi", "rdi",
        "r8", "r9", "r10", "r11", "r12", "r13", "r14", "r15" },
  M = { "mm0", "mm1", "mm2", "mm3", "mm4", "mm5", "mm6", "mm7",
        "mm0", "mm1", "mm2", "mm3", "mm4", "mm5", "mm6", "mm7" }, -- No x64 ext!
  X = { "xmm0", "xmm1", "xmm2", "xmm3", "xmm4", "xmm5", "xmm6", "xmm7",
        "xmm8", "xmm9", "xmm10", "xmm11", "xmm12", "xmm13", "xmm14", "xmm15" },
}
local map_segregs = { "es", "cs", "ss", "ds", "fs", "gs", "segr6", "segr7" }

-- Maps for size names.
local map_sz2n = {
  B = 1, W = 2, D = 4, Q = 8, M = 8, X = 16,
}
local map_sz2prefix = {
  B = "byte", W = "word", D = "dword",
  Q = "qword",
  M = "qword", X = "xword",
  F = "dword", G = "qword", -- No need for sizes/register names for these two.
}

------------------------------------------------------------------------------

-- Output a nicely formatted line with an opcode and operands.
local function putop(ctx, text, operands)
  local code, pos, hex = ctx.code, ctx.pos, ""
  local hmax = ctx.hexdump
  if hmax > 0 then
    for i=ctx.start,pos-1 do
      hex = hex..format("%02X", byte(code, i, i))
    end
    if #hex > hmax then hex = sub(hex, 1, hmax)..". "
    else hex = hex..rep(" ", hmax-#hex+2) end
```

```
  end
  if operands then text = text.." "..operands end
  if ctx.o16 then text = "o16 "..text; ctx.o16 = false end
  if ctx.a32 then text = "a32 "..text; ctx.a32 = false end
  if ctx.rep then text = ctx.rep.." "..text; ctx.rep = false end
  if ctx.rex then
    local t = (ctx.rexw and "w" or "")..(ctx.rexr and "r" or "")..
              (ctx.rexx and "x" or "")..(ctx.rexb and "b" or "")
    if t ~= "" then text = "rex."..t.." "..text end
    ctx.rexw = false; ctx.rexr = false; ctx.rexx = false; ctx.rexb = false
    ctx.rex = false
  end
  if ctx.seg then
    local text2, n = gsub(text, "%[", "["..ctx.seg..":")
    if n == 0 then text = ctx.seg.." "..text else text = text2 end
    ctx.seg = false
  end
  if ctx.lock then text = "lock "..text; ctx.lock = false end
  local imm = ctx.imm
  if imm then
    local sym = ctx.symtab[imm]
    if sym then text = text.."\t->"..sym end
  end
  ctx.out(format("%08x  %s%s\n", ctx.addr+ctx.start, hex, text))
  ctx.mrm = false
  ctx.start = pos
  ctx.imm = nil
end

-- Clear all prefix flags.
local function clearprefixes(ctx)
  ctx.o16 = false; ctx.seg = false; ctx.lock = false; ctx.rep = false
  ctx.rexw = false; ctx.rexr = false; ctx.rexx = false; ctx.rexb = false
  ctx.rex = false; ctx.a32 = false
end

-- Fallback for incomplete opcodes at the end.
local function incomplete(ctx)
  ctx.pos = ctx.stop+1
  clearprefixes(ctx)
  return putop(ctx, "(incomplete)")
end

-- Fallback for unknown opcodes.
local function unknown(ctx)
  clearprefixes(ctx)
  return putop(ctx, "(unknown)")
end

-- Return an immediate of the specified size.
local function getimm(ctx, pos, n)
  if pos+n-1 > ctx.stop then return incomplete(ctx) end
  local code = ctx.code
  if n == 1 then
    local b1 = byte(code, pos, pos)
    return b1
  elseif n == 2 then
    local b1, b2 = byte(code, pos, pos+1)
    return b1+b2*256
  else
    local b1, b2, b3, b4 = byte(code, pos, pos+3)
    local imm = b1+b2*256+b3*65536+b4*16777216
    ctx.imm = imm
    return imm
  end
end

-- Process pattern string and generate the operands.
local function putpat(ctx, name, pat)
  local operands, regs, sz, mode, sp, rm, sc, rx, sdisp
  local code, pos, stop = ctx.code, ctx.pos, ctx.stop

  -- Chars used: 1DFGIMPQRSTUVWXacdfgijmoprstuwxyz
  for p in gmatch(pat, ".") do
    local x = nil
    if p == "V" or p == "U" then
```

```
      if ctx.rexw then sz = "Q"; ctx.rexw = false
      elseif ctx.o16 then sz = "W"; ctx.o16 = false
      elseif p == "U" and ctx.x64 then sz = "Q"
      else sz = "D" end
      regs = map_regs[sz]
    elseif p == "T" then
      if ctx.rexw then sz = "Q"; ctx.rexw = false else sz = "D" end
      regs = map_regs[sz]
    elseif p == "B" then
      sz = "B"
      regs = ctx.rex and map_regs.B64 or map_regs.B
    elseif match(p, "[WDQMXFG]") then
      sz = p
      regs = map_regs[sz]
    elseif p == "P" then
      sz = ctx.o16 and "X" or "M"; ctx.o16 = false
      regs = map_regs[sz]
    elseif p == "S" then
      name = name..lower(sz)
    elseif p == "s" then
      local imm = getimm(ctx, pos, 1); if not imm then return end
      x = imm <= 127 and format("+0x%02x", imm)
                     or format("-0x%02x", 256-imm)
      pos = pos+1
    elseif p == "u" then
      local imm = getimm(ctx, pos, 1); if not imm then return end
      x = format("0x%02x", imm)
      pos = pos+1
    elseif p == "w" then
      local imm = getimm(ctx, pos, 2); if not imm then return end
      x = format("0x%x", imm)
      pos = pos+2
    elseif p == "o" then -- [offset]
      if ctx.x64 then
        local imm1 = getimm(ctx, pos, 4); if not imm1 then return end
        local imm2 = getimm(ctx, pos+4, 4); if not imm2 then return end
        x = format("[0x%08x%08x]", imm2, imm1)
        pos = pos+8
      else
        local imm = getimm(ctx, pos, 4); if not imm then return end
        x = format("[0x%08x]", imm)
        pos = pos+4
      end
    elseif p == "i" or p == "I" then
      local n = map_sz2n[sz]
      if n == 8 and ctx.x64 and p == "I" then
        local imm1 = getimm(ctx, pos, 4); if not imm1 then return end
        local imm2 = getimm(ctx, pos+4, 4); if not imm2 then return end
        x = format("0x%08x%08x", imm2, imm1)
      else
        if n == 8 then n = 4 end
        local imm = getimm(ctx, pos, n); if not imm then return end
        if sz == "Q" and (imm < 0 or imm > 0x7fffffff) then
          imm = (0xffffffff+1)-imm
          x = format(imm > 65535 and "-0x%08x" or "-0x%x", imm)
        else
          x = format(imm > 65535 and "0x%08x" or "0x%x", imm)
        end
      end
      pos = pos+n
    elseif p == "j" then
      local n = map_sz2n[sz]
      if n == 8 then n = 4 end
      local imm = getimm(ctx, pos, n); if not imm then return end
      if sz == "B" and imm > 127 then imm = imm-256
      elseif imm > 2147483647 then imm = imm-4294967296 end
      pos = pos+n
      imm = imm + pos + ctx.addr
      if imm > 4294967295 and not ctx.x64 then imm = imm-4294967296 end
      ctx.imm = imm
      if sz == "W" then
        x = format("word 0x%04x", imm%65536)
      elseif ctx.x64 then
        local lo = imm % 0x1000000
        x = format("0x%02x%06x", (imm-lo) / 0x1000000, lo)
      else
```

```
537         x = "0x"..tohex(imm)
538       end
539     elseif p == "R" then
540       local r = byte(code, pos-1, pos-1)%8
541       if ctx.rexb then r = r + 8; ctx.rexb = false end
542       x = regs[r+1]
543     elseif p == "a" then x = regs[1]
544     elseif p == "c" then x = "cl"
545     elseif p == "d" then x = "dx"
546     elseif p == "1" then x = "1"
547     else
548       if not mode then
549         mode = ctx.mrm
550         if not mode then
551           if pos > stop then return incomplete(ctx) end
552           mode = byte(code, pos, pos)
553           pos = pos+1
554         end
555         rm = mode%8; mode = (mode-rm)/8
556         sp = mode%8; mode = (mode-sp)/8
557         sdisp = ""
558         if mode < 3 then
559           if rm == 4 then
560             if pos > stop then return incomplete(ctx) end
561             sc = byte(code, pos, pos)
562             pos = pos+1
563             rm = sc%8; sc = (sc-rm)/8
564             rx = sc%8; sc = (sc-rx)/8
565             if ctx.rexx then rx = rx + 8; ctx.rexx = false end
566             if rx == 4 then rx = nil end
567           end
568           if mode > 0 or rm == 5 then
569             local dsz = mode
570             if dsz ~= 1 then dsz = 4 end
571             local disp = getimm(ctx, pos, dsz); if not disp then return end
572             if mode == 0 then rm = nil end
573             if rm or rx or (not sc and ctx.x64 and not ctx.a32) then
574               if dsz == 1 and disp > 127 then
575                 sdisp = format("-0x%x", 256-disp)
576               elseif disp >= 0 and disp <= 0x7fffffff then
577                 sdisp = format("+0x%x", disp)
578               else
579                 sdisp = format("-0x%x", (0xffffffff+1)-disp)
580               end
581             else
582               sdisp = format(ctx.x64 and not ctx.a32 and
583                 not (disp >= 0 and disp <= 0x7fffffff)
584                 and "0xffffffff%08x" or "0x%08x", disp)
585             end
586             pos = pos+dsz
587           end
588         end
589         if rm and ctx.rexb then rm = rm + 8; ctx.rexb = false end
590         if ctx.rexr then sp = sp + 8; ctx.rexr = false end
591       end
592       if p == "m" then
593         if mode == 3 then x = regs[rm+1]
594         else
595           local aregs = ctx.a32 and map_regs.D or ctx.aregs
596           local srm, srx = "", ""
597           if rm then srm = aregs[rm+1]
598           elseif not sc and ctx.x64 and not ctx.a32 then srm = "rip" end
599           ctx.a32 = false
600           if rx then
601             if rm then srm = srm.."+" end
602             srx = aregs[rx+1]
603             if sc > 0 then srx = srx.."*"..(2^sc) end
604           end
605           x = format("[%s%s%s]", srm, srx, sdisp)
606         end
607         if mode < 3 and
608            (not match(pat, "[aRrgp]") or match(pat, "t")) then -- Yuck.
609           x = map_sz2prefix[sz].." "..x
610         end
611       elseif p == "r" then x = regs[sp+1]
612       elseif p == "g" then x = map_segregs[sp+1]
```

```
      elseif p == "p" then -- Suppress prefix.
      elseif p == "f" then x = "st"..rm
      elseif p == "x" then
        if sp == 0 and ctx.lock and not ctx.x64 then
          x = "CR8"; ctx.lock = false
        else
          x = "CR"..sp
        end
      elseif p == "y" then x = "DR"..sp
      elseif p == "z" then x = "TR"..sp
      elseif p == "t" then
      else
        error("bad pattern `"..pat.."'")
      end
    end
    if x then operands = operands and operands..", "..x or x end
  end
  ctx.pos = pos
  return putop(ctx, name, operands)
end

-- Forward declaration.
local map_act

-- Fetch and cache MRM byte.
local function getmrm(ctx)
  local mrm = ctx.mrm
  if not mrm then
    local pos = ctx.pos
    if pos > ctx.stop then return nil end
    mrm = byte(ctx.code, pos, pos)
    ctx.pos = pos+1
    ctx.mrm = mrm
  end
  return mrm
end

-- Dispatch to handler depending on pattern.
local function dispatch(ctx, opat, patgrp)
  if not opat then return unknown(ctx) end
  if match(opat, "%|") then -- MMX/SSE variants depending on prefix.
    local p
    if ctx.rep then
      p = ctx.rep=="rep" and "%|([^%|]*)" or "%|[^%|]*%|[^%|]*%|([^%|]*)"
      ctx.rep = false
    elseif ctx.o16 then p = "%|[^%|]*%|([^%|]*)"; ctx.o16 = false
    else p = "^[^%|]*" end
    opat = match(opat, p)
    if not opat then return unknown(ctx) end
--    ctx.rep = false; ctx.o16 = false
    --XXX fails for 66 f2 0f 38 f1 06  crc32 eax,WORD PTR [esi]
    --XXX remove in branches?
  end
  if match(opat, "%$") then -- reg$mem variants.
    local mrm = getmrm(ctx); if not mrm then return incomplete(ctx) end
    opat = match(opat, mrm >= 192 and "^[^%$]*" or "%$(.*)")
    if opat == "" then return unknown(ctx) end
  end
  if opat == "" then return unknown(ctx) end
  local name, pat = match(opat, "^([a-z0-9 ]*)(.*)")
  if pat == "" and patgrp then pat = patgrp end
  return map_act[sub(pat, 1, 1)](ctx, name, pat)
end

-- Get a pattern from an opcode map and dispatch to handler.
local function dispatchmap(ctx, opcmap)
  local pos = ctx.pos
  local opat = opcmap[byte(ctx.code, pos, pos)]
  pos = pos + 1
  ctx.pos = pos
  return dispatch(ctx, opat)
end

-- Map for action codes. The key is the first char after the name.
map_act = {
  -- Simple opcodes without operands.
```

```
  [""] = function(ctx, name, pat)
    return putop(ctx, name)
  end,

  -- Operand size chars fall right through.
  B = putpat, W = putpat, D = putpat, Q = putpat,
  V = putpat, U = putpat, T = putpat,
  M = putpat, X = putpat, P = putpat,
  F = putpat, G = putpat,

  -- Collect prefixes.
  [":"] = function(ctx, name, pat)
    ctx[pat == ":" and name or sub(pat, 2)] = name
    if ctx.pos - ctx.start > 5 then return unknown(ctx) end -- Limit #prefixes.
  end,

  -- Chain to special handler specified by name.
  ["*"] = function(ctx, name, pat)
    return map_act[name](ctx, name, sub(pat, 2))
  end,

  -- Use named subtable for opcode group.
  ["!"] = function(ctx, name, pat)
    local mrm = getmrm(ctx); if not mrm then return incomplete(ctx) end
    return dispatch(ctx, map_opcgroup[name][((mrm-(mrm%8))/8)%8+1], sub(pat, 2))
  end,

  -- o16,o32[,o64] variants.
  sz = function(ctx, name, pat)
    if ctx.o16 then ctx.o16 = false
    else
      pat = match(pat, ",(.*)")
      if ctx.rexw then
        local p = match(pat, ",(.*)")
        if p then pat = p; ctx.rexw = false end
      end
    end
    pat = match(pat, "^[^,]*")
    return dispatch(ctx, pat)
  end,

  -- Two-byte opcode dispatch.
  opc2 = function(ctx, name, pat)
    return dispatchmap(ctx, map_opc2)
  end,

  -- Three-byte opcode dispatch.
  opc3 = function(ctx, name, pat)
    return dispatchmap(ctx, map_opc3[pat])
  end,

  -- VMX/SVM dispatch.
  vm = function(ctx, name, pat)
    return dispatch(ctx, map_opcvm[ctx.mrm])
  end,

  -- Floating point opcode dispatch.
  fp = function(ctx, name, pat)
    local mrm = getmrm(ctx); if not mrm then return incomplete(ctx) end
    local rm = mrm%8
    local idx = pat*8 + ((mrm-rm)/8)%8
    if mrm >= 192 then idx = idx + 64 end
    local opat = map_opcfp[idx]
    if type(opat) == "table" then opat = opat[rm+1] end
    return dispatch(ctx, opat)
  end,

  -- REX prefix.
  rex = function(ctx, name, pat)
    if ctx.rex then return unknown(ctx) end -- Only 1 REX prefix allowed.
    for p in gmatch(pat, ".") do ctx["rex"..p] = true end
    ctx.rex = true
  end,

  -- Special case for nop with REX prefix.
  nop = function(ctx, name, pat)
```

```
765     return dispatch(ctx, ctx.rex and pat or "nop")
766   end,
767 }
768
769 ------------------------------------------------------------------------------
770
771 -- Disassemble a block of code.
772 local function disass_block(ctx, ofs, len)
773   if not ofs then ofs = 0 end
774   local stop = len and ofs+len or #ctx.code
775   ofs = ofs + 1
776   ctx.start = ofs
777   ctx.pos = ofs
778   ctx.stop = stop
779   ctx.imm = nil
780   ctx.mrm = false
781   clearprefixes(ctx)
782   while ctx.pos <= stop do dispatchmap(ctx, ctx.map1) end
783   if ctx.pos ~= ctx.start then incomplete(ctx) end
784 end
785
786 -- Extended API: create a disassembler context. Then call ctx:disass(ofs, len).
787 local function create(code, addr, out)
788   local ctx = {}
789   ctx.code = code
790   ctx.addr = (addr or 0) - 1
791   ctx.out = out or io.write
792   ctx.symtab = {}
793   ctx.disass = disass_block
794   ctx.hexdump = 16
795   ctx.x64 = false
796   ctx.map1 = map_opc1_32
797   ctx.aregs = map_regs.D
798   return ctx
799 end
800
801 local function create64(code, addr, out)
802   local ctx = create(code, addr, out)
803   ctx.x64 = true
804   ctx.map1 = map_opc1_64
805   ctx.aregs = map_regs.Q
806   return ctx
807 end
808
809 -- Simple API: disassemble code (a string) at address and output via out.
810 local function disass(code, addr, out)
811   create(code, addr, out):disass()
812 end
813
814 local function disass64(code, addr, out)
815   create64(code, addr, out):disass()
816 end
817
818 -- Return register name for RID.
819 local function regname(r)
820   if r < 8 then return map_regs.D[r+1] end
821   return map_regs.X[r-7]
822 end
823
824 local function regname64(r)
825   if r < 16 then return map_regs.Q[r+1] end
826   return map_regs.X[r-15]
827 end
828
829 -- Public module functions.
830 return {
831   create = create,
832   create64 = create64,
833   disass = disass,
834   disass64 = disass64,
835   regname = regname,
836   regname64 = regname64
837 }
838
```

One Level Up Top Level

src/jit/dis_mips.lua - luajit-2.0-src

Functions defined

Source code

```
1  ------------------------------------------------------------------------------
2  -- LuaJIT MIPS disassembler module.
3  --
4  -- Copyright (C) 2005-2015 Mike Pall. All rights reserved.
5  -- Released under the MIT/X license. See Copyright Notice in luajit.h
6  ------------------------------------------------------------------------------
7  -- This is a helper module used by the LuaJIT machine code dumper module.
8  --
9  -- It disassembles all standard MIPS32R1/R2 instructions.
10 -- Default mode is big-endian, but see: dis_mipsel.lua
11 ------------------------------------------------------------------------------
12
13 local type = type
14 local sub, byte, format = string.sub, string.byte, string.format
15 local match, gmatch, gsub = string.match, string.gmatch, string.gsub
16 local concat = table.concat
17 local bit = require("bit")
18 local band, bor, tohex = bit.band, bit.bor, bit.tohex
19 local lshift, rshift, arshift = bit.lshift, bit.rshift, bit.arshift
20
21 ------------------------------------------------------------------------------
22 -- Primary and extended opcode maps
23 ------------------------------------------------------------------------------
24
25 local map_movci = { shift = 16, mask = 1, [0] = "movfDSC", "movtDSC", }
26 local map_srl = { shift = 21, mask = 1, [0] = "srlDTA", "rotrDTA", }
27 local map_srlv = { shift = 6, mask = 1, [0] = "srlvDTS", "rotrvDTS", }
28
29 local map_special = {
30   shift = 0, mask = 63,
31   [0] = { shift = 0, mask = -1, [0] = "nop", _ = "sllDTA" },
32   map_movci,        map_srl,        "sraDTA",
33   "sllvDTS",        false,               map_srlv,        "sravDTS",
34   "jrS",        "jalrD1S",        "movzDST",        "movnDST",
35   "syscallY",        "breakY",        false,               "sync",
36   "mfhiD",        "mthiS",        "mfloD",        "mtloS",
37   false,        false,               false,               false,
38   "multST",        "multuST",        "divST",        "divuST",
39   false,        false,               false,               false,
40   "addDST",        "addu|moveDST0", "subDST",        "subu|neguDS0T",
```

```
  "andDST",        "orDST",        "xorDST",        "nor|notDST0",
  false,        false,              "sltDST",        "sltuDST",
  false,        false,              false,              false,
  "tgeSTZ",        "tgeuSTZ",       "tltSTZ",        "tltuSTZ",
  "teqSTZ",        false,              "tneSTZ",
}

local map_special2 = {
  shift = 0, mask = 63,
  [0] = "maddST", "madduST",       "mulDST",        false,
  "msubST",        "msubuST",
  [32] = "clzDS", [33] = "cloDS",
  [63] = "sdbbpY",
}

local map_bshfl = {
  shift = 6, mask = 31,
  [2] = "wsbhDT",
  [16] = "sebDT",
  [24] = "sehDT",
}

local map_special3 = {
  shift = 0, mask = 63,
  [0] = "extTSAK", [4] = "insTSAL",
  [32] = map_bshfl,
  [59] = "rdhwrTD",
}

local map_regimm = {
  shift = 16, mask = 31,
  [0] = "bltzSB",        "bgezSB",        "bltzlSB",        "bgezlSB",
  false,        false,              false,              false,
  "tgeiSI",        "tgeiuSI",       "tltiSI",        "tltiuSI",
  "teqiSI",        false,              "tneiSI",        false,
  "bltzalSB",        "bgezalSB",       "bltzallSB",        "bgezallSB",
  false,        false,              false,              false,
  false,        false,              false,              false,
  false,        false,              false,              "synciSO",
}

local map_cop0 = {
  shift = 25, mask = 1,
  [0] = {
    shift = 21, mask = 15,
    [0] = "mfc0TDW", [4] = "mtc0TDW",
    [10] = "rdpgprDT",
    [11] = { shift = 5, mask = 1, [0] = "diT0", "eiT0", },
    [14] = "wrpgprDT",
  }, {
    shift = 0, mask = 63,
    [1] = "tlbr", [2] = "tlbwi", [6] = "tlbwr", [8] = "tlbp",
    [24] = "eret", [31] = "deret",
    [32] = "wait",
  },
}

local map_cop1s = {
  shift = 0, mask = 63,
   [0] = "add.sFGH",        "sub.sFGH",        "mul.sFGH",        "div.sFGH",
   "sqrt.sFG",              "abs.sFG",         "mov.sFG",         "neg.sFG",
   "round.l.sFG",        "trunc.l.sFG",        "ceil.l.sFG",        "floor.l.sFG",
   "round.w.sFG",        "trunc.w.sFG",        "ceil.w.sFG",        "floor.w.sFG",
   false,
   { shift = 16, mask = 1, [0] = "movf.sFGC", "movt.sFGC" },
   "movz.sFGT",        "movn.sFGT",
   false,        "recip.sFG",        "rsqrt.sFG",        false,
   false,        false,              false,              false,
   false,        false,              false,              false,
   false,        "cvt.d.sFG",        false,              false,
   "cvt.w.sFG",        "cvt.l.sFG",        "cvt.ps.sFGH",        false,
   false,        false,              false,              false,
   false,        false,              false,              false,
   "c.f.sVGH",        "c.un.sVGH",        "c.eq.sVGH",        "c.ueq.sVGH",
   "c.olt.sVGH",        "c.ult.sVGH",        "c.ole.sVGH",        "c.ule.sVGH",
   "c.sf.sVGH",        "c.ngle.sVGH",        "c.seq.sVGH",        "c.ngl.sVGH",
```

```
  "c.lt.sVGH",        "c.nge.sVGH",        "c.le.sVGH",        "c.ngt.sVGH",
}

local map_cop1d = {
  shift = 0, mask = 63,
  [0] = "add.dFGH",        "sub.dFGH",        "mul.dFGH",        "div.dFGH",
  "sqrt.dFG",                "abs.dFG",        "mov.dFG",        "neg.dFG",
  "round.l.dFG",        "trunc.l.dFG",        "ceil.l.dFG",        "floor.l.dFG",
  "round.w.dFG",        "trunc.w.dFG",        "ceil.w.dFG",        "floor.w.dFG",
  false,
  { shift = 16, mask = 1, [0] = "movf.dFGC", "movt.dFGC" },
  "movz.dFGT",        "movn.dFGT",
  false,        "recip.dFG",        "rsqrt.dFG",        false,
  false,        false,                false,                false,
  false,        false,                false,                false,
  "cvt.s.dFG",        false,                false,                false,
  "cvt.w.dFG",        "cvt.l.dFG",        false,                false,
  false,        false,                false,                false,
  false,        false,                false,                false,
  "c.f.dVGH",        "c.un.dVGH",        "c.eq.dVGH",        "c.ueq.dVGH",
  "c.olt.dVGH",        "c.ult.dVGH",        "c.ole.dVGH",        "c.ule.dVGH",
  "c.df.dVGH",        "c.ngle.dVGH",        "c.deq.dVGH",        "c.ngl.dVGH",
  "c.lt.dVGH",        "c.nge.dVGH",        "c.le.dVGH",        "c.ngt.dVGH",
}

local map_cop1ps = {
  shift = 0, mask = 63,
  [0] = "add.psFGH",        "sub.psFGH",        "mul.psFGH",        false,
  false,                "abs.psFG",        "mov.psFG",        "neg.psFG",
  false,                false,                false,                false,
  false,                false,                false,                false,
  false,
  { shift = 16, mask = 1, [0] = "movf.psFGC", "movt.psFGC" },
  "movz.psFGT",        "movn.psFGT",
  false,        false,                false,                false,
  false,        false,                false,                false,
  false,        false,                false,                false,
  "cvt.s.puFG",        false,                false,                false,
  false,        false,                false,                false,
  "cvt.s.plFG",        false,                false,                false,
  "pll.psFGH",        "plu.psFGH",        "pul.psFGH",        "puu.psFGH",
  "c.f.psVGH",        "c.un.psVGH",        "c.eq.psVGH",        "c.ueq.psVGH",
  "c.olt.psVGH", "c.ult.psVGH",        "c.ole.psVGH",        "c.ule.psVGH",
  "c.psf.psVGH", "c.ngle.psVGH", "c.pseq.psVGH", "c.ngl.psVGH",
  "c.lt.psVGH",        "c.nge.psVGH",        "c.le.psVGH",        "c.ngt.psVGH",
}

local map_cop1w = {
  shift = 0, mask = 63,
  [32] = "cvt.s.wFG", [33] = "cvt.d.wFG",
}

local map_cop1l = {
  shift = 0, mask = 63,
  [32] = "cvt.s.lFG", [33] = "cvt.d.lFG",
}

local map_cop1bc = {
  shift = 16, mask = 3,
  [0] = "bc1fCB", "bc1tCB",        "bc1flCB",        "bc1tlCB",
}

local map_cop1 = {
  shift = 21, mask = 31,
  [0] = "mfc1TG", false,        "cfc1TG",        "mfhc1TG",
  "mtc1TG",        false,                "ctc1TG",        "mthc1TG",
  map_cop1bc,        false,                false,                false,
  false,        false,                false,                false,
  map_cop1s,        map_cop1d,        false,                false,
  map_cop1w,        map_cop1l,        map_cop1ps,
}

local map_cop1x = {
  shift = 0, mask = 63,
  [0] = "lwxc1FSX",        "ldxc1FSX",        false,                false,
  false,        "luxc1FSX",        false,                false,
```

```
  "swxc1FSX",        "sdxc1FSX",        false,               false,
  false,        "suxc1FSX",        false,               "prefxMSX",
  false,        false,               false,               false,
  false,        false,               false,               false,
  false,        false,               false,               false,
  false,        false,               "alnv.psFGHS",       false,
  "madd.sFRGH",        "madd.dFRGH",        false,               false,
  false,        false,               "madd.psFRGH",       false,
  "msub.sFRGH",        "msub.dFRGH",        false,               false,
  false,        false,               "msub.psFRGH",       false,
  "nmadd.sFRGH", "nmadd.dFRGH",        false,               false,
  false,        false,               "nmadd.psFRGH",      false,
  "nmsub.sFRGH", "nmsub.dFRGH",        false,               false,
  false,        false,               "nmsub.psFRGH",      false,
}

local map_pri = {
  [0] = map_special,        map_regimm,        "jJ",        "jalJ",
  "beq|beqz|bST00B",        "bne|bnezST0B",               "blezSB",        "bgtzSB",
  "addiTSI",        "addiu|liTS0I",        "sltiTSI",        "sltiuTSI",
  "andiTSU",        "ori|liTS0U",        "xoriTSU",        "luiTU",
  map_cop0,        map_cop1,        false,               map_cop1x,
  "beql|beqzlST0B",        "bnel|bnezlST0B",        "blezlSB",        "bgtzlSB",
  false,        false,               false,               false,
  map_special2,        false,               false,               map_special3,
  "lbTSO",        "lhTSO",        "lwlTSO",        "lwTSO",
  "lbuTSO",        "lhuTSO",        "lwrTSO",        false,
  "sbTSO",        "shTSO",        "swlTSO",        "swTSO",
  false,        false,               "swrTSO",        "cacheNSO",
  "llTSO",        "lwc1HSO",        "lwc2TSO",        "prefNSO",
  false,        "ldc1HSO",        "ldc2TSO",        false,
  "scTSO",        "swc1HSO",        "swc2TSO",        false,
  false,        "sdc1HSO",        "sdc2TSO",        false,
}

------------------------------------------------------------------------------

local map_gpr = {
  [0] = "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7",
  "r8", "r9", "r10", "r11", "r12", "r13", "r14", "r15",
  "r16", "r17", "r18", "r19", "r20", "r21", "r22", "r23",
  "r24", "r25", "r26", "r27", "r28", "sp", "r30", "ra",
}

------------------------------------------------------------------------------

-- Output a nicely formatted line with an opcode and operands.
local function putop(ctx, text, operands)
  local pos = ctx.pos
  local extra = ""
  if ctx.rel then
    local sym = ctx.symtab[ctx.rel]
    if sym then extra = "\t->"..sym end
  end
  if ctx.hexdump > 0 then
    ctx.out(format("%08x  %s  %-7s %s%s\n",
            ctx.addr+pos, tohex(ctx.op), text, concat(operands, ", "), extra))
  else
    ctx.out(format("%08x  %-7s %s%s\n",
            ctx.addr+pos, text, concat(operands, ", "), extra))
  end
  ctx.pos = pos + 4
end

-- Fallback for unknown opcodes.
local function unknown(ctx)
  return putop(ctx, ".long", { "0x"..tohex(ctx.op) })
end

local function get_be(ctx)
  local pos = ctx.pos
  local b0, b1, b2, b3 = byte(ctx.code, pos+1, pos+4)
  return bor(lshift(b0, 24), lshift(b1, 16), lshift(b2, 8), b3)
end

local function get_le(ctx)
```

```
  local pos = ctx.pos
  local b0, b1, b2, b3 = byte(ctx.code, pos+1, pos+4)
  return bor(lshift(b3, 24), lshift(b2, 16), lshift(b1, 8), b0)
end

-- Disassemble a single instruction.
local function disass_ins(ctx)
  local op = ctx:get()
  local operands = {}
  local last = nil
  ctx.op = op
  ctx.rel = nil

  local opat = map_pri[rshift(op, 26)]
  while type(opat) ~= "string" do
    if not opat then return unknown(ctx) end
    opat = opat[band(rshift(op, opat.shift), opat.mask)] or opat._
  end
  local name, pat = match(opat, "^([a-z0-9_.]*)(.*)")
  local altname, pat2 = match(pat, "|([a-z0-9_.|]*)(.*)")
  if altname then pat = pat2 end

  for p in gmatch(pat, ".") do
    local x = nil
    if p == "S" then
      x = map_gpr[band(rshift(op, 21), 31)]
    elseif p == "T" then
      x = map_gpr[band(rshift(op, 16), 31)]
    elseif p == "D" then
      x = map_gpr[band(rshift(op, 11), 31)]
    elseif p == "F" then
      x = "f"..band(rshift(op, 6), 31)
    elseif p == "G" then
      x = "f"..band(rshift(op, 11), 31)
    elseif p == "H" then
      x = "f"..band(rshift(op, 16), 31)
    elseif p == "R" then
      x = "f"..band(rshift(op, 21), 31)
    elseif p == "A" then
      x = band(rshift(op, 6), 31)
    elseif p == "M" then
      x = band(rshift(op, 11), 31)
    elseif p == "N" then
      x = band(rshift(op, 16), 31)
    elseif p == "C" then
      x = band(rshift(op, 18), 7)
      if x == 0 then x = nil end
    elseif p == "K" then
      x = band(rshift(op, 11), 31) + 1
    elseif p == "L" then
      x = band(rshift(op, 11), 31) - last + 1
    elseif p == "I" then
      x = arshift(lshift(op, 16), 16)
    elseif p == "U" then
      x = band(op, 0xffff)
    elseif p == "O" then
      local disp = arshift(lshift(op, 16), 16)
      operands[#operands] = format("%d(%s)", disp, last)
    elseif p == "X" then
      local index = map_gpr[band(rshift(op, 16), 31)]
      operands[#operands] = format("%s(%s)", index, last)
    elseif p == "B" then
      x = ctx.addr + ctx.pos + arshift(lshift(op, 16), 16)*4 + 4
      ctx.rel = x
      x = "0x"..tohex(x)
    elseif p == "J" then
      x = band(ctx.addr + ctx.pos, 0xf0000000) + band(op, 0x03ffffff)*4
      ctx.rel = x
      x = "0x"..tohex(x)
    elseif p == "V" then
      x = band(rshift(op, 8), 7)
      if x == 0 then x = nil end
    elseif p == "W" then
      x = band(op, 7)
      if x == 0 then x = nil end
    elseif p == "Y" then
```

```
      x = band(rshift(op, 6), 0x000fffff)
      if x == 0 then x = nil end
    elseif p == "Z" then
      x = band(rshift(op, 6), 1023)
      if x == 0 then x = nil end
    elseif p == "0" then
      if last == "r0" or last == 0 then
        local n = #operands
        operands[n] = nil
        last = operands[n-1]
        if altname then
          local a1, a2 = match(altname, "([^|]*)|(.*)")
          if a1 then name, altname = a1, a2
          else name = altname end
        end
      end
    elseif p == "1" then
      if last == "ra" then
        operands[#operands] = nil
      end
    else
      assert(false)
    end
    if x then operands[#operands+1] = x; last = x end
  end

  return putop(ctx, name, operands)
end

------------------------------------------------------------------------------

-- Disassemble a block of code.
local function disass_block(ctx, ofs, len)
  if not ofs then ofs = 0 end
  local stop = len and ofs+len or #ctx.code
  stop = stop - stop % 4
  ctx.pos = ofs - ofs % 4
  ctx.rel = nil
  while ctx.pos < stop do disass_ins(ctx) end
end

-- Extended API: create a disassembler context. Then call ctx:disass(ofs, len).
local function create(code, addr, out)
  local ctx = {}
  ctx.code = code
  ctx.addr = addr or 0
  ctx.out = out or io.write
  ctx.symtab = {}
  ctx.disass = disass_block
  ctx.hexdump = 8
  ctx.get = get_be
  return ctx
end

local function create_el(code, addr, out)
  local ctx = create(code, addr, out)
  ctx.get = get_le
  return ctx
end

-- Simple API: disassemble code (a string) at address and output via out.
local function disass(code, addr, out)
  create(code, addr, out):disass()
end

local function disass_el(code, addr, out)
  create_el(code, addr, out):disass()
end

-- Return register name for RID.
local function regname(r)
  if r < 32 then return map_gpr[r] end
  return "f"..(r-32)
end

-- Public module functions.
```

```
return {
  create = create,
  create_el = create_el,
  disass = disass,
  disass_el = disass_el,
  regname = regname
}

```

One Level Up Top Level

src/jit/dis_arm.lua - luajit-2.0-src

Functions defined

Source code

```
1  ------------------------------------------------------------------------------
2  -- LuaJIT ARM disassembler module.
3  --
4  -- Copyright (C) 2005-2015 Mike Pall. All rights reserved.
5  -- Released under the MIT license. See Copyright Notice in luajit.h
6  ------------------------------------------------------------------------------
7  -- This is a helper module used by the LuaJIT machine code dumper module.
8  --
9  -- It disassembles most user-mode ARMv7 instructions
10 -- NYI: Advanced SIMD and VFP instructions.
11 ------------------------------------------------------------------------------
12
13 local type = type
14 local sub, byte, format = string.sub, string.byte, string.format
15 local match, gmatch, gsub = string.match, string.gmatch, string.gsub
16 local concat = table.concat
17 local bit = require("bit")
18 local band, bor, ror, tohex = bit.band, bit.bor, bit.ror, bit.tohex
19 local lshift, rshift, arshift = bit.lshift, bit.rshift, bit.arshift
20
21 ------------------------------------------------------------------------------
22 -- Opcode maps
23 ------------------------------------------------------------------------------
24
25 local map_loadc = {
26   shift = 8, mask = 15,
27   [10] = {
28     shift = 20, mask = 1,
29     [0] = {
30       shift = 23, mask = 3,
31       [0] = "vmovFmDN", "vstmFNdr",
32       _ = {
33         shift = 21, mask = 1,
34         [0] = "vstrFdl",
35         { shift = 16, mask = 15, [13] = "vpushFdr", _ = "vstmdbFNdr", }
36       },
37     },
38     {
39       shift = 23, mask = 3,
40       [0] = "vmovFDNm",
41       { shift = 16, mask = 15, [13] = "vpopFdr", _ = "vldmFNdr", },
42       _ = {
```

```
        shift = 21, mask = 1,
        [0] = "vldrFdl", "vldmdbFNdr",
      },
    },
  },
  [11] = {
    shift = 20, mask = 1,
    [0] = {
      shift = 23, mask = 3,
      [0] = "vmovGmDN", "vstmGNdr",
      _ = {
        shift = 21, mask = 1,
        [0] = "vstrGdl",
        { shift = 16, mask = 15, [13] = "vpushGdr", _ = "vstmdbGNdr", }
      },
    },
    {
      shift = 23, mask = 3,
      [0] = "vmovGDNm",
      { shift = 16, mask = 15, [13] = "vpopGdr", _ = "vldmGNdr", },
      _ = {
        shift = 21, mask = 1,
        [0] = "vldrGdl", "vldmdbGNdr",
      },
    },
  },
  _ = {
    shift = 0, mask = 0 -- NYI ldc, mcrr, mrrc.
  },
}

local map_vfps = {
  shift = 6, mask = 0x2c001,
  [0] = "vmlaF.dnm", "vmlsF.dnm",
  [0x04000] = "vnmlsF.dnm", [0x04001] = "vnmlaF.dnm",
  [0x08000] = "vmulF.dnm", [0x08001] = "vnmulF.dnm",
  [0x0c000] = "vaddF.dnm", [0x0c001] = "vsubF.dnm",
  [0x20000] = "vdivF.dnm",
  [0x24000] = "vfnmsF.dnm", [0x24001] = "vfnmaF.dnm",
  [0x28000] = "vfmaF.dnm", [0x28001] = "vfmsF.dnm",
  [0x2c000] = "vmovF.dY",
  [0x2c001] = {
    shift = 7, mask = 0x1e01,
    [0] = "vmovF.dm", "vabsF.dm",
    [0x0200] = "vnegF.dm", [0x0201] = "vsqrtF.dm",
    [0x0800] = "vcmpF.dm", [0x0801] = "vcmpeF.dm",
    [0x0a00] = "vcmpzF.d", [0x0a01] = "vcmpzeF.d",
    [0x0e01] = "vcvtG.dF.m",
    [0x1000] = "vcvt.f32.u32Fdm", [0x1001] = "vcvt.f32.s32Fdm",
    [0x1800] = "vcvtr.u32F.dm", [0x1801] = "vcvt.u32F.dm",
    [0x1a00] = "vcvtr.s32F.dm", [0x1a01] = "vcvt.s32F.dm",
  },
}

local map_vfpd = {
  shift = 6, mask = 0x2c001,
  [0] = "vmlaG.dnm", "vmlsG.dnm",
   [0x04000] = "vnmlsG.dnm", [0x04001] = "vnmlaG.dnm",
   [0x08000] = "vmulG.dnm", [0x08001] = "vnmulG.dnm",
   [0x0c000] = "vaddG.dnm", [0x0c001] = "vsubG.dnm",
   [0x20000] = "vdivG.dnm",
   [0x24000] = "vfnmsG.dnm", [0x24001] = "vfnmaG.dnm",
   [0x28000] = "vfmaG.dnm", [0x28001] = "vfmsG.dnm",
   [0x2c000] = "vmovG.dY",
   [0x2c001] = {
     shift = 7, mask = 0x1e01,
     [0] = "vmovG.dm", "vabsG.dm",
     [0x0200] = "vnegG.dm", [0x0201] = "vsqrtG.dm",
     [0x0800] = "vcmpG.dm", [0x0801] = "vcmpeG.dm",
     [0x0a00] = "vcmpzG.d", [0x0a01] = "vcmpzeG.d",
     [0x0e01] = "vcvtF.dG.m",
     [0x1000] = "vcvt.f64.u32GdFm", [0x1001] = "vcvt.f64.s32GdFm",
     [0x1800] = "vcvtr.u32FdG.m", [0x1801] = "vcvt.u32FdG.m",
     [0x1a00] = "vcvtr.s32FdG.m", [0x1a01] = "vcvt.s32FdG.m",
   },
 }
```

```
119
120 local map_datac = {
121   shift = 24, mask = 1,
122   [0] = {
123     shift = 4, mask = 1,
124     [0] = {
125       shift = 8, mask = 15,
126       [10] = map_vfps,
127       [11] = map_vfpd,
128       -- NYI cdp, mcr, mrc.
129     },
130     {
131       shift = 8, mask = 15,
132       [10] = {
133         shift = 20, mask = 15,
134         [0] = "vmovFnD", "vmovFDn",
135         [14] = "vmsrD",
136         [15] = { shift = 12, mask = 15, [15] = "vmrs", _ = "vmrsD", },
137       },
138     },
139   },
140   "svcT",
141 }
142
143 local map_loadcu = {
144   shift = 0, mask = 0, -- NYI unconditional CP load/store.
145 }
146
147 local map_datacu = {
148   shift = 0, mask = 0, -- NYI unconditional CP data.
149 }
150
151 local map_simddata = {
152   shift = 0, mask = 0, -- NYI SIMD data.
153 }
154
155 local map_simdload = {
156   shift = 0, mask = 0, -- NYI SIMD load/store, preload.
157 }
158
159 local map_preload = {
160   shift = 0, mask = 0, -- NYI preload.
161 }
162
163 local map_media = {
164   shift = 20, mask = 31,
165   [0] = false,
166   { --01
167     shift = 5, mask = 7,
168     [0] = "sadd16DNM", "sasxDNM", "ssaxDNM", "ssub16DNM",
169     "sadd8DNM", false, false, "ssub8DNM",
170   },
171   { --02
172     shift = 5, mask = 7,
173     [0] = "qadd16DNM", "qasxDNM", "qsaxDNM", "qsub16DNM",
174     "qadd8DNM", false, false, "qsub8DNM",
175   },
176   { --03
177     shift = 5, mask = 7,
178     [0] = "shadd16DNM", "shasxDNM", "shsaxDNM", "shsub16DNM",
179     "shadd8DNM", false, false, "shsub8DNM",
180   },
181   false,
182   { --05
183     shift = 5, mask = 7,
184     [0] = "uadd16DNM", "uasxDNM", "usaxDNM", "usub16DNM",
185     "uadd8DNM", false, false, "usub8DNM",
186   },
187   { --06
188     shift = 5, mask = 7,
189     [0] = "uqadd16DNM", "uqasxDNM", "uqsaxDNM", "uqsub16DNM",
190     "uqadd8DNM", false, false, "uqsub8DNM",
191   },
192   { --07
193     shift = 5, mask = 7,
194     [0] = "uhadd16DNM", "uhasxDNM", "uhsaxDNM", "uhsub16DNM",
```

```
    "uhadd8DNM", false, false, "uhsub8DNM",
  },
  { --08
    shift = 5, mask = 7,
    [0] = "pkhbtDNMU", false, "pkhtbDNMU",
    { shift = 16, mask = 15, [15] = "sxtb16DMU", _ = "sxtab16DNMU", },
    "pkhbtDNMU", "selDNM", "pkhtbDNMU",
  },
  false,
  { --0a
    shift = 5, mask = 7,
    [0] = "ssatDxMu", "ssat16DxM", "ssatDxMu",
    { shift = 16, mask = 15, [15] = "sxtbDMU", _ = "sxtabDNMU", },
    "ssatDxMu", false, "ssatDxMu",
  },
  { --0b
    shift = 5, mask = 7,
    [0] = "ssatDxMu", "revDM", "ssatDxMu",
    { shift = 16, mask = 15, [15] = "sxthDMU", _ = "sxtahDNMU", },
    "ssatDxMu", "rev16DM", "ssatDxMu",
  },
  { --0c
    shift = 5, mask = 7,
    [3] = { shift = 16, mask = 15, [15] = "uxtb16DMU", _ = "uxtab16DNMU", },
  },
  false,
  { --0e
    shift = 5, mask = 7,
    [0] = "usatDwMu", "usat16DwM", "usatDwMu",
    { shift = 16, mask = 15, [15] = "uxtbDMU", _ = "uxtabDNMU", },
    "usatDwMu", false, "usatDwMu",
  },
  { --0f
    shift = 5, mask = 7,
    [0] = "usatDwMu", "rbitDM", "usatDwMu",
    { shift = 16, mask = 15, [15] = "uxthDMU", _ = "uxtahDNMU", },
    "usatDwMu", "revshDM", "usatDwMu",
  },
  { --10
    shift = 12, mask = 15,
    [15] = {
      shift = 5, mask = 7,
      "smuadNMS", "smuadxNMS", "smusdNMS", "smusdxNMS",
    },
    _ = {
      shift = 5, mask = 7,
      [0] = "smladNMSD", "smladxNMSD", "smlsdNMSD", "smlsdxNMSD",
    },
  },
  false, false, false,
  { --14
    shift = 5, mask = 7,
    [0] = "smlaldDNMS", "smlaldxDNMS", "smlsldDNMS", "smlsldxDNMS",
  },
  { --15
    shift = 5, mask = 7,
    [0] = { shift = 12, mask = 15, [15] = "smmulNMS", _ = "smmlaNMSD", },
    { shift = 12, mask = 15, [15] = "smmulrNMS", _ = "smmlarNMSD", },
    false, false, false, false,
    "smmlsNMSD", "smmlsrNMSD",
  },
  false, false,
  { --18
    shift = 5, mask = 7,
    [0] = { shift = 12, mask = 15, [15] = "usad8NMS", _ = "usada8NMSD", },
  },
  false,
  { --1a
    shift = 5, mask = 3, [2] = "sbfxDMvw",
  },
  { --1b
    shift = 5, mask = 3, [2] = "sbfxDMvw",
  },
  { --1c
    shift = 5, mask = 3,
    [0] = { shift = 0, mask = 15, [15] = "bfcDvX", _ = "bfiDMvX", },
```

```
  },
  { --1d
    shift = 5, mask = 3,
    [0] = { shift = 0, mask = 15, [15] = "bfcDvX", _ = "bfiDMvX", },
  },
  { --1e
    shift = 5, mask = 3, [2] = "ubfxDMvw",
  },
  { --1f
    shift = 5, mask = 3, [2] = "ubfxDMvw",
  },
}

local map_load = {
  shift = 21, mask = 9,
  {
    shift = 20, mask = 5,
    [0] = "strtDL", "ldrtDL", [4] = "strbtDL", [5] = "ldrbtDL",
  },
  _ = {
    shift = 20, mask = 5,
    [0] = "strDL", "ldrDL", [4] = "strbDL", [5] = "ldrbDL",
  }
}

local map_load1 = {
  shift = 4, mask = 1,
  [0] = map_load, map_media,
}

local map_loadm = {
  shift = 20, mask = 1,
  [0] = {
    shift = 23, mask = 3,
    [0] = "stmdaNR", "stmNR",
    { shift = 16, mask = 63, [45] = "pushR", _ = "stmdbNR", }, "stmibNR",
  },
  {
    shift = 23, mask = 3,
    [0] = "ldmdaNR", { shift = 16, mask = 63, [61] = "popR", _ = "ldmNR", },
    "ldmdbNR", "ldmibNR",
  },
}

local map_data = {
  shift = 21, mask = 15,
  [0] = "andDNPs", "eorDNPs", "subDNPs", "rsbDNPs",
  "addDNPs", "adcDNPs", "sbcDNPs", "rscDNPs",
  "tstNP", "teqNP", "cmpNP", "cmnNP",
  "orrDNPs", "movDPs", "bicDNPs", "mvnDPs",
}

local map_mul = {
  shift = 21, mask = 7,
  [0] = "mulNMSs", "mlaNMSDs", "umaalDNMS", "mlsDNMS",
  "umullDNMSs", "umlalDNMSs", "smullDNMSs", "smlalDNMSs",
}

local map_sync = {
  shift = 20, mask = 15, -- NYI: brackets around N. R(D+1) for ldrexd/strexd.
  [0] = "swpDMN", false, false, false,
  "swpbDMN", false, false, false,
  "strexDMN", "ldrexDN", "strexdDN", "ldrexdDN",
  "strexbDMN", "ldrexbDN", "strexhDN", "ldrexhDN",
}

local map_mulh = {
  shift = 21, mask = 3,
  [0] = { shift = 5, mask = 3,
    [0] = "smlabbNMSD", "smlatbNMSD", "smlabtNMSD", "smlattNMSD", },
  { shift = 5, mask = 3,
    [0] = "smlawbNMSD", "smulwbNMS", "smlawtNMSD", "smulwtNMS", },
  { shift = 5, mask = 3,
    [0] = "smlalbbDNMS", "smlaltbDNMS", "smlalbtDNMS", "smlalttDNMS", },
  { shift = 5, mask = 3,
    [0] = "smulbbNMS", "smultbNMS", "smulbtNMS", "smulttNMS", },
```

```
}

local map_misc = {
  shift = 4, mask = 7,
  -- NYI: decode PSR bits of msr.
  [0] = { shift = 21, mask = 1, [0] = "mrsD", "msrM", },
  { shift = 21, mask = 3, "bxM", false, "clzDM", },
  { shift = 21, mask = 3, "bxjM", },
  { shift = 21, mask = 3, "blxM", },
  false,
  { shift = 21, mask = 3, [0] = "qaddDMN", "qsubDMN", "qdaddDMN", "qdsubDMN", },
  false,
  { shift = 21, mask = 3, "bkptK", },
}

local map_datar = {
  shift = 4, mask = 9,
  [9] = {
    shift = 5, mask = 3,
    [0] = { shift = 24, mask = 1, [0] = map_mul, map_sync, },
    { shift = 20, mask = 1, [0] = "strhDL", "ldrhDL", },
    { shift = 20, mask = 1, [0] = "ldrdDL", "ldrsbDL", },
    { shift = 20, mask = 1, [0] = "strdDL", "ldrshDL", },
  },
  _ = {
    shift = 20, mask = 25,
    [16] = { shift = 7, mask = 1, [0] = map_misc, map_mulh, },
    _ = {
      shift = 0, mask = 0xffffffff,
      [bor(0xe1a00000)] = "nop",
      _ = map_data,
    }
  },
}

local map_datai = {
  shift = 20, mask = 31, -- NYI: decode PSR bits of msr. Decode imm12.
  [16] = "movwDW", [20] = "movtDW",
  [18] = { shift = 0, mask = 0xf00ff, [0] = "nopv6", _ = "msrNW", },
  [22] = "msrNW",
  _ = map_data,
}

local map_branch = {
  shift = 24, mask = 1,
  [0] = "bB", "blB"
}

local map_condins = {
  [0] = map_datar, map_datai, map_load, map_load1,
  map_loadm, map_branch, map_loadc, map_datac
}

-- NYI: setend.
local map_uncondins = {
  [0] = false, map_simddata, map_simdload, map_preload,
  false, "blxB", map_loadcu, map_datacu,
}

------------------------------------------------------------------------------

local map_gpr = {
  [0] = "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7",
  "r8", "r9", "r10", "r11", "r12", "sp", "lr", "pc",
}

local map_cond = {
  [0] = "eq", "ne", "hs", "lo", "mi", "pl", "vs", "vc",
  "hi", "ls", "ge", "lt", "gt", "le", "al",
}

local map_shift = { [0] = "lsl", "lsr", "asr", "ror", }

------------------------------------------------------------------------------

-- Output a nicely formatted line with an opcode and operands.
```

```
local function putop(ctx, text, operands)
  local pos = ctx.pos
  local extra = ""
  if ctx.rel then
    local sym = ctx.symtab[ctx.rel]
    if sym then
      extra = "\t->"..sym
    elseif band(ctx.op, 0x0e000000) ~= 0x0a000000 then
      extra = "\t; 0x"..tohex(ctx.rel)
    end
  end
  if ctx.hexdump > 0 then
    ctx.out(format("%08x  %s  %-5s %s%s\n",
            ctx.addr+pos, tohex(ctx.op), text, concat(operands, ", "), extra))
  else
    ctx.out(format("%08x  %-5s %s%s\n",
            ctx.addr+pos, text, concat(operands, ", "), extra))
  end
  ctx.pos = pos + 4
end

-- Fallback for unknown opcodes.
local function unknown(ctx)
  return putop(ctx, ".long", { "0x"..tohex(ctx.op) })
end

-- Format operand 2 of load/store opcodes.
local function fmtload(ctx, op, pos)
  local base = map_gpr[band(rshift(op, 16), 15)]
  local x, ofs
  local ext = (band(op, 0x04000000) == 0)
  if not ext and band(op, 0x02000000) == 0 then
    ofs = band(op, 4095)
    if band(op, 0x00800000) == 0 then ofs = -ofs end
    if base == "pc" then ctx.rel = ctx.addr + pos + 8 + ofs end
    ofs = "#"..ofs
  elseif ext and band(op, 0x00400000) ~= 0 then
    ofs = band(op, 15) + band(rshift(op, 4), 0xf0)
    if band(op, 0x00800000) == 0 then ofs = -ofs end
    if base == "pc" then ctx.rel = ctx.addr + pos + 8 + ofs end
    ofs = "#"..ofs
  else
    ofs = map_gpr[band(op, 15)]
    if ext or band(op, 0xfe0) == 0 then
    elseif band(op, 0xfe0) == 0x60 then
      ofs = format("%s, rrx", ofs)
    else
      local sh = band(rshift(op, 7), 31)
      if sh == 0 then sh = 32 end
      ofs = format("%s, %s #%d", ofs, map_shift[band(rshift(op, 5), 3)], sh)
    end
    if band(op, 0x00800000) == 0 then ofs = "-"..ofs end
  end
  if ofs == "#0" then
    x = format("[%s]", base)
  elseif band(op, 0x01000000) == 0 then
    x = format("[%s], %s", base, ofs)
  else
    x = format("[%s, %s]", base, ofs)
  end
  if band(op, 0x01200000) == 0x01200000 then x = x.."!" end
  return x
end

-- Format operand 2 of vector load/store opcodes.
local function fmtvload(ctx, op, pos)
  local base = map_gpr[band(rshift(op, 16), 15)]
  local ofs = band(op, 255)*4
  if band(op, 0x00800000) == 0 then ofs = -ofs end
  if base == "pc" then ctx.rel = ctx.addr + pos + 8 + ofs end
  if ofs == 0 then
    return format("[%s]", base)
  else
    return format("[%s, #%d]", base, ofs)
  end
end
```

```
499
500 local function fmtvr(op, vr, sh0, sh1)
501   if vr == "s" then
502     return format("s%d", 2*band(rshift(op, sh0), 15)+band(rshift(op, sh1), 1))
503   else
504     return format("d%d", band(rshift(op, sh0), 15)+band(rshift(op, sh1-4), 16))
505   end
506 end
507
508 -- Disassemble a single instruction.
509 local function disass_ins(ctx)
510   local pos = ctx.pos
511   local b0, b1, b2, b3 = byte(ctx.code, pos+1, pos+4)
512   local op = bor(lshift(b3, 24), lshift(b2, 16), lshift(b1, 8), b0)
513   local operands = {}
514   local suffix = ""
515   local last, name, pat
516   local vr
517   ctx.op = op
518   ctx.rel = nil
519
520   local cond = rshift(op, 28)
521   local opat
522   if cond == 15 then
523     opat = map_uncondins[band(rshift(op, 25), 7)]
524   else
525     if cond ~= 14 then suffix = map_cond[cond] end
526     opat = map_condins[band(rshift(op, 25), 7)]
527   end
528   while type(opat) ~= "string" do
529     if not opat then return unknown(ctx) end
530     opat = opat[band(rshift(op, opat.shift), opat.mask)] or opat._
531   end
532   name, pat = match(opat, "^([a-z0-9]*)(.*)")
533   if sub(pat, 1, 1) == "." then
534     local s2, p2 = match(pat, "^([a-z0-9.]*)(.*)")
535     suffix = suffix..s2
536     pat = p2
537   end
538
539   for p in gmatch(pat, ".") do
540     local x = nil
541     if p == "D" then
542       x = map_gpr[band(rshift(op, 12), 15)]
543     elseif p == "N" then
544       x = map_gpr[band(rshift(op, 16), 15)]
545     elseif p == "S" then
546       x = map_gpr[band(rshift(op, 8), 15)]
547     elseif p == "M" then
548       x = map_gpr[band(op, 15)]
549     elseif p == "d" then
550       x = fmtvr(op, vr, 12, 22)
551     elseif p == "n" then
552       x = fmtvr(op, vr, 16, 7)
553     elseif p == "m" then
554       x = fmtvr(op, vr, 0, 5)
555     elseif p == "P" then
556       if band(op, 0x02000000) ~= 0 then
557         x = ror(band(op, 255), 2*band(rshift(op, 8), 15))
558       else
559         x = map_gpr[band(op, 15)]
560         if band(op, 0xff0) ~= 0 then
561           operands[#operands+1] = x
562           local s = map_shift[band(rshift(op, 5), 3)]
563           local r = nil
564           if band(op, 0xf90) == 0 then
565             if s == "ror" then s = "rrx" else r = "#32" end
566           elseif band(op, 0x10) == 0 then
567             r = "#"..band(rshift(op, 7), 31)
568           else
569             r = map_gpr[band(rshift(op, 8), 15)]
570           end
571           if name == "mov" then name = s; x = r
572           elseif r then x = format("%s %s", s, r)
573           else x = s end
574         end
```

```
      end
    elseif p == "L" then
      x = fmtload(ctx, op, pos)
    elseif p == "l" then
      x = fmtvload(ctx, op, pos)
    elseif p == "B" then
      local addr = ctx.addr + pos + 8 + arshift(lshift(op, 8), 6)
      if cond == 15 then addr = addr + band(rshift(op, 23), 2) end
      ctx.rel = addr
      x = "0x"..tohex(addr)
    elseif p == "F" then
      vr = "s"
    elseif p == "G" then
      vr = "d"
    elseif p == "." then
      suffix = suffix..(vr == "s" and ".f32" or ".f64")
    elseif p == "R" then
      if band(op, 0x00200000) ~= 0 and #operands == 1 then
        operands[1] = operands[1].."!"
      end
      local t = {}
      for i=0,15 do
        if band(rshift(op, i), 1) == 1 then t[#t+1] = map_gpr[i] end
      end
      x = "{"..concat(t, ", ").."}"
    elseif p == "r" then
      if band(op, 0x00200000) ~= 0 and #operands == 2 then
        operands[1] = operands[1].."!"
      end
      local s = tonumber(sub(last, 2))
      local n = band(op, 255)
      if vr == "d" then n = rshift(n, 1) end
      operands[#operands] = format("{%s-%s%d}", last, vr, s+n-1)
    elseif p == "W" then
      x = band(op, 0x0fff) + band(rshift(op, 4), 0xf000)
    elseif p == "T" then
      x = "#0x"..tohex(band(op, 0x00ffffff), 6)
    elseif p == "U" then
      x = band(rshift(op, 7), 31)
      if x == 0 then x = nil end
    elseif p == "u" then
      x = band(rshift(op, 7), 31)
      if band(op, 0x40) == 0 then
        if x == 0 then x = nil else x = "lsl #"..x end
      else
        if x == 0 then x = "asr #32" else x = "asr #"..x end
      end
    elseif p == "v" then
      x = band(rshift(op, 7), 31)
    elseif p == "w" then
      x = band(rshift(op, 16), 31)
    elseif p == "x" then
      x = band(rshift(op, 16), 31) + 1
    elseif p == "X" then
      x = band(rshift(op, 16), 31) - last + 1
    elseif p == "Y" then
      x = band(rshift(op, 12), 0xf0) + band(op, 0x0f)
    elseif p == "K" then
      x = "#0x"..tohex(band(rshift(op, 4), 0x0000fff0) + band(op, 15), 4)
    elseif p == "s" then
      if band(op, 0x00100000) ~= 0 then suffix = "s"..suffix end
    else
      assert(false)
    end
    if x then
      last = x
      if type(x) == "number" then x = "#"..x end
      operands[#operands+1] = x
    end
  end

  return putop(ctx, name..suffix, operands)
end

------------------------------------------------------------------------------

```

```
651 -- Disassemble a block of code.
652 local function disass_block(ctx, ofs, len)
653   if not ofs then ofs = 0 end
654   local stop = len and ofs+len or #ctx.code
655   ctx.pos = ofs
656   ctx.rel = nil
657   while ctx.pos < stop do disass_ins(ctx) end
658 end
659
660 -- Extended API: create a disassembler context. Then call ctx:disass(ofs, len).
661 local function create(code, addr, out)
662   local ctx = {}
663   ctx.code = code
664   ctx.addr = addr or 0
665   ctx.out = out or io.write
666   ctx.symtab = {}
667   ctx.disass = disass_block
668   ctx.hexdump = 8
669   return ctx
670 end
671
672 -- Simple API: disassemble code (a string) at address and output via out.
673 local function disass(code, addr, out)
674   create(code, addr, out):disass()
675 end
676
677 -- Return register name for RID.
678 local function regname(r)
679   if r < 16 then return map_gpr[r] end
680   return "d"..(r-16)
681 end
682
683 -- Public module functions.
684 return {
685   create = create,
686   disass = disass,
687   regname = regname
688 }
689
```

One Level Up Top Level

src/jit/dis_mipsel.lua - luajit-2.0-src

```
1  ----------------------------------------------------------------------------
2  -- LuaJIT MIPSEL disassembler wrapper module.
3  --
4  -- Copyright (C) 2005-2015 Mike Pall. All rights reserved.
5  -- Released under the MIT license. See Copyright Notice in luajit.h
6  ----------------------------------------------------------------------------
7  -- This module just exports the little-endian functions from the
8  -- MIPS disassembler module. All the interesting stuff is there.
9  ----------------------------------------------------------------------------
10
11 local dis_mips = require((string.match(..., ".*%.") or "")..."dis_mips")
12 return {
13   create = dis_mips.create_el,
14   disass = dis_mips.disass_el,
15   regname = dis_mips.regname
16 }
17
```

src/jit/dis_ppc.lua - luajit-2.0-src

Functions defined

Source code

```
1  ------------------------------------------------------------------------------
2  -- LuaJIT PPC disassembler module.
3  --
4  -- Copyright (C) 2005-2015 Mike Pall. All rights reserved.
5  -- Released under the MIT/X license. See Copyright Notice in luajit.h
6  ------------------------------------------------------------------------------
7  -- This is a helper module used by the LuaJIT machine code dumper module.
8  --
9  -- It disassembles all common, non-privileged 32/64 bit PowerPC instructions
10 -- plus the e500 SPE instructions and some Cell/Xenon extensions.
11 --
12 -- NYI: VMX, VMX128
13 ------------------------------------------------------------------------------
14
15 local type = type
16 local sub, byte, format = string.sub, string.byte, string.format
17 local match, gmatch, gsub = string.match, string.gmatch, string.gsub
18 local concat = table.concat
19 local bit = require("bit")
20 local band, bor, tohex = bit.band, bit.bor, bit.tohex
21 local lshift, rshift, arshift = bit.lshift, bit.rshift, bit.arshift
22
23 ------------------------------------------------------------------------------
24 -- Primary and extended opcode maps
25 ------------------------------------------------------------------------------
26
27 local map_crops = {
28   shift = 1, mask = 1023,
29   [0] = "mcrfXX",
30   [33] = "crnor|crnotCCC=", [129] = "crandcCCC",
31   [193] = "crxor|crclrCCC%", [225] = "crnandCCC",
32   [257] = "crandCCC", [289] = "creqv|crsetCCC%",
33   [417] = "crorcCCC", [449] = "cror|crmoveCCC=",
34   [16] = "b_lrKB", [528] = "b_ctrKB",
35   [150] = "isync",
36 }
37
38 local map_rlwinm = setmetatable({
39   shift = 0, mask = -1,
40 },
41 { __index = function(t, x)
42     local rot = band(rshift(x, 11), 31)
```

```
    local mb = band(rshift(x, 6), 31)
    local me = band(rshift(x, 1), 31)
    if mb == 0 and me == 31-rot then
      return "slwiRR~A."
    elseif me == 31 and mb == 32-rot then
      return "srwiRR~-A."
    else
      return "rlwinmRR~AAA."
    end
  end
})

local map_rld = {
  shift = 2, mask = 7,
  [0] = "rldiclRR~HM.", "rldicrRR~HM.", "rldicRR~HM.", "rldimiRR~HM.",
  {
    shift = 1, mask = 1,
    [0] = "rldclRR~RM.", "rldcrRR~RM.",
  },
}

local map_ext = setmetatable({
  shift = 1, mask = 1023,

  [0] = "cmp_YLRR", [32] = "cmpl_YLRR",
  [4] = "twARR", [68] = "tdARR",

  [8] = "subfcRRR.", [40] = "subfRRR.",
  [104] = "negRR.", [136] = "subfeRRR.",
  [200] = "subfzeRR.", [232] = "subfmeRR.",
  [520] = "subfcoRRR.", [552] = "subfoRRR.",
  [616] = "negoRR.", [648] = "subfeoRRR.",
  [712] = "subfzeoRR.", [744] = "subfmeoRR.",

  [9] = "mulhduRRR.", [73] = "mulhdRRR.", [233] = "mulldRRR.",
  [457] = "divduRRR.", [489] = "divdRRR.",
  [745] = "mulldoRRR.",
  [969] = "divduoRRR.", [1001] = "divdoRRR.",

  [10] = "addcRRR.", [138] = "addeRRR.",
  [202] = "addzeRR.", [234] = "addmeRR.", [266] = "addRRR.",
  [522] = "addcoRRR.", [650] = "addeoRRR.",
  [714] = "addzeoRR.", [746] = "addmeoRR.", [778] = "addoRRR.",

  [11] = "mulhwuRRR.", [75] = "mulhwRRR.", [235] = "mullwRRR.",
  [459] = "divwuRRR.", [491] = "divwRRR.",
  [747] = "mullwoRRR.",
  [971] = "divwouRRR.", [1003] = "divwoRRR.",

  [15] = "iselltRRR", [47] = "iselgtRRR", [79] = "iseleqRRR",

  [144] = { shift = 20, mask = 1, [0] = "mtcrfRZ~", "mtocrfRZ~", },
  [19] = { shift = 20, mask = 1, [0] = "mfcrR", "mfocrfRZ", },
  [371] = { shift = 11, mask = 1023, [392] = "mftbR", [424] = "mftbuR", },
  [339] = {
    shift = 11, mask = 1023,
    [32] = "mferR", [256] = "mflrR", [288] = "mfctrR", [16] = "mfspefscrR",
  },
  [467] = {
    shift = 11, mask = 1023,
    [32] = "mtxerR", [256] = "mtlrR", [288] = "mtctrR", [16] = "mtspefscrR",
  },

  [20] = "lwarxRR0R", [84] = "ldarxRR0R",

  [21] = "ldxRR0R", [53] = "lduxRRR",
  [149] = "stdxRR0R", [181] = "stduxRRR",
  [341] = "lwaxRR0R", [373] = "lwauxRRR",

  [23] = "lwzxRR0R", [55] = "lwzuxRRR",
  [87] = "lbzxRR0R", [119] = "lbzuxRRR",
  [151] = "stwxRR0R", [183] = "stwuxRRR",
  [215] = "stbxRR0R", [247] = "stbuxRRR",
  [279] = "lhzxRR0R", [311] = "lhzuxRRR",
  [343] = "lhaxRR0R", [375] = "lhauxRRR",
  [407] = "sthxRR0R", [439] = "sthuxRRR",
```

```
  [54] = "dcbst-R0R", [86] = "dcbf-R0R",
  [150] = "stwcxRR0R.", [214] = "stdcxRR0R.",
  [246] = "dcbtst-R0R", [278] = "dcbt-R0R",
  [310] = "eciwxRR0R", [438] = "ecowxRR0R",
  [470] = "dcbi-RR",

  [598] = {
    shift = 21, mask = 3,
    [0] = "sync", "lwsync", "ptesync",
  },
  [758] = "dcba-RR",
  [854] = "eieio", [982] = "icbi-R0R", [1014] = "dcbz-R0R",

  [26] = "cntlzwRR~", [58] = "cntlzdRR~",
  [122] = "popcntbRR~",
  [154] = "prtywRR~", [186] = "prtydRR~",

  [28] = "andRR~R.", [60] = "andcRR~R.", [124] = "nor|notRR~R=.",
  [284] = "eqvRR~R.", [316] = "xorRR~R.",
  [412] = "orcRR~R.", [444] = "or|mrRR~R=.", [476] = "nandRR~R.",
  [508] = "cmpbRR~R",

  [512] = "mcrxrX",

  [532] = "ldbrxRR0R", [660] = "stdbrxRR0R",

  [533] = "lswxRR0R", [597] = "lswiRR0A",
  [661] = "stswxRR0R", [725] = "stswiRR0A",

  [534] = "lwbrxRR0R", [662] = "stwbrxRR0R",
  [790] = "lhbrxRR0R", [918] = "sthbrxRR0R",

  [535] = "lfsxFR0R", [567] = "lfsuxFRR",
  [599] = "lfdxFR0R", [631] = "lfduxFRR",
  [663] = "stfsxFR0R", [695] = "stfsuxFRR",
  [727] = "stfdxFR0R", [759] = "stfduxFR0R",
  [855] = "lfiwaxFR0R",
  [983] = "stfiwxFR0R",

  [24] = "slwRR~R.",

  [27] = "sldRR~R.", [536] = "srwRR~R.",
  [792] = "srawRR~R.", [824] = "srawiRR~A.",

  [794] = "sradRR~R.", [826] = "sradiRR~H.", [827] = "sradiRR~H.",
  [922] = "extshRR~.", [954] = "extsbRR~.", [986] = "extswRR~.",

  [539] = "srdRR~R.",
},
{ __index = function(t, x)
    if band(x, 31) == 15 then return "iselRRRC" end
  end
})

local map_ld = {
  shift = 0, mask = 3,
  [0] = "ldRRE", "lduRRE", "lwaRRE",
}

local map_std = {
  shift = 0, mask = 3,
  [0] = "stdRRE", "stduRRE",
}

local map_fps = {
  shift = 5, mask = 1,
  {
    shift = 1, mask = 15,
    [0] = false, false, "fdivsFFF.", false,
    "fsubsFFF.", "faddsFFF.", "fsqrtsF-F.", false,
    "fresF-F.", "fmulsFF-F.", "frsqrtesF-F.", false,
    "fmsubsFFFF~.", "fmaddsFFFF~.", "fnmsubsFFFF~.", "fnmaddsFFFF~.",
  }
}
```

```
local map_fpd = {
  shift = 5, mask = 1,
  [0] = {
    shift = 1, mask = 1023,
    [0] = "fcmpuXFF", [32] = "fcmpoXFF", [64] = "mcrfsXX",
    [38] = "mtfsb1A.", [70] = "mtfsb0A.", [134] = "mtfsfiA>>-A>",
    [8] = "fcpsgnFFF.", [40] = "fnegF-F.", [72] = "fmrF-F.",
    [136] = "fnabsF-F.", [264] = "fabsF-F.",
    [12] = "frspF-F.",
    [14] = "fctiwF-F.", [15] = "fctiwzF-F.",
    [583] = "mffsF.", [711] = "mtfsfZF.",
    [392] = "frinF-F.", [424] = "frizF-F.",
    [456] = "fripF-F.", [488] = "frimF-F.",
    [814] = "fctidF-F.", [815] = "fctidzF-F.", [846] = "fcfidF-F.",
  },
  {
    shift = 1, mask = 15,
    [0] = false, false, "fdivFFF.", false,
    "fsubFFF.", "faddFFF.", "fsqrtF-F.", "fselFFFF~.",
    "freF-F.", "fmulFF-F.", "frsqrteF-F.", false,
    "fmsubFFFF~.", "fmaddFFFF~.", "fnmsubFFFF~.", "fnmaddFFFF~.",
  }
}

local map_spe = {
  shift = 0, mask = 2047,

  [512] = "evaddwRRR", [514] = "evaddiwRAR~",
  [516] = "evsubwRRR~", [518] = "evsubiwRAR~",
  [520] = "evabsRR", [521] = "evnegRR",
  [522] = "evextsbRR", [523] = "evextshRR", [524] = "evrndwRR",
  [525] = "evcntlzwRR", [526] = "evcntlswRR",

  [527] = "brincRRR",

  [529] = "evandRRR", [530] = "evandcRRR", [534] = "evxorRRR",
  [535] = "evor|evmrRRR=", [536] = "evnor|evnotRRR=",
  [537] = "eveqvRRR", [539] = "evorcRRR", [542] = "evnandRRR",

  [544] = "evsrwuRRR", [545] = "evsrwsRRR",
  [546] = "evsrwiuRRA", [547] = "evsrwisRRA",
  [548] = "evslwRRR", [550] = "evslwiRRA",
  [552] = "evrlwRRR", [553] = "evsplatiRS",
  [554] = "evrlwiRRA", [555] = "evsplatfiRS",
  [556] = "evmergehiRRR", [557] = "evmergeloRRR",
  [558] = "evmergehiloRRR", [559] = "evmergelohiRRR",

  [560] = "evcmpgtuYRR", [561] = "evcmpgtsYRR",
  [562] = "evcmpltuYRR", [563] = "evcmpltsYRR",
  [564] = "evcmpeqYRR",

  [632] = "evselRRR", [633] = "evselRRRW",
  [634] = "evselRRRW", [635] = "evselRRRW",
  [636] = "evselRRRW", [637] = "evselRRRW",
  [638] = "evselRRRW", [639] = "evselRRRW",

  [640] = "evfsaddRRR", [641] = "evfssubRRR",
  [644] = "evfsabsRR", [645] = "evfsnabsRR", [646] = "evfsnegRR",
  [648] = "evfsmulRRR", [649] = "evfsdivRRR",
  [652] = "evfscmpgtYRR", [653] = "evfscmpltYRR", [654] = "evfscmpeqYRR",
  [656] = "evfscfuiR-R", [657] = "evfscfsiR-R",
  [658] = "evfscfufR-R", [659] = "evfscfsfR-R",
  [660] = "evfsctuiR-R", [661] = "evfsctsiR-R",
  [662] = "evfsctufR-R", [663] = "evfsctsfR-R",
  [664] = "evfsctuizR-R", [666] = "evfsctsizR-R",
  [668] = "evfststgtYRR", [669] = "evfststltYRR", [670] = "evfststeqYRR",

  [704] = "efsaddRRR", [705] = "efssubRRR",
  [708] = "efsabsRR", [709] = "efsnabsRR", [710] = "efsnegRR",
  [712] = "efsmulRRR", [713] = "efsdivRRR",
  [716] = "efscmpgtYRR", [717] = "efscmpltYRR", [718] = "efscmpeqYRR",
  [719] = "efscfdR-R",
  [720] = "efscfuiR-R", [721] = "efscfsiR-R",
  [722] = "efscfufR-R", [723] = "efscfsfR-R",
  [724] = "efsctuiR-R", [725] = "efsctsiR-R",
  [726] = "efsctufR-R", [727] = "efsctsfR-R",
```

```
    [728] = "efsctuizR-R", [730] = "efsctsizR-R",
    [732] = "efststgtYRR", [733] = "efststltYRR", [734] = "efststeqYRR",

    [736] = "efdaddRRR", [737] = "efdsubRRR",
    [738] = "efdcfuidR-R", [739] = "efdcfsidR-R",
    [740] = "efdabsRR", [741] = "efdnabsRR", [742] = "efdnegRR",
    [744] = "efdmulRRR", [745] = "efddivRRR",
    [746] = "efdctuidzR-R", [747] = "efdctsidzR-R",
    [748] = "efdcmpgtYRR", [749] = "efdcmpltYRR", [750] = "efdcmpeqYRR",
    [751] = "efdcfsR-R",
    [752] = "efdcfuiR-R", [753] = "efdcfsiR-R",
    [754] = "efdcfufR-R", [755] = "efdcfsfR-R",
    [756] = "efdctuiR-R", [757] = "efdctsiR-R",
    [758] = "efdctufR-R", [759] = "efdctsfR-R",
    [760] = "efdctuizR-R", [762] = "efdctsizR-R",
    [764] = "efdtstgtYRR", [765] = "efdtstltYRR", [766] = "efdtsteqYRR",

    [768] = "evlddxRR0R", [769] = "evlddRR8",
    [770] = "evldwxRR0R", [771] = "evldwRR8",
    [772] = "evldhxRR0R", [773] = "evldhRR8",
    [776] = "evlhhesplatxRR0R", [777] = "evlhhesplatRR2",
    [780] = "evlhhousplatxRR0R", [781] = "evlhhousplatRR2",
    [782] = "evlhhossplatxRR0R", [783] = "evlhhossplatRR2",
    [784] = "evlwhexRR0R", [785] = "evlwheRR4",
    [788] = "evlwhouxRR0R", [789] = "evlwhouRR4",
    [790] = "evlwhosxRR0R", [791] = "evlwhosRR4",
    [792] = "evlwwsplatxRR0R", [793] = "evlwwsplatRR4",
    [796] = "evlwhsplatxRR0R", [797] = "evlwhsplatRR4",

    [800] = "evstddxRR0R", [801] = "evstddRR8",
    [802] = "evstdwxRR0R", [803] = "evstdwRR8",
    [804] = "evstdhxRR0R", [805] = "evstdhRR8",
    [816] = "evstwhexRR0R", [817] = "evstwheRR4",
    [820] = "evstwhoxRR0R", [821] = "evstwhoRR4",
    [824] = "evstwwexRR0R", [825] = "evstwweRR4",
    [828] = "evstwwoxRR0R", [829] = "evstwwoRR4",

    [1027] = "evmhessfRRR", [1031] = "evmhossfRRR", [1032] = "evmheumiRRR",
    [1033] = "evmhesmiRRR", [1035] = "evmhesmfRRR", [1036] = "evmhoumiRRR",
    [1037] = "evmhosmiRRR", [1039] = "evmhosmfRRR", [1059] = "evmhessfaRRR",
    [1063] = "evmhossfaRRR", [1064] = "evmheumiaRRR", [1065] = "evmhesmiaRRR",
    [1067] = "evmhesmfaRRR", [1068] = "evmhoumiaRRR", [1069] = "evmhosmiaRRR",
    [1071] = "evmhosmfaRRR", [1095] = "evmwhssfRRR", [1096] = "evmwlumiRRR",
    [1100] = "evmwhumiRRR", [1101] = "evmwhsmiRRR", [1103] = "evmwhsmfRRR",
    [1107] = "evmwssfRRR", [1112] = "evmwumiRRR", [1113] = "evmwsmiRRR",
    [1115] = "evmwsmfRRR", [1127] = "evmwhssfaRRR", [1128] = "evmwlumiaRRR",
    [1132] = "evmwhumiaRRR", [1133] = "evmwhsmiaRRR", [1135] = "evmwhsmfaRRR",
    [1139] = "evmwssfaRRR", [1144] = "evmwumiaRRR", [1145] = "evmwsmiaRRR",
    [1147] = "evmwsmfaRRR",

    [1216] = "evaddusiaawRR", [1217] = "evaddssiaawRR",
    [1218] = "evsubfusiaawRR", [1219] = "evsubfssiaawRR",
    [1220] = "evmraRR",
    [1222] = "evdivwsRRR", [1223] = "evdivwuRRR",
    [1224] = "evaddumiaawRR", [1225] = "evaddsmiaawRR",
    [1226] = "evsubfumiaawRR", [1227] = "evsubfsmiaawRR",

    [1280] = "evmheusiaawRRR", [1281] = "evmhessiaawRRR",
    [1283] = "evmhessfaawRRR", [1284] = "evmhousiaawRRR",
    [1285] = "evmhossiaawRRR", [1287] = "evmhossfaawRRR",
    [1288] = "evmheumiaawRRR", [1289] = "evmhesmiaawRRR",
    [1291] = "evmhesmfaawRRR", [1292] = "evmhoumiaawRRR",
    [1293] = "evmhosmiaawRRR", [1295] = "evmhosmfaawRRR",
    [1320] = "evmhegumiaaRRR", [1321] = "evmhegsmiaaRRR",
    [1323] = "evmhegsmfaaRRR", [1324] = "evmhogumiaaRRR",
    [1325] = "evmhogsmiaaRRR", [1327] = "evmhogsmfaaRRR",
    [1344] = "evmwlusiaawRRR", [1345] = "evmwlssiaawRRR",
    [1352] = "evmwlumiaawRRR", [1353] = "evmwlsmiaawRRR",
    [1363] = "evmwssfaaRRR", [1368] = "evmwumiaaRRR",
    [1369] = "evmwsmiaaRRR", [1371] = "evmwsmfaaRRR",
    [1408] = "evmheusianwRRR", [1409] = "evmhessianwRRR",
    [1411] = "evmhessfanwRRR", [1412] = "evmhousianwRRR",
    [1413] = "evmhossianwRRR", [1415] = "evmhossfanwRRR",
    [1416] = "evmheumianwRRR", [1417] = "evmhesmianwRRR",
    [1419] = "evmhesmfanwRRR", [1420] = "evmhoumianwRRR",
    [1421] = "evmhosmianwRRR", [1423] = "evmhosmfanwRRR",
```

```
  [1448] = "evmhegumianRRR", [1449] = "evmhegsmianRRR",
  [1451] = "evmhegsmfanRRR", [1452] = "evmhogumianRRR",
  [1453] = "evmhogsmianRRR", [1455] = "evmhogsmfanRRR",
  [1472] = "evmwlusianwRRR", [1473] = "evmwlssianwRRR",
  [1480] = "evmwlumianwRRR", [1481] = "evmwlsmianwRRR",
  [1491] = "evmwssfanRRR", [1496] = "evmwumianRRR",
  [1497] = "evmwsmianRRR", [1499] = "evmwsmfanRRR",
}

local map_pri = {
  [0] = false,        false,               "tdiARI",        "twiARI",
  map_spe,        false,               false,               "mulliRRI",
  "subficRRI",        false,               "cmpl_iYLRU",        "cmp_iYLRI",
  "addicRRI",        "addic.RRI",        "addi|liRR0I",        "addis|lisRR0I",
  "b_KBJ",        "sc",               "bKJ",               map_crops,
  "rlwimiRR~AAA.", map_rlwinm,        false,               "rlwnmRR~RAA.",
  "oriNRR~U",        "orisRR~U",        "xoriRR~U",        "xorisRR~U",
  "andi.RR~U",        "andis.RR~U",        map_rld,        map_ext,
  "lwzRRD",        "lwzuRRD",        "lbzRRD",        "lbzuRRD",
  "stwRRD",        "stwuRRD",        "stbRRD",        "stbuRRD",
  "lhzRRD",        "lhzuRRD",        "lhaRRD",        "lhauRRD",
  "sthRRD",        "sthuRRD",        "lmwRRD",        "stmwRRD",
  "lfsFRD",        "lfsuFRD",        "lfdFRD",        "lfduFRD",
  "stfsFRD",        "stfsuFRD",        "stfdFRD",        "stfduFRD",
  false,        false,               map_ld,               map_fps,
  false,        false,               map_std,        map_fpd,
}

------------------------------------------------------------------------------

local map_gpr = {
  [0] = "r0", "sp", "r2", "r3", "r4", "r5", "r6", "r7",
  "r8", "r9", "r10", "r11", "r12", "r13", "r14", "r15",
  "r16", "r17", "r18", "r19", "r20", "r21", "r22", "r23",
  "r24", "r25", "r26", "r27", "r28", "r29", "r30", "r31",
}

local map_cond = { [0] = "lt", "gt", "eq", "so", "ge", "le", "ne", "ns", }

-- Format a condition bit.
local function condfmt(cond)
  if cond <= 3 then
    return map_cond[band(cond, 3)]
  else
    return format("4*cr%d+%s", rshift(cond, 2), map_cond[band(cond, 3)])
  end
end

------------------------------------------------------------------------------

-- Output a nicely formatted line with an opcode and operands.
local function putop(ctx, text, operands)
  local pos = ctx.pos
  local extra = ""
  if ctx.rel then
    local sym = ctx.symtab[ctx.rel]
    if sym then extra = "\t->"..sym end
  end
  if ctx.hexdump > 0 then
    ctx.out(format("%08x  %s  %-7s %s%s\n",
            ctx.addr+pos, tohex(ctx.op), text, concat(operands, ", "), extra))
  else
    ctx.out(format("%08x  %-7s %s%s\n",
            ctx.addr+pos, text, concat(operands, ", "), extra))
  end
  ctx.pos = pos + 4
end

-- Fallback for unknown opcodes.
local function unknown(ctx)
  return putop(ctx, ".long", { "0x"..tohex(ctx.op) })
end

-- Disassemble a single instruction.
local function disass_ins(ctx)
  local pos = ctx.pos
```

```
  local b0, b1, b2, b3 = byte(ctx.code, pos+1, pos+4)
  local op = bor(lshift(b0, 24), lshift(b1, 16), lshift(b2, 8), b3)
  local operands = {}
  local last = nil
  local rs = 21
  ctx.op = op
  ctx.rel = nil

  local opat = map_pri[rshift(b0, 2)]
  while type(opat) ~= "string" do
    if not opat then return unknown(ctx) end
    opat = opat[band(rshift(op, opat.shift), opat.mask)]
  end
  local name, pat = match(opat, "^([a-z0-9_.]*)(.*)")
  local altname, pat2 = match(pat, "|([a-z0-9_.]*)(.*)")
  if altname then pat = pat2 end

  for p in gmatch(pat, ".") do
    local x = nil
    if p == "R" then
      x = map_gpr[band(rshift(op, rs), 31)]
      rs = rs - 5
    elseif p == "F" then
      x = "f"..band(rshift(op, rs), 31)
      rs = rs - 5
    elseif p == "A" then
      x = band(rshift(op, rs), 31)
      rs = rs - 5
    elseif p == "S" then
      x = arshift(lshift(op, 27-rs), 27)
      rs = rs - 5
    elseif p == "I" then
      x = arshift(lshift(op, 16), 16)
    elseif p == "U" then
      x = band(op, 0xffff)
    elseif p == "D" or p == "E" then
      local disp = arshift(lshift(op, 16), 16)
      if p == "E" then disp = band(disp, -4) end
      if last == "r0" then last = "0" end
      operands[#operands] = format("%d(%s)", disp, last)
    elseif p >= "2" and p <= "8" then
      local disp = band(rshift(op, rs), 31) * p
      if last == "r0" then last = "0" end
      operands[#operands] = format("%d(%s)", disp, last)
    elseif p == "H" then
      x = band(rshift(op, rs), 31) + lshift(band(op, 2), 4)
      rs = rs - 5
    elseif p == "M" then
      x = band(rshift(op, rs), 31) + band(op, 0x20)
    elseif p == "C" then
      x = condfmt(band(rshift(op, rs), 31))
      rs = rs - 5
    elseif p == "B" then
      local bo = rshift(op, 21)
      local cond = band(rshift(op, 16), 31)
      local cn = ""
      rs = rs - 10
      if band(bo, 4) == 0 then
        cn = band(bo, 2) == 0 and "dnz" or "dz"
        if band(bo, 0x10) == 0 then
          cn = cn..(band(bo, 8) == 0 and "f" or "t")
        end
        if band(bo, 0x10) == 0 then x = condfmt(cond) end
        name = name..(band(bo, 1) == band(rshift(op, 15), 1) and "-" or "+")
      elseif band(bo, 0x10) == 0 then
        cn = map_cond[band(cond, 3) + (band(bo, 8) == 0 and 4 or 0)]
        if cond > 3 then x = "cr"..rshift(cond, 2) end
        name = name..(band(bo, 1) == band(rshift(op, 15), 1) and "-" or "+")
      end
      name = gsub(name, "_", cn)
    elseif p == "J" then
      x = arshift(lshift(op, 27-rs), 29-rs)*4
      if band(op, 2) == 0 then x = ctx.addr + pos + x end
      ctx.rel = x
      x = "0x"..tohex(x)
    elseif p == "K" then
```

```
      if band(op, 1) ~= 0 then name = name.."l" end
      if band(op, 2) ~= 0 then name = name.."a" end
    elseif p == "X" or p == "Y" then
      x = band(rshift(op, rs+2), 7)
      if x == 0 and p == "Y" then x = nil else x = "cr"..x end
      rs = rs - 5
    elseif p == "W" then
      x = "cr"..band(op, 7)
    elseif p == "Z" then
      x = band(rshift(op, rs-4), 255)
      rs = rs - 10
    elseif p == ">" then
      operands[#operands] = rshift(operands[#operands], 1)
    elseif p == "0" then
      if last == "r0" then
        operands[#operands] = nil
        if altname then name = altname end
      end
    elseif p == "L" then
      name = gsub(name, "_", band(op, 0x00200000) ~= 0 and "d" or "w")
    elseif p == "." then
      if band(op, 1) == 1 then name = name.."." end
    elseif p == "N" then
      if op == 0x60000000 then name = "nop"; break end
    elseif p == "~" then
      local n = #operands
      operands[n-1],  operands[n] = operands[n], operands[n-1]
    elseif p == "=" then
      local n = #operands
      if last == operands[n-1] then
        operands[n] = nil
        name = altname
      end
    elseif p == "%" then
      local n = #operands
      if last == operands[n-1] and last == operands[n-2] then
        operands[n] = nil
        operands[n-1] = nil
        name = altname
      end
    elseif p == "-" then
      rs = rs - 5
    else
      assert(false)
    end
    if x then operands[#operands+1] = x; last = x end
  end

  return putop(ctx, name, operands)
end

------------------------------------------------------------------------------

-- Disassemble a block of code.
local function disass_block(ctx, ofs, len)
  if not ofs then ofs = 0 end
  local stop = len and ofs+len or #ctx.code
  stop = stop - stop % 4
  ctx.pos = ofs - ofs % 4
  ctx.rel = nil
  while ctx.pos < stop do disass_ins(ctx) end
end

-- Extended API: create a disassembler context. Then call ctx:disass(ofs, len).
local function create(code, addr, out)
  local ctx = {}
  ctx.code = code
  ctx.addr = addr or 0
  ctx.out = out or io.write
  ctx.symtab = {}
  ctx.disass = disass_block
  ctx.hexdump = 8
  return ctx
end

-- Simple API: disassemble code (a string) at address and output via out.
```

```
local function disass(code, addr, out)
  create(code, addr, out):disass()
end

-- Return register name for RID.
local function regname(r)
  if r < 32 then return map_gpr[r] end
  return "f"..(r-32)
end

-- Public module functions.
return {
  create = create,
  disass = disass,
  regname = regname
}

```

One Level Up Top Level

src/jit/dis_x64.lua - luajit-2.0-src

```
1  ------------------------------------------------------------------------------
2  -- LuaJIT x64 disassembler wrapper module.
3  --
4  -- Copyright (C) 2005-2015 Mike Pall. All rights reserved.
5  -- Released under the MIT license. See Copyright Notice in luajit.h
6  ------------------------------------------------------------------------------
7  -- This module just exports the 64 bit functions from the combined
8  -- x86/x64 disassembler module. All the interesting stuff is there.
9  ------------------------------------------------------------------------------
10
11 local dis_x86 = require((string.match(..., ".*%.") or "")..."dis_x86")
12 return {
13   create = dis_x86.create64,
14   disass = dis_x86.disass64,
15   regname = dis_x86.regname64
16 }
17
```

src/jit/dump.lua - luajit-2.0-src

Functions defined

Source code

```
----------------------------------------------------------------------------
-- LuaJIT compiler dump module.
--
-- Copyright (C) 2005-2015 Mike Pall. All rights reserved.
-- Released under the MIT license. See Copyright Notice in luajit.h
----------------------------------------------------------------------------
--
-- This module can be used to debug the JIT compiler itself. It dumps the
-- code representations and structures used in various compiler stages.
--
-- Example usage:
--
--   luajit -jdump -e "local x=0; for i=1,1e6 do x=x+i end; print(x)"
--   luajit -jdump=im -e "for i=1,1000 do for j=1,1000 do end end" | less -R
--   luajit -jdump=is myapp.lua | less -R
--   luajit -jdump=-b myapp.lua
--   luajit -jdump=+aH,myapp.html myapp.lua
--   luajit -jdump=ixT,myapp.dump myapp.lua
--
-- The first argument specifies the dump mode. The second argument gives
-- the output file name. Default output is to stdout, unless the environment
-- variable LUAJIT_DUMPFILE is set. The file is overwritten every time the
-- module is started.
--
-- Different features can be turned on or off with the dump mode. If the
-- mode starts with a '+', the following features are added to the default
-- set of features; a '-' removes them. Otherwise the features are replaced.
--
-- The following dump features are available (* marks the default):
--
--   * t  Print a line for each started, ended or aborted trace (see also -jv).
--   * b  Dump the traced bytecode.
--   * i  Dump the IR (intermediate representation).
--     r  Augment the IR with register/stack slots.
--     s  Dump the snapshot map.
--   * m  Dump the generated machine code.
--     x  Print each taken trace exit.
--     X  Print each taken trace exit and the contents of all registers.
--     a  Print the IR of aborted traces, too.
--
-- The output format can be set with the following characters:
--
--     T  Plain text output.
--     A  ANSI-colored text output
--     H  Colorized HTML + CSS output.
--
-- The default output format is plain text. It's set to ANSI-colored text
-- if the COLORTERM variable is set. Note: this is independent of any output
-- redirection, which is actually considered a feature.
--
-- You probably want to use less -R to enjoy viewing ANSI-colored text from
-- a pipe or a file. Add this to your ~/.bashrc: export LESS="-R"
--
----------------------------------------------------------------------------

-- Cache some library functions and objects.
local jit = require("jit")
assert(jit.version_num == 20100, "LuaJIT core/library version mismatch")
local jutil = require("jit.util")
local vmdef = require("jit.vmdef")
local funcinfo, funcbc = jutil.funcinfo, jutil.funcbc
local traceinfo, traceir, tracek = jutil.traceinfo, jutil.traceir, jutil.tracek
local tracemc, tracesnap = jutil.tracemc, jutil.tracesnap
local traceexitstub, ircalladdr = jutil.traceexitstub, jutil.ircalladdr
local bit = require("bit")
local band, shl, shr, tohex = bit.band, bit.lshift, bit.rshift, bit.tohex
local sub, gsub, format = string.sub, string.gsub, string.format
local byte, char, rep = string.byte, string.char, string.rep
local type, tostring = type, tostring
local stdout, stderr = io.stdout, io.stderr

-- Load other modules on-demand.
local bcline, disass

-- Active flag, output file handle and dump mode.
local active, out, dumpmode
```

```
77
78  ------------------------------------------------------------------------------
79
80  local symtabmt = { __index = false }
81  local symtab = {}
82  local nexitsym = 0
83
84  -- Fill nested symbol table with per-trace exit stub addresses.
85  local function fillsymtab_tr(tr, nexit)
86    local t = {}
87    symtabmt.__index = t
88    if jit.arch == "mips" or jit.arch == "mipsel" then
89      t[traceexitstub(tr, 0)] = "exit"
90      return
91    end
92    for i=0,nexit-1 do
93      local addr = traceexitstub(tr, i)
94      if addr < 0 then addr = addr + 2^32 end
95      t[addr] = tostring(i)
96    end
97    local addr = traceexitstub(tr, nexit)
98    if addr then t[addr] = "stack_check" end
99  end
100
101  -- Fill symbol table with trace exit stub addresses.
102  local function fillsymtab(tr, nexit)
103    local t = symtab
104    if nexitsym == 0 then
105      local ircall = vmdef.ircall
106      for i=0,#ircall do
107        local addr = ircalladdr(i)
108        if addr ~= 0 then
109          if addr < 0 then addr = addr + 2^32 end
110          t[addr] = ircall[i]
111        end
112      end
113    end
114    if nexitsym == 1000000 then -- Per-trace exit stubs.
115      fillsymtab_tr(tr, nexit)
116    elseif nexit > nexitsym then -- Shared exit stubs.
117      for i=nexitsym,nexit-1 do
118        local addr = traceexitstub(i)
119        if addr == nil then -- Fall back to per-trace exit stubs.
120          fillsymtab_tr(tr, nexit)
121          setmetatable(symtab, symtabmt)
122          nexit = 1000000
123          break
124        end
125        if addr < 0 then addr = addr + 2^32 end
126        t[addr] = tostring(i)
127      end
128      nexitsym = nexit
129    end
130    return t
131  end
132
133  local function dumpwrite(s)
134    out:write(s)
135  end
136
137  -- Disassemble machine code.
138  local function dump_mcode(tr)
139    local info = traceinfo(tr)
140    if not info then return end
141    local mcode, addr, loop = tracemc(tr)
142    if not mcode then return end
143    if not disass then disass = require("jit.dis_"..jit.arch) end
144    if addr < 0 then addr = addr + 2^32 end
145    out:write("---- TRACE ", tr, " mcode ", #mcode, "\n")
146    local ctx = disass.create(mcode, addr, dumpwrite)
147    ctx.hexdump = 0
148    ctx.symtab = fillsymtab(tr, info.nexit)
149    if loop ~= 0 then
150      symtab[addr+loop] = "LOOP"
151      ctx:disass(0, loop)
152      out:write("->LOOP:\n")
```

```
    ctx:disass(loop, #mcode-loop)
    symtab[addr+loop] = nil
  else
    ctx:disass(0, #mcode)
  end
end

--------------------------------------------------------------------------------

local irtype_text = {
  [0] = "nil",
  "fal",
  "tru",
  "lud",
  "str",
  "p32",
  "thr",
  "pro",
  "fun",
  "p64",
  "cdt",
  "tab",
  "udt",
  "flt",
  "num",
  "i8 ",
  "u8 ",
  "i16",
  "u16",
  "int",
  "u32",
  "i64",
  "u64",
  "sfp",
}

local colortype_ansi = {
  [0] = "%s",
  "%s",
  "%s",
  "\027[36m%s\027[m",
  "\027[32m%s\027[m",
  "%s",
  "\027[1m%s\027[m",
  "%s",
  "\027[1m%s\027[m",
  "%s",
  "\027[33m%s\027[m",
  "\027[31m%s\027[m",
  "\027[36m%s\027[m",
  "\027[34m%s\027[m",
  "\027[34m%s\027[m",
  "\027[35m%s\027[m",
  "\027[35m%s\027[m",
  "\027[35m%s\027[m",
  "\027[35m%s\027[m",
  "\027[35m%s\027[m",
  "\027[35m%s\027[m",
  "\027[35m%s\027[m",
  "\027[35m%s\027[m",
  "\027[35m%s\027[m",
}

local function colorize_text(s, t)
  return s
end

local function colorize_ansi(s, t)
  return format(colortype_ansi[t], s)
end

local irtype_ansi = setmetatable({},
  { __index = function(tab, t)
      local s = colorize_ansi(irtype_text[t], t); tab[t] = s; return s; end })

local html_escape = { ["<"] = "&lt;", [">"] = "&gt;", ["&"] = "&amp;", }
```

```
229
230 local function colorize_html(s, t)
231   s = gsub(s, "[<>&]", html_escape)
232   return format('<span class="irt_%s">%s</span>', irtype_text[t], s)
233 end
234
235 local irtype_html = setmetatable({},
236   { __index = function(tab, t)
237       local s = colorize_html(irtype_text[t], t); tab[t] = s; return s; end })
238
239 local header_html = [[
240 <style type="text/css">
241 background { background: #ffffff; color: #000000; }
242 pre.ljdump {
243 font-size: 10pt;
244 background: #f0f4ff;
245 color: #000000;
246 border: 1px solid #bfcfff;
247 padding: 0.5em;
248 margin-left: 2em;
249 margin-right: 2em;
250 }
251 span.irt_str { color: #00a000; }
252 span.irt_thr, span.irt_fun { color: #404040; font-weight: bold; }
253 span.irt_tab { color: #c00000; }
254 span.irt_udt, span.irt_lud { color: #00c0c0; }
255 span.irt_num { color: #4040c0; }
256 span.irt_int, span.irt_i8, span.irt_u8, span.irt_i16, span.irt_u16 { color: #b040b0; }
257 </style>
258 ]]
259
260 local colorize, irtype
261
262 -- Lookup tables to convert some literals into names.
263 local litname = {
264   ["SLOAD "] = setmetatable({}, { __index = function(t, mode)
265     local s = ""
266     if band(mode, 1) ~= 0 then s = s.."P" end
267     if band(mode, 2) ~= 0 then s = s.."F" end
268     if band(mode, 4) ~= 0 then s = s.."T" end
269     if band(mode, 8) ~= 0 then s = s.."C" end
270     if band(mode, 16) ~= 0 then s = s.."R" end
271     if band(mode, 32) ~= 0 then s = s.."I" end
272     t[mode] = s
273     return s
274   end}),
275   ["XLOAD "] = { [0] = "", "R", "V", "RV", "U", "RU", "VU", "RVU", },
276   ["CONV  "] = setmetatable({}, { __index = function(t, mode)
277     local s = irtype[band(mode, 31)]
278     s = irtype[band(shr(mode, 5), 31)].."."..s
279     if band(mode, 0x800) ~= 0 then s = s.." sext" end
280     local c = shr(mode, 14)
281     if c == 2 then s = s.." index" elseif c == 3 then s = s.." check" end
282     t[mode] = s
283     return s
284   end}),
285   ["FLOAD "] = vmdef.irfield,
286   ["FREF  "] = vmdef.irfield,
287   ["FPMATH"] = vmdef.irfpm,
288   ["BUFHDR"] = { [0] = "RESET", "APPEND" },
289   ["TOSTR "] = { [0] = "INT", "NUM", "CHAR" },
290 }
291
292 local function ctlsub(c)
293   if c == "\n" then return "\\n"
294   elseif c == "\r" then return "\\r"
295   elseif c == "\t" then return "\\t"
296   else return format("\\%03d", byte(c))
297   end
298 end
299
300 local function fmtfunc(func, pc)
301   local fi = funcinfo(func, pc)
302   if fi.loc then
303     return fi.loc
304   elseif fi.ffid then
```

```
    return vmdef.ffnames[fi.ffid]
  elseif fi.addr then
    return format("C:%x", fi.addr)
  else
    return "(?)"
  end
end

local function formatk(tr, idx)
  local k, t, slot = tracek(tr, idx)
  local tn = type(k)
  local s
  if tn == "number" then
    if k == 2^52+2^51 then
      s = "bias"
    else
      s = format("%+.14g", k)
    end
  elseif tn == "string" then
    s = format(#k > 20 and '"%.20s"~' or '"%s"', gsub(k, "%c", ctlsub))
  elseif tn == "function" then
    s = fmtfunc(k)
  elseif tn == "table" then
    s = format("{%p}", k)
  elseif tn == "userdata" then
    if t == 12 then
      s = format("userdata:%p", k)
    else
      s = format("[%p]", k)
      if s == "[0x00000000]" then s = "NULL" end
    end
  elseif t == 21 then -- int64_t
    s = sub(tostring(k), 1, -3)
    if sub(s, 1, 1) ~= "-" then s = "+"..s end
  else
    s = tostring(k) -- For primitives.
  end
  s = colorize(format("%-4s", s), t)
  if slot then
    s = format("%s @%d", s, slot)
  end
  return s
end

local function printsnap(tr, snap)
  local n = 2
  for s=0,snap[1]-1 do
    local sn = snap[n]
    if shr(sn, 24) == s then
      n = n + 1
      local ref = band(sn, 0xffff) - 0x8000 -- REF_BIAS
      if ref < 0 then
        out:write(formatk(tr, ref))
      elseif band(sn, 0x80000) ~= 0 then -- SNAP_SOFTFPNUM
        out:write(colorize(format("%04d/%04d", ref, ref+1), 14))
      else
        local m, ot, op1, op2 = traceir(tr, ref)
        out:write(colorize(format("%04d", ref), band(ot, 31)))
      end
      out:write(band(sn, 0x10000) == 0 and " " or "|") -- SNAP_FRAME
    else
      out:write("---- ")
    end
  end
  out:write("]\n")
end

-- Dump snapshots (not interleaved with IR).
local function dump_snap(tr)
  out:write("---- TRACE ", tr, " snapshots\n")
  for i=0,1000000000 do
    local snap = tracesnap(tr, i)
    if not snap then break end
    out:write(format("#%-3d %04d [ ", i, snap[0]))
    printsnap(tr, snap)
  end
```

```
end

-- Return a register name or stack slot for a rid/sp location.
local function ridsp_name(ridsp, ins)
  if not disass then disass = require("jit.dis_"..jit.arch) end
  local rid, slot = band(ridsp, 0xff), shr(ridsp, 8)
  if rid == 253 or rid == 254 then
    return (slot == 0 or slot == 255) and " {sink" or format(" {%04d", ins-slot)
  end
  if ridsp > 255 then return format("[%x]", slot*4) end
  if rid < 128 then return disass.regname(rid) end
  return ""
end

-- Dump CALL* function ref and return optional ctype.
local function dumpcallfunc(tr, ins)
  local ctype
  if ins > 0 then
    local m, ot, op1, op2 = traceir(tr, ins)
    if band(ot, 31) == 0 then -- nil type means CARG(func, ctype).
      ins = op1
      ctype = formatk(tr, op2)
    end
  end
  if ins < 0 then
    out:write(format("[0x%x](", tonumber((tracek(tr, ins)))))
  else
    out:write(format("%04d (", ins))
  end
  return ctype
end

-- Recursively gather CALL* args and dump them.
local function dumpcallargs(tr, ins)
  if ins < 0 then
    out:write(formatk(tr, ins))
  else
    local m, ot, op1, op2 = traceir(tr, ins)
    local oidx = 6*shr(ot, 8)
    local op = sub(vmdef.irnames, oidx+1, oidx+6)
    if op == "CARG  " then
      dumpcallargs(tr, op1)
      if op2 < 0 then
        out:write(" ", formatk(tr, op2))
      else
        out:write(" ", format("%04d", op2))
      end
    else
      out:write(format("%04d", ins))
    end
  end
end

-- Dump IR and interleaved snapshots.
local function dump_ir(tr, dumpsnap, dumpreg)
  local info = traceinfo(tr)
  if not info then return end
  local nins = info.nins
  out:write("---- TRACE ", tr, " IR\n")
  local irnames = vmdef.irnames
  local snapref = 65536
  local snap, snapno
  if dumpsnap then
    snap = tracesnap(tr, 0)
    snapref = snap[0]
    snapno = 0
  end
  for ins=1,nins do
    if ins >= snapref then
      if dumpreg then
        out:write(format("....              SNAP   #%-3d [ ", snapno))
      else
        out:write(format("....        SNAP   #%-3d [ ", snapno))
      end
      printsnap(tr, snap)
      snapno = snapno + 1
```

```
      snap = tracesnap(tr, snapno)
      snapref = snap and snap[0] or 65536
    end
    local m, ot, op1, op2, ridsp = traceir(tr, ins)
    local oidx, t = 6*shr(ot, 8), band(ot, 31)
    local op = sub(irnames, oidx+1, oidx+6)
    if op == "LOOP  " then
      if dumpreg then
        out:write(format("%04d ------------ LOOP ------------\n", ins))
      else
        out:write(format("%04d ------ LOOP ------------\n", ins))
      end
    elseif op ~= "NOP   " and op ~= "CARG  " and
           (dumpreg or op ~= "RENAME") then
      local rid = band(ridsp, 255)
      if dumpreg then
        out:write(format("%04d %-6s", ins, ridsp_name(ridsp, ins)))
      else
        out:write(format("%04d ", ins))
      end
      out:write(format("%s%s %s %s ",
                       (rid == 254 or rid == 253) and "}" or
                       (band(ot, 128) == 0 and " " or ">"),
                       band(ot, 64) == 0 and " " or "+",
                       irtype[t], op))
      local m1, m2 = band(m, 3), band(m, 3*4)
      if sub(op, 1, 4) == "CALL" then
        local ctype
        if m2 == 1*4 then -- op2 == IRMlit
          out:write(format("%-10s  (", vmdef.ircall[op2]))
        else
          ctype = dumpcallfunc(tr, op2)
        end
        if op1 ~= -1 then dumpcallargs(tr, op1) end
        out:write(")")
        if ctype then out:write(" ctype ", ctype) end
      elseif op == "CNEW  " and op2 == -1 then
        out:write(formatk(tr, op1))
      elseif m1 ~= 3 then -- op1 != IRMnone
        if op1 < 0 then
          out:write(formatk(tr, op1))
        else
          out:write(format(m1 == 0 and "%04d" or "#%-3d", op1))
        end
        if m2 ~= 3*4 then -- op2 != IRMnone
          if m2 == 1*4 then -- op2 == IRMlit
            local litn = litname[op]
            if litn and litn[op2] then
              out:write("  ", litn[op2])
            elseif op == "UREFO " or op == "UREFC " then
              out:write(format("  #%-3d", shr(op2, 8)))
            else
              out:write(format("  #%-3d", op2))
            end
          elseif op2 < 0 then
            out:write("  ", formatk(tr, op2))
          else
            out:write(format("  %04d", op2))
          end
        end
      end
      out:write("\n")
    end
  end
  if snap then
    if dumpreg then
      out:write(format("....              SNAP   #%-3d [ ", snapno))
    else
      out:write(format("....        SNAP   #%-3d [ ", snapno))
    end
    printsnap(tr, snap)
  end
end

------------------------------------------------------------------------------

```

```
local recprefix = ""
local recdepth = 0

-- Format trace error message.
local function fmterr(err, info)
  if type(err) == "number" then
    if type(info) == "function" then info = fmtfunc(info) end
    err = format(vmdef.traceerr[err], info)
  end
  return err
end

-- Dump trace states.
local function dump_trace(what, tr, func, pc, otr, oex)
  if what == "stop" or (what == "abort" and dumpmode.a) then
    if dumpmode.i then dump_ir(tr, dumpmode.s, dumpmode.r and what == "stop")
    elseif dumpmode.s then dump_snap(tr) end
    if dumpmode.m then dump_mcode(tr) end
  end
  if what == "start" then
    if dumpmode.H then out:write('<pre class="ljdump">\n') end
    out:write("---- TRACE ", tr, " ", what)
    if otr then out:write(" ", otr, "/", oex) end
    out:write(" ", fmtfunc(func, pc), "\n")
  elseif what == "stop" or what == "abort" then
    out:write("---- TRACE ", tr, " ", what)
    if what == "abort" then
      out:write(" ", fmtfunc(func, pc), " -- ", fmterr(otr, oex), "\n")
    else
      local info = traceinfo(tr)
      local link, ltype = info.link, info.linktype
      if link == tr or link == 0 then
        out:write(" -> ", ltype, "\n")
      elseif ltype == "root" then
        out:write(" -> ", link, "\n")
      else
        out:write(" -> ", link, " ", ltype, "\n")
      end
    end
    if dumpmode.H then out:write("</pre>\n\n") else out:write("\n") end
  else
    out:write("---- TRACE ", what, "\n\n")
  end
  out:flush()
end

-- Dump recorded bytecode.
local function dump_record(tr, func, pc, depth, callee)
  if depth ~= recdepth then
    recdepth = depth
    recprefix = rep(" .", depth)
  end
  local line
  if pc >= 0 then
    line = bcline(func, pc, recprefix)
    if dumpmode.H then line = gsub(line, "[<>&]", html_escape) end
  else
    line = "0000 "..recprefix.." FUNCC      \n"
    callee = func
  end
  if pc <= 0 then
    out:write(sub(line, 1, -2), "         ; ", fmtfunc(func), "\n")
  else
    out:write(line)
  end
  if pc >= 0 and band(funcbc(func, pc), 0xff) < 16 then -- ORDER BC
    out:write(bcline(func, pc+1, recprefix)) -- Write JMP for cond.
  end
end

------------------------------------------------------------------------------

-- Dump taken trace exits.
local function dump_texit(tr, ex, ngpr, nfpr, ...)
  out:write("---- TRACE ", tr, " exit ", ex, "\n")
  if dumpmode.X then
```

```
    local regs = {...}
    if jit.arch == "x64" then
      for i=1,ngpr do
        out:write(format(" %016x", regs[i]))
        if i % 4 == 0 then out:write("\n") end
      end
    else
      for i=1,ngpr do
        out:write(" ", tohex(regs[i]))
        if i % 8 == 0 then out:write("\n") end
      end
    end
    if jit.arch == "mips" or jit.arch == "mipsel" then
      for i=1,nfpr,2 do
        out:write(format(" %+17.14g", regs[ngpr+i]))
        if i % 8 == 7 then out:write("\n") end
      end
    else
      for i=1,nfpr do
        out:write(format(" %+17.14g", regs[ngpr+i]))
        if i % 4 == 0 then out:write("\n") end
      end
    end
  end
end

------------------------------------------------------------------------------

-- Detach dump handlers.
local function dumpoff()
  if active then
    active = false
    jit.attach(dump_texit)
    jit.attach(dump_record)
    jit.attach(dump_trace)
    if out and out ~= stdout and out ~= stderr then out:close() end
    out = nil
  end
end

-- Open the output file and attach dump handlers.
local function dumpon(opt, outfile)
  if active then dumpoff() end

  local colormode = os.getenv("COLORTERM") and "A" or "T"
  if opt then
    opt = gsub(opt, "[TAH]", function(mode) colormode = mode; return ""; end)
  end

  local m = { t=true, b=true, i=true, m=true, }
  if opt and opt ~= "" then
    local o = sub(opt, 1, 1)
    if o ~= "+" and o ~= "-" then m = {} end
    for i=1,#opt do m[sub(opt, i, i)] = (o ~= "-") end
  end
  dumpmode = m

  if m.t or m.b or m.i or m.s or m.m then
    jit.attach(dump_trace, "trace")
  end
  if m.b then
    jit.attach(dump_record, "record")
    if not bcline then bcline = require("jit.bc").line end
  end
  if m.x or m.X then
    jit.attach(dump_texit, "texit")
  end

  if not outfile then outfile = os.getenv("LUAJIT_DUMPFILE") end
  if outfile then
    out = outfile == "-" and stdout or assert(io.open(outfile, "w"))
  else
    out = stdout
  end

  m[colormode] = true
```

```
685   if colormode == "A" then
686     colorize = colorize_ansi
687     irtype = irtype_ansi
688   elseif colormode == "H" then
689     colorize = colorize_html
690     irtype = irtype_html
691     out:write(header_html)
692   else
693     colorize = colorize_text
694     irtype = irtype_text
695   end
696
697   active = true
698 end
699
700 -- Public module functions.
701 return {
702   on = dumpon,
703   off = dumpoff,
704   start = dumpon -- For -j command line option.
705 }
706
```

One Level Up Top Level

src/jit/v.lua - luajit-2.0-src

Functions defined

Source code

```
1  ----------------------------------------------------------------------------
2  -- Verbose mode of the LuaJIT compiler.
3  --
4  -- Copyright (C) 2005-2015 Mike Pall. All rights reserved.
5  -- Released under the MIT license. See Copyright Notice in luajit.h
6  ----------------------------------------------------------------------------
7  --
8  -- This module shows verbose information about the progress of the
9  -- JIT compiler. It prints one line for each generated trace. This module
10 -- is useful to see which code has been compiled or where the compiler
11 -- punts and falls back to the interpreter.
12 --
13 -- Example usage:
14 --
15 --   luajit -jv -e "for i=1,1000 do for j=1,1000 do end end"
16 --   luajit -jv=myapp.out myapp.lua
17 --
18 -- Default output is to stderr. To redirect the output to a file, pass a
19 -- filename as an argument (use '-' for stdout) or set the environment
20 -- variable LUAJIT_VERBOSEFILE. The file is overwritten every time the
21 -- module is started.
22 --
23 -- The output from the first example should look like this:
24 --
25 -- [TRACE   1 (command line):1 loop]
26 -- [TRACE   2 (1/3) (command line):1 -> 1]
27 --
28 -- The first number in each line is the internal trace number. Next are
29 -- the file name ('(command line)') and the line number (':1') where the
30 -- trace has started. Side traces also show the parent trace number and
31 -- the exit number where they are attached to in parentheses ('(1/3)').
32 -- An arrow at the end shows where the trace links to ('-> 1'), unless
33 -- it loops to itself.
34 --
35 -- In this case the inner loop gets hot and is traced first, generating
36 -- a root trace. Then the last exit from the 1st trace gets hot, too,
37 -- and triggers generation of the 2nd trace. The side trace follows the
38 -- path along the outer loop and *around* the inner loop, back to its
39 -- start, and then links to the 1st trace. Yes, this may seem unusual,
40 -- if you know how traditional compilers work. Trace compilers are full
41 -- of surprises like this -- have fun! :-)
42 --
43 -- Aborted traces are shown like this:
44 --
45 -- [TRACE --- foo.lua:44 -- leaving loop in root trace at foo:lua:50]
46 --
47 -- Don't worry -- trace aborts are quite common, even in programs which
48 -- can be fully compiled. The compiler may retry several times until it
49 -- finds a suitable trace.
50 --
51 -- Of course this doesn't work with features that are not-yet-implemented
```

```
-- (NYI error messages). The VM simply falls back to the interpreter. This
-- may not matter at all if the particular trace is not very high up in
-- the CPU usage profile. Oh, and the interpreter is quite fast, too.
--
-- Also check out the -jdump module, which prints all the gory details.
--
------------------------------------------------------------------------------

-- Cache some library functions and objects.
local jit = require("jit")
assert(jit.version_num == 20100, "LuaJIT core/library version mismatch")
local jutil = require("jit.util")
local vmdef = require("jit.vmdef")
local funcinfo, traceinfo = jutil.funcinfo, jutil.traceinfo
local type, format = type, string.format
local stdout, stderr = io.stdout, io.stderr

-- Active flag and output file handle.
local active, out

------------------------------------------------------------------------------

local startloc, startex

local function fmtfunc(func, pc)
  local fi = funcinfo(func, pc)
  if fi.loc then
    return fi.loc
  elseif fi.ffid then
    return vmdef.ffnames[fi.ffid]
  elseif fi.addr then
    return format("C:%x", fi.addr)
  else
    return "(?)"
  end
end

-- Format trace error message.
local function fmterr(err, info)
  if type(err) == "number" then
    if type(info) == "function" then info = fmtfunc(info) end
    err = format(vmdef.traceerr[err], info)
  end
  return err
end

-- Dump trace states.
local function dump_trace(what, tr, func, pc, otr, oex)
   if what == "start" then
     startloc = fmtfunc(func, pc)
     startex = otr and "("..otr.."/"..oex..") " or ""
   else
     if what == "abort" then
       local loc = fmtfunc(func, pc)
       if loc ~= startloc then
         out:write(format("[TRACE --- %s%s -- %s at %s]\n",
           startex, startloc, fmterr(otr, oex), loc))
       else
         out:write(format("[TRACE --- %s%s -- %s]\n",
           startex, startloc, fmterr(otr, oex)))
       end
     elseif what == "stop" then
       local info = traceinfo(tr)
       local link, ltype = info.link, info.linktype
       if ltype == "interpreter" then
         out:write(format("[TRACE %3s %s%s -- fallback to interpreter]\n",
           tr, startex, startloc))
       elseif ltype == "stitch" then
         out:write(format("[TRACE %3s %s%s %s %s]\n",
           tr, startex, startloc, ltype, fmtfunc(func, pc)))
       elseif link == tr or link == 0 then
         out:write(format("[TRACE %3s %s%s %s]\n",
           tr, startex, startloc, ltype))
       elseif ltype == "root" then
         out:write(format("[TRACE %3s %s%s -> %d]\n",
           tr, startex, startloc, link))
```

```
      else
        out:write(format("[TRACE %3s %s%s -> %d %s]\n",
          tr, startex, startloc, link, ltype))
      end
    else
      out:write(format("[TRACE %s]\n", what))
    end
    out:flush()
  end
end

------------------------------------------------------------------------------

-- Detach dump handlers.
local function dumpoff()
  if active then
    active = false
    jit.attach(dump_trace)
    if out and out ~= stdout and out ~= stderr then out:close() end
    out = nil
  end
end

-- Open the output file and attach dump handlers.
local function dumpon(outfile)
  if active then dumpoff() end
  if not outfile then outfile = os.getenv("LUAJIT_VERBOSEFILE") end
  if outfile then
    out = outfile == "-" and stdout or assert(io.open(outfile, "w"))
  else
    out = stderr
  end
  jit.attach(dump_trace, "trace")
  active = true
end

-- Public module functions.
return {
  on = dumpon,
  off = dumpoff,
  start = dumpon -- For -j command line option.
}

```

One Level Up Top Level

src/jit/vmdef.lua - luajit-2.0-src

```
-- This is a generated file. DO NOT EDIT!

return {

bcnames = "ISLT  ISGE  ISLE  ISGT  ISEQV ISNEV ISEQS ISNES ISEQN ISNEN ISEQP ISNEP ISTC  ISFC  IST   ISF
ISTYPEISNUM MOV   NOT   UNM   LEN   ADDVN SUBVN MULVN DIVVN MODVN ADDNV SUBNV MULNV DIVNV MODNV ADDVV SUBVV MULVV
DIVVV MODVV POW   CAT   KSTR  KCDATAKSHORTKNUM  KPRI  KNIL  UGET  USETV USETS USETN USETP UCLO  FNEW  TNEW  TDUP
GGET  GSET  TGETV TGETS TGETB TGETR TSETV TSETS TSETB TSETM TSETR CALLM CALL  CALLMTCALLT ITERC ITERN VARG
ISNEXTRETM  RET   RET0  RET1  FORI  JFORI FORL  IFORL JFORL ITERL IITERLJITERLLOOP  ILOOP JLOOP JMP   FUNCF
IFUNCFJFUNCFFUNCV IFUNCVJFUNCVFUNCC FUNCCW",

irnames = "LT    GE    LE    GT    ULT   UGE   ULE   UGT   EQ    NE    ABC   RETF  NOP   BASE  PVAL
GCSTEPHIOP  LOOP  USE   PHI   RENAMEPROF  KPRI  KINT  KGC   KPTR  KKPTR KNULL KNUM  KINT64KSLOT BNOT  BSWAP BAND
BOR   BXOR  BSHL  BSHR  BSAR  BROL  BROR  ADD   SUB   MUL   DIV   MOD   POW   NEG   ABS   ATAN2 LDEXP MIN   MAX
FPMATHADDOV SUBOV MULOV AREF  HREFK HREF  NEWREFUREFO UREFC FREF  STRREFLREF  ALOAD HLOAD ULOAD FLOAD XLOAD SLOAD
VLOAD ASTOREHSTOREUSTOREFSTOREXSTORESNEW  XSNEW TNEW  TDUP  CNEW  CNEWI BUFHDRBUFPUTBUFSTRTBAR  OBAR  XBAR  CONV
TOBIT TOSTR STRTO CALLN CALLA CALLL CALLS CALLXSCARG  ",

irfpm = { [0]="floor", "ceil", "trunc", "sqrt", "exp", "exp2", "log", "log2", "log10", "sin", "cos", "tan",
"other", },

irfield = { [0]="str.len", "func.env", "func.pc", "func.ffid", "thread.env", "tab.meta", "tab.array",
"tab.node", "tab.asize", "tab.hmask", "tab.nomm", "udata.meta", "udata.udtype", "udata.file", "cdata.ctypeid",
"cdata.ptr", "cdata.int", "cdata.int64", "cdata.int64_4", },

ircall = {
[0]="lj_str_cmp",
"lj_str_find",
"lj_str_new",
"lj_strscan_num",
"lj_strfmt_int",
"lj_strfmt_num",
"lj_strfmt_char",
"lj_strfmt_putint",
"lj_strfmt_putnum",
"lj_strfmt_putquoted",
"lj_strfmt_putfxint",
"lj_strfmt_putfnum_int",
"lj_strfmt_putfnum_uint",
"lj_strfmt_putfnum",
"lj_strfmt_putfstr",
"lj_strfmt_putfchar",
"lj_buf_putmem",
"lj_buf_putstr",
"lj_buf_putchar",
"lj_buf_putstr_reverse",
"lj_buf_putstr_lower",
"lj_buf_putstr_upper",
"lj_buf_putstr_rep",
"lj_buf_puttab",
"lj_buf_tostr",
"lj_tab_new_ah",
"lj_tab_new1",
"lj_tab_dup",
"lj_tab_clear",
"lj_tab_newkey",
"lj_tab_len",
"lj_gc_step_jit",
"lj_gc_barrieruv",
"lj_mem_newgco",
"lj_math_random_step",
"lj_vm_modi",
"sinh",
"cosh",
"tanh",
"fputc",
"fwrite",
"fflush",
"lj_vm_floor",
"lj_vm_ceil",
"lj_vm_trunc",
```

```
"sqrt",
"exp",
"lj_vm_exp2",
"log",
"lj_vm_log2",
"log10",
"sin",
"cos",
"tan",
"lj_vm_powi",
"pow",
"atan2",
"ldexp",
"lj_vm_tobit",
"softfp_add",
"softfp_sub",
"softfp_mul",
"softfp_div",
"softfp_cmp",
"softfp_i2d",
"softfp_d2i",
"softfp_ui2d",
"softfp_f2d",
"softfp_d2ui",
"softfp_d2f",
"softfp_i2f",
"softfp_ui2f",
"softfp_f2i",
"softfp_f2ui",
"fp64_l2d",
"fp64_ul2d",
"fp64_l2f",
"fp64_ul2f",
"fp64_d2l",
"fp64_d2ul",
"fp64_f2l",
"fp64_f2ul",
"lj_carith_divi64",
"lj_carith_divu64",
"lj_carith_modi64",
"lj_carith_modu64",
 "lj_carith_powi64",
 "lj_carith_powu64",
 "lj_cdata_newv",
 "lj_cdata_setfin",
 "strlen",
 "memcpy",
 "memset",
 "lj_vm_errno",
 "lj_carith_mul64",
 "lj_carith_shl64",
 "lj_carith_shr64",
 "lj_carith_sar64",
 "lj_carith_rol64",
 "lj_carith_ror64",
 },

 traceerr = {
 [0]="error thrown or hook called during recording",
 "trace too short",
 "trace too long",
 "trace too deep",
 "too many snapshots",
 "blacklisted",
 "NYI: bytecode %d",
 "leaving loop in root trace",
 "inner loop in root trace",
 "loop unroll limit reached",
 "bad argument type",
 "JIT compilation disabled for function",
 "call unroll limit reached",
 "down-recursion, restarting",
 "NYI: unsupported variant of FastFunc %s",
 "NYI: return to lower frame",
 "store with nil or NaN key",
 "missing metamethod",
```

```
"looping index lookup",
"NYI: mixed sparse/dense table",
"symbol not in cache",
"NYI: unsupported C type conversion",
"NYI: unsupported C function type",
"guard would always fail",
"too many PHIs",
"persistent type instability",
"failed to allocate mcode memory",
"machine code too long",
"hit mcode limit (retrying)",
"too many spill slots",
"inconsistent register allocation",
"NYI: cannot assemble IR instruction %d",
"NYI: PHI shuffling too complex",
"NYI: register coalescing too complex",
},

ffnames = {
[0]="Lua",
"C",
"assert",
"type",
"next",
"pairs",
"ipairs_aux",
"ipairs",
"getmetatable",
"setmetatable",
"getfenv",
"setfenv",
"rawget",
"rawset",
"rawequal",
"unpack",
"select",
"tonumber",
"tostring",
"error",
"pcall",
"xpcall",
"loadfile",
"load",
"loadstring",
"dofile",
"gcinfo",
"collectgarbage",
"newproxy",
"print",
"coroutine.status",
"coroutine.running",
"coroutine.create",
"coroutine.yield",
"coroutine.resume",
"coroutine.wrap_aux",
"coroutine.wrap",
"math.abs",
"math.floor",
"math.ceil",
"math.sqrt",
"math.log10",
"math.exp",
"math.sin",
"math.cos",
"math.tan",
"math.asin",
"math.acos",
"math.atan",
"math.sinh",
"math.cosh",
"math.tanh",
"math.frexp",
"math.modf",
"math.log",
"math.atan2",
"math.pow",
```

```
"math.fmod",
"math.ldexp",
"math.min",
"math.max",
"math.random",
"math.randomseed",
"bit.tobit",
"bit.bnot",
"bit.bswap",
"bit.lshift",
"bit.rshift",
"bit.arshift",
"bit.rol",
"bit.ror",
"bit.band",
"bit.bor",
"bit.bxor",
"bit.tohex",
"string.byte",
"string.char",
"string.sub",
"string.rep",
"string.reverse",
"string.lower",
"string.upper",
"string.dump",
"string.find",
"string.match",
"string.gmatch_aux",
"string.gmatch",
"string.gsub",
"string.format",
"table.maxn",
"table.insert",
"table.concat",
"table.sort",
"table.new",
"table.clear",
"io.method.close",
"io.method.read",
"io.method.write",
"io.method.flush",
"io.method.seek",
"io.method.setvbuf",
"io.method.lines",
"io.method.__gc",
"io.method.__tostring",
"io.open",
"io.popen",
"io.tmpfile",
"io.close",
"io.read",
"io.write",
"io.flush",
"io.input",
"io.output",
"io.lines",
"io.type",
"os.execute",
"os.remove",
"os.rename",
"os.tmpname",
"os.getenv",
"os.exit",
"os.clock",
"os.date",
"os.time",
"os.difftime",
"os.setlocale",
"debug.getregistry",
"debug.getmetatable",
"debug.setmetatable",
"debug.getfenv",
"debug.setfenv",
"debug.getinfo",
"debug.getlocal",
```

```
"debug.setlocal",
"debug.getupvalue",
"debug.setupvalue",
"debug.upvalueid",
"debug.upvaluejoin",
"debug.sethook",
"debug.gethook",
"debug.debug",
"debug.traceback",
"jit.on",
"jit.off",
"jit.flush",
"jit.status",
"jit.attach",
"jit.util.funcinfo",
"jit.util.funcbc",
"jit.util.funck",
"jit.util.funcuvname",
"jit.util.traceinfo",
"jit.util.traceir",
"jit.util.tracek",
"jit.util.tracesnap",
"jit.util.tracemc",
"jit.util.traceexitstub",
"jit.util.ircalladdr",
"jit.opt.start",
"jit.profile.start",
"jit.profile.stop",
"jit.profile.dumpstack",
"ffi.meta.__index",
"ffi.meta.__newindex",
"ffi.meta.__eq",
"ffi.meta.__len",
"ffi.meta.__lt",
"ffi.meta.__le",
"ffi.meta.__concat",
"ffi.meta.__call",
"ffi.meta.__add",
"ffi.meta.__sub",
"ffi.meta.__mul",
"ffi.meta.__div",
"ffi.meta.__mod",
"ffi.meta.__pow",
"ffi.meta.__unm",
"ffi.meta.__tostring",
"ffi.meta.__pairs",
"ffi.meta.__ipairs",
"ffi.clib.__index",
"ffi.clib.__newindex",
"ffi.clib.__gc",
"ffi.callback.free",
"ffi.callback.set",
"ffi.cdef",
"ffi.new",
"ffi.cast",
"ffi.typeof",
"ffi.typeinfo",
"ffi.istype",
"ffi.sizeof",
"ffi.alignof",
"ffi.offsetof",
"ffi.errno",
"ffi.string",
"ffi.copy",
"ffi.fill",
"ffi.abi",
"ffi.metatype",
"ffi.gc",
"ffi.load",
},

}

```

One Level Up Top Level

src/lib_os.c - luajit-2.0-src

Functions defined

- LJLIB_CF(os_execute)
- LJLIB_CF(os_remove)
- LJLIB_CF(os_rename)
- LJLIB_CF(os_tmpname)
- LJLIB_CF(os_getenv)
- LJLIB_CF(os_exit)
- LJLIB_CF(os_clock)
- LJLIB_CF(os_date)
- LJLIB_CF(os_time)
- LJLIB_CF(os_difftime)
- LJLIB_CF(os_setlocale)
- getboolfield
- getfield
- luaopen_os
- setboolfield
- setfield

Macros defined

- LJLIB_MODULE_os
- LUA_LIB
- lib_os_c

Source code

```
1  /*
2  ** OS library.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  **
5  ** Major portions taken verbatim or adapted from the Lua interpreter.
6  ** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
7  */
8
9  #include <errno.h>
10  #include <time.h>
11
12  #define lib_os_c
13  #define LUA_LIB
14
15  #include "lua.h"
16  #include "lauxlib.h"
17  #include "lualib.h"
18
```

```c
#include "lj_obj.h"
#include "lj_gc.h"
#include "lj_err.h"
#include "lj_buf.h"
#include "lj_str.h"
#include "lj_lib.h"

#if LJ_TARGET_POSIX
#include <unistd.h>
#else
#include <stdio.h>
#endif

#if !LJ_TARGET_PSVITA
#include <locale.h>
#endif

/* ------------------------------------------------------------------------ */

#define LJLIB_MODULE_os

LJLIB_CF(os_execute)
{
#if LJ_TARGET_CONSOLE
#if LJ_52
  errno = ENOSYS;
  return luaL_fileresult(L, 0, NULL);
#else
  lua_pushinteger(L, -1);
  return 1;
#endif
#else
  const char *cmd = luaL_optstring(L, 1, NULL);
  int stat = system(cmd);
#if LJ_52
  if (cmd)
    return luaL_execresult(L, stat);
  setboolV(L->top++, 1);
#else
  setintV(L->top++, stat);
#endif
  return 1;
#endif
}

LJLIB_CF(os_remove)
{
  const char *filename = luaL_checkstring(L, 1);
  return luaL_fileresult(L, remove(filename) == 0, filename);
}

LJLIB_CF(os_rename)
{
  const char *fromname = luaL_checkstring(L, 1);
  const char *toname = luaL_checkstring(L, 2);
  return luaL_fileresult(L, rename(fromname, toname) == 0, fromname);
}

LJLIB_CF(os_tmpname)
{
#if LJ_TARGET_PS3 || LJ_TARGET_PS4 || LJ_TARGET_PSVITA
  lj_err_caller(L, LJ_ERR_OSUNIQF);
  return 0;
#else
#if LJ_TARGET_POSIX
  char buf[15+1];
  int fp;
  strcpy(buf, "/tmp/lua_XXXXXX");
  fp = mkstemp(buf);
  if (fp != -1)
    close(fp);
  else
    lj_err_caller(L, LJ_ERR_OSUNIQF);
#else
  char buf[L_tmpnam];
  if (tmpnam(buf) == NULL)
```

```
    lj_err_caller(L, LJ_ERR_OSUNIQF);
#endif
  lua_pushstring(L, buf);
  return 1;
#endif
}

LJLIB_CF(os_getenv)
{
#if LJ_TARGET_CONSOLE
  lua_pushnil(L);
#else
  lua_pushstring(L, getenv(luaL_checkstring(L, 1)));  /* if NULL push nil */
#endif
  return 1;
}

LJLIB_CF(os_exit)
{
  int status;
  if (L->base < L->top && tvisbool(L->base))
    status = boolV(L->base) ? EXIT_SUCCESS : EXIT_FAILURE;
  else
    status = lj_lib_optint(L, 1, EXIT_SUCCESS);
  if (L->base+1 < L->top && tvistruecond(L->base+1))
    lua_close(L);
  exit(status);
  return 0;  /* Unreachable. */
}

LJLIB_CF(os_clock)
{
  setnumV(L->top++, ((lua_Number)clock())*(1.0/(lua_Number)CLOCKS_PER_SEC));
  return 1;
}

/* ------------------------------------------------------------------------ */

static void setfield(lua_State *L, const char *key, int value)
{
  lua_pushinteger(L, value);
  lua_setfield(L, -2, key);
}

static void setboolfield(lua_State *L, const char *key, int value)
{
  if (value < 0)  /* undefined? */
    return;  /* does not set field */
  lua_pushboolean(L, value);
  lua_setfield(L, -2, key);
}

static int getboolfield(lua_State *L, const char *key)
{
  int res;
  lua_getfield(L, -1, key);
  res = lua_isnil(L, -1) ? -1 : lua_toboolean(L, -1);
  lua_pop(L, 1);
  return res;
}

static int getfield(lua_State *L, const char *key, int d)
{
  int res;
  lua_getfield(L, -1, key);
  if (lua_isnumber(L, -1)) {
    res = (int)lua_tointeger(L, -1);
  } else {
    if (d < 0)
      lj_err_callerv(L, LJ_ERR_OSDATEF, key);
    res = d;
  }
  lua_pop(L, 1);
  return res;
}
```

```c
LJLIB_CF(os_date)
{
  const char *s = luaL_optstring(L, 1, "%c");
  time_t t = luaL_opt(L, (time_t)luaL_checknumber, 2, time(NULL));
  struct tm *stm;
#if LJ_TARGET_POSIX
  struct tm rtm;
#endif
  if (*s == '!') {  /* UTC? */
    s++;  /* Skip '!' */
#if LJ_TARGET_POSIX
    stm = gmtime_r(&t, &rtm);
#else
    stm = gmtime(&t);
#endif
  } else {
#if LJ_TARGET_POSIX
    stm = localtime_r(&t, &rtm);
#else
    stm = localtime(&t);
#endif
  }
  if (stm == NULL) {  /* Invalid date? */
    setnilV(L->top++);
  } else if (strcmp(s, "*t") == 0) {
    lua_createtable(L, 0, 9);  /* 9 = number of fields */
    setfield(L, "sec", stm->tm_sec);
    setfield(L, "min", stm->tm_min);
    setfield(L, "hour", stm->tm_hour);
    setfield(L, "day", stm->tm_mday);
    setfield(L, "month", stm->tm_mon+1);
    setfield(L, "year", stm->tm_year+1900);
    setfield(L, "wday", stm->tm_wday+1);
    setfield(L, "yday", stm->tm_yday+1);
    setboolfield(L, "isdst", stm->tm_isdst);
  } else if (*s) {
    SBuf *sb = &G(L)->tmpbuf;
    MSize sz = 0;
    const char *q;
    for (q = s; *q; q++)
      sz += (*q == '%') ? 30 : 1;  /* Overflow doesn't matter. */
    setsbufL(sb, L);
    for (;;) {
      char *buf = lj_buf_need(sb, sz);
      size_t len = strftime(buf, sbufsz(sb), s, stm);
      if (len) {
        setstrV(L, L->top++, lj_str_new(L, buf, len));
        lj_gc_check(L);
        break;
      }
      sz += (sz|1);
    }
  } else {
    setstrV(L, L->top++, &G(L)->strempty);
  }
  return 1;
}

LJLIB_CF(os_time)
{
  time_t t;
  if (lua_isnoneornil(L, 1)) {  /* called without args? */
    t = time(NULL);  /* get current time */
  } else {
    struct tm ts;
    luaL_checktype(L, 1, LUA_TTABLE);
    lua_settop(L, 1);  /* make sure table is at the top */
    ts.tm_sec = getfield(L, "sec", 0);
    ts.tm_min = getfield(L, "min", 0);
    ts.tm_hour = getfield(L, "hour", 12);
    ts.tm_mday = getfield(L, "day", -1);
    ts.tm_mon = getfield(L, "month", -1) - 1;
    ts.tm_year = getfield(L, "year", -1) - 1900;
    ts.tm_isdst = getboolfield(L, "isdst");
    t = mktime(&ts);
  }
```

```c
  if (t == (time_t)(-1))
    lua_pushnil(L);
  else
    lua_pushnumber(L, (lua_Number)t);
  return 1;
}

LJLIB_CF(os_difftime)
{
  lua_pushnumber(L, difftime((time_t)(luaL_checknumber(L, 1)),
                             (time_t)(luaL_optnumber(L, 2, (lua_Number)0))));
  return 1;
}

/* ------------------------------------------------------------------------ */

LJLIB_CF(os_setlocale)
{
#if LJ_TARGET_PSVITA
  lua_pushliteral(L, "C");
#else
  GCstr *s = lj_lib_optstr(L, 1);
  const char *str = s ? strdata(s) : NULL;
  int opt = lj_lib_checkopt(L, 2, 6,
    "\5ctype\7numeric\4time\7collate\10monetary\1\377\3all");
  if (opt == 0) opt = LC_CTYPE;
  else if (opt == 1) opt = LC_NUMERIC;
  else if (opt == 2) opt = LC_TIME;
  else if (opt == 3) opt = LC_COLLATE;
  else if (opt == 4) opt = LC_MONETARY;
  else if (opt == 6) opt = LC_ALL;
  lua_pushstring(L, setlocale(opt, str));
#endif
  return 1;
}

/* ------------------------------------------------------------------------ */

#include "lj_libdef.h"

LUALIB_API int luaopen_os(lua_State *L)
{
  LJ_LIB_REG(L, LUA_OSLIBNAME, os);
  return 1;
}

```

One Level Up Top Level

src/lib_table.c - luajit-2.0-src

Functions defined

Macros defined

Source code

```
1  /*
2  ** Table library.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  **
5  ** Major portions taken verbatim or adapted from the Lua interpreter.
6  ** Copyright (C) 1994-2008 Lua.org, PUC-Rio. See Copyright Notice in lua.h
7  */
8
9  #define lib_table_c
10  #define LUA_LIB
11
12  #include "lua.h"
13  #include "lauxlib.h"
14  #include "lualib.h"
15
16  #include "lj_obj.h"
17  #include "lj_gc.h"
18  #include "lj_err.h"
19  #include "lj_buf.h"
20  #include "lj_tab.h"
21  #include "lj_ff.h"
22  #include "lj_lib.h"
23
24  /* ------------------------------------------------------------------------ */
25
```

```
#define LJLIB_MODULE_table

LJLIB_LUA(table_foreachi) /*
  function(t, f)
    CHECK_tab(t)
    CHECK_func(f)
    for i=1,#t do
      local r = f(i, t[i])
      if r ~= nil then return r end
    end
  end
*/

LJLIB_LUA(table_foreach) /*
  function(t, f)
    CHECK_tab(t)
    CHECK_func(f)
    for k, v in PAIRS(t) do
      local r = f(k, v)
      if r ~= nil then return r end
    end
  end
*/

LJLIB_LUA(table_getn) /*
  function(t)
    CHECK_tab(t)
    return #t
  end
*/

LJLIB_CF(table_maxn)
{
  GCtab *t = lj_lib_checktab(L, 1);
  TValue *array = tvref(t->array);
  Node *node;
  lua_Number m = 0;
  ptrdiff_t i;
  for (i = (ptrdiff_t)t->asize - 1; i >= 0; i--)
    if (!tvisnil(&array[i])) {
      m = (lua_Number)(int32_t)i;
      break;
    }
  node = noderef(t->node);
  for (i = (ptrdiff_t)t->hmask; i >= 0; i--)
    if (!tvisnil(&node[i].val) && tvisnumber(&node[i].key)) {
      lua_Number n = numberVnum(&node[i].key);
      if (n > m) m = n;
    }
  setnumV(L->top-1, m);
  return 1;
}

LJLIB_CF(table_insert)               LJLIB_REC(.)
{
  GCtab *t = lj_lib_checktab(L, 1);
  int32_t n, i = (int32_t)lj_tab_len(t) + 1;
  int nargs = (int)((char *)L->top - (char *)L->base);
  if (nargs != 2*sizeof(TValue)) {
    if (nargs != 3*sizeof(TValue))
      lj_err_caller(L, LJ_ERR_TABINS);
    /* NOBARRIER: This just moves existing elements around. */
    for (n = lj_lib_checkint(L, 2); i > n; i--) {
      /* The set may invalidate the get pointer, so need to do it first! */
      TValue *dst = lj_tab_setint(L, t, i);
      cTValue *src = lj_tab_getint(t, i-1);
      if (src) {
        copyTV(L, dst, src);
      } else {
        setnilV(dst);
      }
    }
    i = n;
  }
   {
     TValue *dst = lj_tab_setint(L, t, i);
```

```
    copyTV(L, dst, L->top-1);  /* Set new value. */
    lj_gc_barriert(L, t, dst);
  }
  return 0;
}

LJLIB_LUA(table_remove) /*
  function(t, pos)
    CHECK_tab(t)
    local len = #t
    if pos == nil then
      if len ~= 0 then
        local old = t[len]
        t[len] = nil
        return old
      end
    else
      CHECK_int(pos)
      if pos >= 1 and pos <= len then
        local old = t[pos]
        for i=pos+1,len do
          t[i-1] = t[i]
        end
        t[len] = nil
        return old
      end
    end
  end
*/

LJLIB_CF(table_concat)               LJLIB_REC(.)
{
  GCtab *t = lj_lib_checktab(L, 1);
  GCstr *sep = lj_lib_optstr(L, 2);
  int32_t i = lj_lib_optint(L, 3, 1);
  int32_t e = (L->base+3 < L->top && !tvisnil(L->base+3)) ?
              lj_lib_checkint(L, 4) : (int32_t)lj_tab_len(t);
  SBuf *sb = lj_buf_tmp_(L);
  SBuf *sbx = lj_buf_puttab(sb, t, sep, i, e);
  if (LJ_UNLIKELY(!sbx)) {  /* Error: bad element type. */
    int32_t idx = (int32_t)(intptr_t)sbufP(sb);
    cTValue *o = lj_tab_getint(t, idx);
    lj_err_callerv(L, LJ_ERR_TABCAT,
                   lj_obj_itypename[o ? itypemap(o) : ~LJ_TNIL], idx);
  }
  setstrV(L, L->top-1, lj_buf_str(L, sbx));
  lj_gc_check(L);
  return 1;
}

/* ------------------------------------------------------------------------ */

static void set2(lua_State *L, int i, int j)
{
  lua_rawseti(L, 1, i);
  lua_rawseti(L, 1, j);
}

static int sort_comp(lua_State *L, int a, int b)
{
  if (!lua_isnil(L, 2)) {  /* function? */
    int res;
    lua_pushvalue(L, 2);
    lua_pushvalue(L, a-1);  /* -1 to compensate function */
    lua_pushvalue(L, b-2);  /* -2 to compensate function and `a' */
    lua_call(L, 2, 1);
    res = lua_toboolean(L, -1);
    lua_pop(L, 1);
    return res;
  } else {  /* a < b? */
    return lua_lessthan(L, a, b);
  }
}

static void auxsort(lua_State *L, int l, int u)
{
```

```
  while (l < u) {  /* for tail recursion */
    int i, j;
    /* sort elements a[l], a[(l+u)/2] and a[u] */
    lua_rawgeti(L, 1, l);
    lua_rawgeti(L, 1, u);
    if (sort_comp(L, -1, -2))  /* a[u] < a[l]? */
      set2(L, l, u);  /* swap a[l] - a[u] */
    else
      lua_pop(L, 2);
    if (u-l == 1) break;  /* only 2 elements */
    i = (l+u)/2;
    lua_rawgeti(L, 1, i);
    lua_rawgeti(L, 1, l);
    if (sort_comp(L, -2, -1)) {  /* a[i]<a[l]? */
      set2(L, i, l);
    } else {
      lua_pop(L, 1);  /* remove a[l] */
      lua_rawgeti(L, 1, u);
      if (sort_comp(L, -1, -2))  /* a[u]<a[i]? */
        set2(L, i, u);
      else
        lua_pop(L, 2);
    }
    if (u-l == 2) break;  /* only 3 elements */
    lua_rawgeti(L, 1, i);  /* Pivot */
    lua_pushvalue(L, -1);
    lua_rawgeti(L, 1, u-1);
    set2(L, i, u-1);
    /* a[l] <= P == a[u-1] <= a[u], only need to sort from l+1 to u-2 */
    i = l; j = u-1;
    for (;;) {  /* invariant: a[l..i] <= P <= a[j..u] */
      /* repeat ++i until a[i] >= P */
      while (lua_rawgeti(L, 1, ++i), sort_comp(L, -1, -2)) {
        if (i>=u) lj_err_caller(L, LJ_ERR_TABSORT);
        lua_pop(L, 1);  /* remove a[i] */
      }
      /* repeat --j until a[j] <= P */
      while (lua_rawgeti(L, 1, --j), sort_comp(L, -3, -1)) {
        if (j<=l) lj_err_caller(L, LJ_ERR_TABSORT);
        lua_pop(L, 1);  /* remove a[j] */
      }
      if (j<i) {
        lua_pop(L, 3);  /* pop pivot, a[i], a[j] */
        break;
      }
      set2(L, i, j);
    }
    lua_rawgeti(L, 1, u-1);
    lua_rawgeti(L, 1, i);
    set2(L, u-1, i);  /* swap pivot (a[u-1]) with a[i] */
    /* a[l..i-1] <= a[i] == P <= a[i+1..u] */
    /* adjust so that smaller half is in [j..i] and larger one in [l..u] */
    if (i-l < u-i) {
      j=l; i=i-1; l=i+2;
    } else {
      j=i+1; i=u; u=j-2;
    }
    auxsort(L, j, i);  /* call recursively the smaller one */
  }  /* repeat the routine for the larger one */
}

LJLIB_CF(table_sort)
{
  GCtab *t = lj_lib_checktab(L, 1);
  int32_t n = (int32_t)lj_tab_len(t);
  lua_settop(L, 2);
  if (!tvisnil(L->base+1))
    lj_lib_checkfunc(L, 2);
  auxsort(L, 1, n);
  return 0;
}

#if LJ_52
LJLIB_PUSH("n")
LJLIB_CF(table_pack)
{
```

```
  TValue *array, *base = L->base;
  MSize i, n = (uint32_t)(L->top - base);
  GCtab *t = lj_tab_new(L, n ? n+1 : 0, 1);
  /* NOBARRIER: The table is new (marked white). */
  setintV(lj_tab_setstr(L, t, strV(lj_lib_upvalue(L, 1))), (int32_t)n);
  for (array = tvref(t->array) + 1, i = 0; i < n; i++)
    copyTV(L, &array[i], &base[i]);
  settabV(L, base, t);
  L->top = base+1;
  lj_gc_check(L);
  return 1;
}
#endif

LJLIB_NOREG LJLIB_CF(table_new)                LJLIB_REC(.)
{
  int32_t a = lj_lib_checkint(L, 1);
  int32_t h = lj_lib_checkint(L, 2);
  lua_createtable(L, a, h);
  return 1;
}

LJLIB_NOREG LJLIB_CF(table_clear)        LJLIB_REC(.)
{
  lj_tab_clear(lj_lib_checktab(L, 1));
  return 0;
}

static int luaopen_table_new(lua_State *L)
{
  return lj_lib_postreg(L, lj_cf_table_new, FF_table_new, "new");
}

static int luaopen_table_clear(lua_State *L)
{
  return lj_lib_postreg(L, lj_cf_table_clear, FF_table_clear, "clear");
}

/* ------------------------------------------------------------------------ */

#include "lj_libdef.h"

LUALIB_API int luaopen_table(lua_State *L)
{
  LJ_LIB_REG(L, LUA_TABLIBNAME, table);
#if LJ_52
  lua_getglobal(L, "unpack");
  lua_setfield(L, -2, "unpack");
#endif
  lj_lib_prereg(L, LUA_TABLIBNAME ".new", luaopen_table_new, tabV(L->top-1));
  lj_lib_prereg(L, LUA_TABLIBNAME ".clear", luaopen_table_clear, tabV(L->top-1));
  return 1;
}

```

One Level Up Top Level

src/lj_alloc.h - luajit-2.0-src

Macros defined

- _LJ_ALLOC_H

Source code

```
1  /*
2  ** Bundled memory allocator.
3  ** Donated to the public domain.
4  */
5
6  #ifndef _LJ_ALLOC_H
7  #define _LJ_ALLOC_H
8
9  #include "lj_def.h"
10
11  #ifndef LUAJIT_USE_SYSMALLOC
12  LJ_FUNC void *lj_alloc_create(void);
13  LJ_FUNC void lj_alloc_destroy(void *msp);
14  LJ_FUNC void *lj_alloc_f(void *msp, void *ptr, size_t osize, size_t nsize);
15  #endif
16
17  #endif
```

src/lj_asm.h - luajit-2.0-src

Macros defined

- _LJ_ASM_H

Source code

```
/*
** IR assembler (SSA IR -> machine code).
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_ASM_H
#define _LJ_ASM_H

#include "lj_jit.h"

 #if LJ_HASJIT
 LJ_FUNC void lj_asm_trace(jit_State *J, GCtrace *T);
 LJ_FUNC void lj_asm_patchexit(jit_State *J, GCtrace *T, ExitNo exitno,
                               MCode *target);
 #endif

 #endif
```

src/lj_asm_mips.h - luajit-2.0-src

Functions defined

- asm_add
- asm_add64
- asm_ahustore
- asm_ahuvload
- asm_aref
- asm_arithov
- asm_bitop
- asm_bitshift
- asm_bnot
- asm_bror
- asm_bswap
- asm_callround
- asm_callx
- asm_cnew
- asm_comp
- asm_comp64
- asm_comp64eq
- asm_conv
- asm_equal
- asm_exitstub_setup
- asm_fload
- asm_fparith
- asm_fpmath
- asm_fpunary
- asm_fref
- asm_fstore
- asm_fuseabase
- asm_fuseahuref
- asm_fusexref
- asm_fxloadins

- asm_fxstoreins
- asm_gc_check
- asm_gencall
- asm_guard
- asm_head_root_base
- asm_head_side_base
- asm_hiop
- asm_href
- asm_hrefk
- asm_loop_fixup
- asm_min_max
- asm_mul
- asm_mulov
- asm_neg
- asm_neg64
- asm_obar
- asm_prof
- asm_retf
- asm_setup_call_slots
- asm_setup_target
- asm_setupresult
- asm_sload
- asm_sparejump_setup
- asm_stack_check
- asm_stack_restore
- asm_strref
- asm_strto
- asm_sub
- asm_sub64
- asm_tail_fixup
- asm_tail_prep
- asm_tbar
- asm_tobit

- asm_tointg
- asm_tvptr
- asm_uref
- asm_xload
- asm_xstore_
- lj_asm_patchexit
- noconflict
- ra_alloc1z
- ra_alloc2
- ra_hintalloc

Macros defined

- CONFLICT_SEARCH_LIM
- MIPS_SPAREJUMP
- asm_abs
- asm_addov
- asm_atan2
- asm_band
- asm_bor
- asm_brol
- asm_bsar
- asm_bshl
- asm_bshr
- asm_bxor
- asm_cnew
- asm_div
- asm_exitstub_addr
- asm_ldexp
- asm_max
- asm_min
- asm_mod
- asm_pow
- asm_subov
- asm_xstore

Source code

```
/*
** MIPS IR assembler (SSA IR -> machine code).
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

/* -- Register allocator extensions --------------------------------------- */

/* Allocate a register with a hint. */
static Reg ra_hintalloc(ASMState *as, IRRef ref, Reg hint, RegSet allow)
{
  Reg r = IR(ref)->r;
  if (ra_noreg(r)) {
    if (!ra_hashint(r) && !iscrossref(as, ref))
      ra_sethint(IR(ref)->r, hint);  /* Propagate register hint. */
    r = ra_allocref(as, ref, allow);
  }
  ra_noweak(as, r);
  return r;
}

/* Allocate a register or RID_ZERO. */
static Reg ra_alloc1z(ASMState *as, IRRef ref, RegSet allow)
{
  Reg r = IR(ref)->r;
  if (ra_noreg(r)) {
    if (!(allow & RSET_FPR) && irref_isk(ref) && IR(ref)->i == 0)
      return RID_ZERO;
    r = ra_allocref(as, ref, allow);
  } else {
    ra_noweak(as, r);
  }
  return r;
}

/* Allocate two source registers for three-operand instructions. */
static Reg ra_alloc2(ASMState *as, IRIns *ir, RegSet allow)
{
  IRIns *irl = IR(ir->op1), *irr = IR(ir->op2);
  Reg left = irl->r, right = irr->r;
  if (ra_hasreg(left)) {
    ra_noweak(as, left);
    if (ra_noreg(right))
      right = ra_alloc1z(as, ir->op2, rset_exclude(allow, left));
    else
      ra_noweak(as, right);
  } else if (ra_hasreg(right)) {
    ra_noweak(as, right);
    left = ra_alloc1z(as, ir->op1, rset_exclude(allow, right));
  } else if (ra_hashint(right)) {
    right = ra_alloc1z(as, ir->op2, allow);
    left = ra_alloc1z(as, ir->op1, rset_exclude(allow, right));
  } else {
    left = ra_alloc1z(as, ir->op1, allow);
    right = ra_alloc1z(as, ir->op2, rset_exclude(allow, left));
  }
  return left | (right << 8);
}

/* -- Guard handling ------------------------------------------------------ */

/* Need some spare long-range jump slots, for out-of-range branches. */
#define MIPS_SPAREJUMP                4

/* Setup spare long-range jump slots per mcarea. */
static void asm_sparejump_setup(ASMState *as)
{
  MCode *mxp = as->mcbot;
  /* Assumes sizeof(MCLink) == 8. */
  if (((uintptr_t)mxp & (LJ_PAGESIZE-1)) == 8) {
    lua_assert(MIPSI_NOP == 0);
    memset(mxp+2, 0, MIPS_SPAREJUMP*8);
    mxp += MIPS_SPAREJUMP*2;
    lua_assert(mxp < as->mctop);
```

```
    lj_mcode_sync(as->mcbot, mxp);
    lj_mcode_commitbot(as->J, mxp);
    as->mcbot = mxp;
    as->mclim = as->mcbot + MCLIM_REDZONE;
  }
}

/* Setup exit stub after the end of each trace. */
static void asm_exitstub_setup(ASMState *as)
{
  MCode *mxp = as->mctop;
  /* sw TMP, 0(sp); j ->vm_exit_handler; li TMP, traceno */
  *--mxp = MIPSI_LI|MIPSF_T(RID_TMP)|as->T->traceno;
  *--mxp = MIPSI_J|((((uintptr_t)(void *)lj_vm_exit_handler)>>2)&0x03ffffffu);
  lua_assert(((uintptr_t)mxp ^ (uintptr_t)(void *)lj_vm_exit_handler)>>28 == 0);
  *--mxp = MIPSI_SW|MIPSF_T(RID_TMP)|MIPSF_S(RID_SP)|0;
  as->mctop = mxp;
}

/* Keep this in-sync with exitstub_trace_addr(). */
#define asm_exitstub_addr(as)		((as)->mctop)

/* Emit conditional branch to exit for guard. */
static void asm_guard(ASMState *as, MIPSIns mi, Reg rs, Reg rt)
{
  MCode *target = asm_exitstub_addr(as);
  MCode *p = as->mcp;
  if (LJ_UNLIKELY(p == as->invmcp)) {
    as->invmcp = NULL;
    as->loopinv = 1;
    as->mcp = p+1;
    mi = mi ^ ((mi>>28) == 1 ? 0x04000000u : 0x00010000u);  /* Invert cond. */
    target = p;  /* Patch target later in asm_loop_fixup. */
  }
  emit_ti(as, MIPSI_LI, RID_TMP, as->snapno);
  emit_branch(as, mi, rs, rt, target);
}

/* -- Operand fusion ------------------------------------------------------ */

/* Limit linear search to this distance. Avoids O(n^2) behavior. */
#define CONFLICT_SEARCH_LIM		31

/* Check if there's no conflicting instruction between curins and ref. */
static int noconflict(ASMState *as, IRRef ref, IROp conflict)
{
  IRIns *ir = as->ir;
  IRRef i = as->curins;
  if (i > ref + CONFLICT_SEARCH_LIM)
    return 0;  /* Give up, ref is too far away. */
  while (--i > ref)
    if (ir[i].o == conflict)
      return 0;  /* Conflict found. */
  return 1;  /* Ok, no conflict. */
}

/* Fuse the array base of colocated arrays. */
static int32_t asm_fuseabase(ASMState *as, IRRef ref)
{
  IRIns *ir = IR(ref);
  if (ir->o == IR_TNEW && ir->op1 <= LJ_MAX_COLOSIZE &&
      !neverfuse(as) && noconflict(as, ref, IR_NEWREF))
    return (int32_t)sizeof(GCtab);
  return 0;
}

/* Fuse array/hash/upvalue reference into register+offset operand. */
static Reg asm_fuseahuref(ASMState *as, IRRef ref, int32_t *ofsp, RegSet allow)
{
  IRIns *ir = IR(ref);
  if (ra_noreg(ir->r)) {
    if (ir->o == IR_AREF) {
      if (mayfuse(as, ref)) {
        if (irref_isk(ir->op2)) {
          IRRef tab = IR(ir->op1)->op1;
          int32_t ofs = asm_fuseabase(as, tab);
```

```
150           IRRef refa = ofs ? tab : ir->op1;
151           ofs += 8*IR(ir->op2)->i;
152           if (checki16(ofs)) {
153             *ofsp = ofs;
154             return ra_alloc1(as, refa, allow);
155           }
156         }
157       }
158     } else if (ir->o == IR_HREFK) {
159       if (mayfuse(as, ref)) {
160         int32_t ofs = (int32_t)(IR(ir->op2)->op2 * sizeof(Node));
161         if (checki16(ofs)) {
162           *ofsp = ofs;
163           return ra_alloc1(as, ir->op1, allow);
164         }
165       }
166     } else if (ir->o == IR_UREFC) {
167       if (irref_isk(ir->op1)) {
168         GCfunc *fn = ir_kfunc(IR(ir->op1));
169         int32_t ofs = i32ptr(&gcref(fn->l.uvptr[(ir->op2 >> 8)])->uv.tv);
170         int32_t jgl = (intptr_t)J2G(as->J);
171         if ((uint32_t)(ofs-jgl) < 65536) {
172           *ofsp = ofs-jgl-32768;
173           return RID_JGL;
174         } else {
175           *ofsp = (int16_t)ofs;
176           return ra_allock(as, ofs-(int16_t)ofs, allow);
177         }
178       }
179     }
180   }
181   *ofsp = 0;
182   return ra_alloc1(as, ref, allow);
183 }
184
185 /* Fuse XLOAD/XSTORE reference into load/store operand. */
186 static void asm_fusexref(ASMState *as, MIPSIns mi, Reg rt, IRRef ref,
187                          RegSet allow, int32_t ofs)
188 {
189   IRIns *ir = IR(ref);
190   Reg base;
191   if (ra_noreg(ir->r) && canfuse(as, ir)) {
192     if (ir->o == IR_ADD) {
193       int32_t ofs2;
194       if (irref_isk(ir->op2) && (ofs2 = ofs + IR(ir->op2)->i, checki16(ofs2))) {
195         ref = ir->op1;
196         ofs = ofs2;
197       }
198     } else if (ir->o == IR_STRREF) {
199       int32_t ofs2 = 65536;
200       lua_assert(ofs == 0);
201       ofs = (int32_t)sizeof(GCstr);
202       if (irref_isk(ir->op2)) {
203         ofs2 = ofs + IR(ir->op2)->i;
204         ref = ir->op1;
205       } else if (irref_isk(ir->op1)) {
206         ofs2 = ofs + IR(ir->op1)->i;
207         ref = ir->op2;
208       }
209       if (!checki16(ofs2)) {
210         /* NYI: Fuse ADD with constant. */
211         Reg right, left = ra_alloc2(as, ir, allow);
212         right = (left >> 8); left &= 255;
213         emit_hsi(as, mi, rt, RID_TMP, ofs);
214         emit_dst(as, MIPSI_ADDU, RID_TMP, left, right);
215         return;
216       }
217       ofs = ofs2;
218     }
219   }
220   base = ra_alloc1(as, ref, allow);
221   emit_hsi(as, mi, rt, base, ofs);
222 }
223
224 /* -- Calls ---------------------------------------------------------- */
225
```

```
/* Generate a call to a C function. */
static void asm_gencall(ASMState *as, const CCallInfo *ci, IRRef *args)
{
  uint32_t n, nargs = CCI_XNARGS(ci);
  int32_t ofs = 16;
  Reg gpr, fpr = REGARG_FIRSTFPR;
  if ((void *)ci->func)
    emit_call(as, (void *)ci->func);
  for (gpr = REGARG_FIRSTGPR; gpr <= REGARG_LASTGPR; gpr++)
    as->cost[gpr] = REGCOST(~0u, ASMREF_L);
  gpr = REGARG_FIRSTGPR;
  for (n = 0; n < nargs; n++) {  /* Setup args. */
    IRRef ref = args[n];
    if (ref) {
      IRIns *ir = IR(ref);
      if (irt_isfp(ir->t) && fpr <= REGARG_LASTFPR &&
          !(ci->flags & CCI_VARARG)) {
        lua_assert(rset_test(as->freeset, fpr));  /* Already evicted. */
        ra_leftov(as, fpr, ref);
        fpr += 2;
        gpr += irt_isnum(ir->t) ? 2 : 1;
      } else {
        fpr = REGARG_LASTFPR+1;
        if (irt_isnum(ir->t)) gpr = (gpr+1) & ~1;
        if (gpr <= REGARG_LASTGPR) {
          lua_assert(rset_test(as->freeset, gpr));  /* Already evicted. */
          if (irt_isfp(ir->t)) {
            RegSet of = as->freeset;
            Reg r;
            /* Workaround to protect argument GPRs from being used for remat. */
            as->freeset &= ~RSET_RANGE(REGARG_FIRSTGPR, REGARG_LASTGPR+1);
            r = ra_alloc1(as, ref, RSET_FPR);
            as->freeset |= (of & RSET_RANGE(REGARG_FIRSTGPR, REGARG_LASTGPR+1));
            if (irt_isnum(ir->t)) {
              emit_tg(as, MIPSI_MFC1, gpr+(LJ_BE?0:1), r+1);
              emit_tg(as, MIPSI_MFC1, gpr+(LJ_BE?1:0), r);
              lua_assert(rset_test(as->freeset, gpr+1));  /* Already evicted. */
              gpr += 2;
            } else if (irt_isfloat(ir->t)) {
              emit_tg(as, MIPSI_MFC1, gpr, r);
              gpr++;
            }
          } else {
            ra_leftov(as, gpr, ref);
            gpr++;
          }
        } else {
          Reg r = ra_alloc1z(as, ref, irt_isfp(ir->t) ? RSET_FPR : RSET_GPR);
          if (irt_isnum(ir->t)) ofs = (ofs + 4) & ~4;
          emit_spstore(as, ir, r, ofs);
          ofs += irt_isnum(ir->t) ? 8 : 4;
        }
      }
    } else {
      fpr = REGARG_LASTFPR+1;
      if (gpr <= REGARG_LASTGPR)
        gpr++;
      else
        ofs += 4;
    }
    checkmclim(as);
  }
}

/* Setup result reg/sp for call. Evict scratch regs. */
static void asm_setupresult(ASMState *as, IRIns *ir, const CCallInfo *ci)
{
  RegSet drop = RSET_SCRATCH;
  int hiop = ((ir+1)->o == IR_HIOP);
  if ((ci->flags & CCI_NOFPRCLOBBER))
    drop &= ~RSET_FPR;
  if (ra_hasreg(ir->r))
    rset_clear(drop, ir->r);  /* Dest reg handled below. */
  if (hiop && ra_hasreg((ir+1)->r))
    rset_clear(drop, (ir+1)->r);  /* Dest reg handled below. */
  ra_evictset(as, drop);  /* Evictions must be performed first. */
```

```
  if (ra_used(ir)) {
    lua_assert(!irt_ispri(ir->t));
    if (irt_isfp(ir->t)) {
      if ((ci->flags & CCI_CASTU64)) {
        int32_t ofs = sps_scale(ir->s);
        Reg dest = ir->r;
        if (ra_hasreg(dest)) {
          ra_free(as, dest);
          ra_modified(as, dest);
          emit_tg(as, MIPSI_MTC1, RID_RETHI, dest+1);
          emit_tg(as, MIPSI_MTC1, RID_RETLO, dest);
        }
        if (ofs) {
          emit_tsi(as, MIPSI_SW, RID_RETLO, RID_SP, ofs+(LJ_BE?4:0));
          emit_tsi(as, MIPSI_SW, RID_RETHI, RID_SP, ofs+(LJ_BE?0:4));
        }
      } else {
        ra_destreg(as, ir, RID_FPRET);
      }
    } else if (hiop) {
      ra_destpair(as, ir);
    } else {
      ra_destreg(as, ir, RID_RET);
    }
  }
}

static void asm_callx(ASMState *as, IRIns *ir)
{
  IRRef args[CCI_NARGS_MAX*2];
  CCallInfo ci;
  IRRef func;
  IRIns *irf;
  ci.flags = asm_callx_flags(as, ir);
  asm_collectargs(as, ir, &ci, args);
  asm_setupresult(as, ir, &ci);
  func = ir->op2; irf = IR(func);
  if (irf->o == IR_CARG) { func = irf->op1; irf = IR(func); }
  if (irref_isk(func)) {  /* Call to constant address. */
    ci.func = (ASMFunction)(void *)(irf->i);
  } else {  /* Need specific register for indirect calls. */
    Reg r = ra_alloc1(as, func, RID2RSET(RID_CFUNCADDR));
    MCode *p = as->mcp;
    if (r == RID_CFUNCADDR)
      *--p = MIPSI_NOP;
    else
      *--p = MIPSI_MOVE | MIPSF_D(RID_CFUNCADDR) | MIPSF_S(r);
    *--p = MIPSI_JALR | MIPSF_S(r);
    as->mcp = p;
    ci.func = (ASMFunction)(void *)0;
  }
  asm_gencall(as, &ci, args);
}

static void asm_callround(ASMState *as, IRIns *ir, IRCallID id)
{
  /* The modified regs must match with the *.dasc implementation. */
  RegSet drop = RID2RSET(RID_R1)|RID2RSET(RID_R12)|RID2RSET(RID_FPRET)|
                RID2RSET(RID_F2)|RID2RSET(RID_F4)|RID2RSET(REGARG_FIRSTFPR);
  if (ra_hasreg(ir->r)) rset_clear(drop, ir->r);
  ra_evictset(as, drop);
  ra_destreg(as, ir, RID_FPRET);
  emit_call(as, (void *)lj_ir_callinfo[id].func);
  ra_leftov(as, REGARG_FIRSTFPR, ir->op1);
}

/* -- Returns ----------------------------------------------------------- */

/* Return to lower frame. Guard that it goes to the right spot. */
static void asm_retf(ASMState *as, IRIns *ir)
{
  Reg base = ra_alloc1(as, REF_BASE, RSET_GPR);
  void *pc = ir_kptr(IR(ir->op2));
  int32_t delta = 1+LJ_FR2+bc_a(*((const BCIns *)pc - 1));
  as->topslot -= (BCReg)delta;
  if ((int32_t)as->topslot < 0) as->topslot = 0;
```

```
  irt_setmark(IR(REF_BASE)->t);  /* Children must not coalesce with BASE reg. */
  emit_setgl(as, base, jit_base);
  emit_addptr(as, base, -8*delta);
  asm_guard(as, MIPSI_BNE, RID_TMP,
            ra_allock(as, i32ptr(pc), rset_exclude(RSET_GPR, base)));
  emit_tsi(as, MIPSI_LW, RID_TMP, base, -8);
}

/* -- Type conversions ---------------------------------------------------- */

static void asm_tointg(ASMState *as, IRIns *ir, Reg left)
{
  Reg tmp = ra_scratch(as, rset_exclude(RSET_FPR, left));
  Reg dest = ra_dest(as, ir, RSET_GPR);
  asm_guard(as, MIPSI_BC1F, 0, 0);
  emit_fgh(as, MIPSI_C_EQ_D, 0, tmp, left);
  emit_fg(as, MIPSI_CVT_D_W, tmp, tmp);
  emit_tg(as, MIPSI_MFC1, dest, tmp);
  emit_fg(as, MIPSI_CVT_W_D, tmp, left);
}

static void asm_tobit(ASMState *as, IRIns *ir)
{
  RegSet allow = RSET_FPR;
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg left = ra_alloc1(as, ir->op1, allow);
  Reg right = ra_alloc1(as, ir->op2, rset_clear(allow, left));
  Reg tmp = ra_scratch(as, rset_clear(allow, right));
  emit_tg(as, MIPSI_MFC1, dest, tmp);
  emit_fgh(as, MIPSI_ADD_D, tmp, left, right);
}

static void asm_conv(ASMState *as, IRIns *ir)
{
  IRType st = (IRType)(ir->op2 & IRCONV_SRCMASK);
  int stfp = (st == IRT_NUM || st == IRT_FLOAT);
  IRRef lref = ir->op1;
  lua_assert(irt_type(ir->t) != st);
  lua_assert(!(irt_isint64(ir->t) ||
               (st == IRT_I64 || st == IRT_U64))); /* Handled by SPLIT. */
  if (irt_isfp(ir->t)) {
    Reg dest = ra_dest(as, ir, RSET_FPR);
    if (stfp) {  /* FP to FP conversion. */
      emit_fg(as, st == IRT_NUM ? MIPSI_CVT_S_D : MIPSI_CVT_D_S,
              dest, ra_alloc1(as, lref, RSET_FPR));
    } else if (st == IRT_U32) {  /* U32 to FP conversion. */
      /* y = (x ^ 0x8000000) + 2147483648.0 */
      Reg left = ra_alloc1(as, lref, RSET_GPR);
      Reg tmp = ra_scratch(as, rset_exclude(RSET_FPR, dest));
      emit_fgh(as, irt_isfloat(ir->t) ? MIPSI_ADD_S : MIPSI_ADD_D,
               dest, dest, tmp);
      emit_fg(as, irt_isfloat(ir->t) ? MIPSI_CVT_S_W : MIPSI_CVT_D_W,
              dest, dest);
      if (irt_isfloat(ir->t))
        emit_lsptr(as, MIPSI_LWC1, (tmp & 31),
                   (void *)lj_ir_k64_find(as->J, U64x(4f000000,4f000000)),
                   RSET_GPR);
      else
        emit_lsptr(as, MIPSI_LDC1, (tmp & 31),
                   (void *)lj_ir_k64_find(as->J, U64x(41e00000,00000000)),
                   RSET_GPR);
      emit_tg(as, MIPSI_MTC1, RID_TMP, dest);
      emit_dst(as, MIPSI_XOR, RID_TMP, RID_TMP, left);
      emit_ti(as, MIPSI_LUI, RID_TMP, 0x8000);
    } else {  /* Integer to FP conversion. */
      Reg left = ra_alloc1(as, lref, RSET_GPR);
      emit_fg(as, irt_isfloat(ir->t) ? MIPSI_CVT_S_W : MIPSI_CVT_D_W,
              dest, dest);
      emit_tg(as, MIPSI_MTC1, left, dest);
    }
  } else if (stfp) {  /* FP to integer conversion. */
    if (irt_isguard(ir->t)) {
      /* Checked conversions are only supported from number to int. */
      lua_assert(irt_isint(ir->t) && st == IRT_NUM);
      asm_tointg(as, ir, ra_alloc1(as, lref, RSET_FPR));
    } else {
```

```
      Reg dest = ra_dest(as, ir, RSET_GPR);
      Reg left = ra_alloc1(as, lref, RSET_FPR);
      Reg tmp = ra_scratch(as, rset_exclude(RSET_FPR, left));
      if (irt_isu32(ir->t)) {
        /* y = (int)floor(x - 2147483648.0) ^ 0x80000000 */
        emit_dst(as, MIPSI_XOR, dest, dest, RID_TMP);
        emit_ti(as, MIPSI_LUI, RID_TMP, 0x8000);
        emit_tg(as, MIPSI_MFC1, dest, tmp);
        emit_fg(as, st == IRT_FLOAT ? MIPSI_FLOOR_W_S : MIPSI_FLOOR_W_D,
                tmp, tmp);
        emit_fgh(as, st == IRT_FLOAT ? MIPSI_SUB_S : MIPSI_SUB_D,
                 tmp, left, tmp);
        if (st == IRT_FLOAT)
          emit_lsptr(as, MIPSI_LWC1, (tmp & 31),
                     (void *)lj_ir_k64_find(as->J, U64x(4f000000,4f000000)),
                     RSET_GPR);
        else
          emit_lsptr(as, MIPSI_LDC1, (tmp & 31),
                     (void *)lj_ir_k64_find(as->J, U64x(41e00000,00000000)),
                     RSET_GPR);
      } else {
        emit_tg(as, MIPSI_MFC1, dest, tmp);
        emit_fg(as, st == IRT_FLOAT ? MIPSI_TRUNC_W_S : MIPSI_TRUNC_W_D,
                tmp, left);
      }
    }
  } else {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    if (st >= IRT_I8 && st <= IRT_U16) {  /* Extend to 32 bit integer. */
      Reg left = ra_alloc1(as, ir->op1, RSET_GPR);
      lua_assert(irt_isint(ir->t) || irt_isu32(ir->t));
      if ((ir->op2 & IRCONV_SEXT)) {
        if ((as->flags & JIT_F_MIPS32R2)) {
          emit_dst(as, st == IRT_I8 ? MIPSI_SEB : MIPSI_SEH, dest, 0, left);
        } else {
          uint32_t shift = st == IRT_I8 ? 24 : 16;
          emit_dta(as, MIPSI_SRA, dest, dest, shift);
          emit_dta(as, MIPSI_SLL, dest, left, shift);
        }
      } else {
        emit_tsi(as, MIPSI_ANDI, dest, left,
                 (int32_t)(st == IRT_U8 ? 0xff : 0xffff));
      }
    } else {  /* 32/64 bit integer conversions. */
      /* Only need to handle 32/32 bit no-op (cast) on 32 bit archs. */
      ra_leftov(as, dest, lref);  /* Do nothing, but may need to move regs. */
    }
  }
}

static void asm_strto(ASMState *as, IRIns *ir)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_strscan_num];
  IRRef args[2];
  RegSet drop = RSET_SCRATCH;
  if (ra_hasreg(ir->r)) rset_set(drop, ir->r);  /* Spill dest reg (if any). */
  ra_evictset(as, drop);
  asm_guard(as, MIPSI_BEQ, RID_RET, RID_ZERO);  /* Test return status. */
  args[0] = ir->op1;      /* GCstr *str */
  args[1] = ASMREF_TMP1;  /* TValue *n  */
  asm_gencall(as, ci, args);
  /* Store the result to the spill slot or temp slots. */
  emit_tsi(as, MIPSI_ADDIU, ra_releasetmp(as, ASMREF_TMP1),
           RID_SP, sps_scale(ir->s));
}

/* -- Memory references --------------------------------------------------- */

/* Get pointer to TValue. */
static void asm_tvptr(ASMState *as, Reg dest, IRRef ref)
{
  IRIns *ir = IR(ref);
  if (irt_isnum(ir->t)) {
    if (irref_isk(ref))  /* Use the number constant itself as a TValue. */
      ra_allockreg(as, i32ptr(ir_knum(ir)), dest);
    else  /* Otherwise force a spill and use the spill slot. */
```

```
      emit_tsi(as, MIPSI_ADDIU, dest, RID_SP, ra_spill(as, ir));
  } else {
    /* Otherwise use g->tmptv to hold the TValue. */
    RegSet allow = rset_exclude(RSET_GPR, dest);
    Reg type;
    emit_tsi(as, MIPSI_ADDIU, dest, RID_JGL, offsetof(global_State, tmptv)-32768);
    if (!irt_ispri(ir->t)) {
      Reg src = ra_alloc1(as, ref, allow);
      emit_setgl(as, src, tmptv.gcr);
    }
    type = ra_allock(as, irt_toitype(ir->t), allow);
    emit_setgl(as, type, tmptv.it);
  }
}

static void asm_aref(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg idx, base;
  if (irref_isk(ir->op2)) {
    IRRef tab = IR(ir->op1)->op1;
    int32_t ofs = asm_fuseabase(as, tab);
    IRRef refa = ofs ? tab : ir->op1;
    ofs += 8*IR(ir->op2)->i;
    if (checki16(ofs)) {
      base = ra_alloc1(as, refa, RSET_GPR);
      emit_tsi(as, MIPSI_ADDIU, dest, base, ofs);
      return;
    }
  }
  base = ra_alloc1(as, ir->op1, RSET_GPR);
  idx = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, base));
  emit_dst(as, MIPSI_ADDU, dest, RID_TMP, base);
  emit_dta(as, MIPSI_SLL, RID_TMP, idx, 3);
}

/* Inlined hash lookup. Specialized for key type and for const keys.
** The equivalent C code is:
**   Node *n = hashkey(t, key);
**   do {
**     if (lj_obj_equal(&n->key, key)) return &n->val;
**   } while ((n = nextnode(n)));
**   return niltv(L);
*/
static void asm_href(ASMState *as, IRIns *ir, IROp merge)
{
  RegSet allow = RSET_GPR;
  int destused = ra_used(ir);
  Reg dest = ra_dest(as, ir, allow);
  Reg tab = ra_alloc1(as, ir->op1, rset_clear(allow, dest));
  Reg key = RID_NONE, type = RID_NONE, tmpnum = RID_NONE, tmp1 = RID_TMP, tmp2;
  IRRef refkey = ir->op2;
  IRIns *irkey = IR(refkey);
  IRType1 kt = irkey->t;
  uint32_t khash;
  MCLabel l_end, l_loop, l_next;

  rset_clear(allow, tab);
  if (irt_isnum(kt)) {
    key = ra_alloc1(as, refkey, RSET_FPR);
    tmpnum = ra_scratch(as, rset_exclude(RSET_FPR, key));
  } else if (!irt_ispri(kt)) {
    key = ra_alloc1(as, refkey, allow);
    rset_clear(allow, key);
    type = ra_allock(as, irt_toitype(irkey->t), allow);
    rset_clear(allow, type);
  }
  tmp2 = ra_scratch(as, allow);
  rset_clear(allow, tmp2);

  /* Key not found in chain: jump to exit (if merged) or load niltv. */
  l_end = emit_label(as);
  as->invmcp = NULL;
  if (merge == IR_NE)
    asm_guard(as, MIPSI_B, RID_ZERO, RID_ZERO);
  else if (destused)
```

```
    emit_loada(as, dest, niltvg(J2G(as->J)));
  /* Follow hash chain until the end. */
  emit_move(as, dest, tmp2);
  l_loop = --as->mcp;
  emit_tsi(as, MIPSI_LW, tmp2, dest, (int32_t)offsetof(Node, next));
  l_next = emit_label(as);

  /* Type and value comparison. */
  if (merge == IR_EQ) {  /* Must match asm_guard(). */
    emit_ti(as, MIPSI_LI, RID_TMP, as->snapno);
    l_end = asm_exitstub_addr(as);
  }
  if (irt_isnum(kt)) {
    emit_branch(as, MIPSI_BC1T, 0, 0, l_end);
    emit_fgh(as, MIPSI_C_EQ_D, 0, tmpnum, key);
    *--as->mcp = MIPSI_NOP;  /* Avoid NaN comparison overhead. */
    emit_branch(as, MIPSI_BEQ, tmp2, RID_ZERO, l_next);
    emit_tsi(as, MIPSI_SLTIU, tmp2, tmp2, (int32_t)LJ_TISNUM);
    emit_hsi(as, MIPSI_LDC1, tmpnum, dest, (int32_t)offsetof(Node, key.n));
  } else {
    if (irt_ispri(kt)) {
      emit_branch(as, MIPSI_BEQ, tmp2, type, l_end);
    } else {
      emit_branch(as, MIPSI_BEQ, tmp1, key, l_end);
      emit_tsi(as, MIPSI_LW, tmp1, dest, (int32_t)offsetof(Node, key.gcr));
      emit_branch(as, MIPSI_BNE, tmp2, type, l_next);
    }
  }
  emit_tsi(as, MIPSI_LW, tmp2, dest, (int32_t)offsetof(Node, key.it));
  *l_loop = MIPSI_BNE | MIPSF_S(tmp2) | ((as->mcp-l_loop-1) & 0xffffu);

  /* Load main position relative to tab->node into dest. */
  khash = irref_isk(refkey) ? ir_khash(irkey) : 1;
  if (khash == 0) {
    emit_tsi(as, MIPSI_LW, dest, tab, (int32_t)offsetof(GCtab, node));
  } else {
    Reg tmphash = tmp1;
    if (irref_isk(refkey))
      tmphash = ra_allock(as, khash, allow);
    emit_dst(as, MIPSI_ADDU, dest, dest, tmp1);
    lua_assert(sizeof(Node) == 24);
    emit_dst(as, MIPSI_SUBU, tmp1, tmp2, tmp1);
    emit_dta(as, MIPSI_SLL, tmp1, tmp1, 3);
    emit_dta(as, MIPSI_SLL, tmp2, tmp1, 5);
    emit_dst(as, MIPSI_AND, tmp1, tmp2, tmphash);
    emit_tsi(as, MIPSI_LW, dest, tab, (int32_t)offsetof(GCtab, node));
    emit_tsi(as, MIPSI_LW, tmp2, tab, (int32_t)offsetof(GCtab, hmask));
    if (irref_isk(refkey)) {
      /* Nothing to do. */
    } else if (irt_isstr(kt)) {
      emit_tsi(as, MIPSI_LW, tmp1, key, (int32_t)offsetof(GCstr, hash));
    } else {  /* Must match with hash*() in lj_tab.c. */
      emit_dst(as, MIPSI_SUBU, tmp1, tmp1, tmp2);
      emit_rotr(as, tmp2, tmp2, dest, (-HASH_ROT3)&31);
      emit_dst(as, MIPSI_XOR, tmp1, tmp1, tmp2);
      emit_rotr(as, tmp1, tmp1, dest, (-HASH_ROT2-HASH_ROT1)&31);
      emit_dst(as, MIPSI_SUBU, tmp2, tmp2, dest);
      if (irt_isnum(kt)) {
        emit_dst(as, MIPSI_XOR, tmp2, tmp2, tmp1);
        if ((as->flags & JIT_F_MIPS32R2)) {
          emit_dta(as, MIPSI_ROTR, dest, tmp1, (-HASH_ROT1)&31);
        } else {
          emit_dst(as, MIPSI_OR, dest, dest, tmp1);
          emit_dta(as, MIPSI_SLL, tmp1, tmp1, HASH_ROT1);
          emit_dta(as, MIPSI_SRL, dest, tmp1, (-HASH_ROT1)&31);
        }
        emit_dst(as, MIPSI_ADDU, tmp1, tmp1, tmp1);
        emit_tg(as, MIPSI_MFC1, tmp2, key);
        emit_tg(as, MIPSI_MFC1, tmp1, key+1);
      } else {
        emit_dst(as, MIPSI_XOR, tmp2, key, tmp1);
        emit_rotr(as, dest, tmp1, tmp2, (-HASH_ROT1)&31);
        emit_dst(as, MIPSI_ADDU, tmp1, key, ra_allock(as, HASH_BIAS, allow));
      }
    }
  }
```

```
}

static void asm_hrefk(ASMState *as, IRIns *ir)
{
  IRIns *kslot = IR(ir->op2);
  IRIns *irkey = IR(kslot->op1);
  int32_t ofs = (int32_t)(kslot->op2 * sizeof(Node));
  int32_t kofs = ofs + (int32_t)offsetof(Node, key);
  Reg dest = (ra_used(ir)||ofs > 32736) ? ra_dest(as, ir, RSET_GPR) : RID_NONE;
  Reg node = ra_alloc1(as, ir->op1, RSET_GPR);
  Reg key = RID_NONE, type = RID_TMP, idx = node;
  RegSet allow = rset_exclude(RSET_GPR, node);
  int32_t lo, hi;
  lua_assert(ofs % sizeof(Node) == 0);
  if (ofs > 32736) {
    idx = dest;
    rset_clear(allow, dest);
    kofs = (int32_t)offsetof(Node, key);
  } else if (ra_hasreg(dest)) {
    emit_tsi(as, MIPSI_ADDIU, dest, node, ofs);
  }
  if (!irt_ispri(irkey->t)) {
    key = ra_scratch(as, allow);
    rset_clear(allow, key);
  }
  if (irt_isnum(irkey->t)) {
    lo = (int32_t)ir_knum(irkey)->u32.lo;
    hi = (int32_t)ir_knum(irkey)->u32.hi;
  } else {
    lo = irkey->i;
    hi = irt_toitype(irkey->t);
    if (!ra_hasreg(key))
      goto nolo;
  }
  asm_guard(as, MIPSI_BNE, key, lo ? ra_allock(as, lo, allow) : RID_ZERO);
nolo:
  asm_guard(as, MIPSI_BNE, type, hi ? ra_allock(as, hi, allow) : RID_ZERO);
  if (ra_hasreg(key)) emit_tsi(as, MIPSI_LW, key, idx, kofs+(LJ_BE?4:0));
  emit_tsi(as, MIPSI_LW, type, idx, kofs+(LJ_BE?0:4));
  if (ofs > 32736)
    emit_tsi(as, MIPSI_ADDU, dest, node, ra_allock(as, ofs, allow));
}

static void asm_uref(ASMState *as, IRIns *ir)
{
  /* NYI: Check that UREFO is still open and not aliasing a slot. */
  Reg dest = ra_dest(as, ir, RSET_GPR);
  if (irref_isk(ir->op1)) {
    GCfunc *fn = ir_kfunc(IR(ir->op1));
    MRef *v = &gcref(fn->l.uvptr[(ir->op2 >> 8)])->uv.v;
    emit_lsptr(as, MIPSI_LW, dest, v, RSET_GPR);
  } else {
    Reg uv = ra_scratch(as, RSET_GPR);
    Reg func = ra_alloc1(as, ir->op1, RSET_GPR);
    if (ir->o == IR_UREFC) {
      asm_guard(as, MIPSI_BEQ, RID_TMP, RID_ZERO);
      emit_tsi(as, MIPSI_ADDIU, dest, uv, (int32_t)offsetof(GCupval, tv));
      emit_tsi(as, MIPSI_LBU, RID_TMP, uv, (int32_t)offsetof(GCupval, closed));
    } else {
      emit_tsi(as, MIPSI_LW, dest, uv, (int32_t)offsetof(GCupval, v));
    }
    emit_tsi(as, MIPSI_LW, uv, func,
             (int32_t)offsetof(GCfuncL, uvptr) + 4*(int32_t)(ir->op2 >> 8));
  }
}

static void asm_fref(ASMState *as, IRIns *ir)
{
  UNUSED(as); UNUSED(ir);
  lua_assert(!ra_used(ir));
}

static void asm_strref(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  IRRef ref = ir->op2, refk = ir->op1;
```

```
  int32_t ofs = (int32_t)sizeof(GCstr);
  Reg r;
  if (irref_isk(ref)) {
    IRRef tmp = refk; refk = ref; ref = tmp;
  } else if (!irref_isk(refk)) {
    Reg right, left = ra_alloc1(as, ir->op1, RSET_GPR);
    IRIns *irr = IR(ir->op2);
    if (ra_hasreg(irr->r)) {
      ra_noweak(as, irr->r);
      right = irr->r;
    } else if (mayfuse(as, irr->op2) &&
               irr->o == IR_ADD && irref_isk(irr->op2) &&
               checki16(ofs + IR(irr->op2)->i)) {
      ofs += IR(irr->op2)->i;
      right = ra_alloc1(as, irr->op1, rset_exclude(RSET_GPR, left));
    } else {
      right = ra_allocref(as, ir->op2, rset_exclude(RSET_GPR, left));
    }
    emit_tsi(as, MIPSI_ADDIU, dest, dest, ofs);
    emit_dst(as, MIPSI_ADDU, dest, left, right);
    return;
  }
  r = ra_alloc1(as, ref, RSET_GPR);
  ofs += IR(refk)->i;
  if (checki16(ofs))
    emit_tsi(as, MIPSI_ADDIU, dest, r, ofs);
  else
    emit_dst(as, MIPSI_ADDU, dest, r,
             ra_allock(as, ofs, rset_exclude(RSET_GPR, r)));
}

/* -- Loads and stores ---------------------------------------------------- */

static MIPSIns asm_fxloadins(IRIns *ir)
{
  switch (irt_type(ir->t)) {
  case IRT_I8: return MIPSI_LB;
  case IRT_U8: return MIPSI_LBU;
  case IRT_I16: return MIPSI_LH;
  case IRT_U16: return MIPSI_LHU;
  case IRT_NUM: return MIPSI_LDC1;
  case IRT_FLOAT: return MIPSI_LWC1;
  default: return MIPSI_LW;
  }
}

static MIPSIns asm_fxstoreins(IRIns *ir)
{
  switch (irt_type(ir->t)) {
  case IRT_I8: case IRT_U8: return MIPSI_SB;
  case IRT_I16: case IRT_U16: return MIPSI_SH;
  case IRT_NUM: return MIPSI_SDC1;
  case IRT_FLOAT: return MIPSI_SWC1;
  default: return MIPSI_SW;
  }
}

static void asm_fload(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg idx = ra_alloc1(as, ir->op1, RSET_GPR);
  MIPSIns mi = asm_fxloadins(ir);
  int32_t ofs;
  if (ir->op2 == IRFL_TAB_ARRAY) {
    ofs = asm_fuseabase(as, ir->op1);
    if (ofs) {  /* Turn the t->array load into an add for colocated arrays. */
      emit_tsi(as, MIPSI_ADDIU, dest, idx, ofs);
      return;
    }
  }
  ofs = field_ofs[ir->op2];
  lua_assert(!irt_isfp(ir->t));
  emit_tsi(as, mi, dest, idx, ofs);
}

static void asm_fstore(ASMState *as, IRIns *ir)
```

```
{
  if (ir->r != RID_SINK) {
    Reg src = ra_alloc1z(as, ir->op2, RSET_GPR);
    IRIns *irf = IR(ir->op1);
    Reg idx = ra_alloc1(as, irf->op1, rset_exclude(RSET_GPR, src));
    int32_t ofs = field_ofs[irf->op2];
    MIPSIns mi = asm_fxstoreins(ir);
    lua_assert(!irt_isfp(ir->t));
    emit_tsi(as, mi, src, idx, ofs);
  }
}

static void asm_xload(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, irt_isfp(ir->t) ? RSET_FPR : RSET_GPR);
  lua_assert(!(ir->op2 & IRXLOAD_UNALIGNED));
  asm_fusexref(as, asm_fxloadins(ir), dest, ir->op1, RSET_GPR, 0);
}

static void asm_xstore_(ASMState *as, IRIns *ir, int32_t ofs)
{
  if (ir->r != RID_SINK) {
    Reg src = ra_alloc1z(as, ir->op2, irt_isfp(ir->t) ? RSET_FPR : RSET_GPR);
    asm_fusexref(as, asm_fxstoreins(ir), src, ir->op1,
                 rset_exclude(RSET_GPR, src), ofs);
  }
}

#define asm_xstore(as, ir)        asm_xstore_(as, ir, 0)

static void asm_ahuvload(ASMState *as, IRIns *ir)
{
  IRType1 t = ir->t;
  Reg dest = RID_NONE, type = RID_TMP, idx;
  RegSet allow = RSET_GPR;
  int32_t ofs = 0;
  if (ra_used(ir)) {
    lua_assert(irt_isnum(t) || irt_isint(t) || irt_isaddr(t));
    dest = ra_dest(as, ir, irt_isnum(t) ? RSET_FPR : RSET_GPR);
    rset_clear(allow, dest);
  }
  idx = asm_fuseahuref(as, ir->op1, &ofs, allow);
  rset_clear(allow, idx);
  if (irt_isnum(t)) {
    asm_guard(as, MIPSI_BEQ, type, RID_ZERO);
    emit_tsi(as, MIPSI_SLTIU, type, type, (int32_t)LJ_TISNUM);
    if (ra_hasreg(dest))
      emit_hsi(as, MIPSI_LDC1, dest, idx, ofs);
  } else {
    asm_guard(as, MIPSI_BNE, type, ra_allock(as, irt_toitype(t), allow));
    if (ra_hasreg(dest)) emit_tsi(as, MIPSI_LW, dest, idx, ofs+(LJ_BE?4:0));
  }
  emit_tsi(as, MIPSI_LW, type, idx, ofs+(LJ_BE?0:4));
}

static void asm_ahustore(ASMState *as, IRIns *ir)
{
  RegSet allow = RSET_GPR;
  Reg idx, src = RID_NONE, type = RID_NONE;
  int32_t ofs = 0;
  if (ir->r == RID_SINK)
    return;
  if (irt_isnum(ir->t)) {
    src = ra_alloc1(as, ir->op2, RSET_FPR);
  } else {
    if (!irt_ispri(ir->t)) {
      src = ra_alloc1(as, ir->op2, allow);
      rset_clear(allow, src);
    }
    type = ra_allock(as, (int32_t)irt_toitype(ir->t), allow);
    rset_clear(allow, type);
  }
  idx = asm_fuseahuref(as, ir->op1, &ofs, allow);
  if (irt_isnum(ir->t)) {
    emit_hsi(as, MIPSI_SDC1, src, idx, ofs);
  } else {
```

```
    if (ra_hasreg(src))
      emit_tsi(as, MIPSI_SW, src, idx, ofs+(LJ_BE?4:0));
    emit_tsi(as, MIPSI_SW, type, idx, ofs+(LJ_BE?0:4));
  }
}

static void asm_sload(ASMState *as, IRIns *ir)
{
  int32_t ofs = 8*((int32_t)ir->op1-1) + ((ir->op2 & IRSLOAD_FRAME) ? 4 : 0);
  IRType1 t = ir->t;
  Reg dest = RID_NONE, type = RID_NONE, base;
  RegSet allow = RSET_GPR;
  lua_assert(!(ir->op2 & IRSLOAD_PARENT));  /* Handled by asm_head_side(). */
  lua_assert(irt_isguard(t) || !(ir->op2 & IRSLOAD_TYPECHECK));
  lua_assert(!irt_isint(t) || (ir->op2 & (IRSLOAD_CONVERT|IRSLOAD_FRAME)));
  if ((ir->op2 & IRSLOAD_CONVERT) && irt_isguard(t) && irt_isint(t)) {
    dest = ra_scratch(as, RSET_FPR);
    asm_tointg(as, ir, dest);
    t.irt = IRT_NUM;  /* Continue with a regular number type check. */
  } else if (ra_used(ir)) {
    lua_assert(irt_isnum(t) || irt_isint(t) || irt_isaddr(t));
    dest = ra_dest(as, ir, irt_isnum(t) ? RSET_FPR : RSET_GPR);
    rset_clear(allow, dest);
    base = ra_alloc1(as, REF_BASE, allow);
    rset_clear(allow, base);
    if ((ir->op2 & IRSLOAD_CONVERT)) {
      if (irt_isint(t)) {
        Reg tmp = ra_scratch(as, RSET_FPR);
        emit_tg(as, MIPSI_MFC1, dest, tmp);
        emit_fg(as, MIPSI_TRUNC_W_D, tmp, tmp);
        dest = tmp;
        t.irt = IRT_NUM;  /* Check for original type. */
      } else {
        Reg tmp = ra_scratch(as, RSET_GPR);
        emit_fg(as, MIPSI_CVT_D_W, dest, dest);
        emit_tg(as, MIPSI_MTC1, tmp, dest);
        dest = tmp;
        t.irt = IRT_INT;  /* Check for original type. */
      }
    }
    goto dotypecheck;
  }
  base = ra_alloc1(as, REF_BASE, allow);
  rset_clear(allow, base);
dotypecheck:
  if (irt_isnum(t)) {
    if ((ir->op2 & IRSLOAD_TYPECHECK)) {
      asm_guard(as, MIPSI_BEQ, RID_TMP, RID_ZERO);
      emit_tsi(as, MIPSI_SLTIU, RID_TMP, RID_TMP, (int32_t)LJ_TISNUM);
      type = RID_TMP;
    }
    if (ra_hasreg(dest)) emit_hsi(as, MIPSI_LDC1, dest, base, ofs);
  } else {
    if ((ir->op2 & IRSLOAD_TYPECHECK)) {
      Reg ktype = ra_allock(as, irt_toitype(t), allow);
      asm_guard(as, MIPSI_BNE, RID_TMP, ktype);
      type = RID_TMP;
    }
    if (ra_hasreg(dest)) emit_tsi(as, MIPSI_LW, dest, base, ofs ^ (LJ_BE?4:0));
  }
  if (ra_hasreg(type)) emit_tsi(as, MIPSI_LW, type, base, ofs ^ (LJ_BE?0:4));
}

/* -- Allocations ---------------------------------------------------------- */

#if LJ_HASFFI
static void asm_cnew(ASMState *as, IRIns *ir)
{
  CTState *cts = ctype_ctsG(J2G(as->J));
  CTypeID id = (CTypeID)IR(ir->op1)->i;
  CTSize sz;
  CTInfo info = lj_ctype_info(cts, id, &sz);
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_mem_newgco];
  IRRef args[4];
  RegSet drop = RSET_SCRATCH;
  lua_assert(sz != CTSIZE_INVALID || (ir->o == IR_CNEW && ir->op2 != REF_NIL));
```

```
986
987    as->gcsteps++;
988    if (ra_hasreg(ir->r))
989      rset_clear(drop, ir->r);  /* Dest reg handled below. */
990    ra_evictset(as, drop);
991    if (ra_used(ir))
992      ra_destreg(as, ir, RID_RET);  /* GCcdata * */
993
994    /* Initialize immutable cdata object. */
995    if (ir->o == IR_CNEWI) {
996      RegSet allow = (RSET_GPR & ~RSET_SCRATCH);
997      int32_t ofs = sizeof(GCcdata);
998      lua_assert(sz == 4 || sz == 8);
999      if (sz == 8) {
1000       ofs += 4;
1001       lua_assert((ir+1)->o == IR_HIOP);
1002       if (LJ_LE) ir++;
1003     }
1004     for (;;) {
1005       Reg r = ra_alloc1z(as, ir->op2, allow);
1006       emit_tsi(as, MIPSI_SW, r, RID_RET, ofs);
1007       rset_clear(allow, r);
1008       if (ofs == sizeof(GCcdata)) break;
1009       ofs -= 4; if (LJ_BE) ir++; else ir--;
1010     }
1011   } else if (ir->op2 != REF_NIL) {  /* Create VLA/VLS/aligned cdata. */
1012     ci = &lj_ir_callinfo[IRCALL_lj_cdata_newv];
1013     args[0] = ASMREF_L;     /* lua_State *L */
1014     args[1] = ir->op1;      /* CTypeID id   */
1015     args[2] = ir->op2;      /* CTSize sz    */
1016     args[3] = ASMREF_TMP1;  /* CTSize align */
1017     asm_gencall(as, ci, args);
1018     emit_loadi(as, ra_releasetmp(as, ASMREF_TMP1), (int32_t)ctype_align(info));
1019     return;
1020   }
1021
1022   /* Initialize gct and ctypeid. lj_mem_newgco() already sets marked. */
1023   emit_tsi(as, MIPSI_SB, RID_RET+1, RID_RET, offsetof(GCcdata, gct));
1024   emit_tsi(as, MIPSI_SH, RID_TMP, RID_RET, offsetof(GCcdata, ctypeid));
1025   emit_ti(as, MIPSI_LI, RID_RET+1, ~LJ_TCDATA);
1026   emit_ti(as, MIPSI_LI, RID_TMP, id); /* Lower 16 bit used. Sign-ext ok. */
1027   args[0] = ASMREF_L;     /* lua_State *L */
1028   args[1] = ASMREF_TMP1;  /* MSize size   */
1029   asm_gencall(as, ci, args);
1030   ra_allockreg(as, (int32_t)(sz+sizeof(GCcdata)),
1031                ra_releasetmp(as, ASMREF_TMP1));
1032 }
1033 #else
1034 #define asm_cnew(as, ir)        ((void)0)
1035 #endif
1036
1037 /* -- Write barriers ----------------------------------------------------- */
1038
1039 static void asm_tbar(ASMState *as, IRIns *ir)
1040 {
1041   Reg tab = ra_alloc1(as, ir->op1, RSET_GPR);
1042   Reg mark = ra_scratch(as, rset_exclude(RSET_GPR, tab));
1043   Reg link = RID_TMP;
1044   MCLabel l_end = emit_label(as);
1045   emit_tsi(as, MIPSI_SW, link, tab, (int32_t)offsetof(GCtab, gclist));
1046   emit_tsi(as, MIPSI_SB, mark, tab, (int32_t)offsetof(GCtab, marked));
1047   emit_setgl(as, tab, gc.grayagain);
1048   emit_getgl(as, link, gc.grayagain);
1049   emit_dst(as, MIPSI_XOR, mark, mark, RID_TMP);  /* Clear black bit. */
1050   emit_branch(as, MIPSI_BEQ, RID_TMP, RID_ZERO, l_end);
1051   emit_tsi(as, MIPSI_ANDI, RID_TMP, mark, LJ_GC_BLACK);
1052   emit_tsi(as, MIPSI_LBU, mark, tab, (int32_t)offsetof(GCtab, marked));
1053 }
1054
1055 static void asm_obar(ASMState *as, IRIns *ir)
1056 {
1057   const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_gc_barrieruv];
1058   IRRef args[2];
1059   MCLabel l_end;
1060   Reg obj, val, tmp;
1061   /* No need for other object barriers (yet). */
```

```
  lua_assert(IR(ir->op1)->o == IR_UREFC);
  ra_evictset(as, RSET_SCRATCH);
  l_end = emit_label(as);
  args[0] = ASMREF_TMP1;  /* global_State *g */
  args[1] = ir->op1;      /* TValue *tv      */
  asm_gencall(as, ci, args);
  emit_tsi(as, MIPSI_ADDIU, ra_releasetmp(as, ASMREF_TMP1), RID_JGL, -32768);
  obj = IR(ir->op1)->r;
  tmp = ra_scratch(as, rset_exclude(RSET_GPR, obj));
  emit_branch(as, MIPSI_BEQ, RID_TMP, RID_ZERO, l_end);
  emit_tsi(as, MIPSI_ANDI, tmp, tmp, LJ_GC_BLACK);
  emit_branch(as, MIPSI_BEQ, RID_TMP, RID_ZERO, l_end);
  emit_tsi(as, MIPSI_ANDI, RID_TMP, RID_TMP, LJ_GC_WHITES);
  val = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, obj));
  emit_tsi(as, MIPSI_LBU, tmp, obj,
           (int32_t)offsetof(GCupval, marked)-(int32_t)offsetof(GCupval, tv));
  emit_tsi(as, MIPSI_LBU, RID_TMP, val, (int32_t)offsetof(GChead, marked));
}

/* -- Arithmetic and logic operations ------------------------------------ */

static void asm_fparith(ASMState *as, IRIns *ir, MIPSIns mi)
{
  Reg dest = ra_dest(as, ir, RSET_FPR);
  Reg right, left = ra_alloc2(as, ir, RSET_FPR);
  right = (left >> 8); left &= 255;
  emit_fgh(as, mi, dest, left, right);
}

static void asm_fpunary(ASMState *as, IRIns *ir, MIPSIns mi)
{
  Reg dest = ra_dest(as, ir, RSET_FPR);
  Reg left = ra_hintalloc(as, ir->op1, dest, RSET_FPR);
  emit_fg(as, mi, dest, left);
}

static void asm_fpmath(ASMState *as, IRIns *ir)
{
  if (ir->op2 == IRFPM_EXP2 && asm_fpjoin_pow(as, ir))
    return;
  if (ir->op2 <= IRFPM_TRUNC)
    asm_callround(as, ir, IRCALL_lj_vm_floor + ir->op2);
  else if (ir->op2 == IRFPM_SQRT)
    asm_fpunary(as, ir, MIPSI_SQRT_D);
  else
    asm_callid(as, ir, IRCALL_lj_vm_floor + ir->op2);
}

static void asm_add(ASMState *as, IRIns *ir)
{
  if (irt_isnum(ir->t)) {
    asm_fparith(as, ir, MIPSI_ADD_D);
  } else {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    Reg right, left = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
    if (irref_isk(ir->op2)) {
      int32_t k = IR(ir->op2)->i;
      if (checki16(k)) {
        emit_tsi(as, MIPSI_ADDIU, dest, left, k);
        return;
      }
    }
    right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
    emit_dst(as, MIPSI_ADDU, dest, left, right);
  }
}

static void asm_sub(ASMState *as, IRIns *ir)
{
  if (irt_isnum(ir->t)) {
    asm_fparith(as, ir, MIPSI_SUB_D);
  } else {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    Reg right, left = ra_alloc2(as, ir, RSET_GPR);
    right = (left >> 8); left &= 255;
    emit_dst(as, MIPSI_SUBU, dest, left, right);
```

```
  }
}

static void asm_mul(ASMState *as, IRIns *ir)
{
  if (irt_isnum(ir->t)) {
    asm_fparith(as, ir, MIPSI_MUL_D);
  } else {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    Reg right, left = ra_alloc2(as, ir, RSET_GPR);
    right = (left >> 8); left &= 255;
    emit_dst(as, MIPSI_MUL, dest, left, right);
  }
}

#define asm_div(as, ir)               asm_fparith(as, ir, MIPSI_DIV_D)
#define asm_mod(as, ir)               asm_callid(as, ir, IRCALL_lj_vm_modi)
#define asm_pow(as, ir)               asm_callid(as, ir, IRCALL_lj_vm_powi)

static void asm_neg(ASMState *as, IRIns *ir)
{
  if (irt_isnum(ir->t)) {
    asm_fpunary(as, ir, MIPSI_NEG_D);
  } else {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    Reg left = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
    emit_dst(as, MIPSI_SUBU, dest, RID_ZERO, left);
  }
}

#define asm_abs(as, ir)               asm_fpunary(as, ir, MIPSI_ABS_D)
#define asm_atan2(as, ir)        asm_callid(as, ir, IRCALL_atan2)
#define asm_ldexp(as, ir)        asm_callid(as, ir, IRCALL_ldexp)

static void asm_arithov(ASMState *as, IRIns *ir)
{
  Reg right, left, tmp, dest = ra_dest(as, ir, RSET_GPR);
  if (irref_isk(ir->op2)) {
    int k = IR(ir->op2)->i;
    if (ir->o == IR_SUBOV) k = -k;
    if (checki16(k)) {  /* (dest < left) == (k >= 0 ? 1 : 0) */
      left = ra_alloc1(as, ir->op1, RSET_GPR);
      asm_guard(as, k >= 0 ? MIPSI_BNE : MIPSI_BEQ, RID_TMP, RID_ZERO);
      emit_dst(as, MIPSI_SLT, RID_TMP, dest, dest == left ? RID_TMP : left);
      emit_tsi(as, MIPSI_ADDIU, dest, left, k);
      if (dest == left) emit_move(as, RID_TMP, left);
      return;
    }
  }
  left = ra_alloc2(as, ir, RSET_GPR);
  right = (left >> 8); left &= 255;
  tmp = ra_scratch(as, rset_exclude(rset_exclude(rset_exclude(RSET_GPR, left),
                                                 right), dest));
  asm_guard(as, MIPSI_BLTZ, RID_TMP, 0);
  emit_dst(as, MIPSI_AND, RID_TMP, RID_TMP, tmp);
  if (ir->o == IR_ADDOV) {  /* ((dest^left) & (dest^right)) < 0 */
    emit_dst(as, MIPSI_XOR, RID_TMP, dest, dest == right ? RID_TMP : right);
  } else {  /* ((dest^left) & (dest^~right)) < 0 */
    emit_dst(as, MIPSI_XOR, RID_TMP, RID_TMP, dest);
    emit_dst(as, MIPSI_NOR, RID_TMP, dest == right ? RID_TMP : right, RID_ZERO);
  }
  emit_dst(as, MIPSI_XOR, tmp, dest, dest == left ? RID_TMP : left);
  emit_dst(as, ir->o == IR_ADDOV ? MIPSI_ADDU : MIPSI_SUBU, dest, left, right);
  if (dest == left || dest == right)
    emit_move(as, RID_TMP, dest == left ? left : right);
}

#define asm_addov(as, ir)        asm_arithov(as, ir)
#define asm_subov(as, ir)        asm_arithov(as, ir)

static void asm_mulov(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg tmp, right, left = ra_alloc2(as, ir, RSET_GPR);
  right = (left >> 8); left &= 255;
  tmp = ra_scratch(as, rset_exclude(rset_exclude(rset_exclude(RSET_GPR, left),
```

```
                                                right), dest));
  asm_guard(as, MIPSI_BNE, RID_TMP, tmp);
  emit_dta(as, MIPSI_SRA, RID_TMP, dest, 31);
  emit_dst(as, MIPSI_MFHI, tmp, 0, 0);
  emit_dst(as, MIPSI_MFLO, dest, 0, 0);
  emit_dst(as, MIPSI_MULT, 0, left, right);
}

#if LJ_HASFFI
static void asm_add64(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg right, left = ra_alloc1(as, ir->op1, RSET_GPR);
  if (irref_isk(ir->op2)) {
    int32_t k = IR(ir->op2)->i;
    if (k == 0) {
      emit_dst(as, MIPSI_ADDU, dest, left, RID_TMP);
      goto loarith;
    } else if (checki16(k)) {
      emit_dst(as, MIPSI_ADDU, dest, dest, RID_TMP);
      emit_tsi(as, MIPSI_ADDIU, dest, left, k);
      goto loarith;
    }
  }
  emit_dst(as, MIPSI_ADDU, dest, dest, RID_TMP);
  right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
  emit_dst(as, MIPSI_ADDU, dest, left, right);
loarith:
  ir--;
  dest = ra_dest(as, ir, RSET_GPR);
  left = ra_alloc1(as, ir->op1, RSET_GPR);
  if (irref_isk(ir->op2)) {
    int32_t k = IR(ir->op2)->i;
    if (k == 0) {
      if (dest != left)
        emit_move(as, dest, left);
      return;
    } else if (checki16(k)) {
      if (dest == left) {
        Reg tmp = ra_scratch(as, rset_exclude(RSET_GPR, left));
        emit_move(as, dest, tmp);
        dest = tmp;
      }
      emit_dst(as, MIPSI_SLTU, RID_TMP, dest, left);
      emit_tsi(as, MIPSI_ADDIU, dest, left, k);
      return;
    }
  }
  right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
  if (dest == left && dest == right) {
    Reg tmp = ra_scratch(as, rset_exclude(rset_exclude(RSET_GPR, left), right));
    emit_move(as, dest, tmp);
    dest = tmp;
  }
  emit_dst(as, MIPSI_SLTU, RID_TMP, dest, dest == left ? right : left);
  emit_dst(as, MIPSI_ADDU, dest, left, right);
}

static void asm_sub64(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg right, left = ra_alloc2(as, ir, RSET_GPR);
  right = (left >> 8); left &= 255;
  emit_dst(as, MIPSI_SUBU, dest, dest, RID_TMP);
  emit_dst(as, MIPSI_SUBU, dest, left, right);
  ir--;
  dest = ra_dest(as, ir, RSET_GPR);
  left = ra_alloc2(as, ir, RSET_GPR);
  right = (left >> 8); left &= 255;
  if (dest == left) {
    Reg tmp = ra_scratch(as, rset_exclude(rset_exclude(RSET_GPR, left), right));
    emit_move(as, dest, tmp);
    dest = tmp;
  }
  emit_dst(as, MIPSI_SLTU, RID_TMP, left, dest);
  emit_dst(as, MIPSI_SUBU, dest, left, right);
```

```
}

static void asm_neg64(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg left = ra_alloc1(as, ir->op1, RSET_GPR);
  emit_dst(as, MIPSI_SUBU, dest, dest, RID_TMP);
  emit_dst(as, MIPSI_SUBU, dest, RID_ZERO, left);
  ir--;
  dest = ra_dest(as, ir, RSET_GPR);
  left = ra_alloc1(as, ir->op1, RSET_GPR);
  emit_dst(as, MIPSI_SLTU, RID_TMP, RID_ZERO, dest);
  emit_dst(as, MIPSI_SUBU, dest, RID_ZERO, left);
}
#endif

static void asm_bnot(ASMState *as, IRIns *ir)
{
  Reg left, right, dest = ra_dest(as, ir, RSET_GPR);
  IRIns *irl = IR(ir->op1);
  if (mayfuse(as, ir->op1) && irl->o == IR_BOR) {
    left = ra_alloc2(as, irl, RSET_GPR);
    right = (left >> 8); left &= 255;
  } else {
    left = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
    right = RID_ZERO;
  }
  emit_dst(as, MIPSI_NOR, dest, left, right);
}

static void asm_bswap(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg left = ra_alloc1(as, ir->op1, RSET_GPR);
  if ((as->flags & JIT_F_MIPS32R2)) {
    emit_dta(as, MIPSI_ROTR, dest, RID_TMP, 16);
    emit_dst(as, MIPSI_WSBH, RID_TMP, 0, left);
  } else {
    Reg tmp = ra_scratch(as, rset_exclude(rset_exclude(RSET_GPR, left), dest));
    emit_dst(as, MIPSI_OR, dest, dest, tmp);
    emit_dst(as, MIPSI_OR, dest, dest, RID_TMP);
    emit_tsi(as, MIPSI_ANDI, dest, dest, 0xff00);
    emit_dta(as, MIPSI_SLL, RID_TMP, RID_TMP, 8);
    emit_dta(as, MIPSI_SRL, dest, left, 8);
    emit_tsi(as, MIPSI_ANDI, RID_TMP, left, 0xff00);
    emit_dst(as, MIPSI_OR, tmp, tmp, RID_TMP);
    emit_dta(as, MIPSI_SRL, tmp, left, 24);
    emit_dta(as, MIPSI_SLL, RID_TMP, left, 24);
  }
}

static void asm_bitop(ASMState *as, IRIns *ir, MIPSIns mi, MIPSIns mik)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg right, left = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
  if (irref_isk(ir->op2)) {
    int32_t k = IR(ir->op2)->i;
    if (checku16(k)) {
      emit_tsi(as, mik, dest, left, k);
      return;
    }
  }
  right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
  emit_dst(as, mi, dest, left, right);
}

#define asm_band(as, ir)        asm_bitop(as, ir, MIPSI_AND, MIPSI_ANDI)
#define asm_bor(as, ir)                asm_bitop(as, ir, MIPSI_OR, MIPSI_ORI)
#define asm_bxor(as, ir)        asm_bitop(as, ir, MIPSI_XOR, MIPSI_XORI)

static void asm_bitshift(ASMState *as, IRIns *ir, MIPSIns mi, MIPSIns mik)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  if (irref_isk(ir->op2)) {  /* Constant shifts. */
    uint32_t shift = (uint32_t)(IR(ir->op2)->i & 31);
    emit_dta(as, mik, dest, ra_hintalloc(as, ir->op1, dest, RSET_GPR), shift);
```

```
  } else {
    Reg right, left = ra_alloc2(as, ir, RSET_GPR);
    right = (left >> 8); left &= 255;
    emit_dst(as, mi, dest, right, left);  /* Shift amount is in rs. */
  }
}

#define asm_bshl(as, ir)        asm_bitshift(as, ir, MIPSI_SLLV, MIPSI_SLL)
#define asm_bshr(as, ir)        asm_bitshift(as, ir, MIPSI_SRLV, MIPSI_SRL)
#define asm_bsar(as, ir)        asm_bitshift(as, ir, MIPSI_SRAV, MIPSI_SRA)
#define asm_brol(as, ir)        lua_assert(0)

static void asm_bror(ASMState *as, IRIns *ir)
{
  if ((as->flags & JIT_F_MIPS32R2)) {
    asm_bitshift(as, ir, MIPSI_ROTRV, MIPSI_ROTR);
  } else {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    if (irref_isk(ir->op2)) {  /* Constant shifts. */
      uint32_t shift = (uint32_t)(IR(ir->op2)->i & 31);
      Reg left = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
      emit_rotr(as, dest, left, RID_TMP, shift);
    } else {
      Reg right, left = ra_alloc2(as, ir, RSET_GPR);
      right = (left >> 8); left &= 255;
      emit_dst(as, MIPSI_OR, dest, dest, RID_TMP);
      emit_dst(as, MIPSI_SRLV, dest, right, left);
      emit_dst(as, MIPSI_SLLV, RID_TMP, RID_TMP, left);
      emit_dst(as, MIPSI_SUBU, RID_TMP, ra_allock(as, 32, RSET_GPR), right);
    }
  }
}

static void asm_min_max(ASMState *as, IRIns *ir, int ismax)
{
  if (irt_isnum(ir->t)) {
    Reg dest = ra_dest(as, ir, RSET_FPR);
    Reg right, left = ra_alloc2(as, ir, RSET_FPR);
    right = (left >> 8); left &= 255;
    if (dest == left) {
      emit_fg(as, MIPSI_MOVT_D, dest, right);
    } else {
      emit_fg(as, MIPSI_MOVF_D, dest, left);
      if (dest != right) emit_fg(as, MIPSI_MOV_D, dest, right);
    }
    emit_fgh(as, MIPSI_C_OLT_D, 0, ismax ? left : right, ismax ? right : left);
  } else {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    Reg right, left = ra_alloc2(as, ir, RSET_GPR);
    right = (left >> 8); left &= 255;
    if (dest == left) {
      emit_dst(as, MIPSI_MOVN, dest, right, RID_TMP);
    } else {
      emit_dst(as, MIPSI_MOVZ, dest, left, RID_TMP);
      if (dest != right) emit_move(as, dest, right);
    }
    emit_dst(as, MIPSI_SLT, RID_TMP,
             ismax ? left : right, ismax ? right : left);
  }
}

#define asm_min(as, ir)                asm_min_max(as, ir, 0)
#define asm_max(as, ir)                asm_min_max(as, ir, 1)

/* -- Comparisons ---------------------------------------------------------- */

static void asm_comp(ASMState *as, IRIns *ir)
{
  /* ORDER IR: LT GE LE GT  ULT UGE ULE UGT. */
  IROp op = ir->o;
  if (irt_isnum(ir->t)) {
    Reg right, left = ra_alloc2(as, ir, RSET_FPR);
    right = (left >> 8); left &= 255;
    asm_guard(as, (op&1) ? MIPSI_BC1T : MIPSI_BC1F, 0, 0);
    emit_fgh(as, MIPSI_C_OLT_D + ((op&3) ^ ((op>>2)&1)), 0, left, right);
  } else {
```

```
    Reg right, left = ra_alloc1(as, ir->op1, RSET_GPR);
    if (op == IR_ABC) op = IR_UGT;
    if ((op&4) == 0 && irref_isk(ir->op2) && IR(ir->op2)->i == 0) {
      MIPSIns mi = (op&2) ? ((op&1) ? MIPSI_BLEZ : MIPSI_BGTZ) :
                            ((op&1) ? MIPSI_BLTZ : MIPSI_BGEZ);
      asm_guard(as, mi, left, 0);
    } else {
      if (irref_isk(ir->op2)) {
        int32_t k = IR(ir->op2)->i;
        if ((op&2)) k++;
        if (checki16(k)) {
          asm_guard(as, (op&1) ? MIPSI_BNE : MIPSI_BEQ, RID_TMP, RID_ZERO);
          emit_tsi(as, (op&4) ? MIPSI_SLTIU : MIPSI_SLTI,
                   RID_TMP, left, k);
          return;
        }
      }
      right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
      asm_guard(as, ((op^(op>>1))&1) ? MIPSI_BNE : MIPSI_BEQ, RID_TMP, RID_ZERO);
      emit_dst(as, (op&4) ? MIPSI_SLTU : MIPSI_SLT,
               RID_TMP, (op&2) ? right : left, (op&2) ? left : right);
    }
  }
}

static void asm_equal(ASMState *as, IRIns *ir)
{
  Reg right, left = ra_alloc2(as, ir, irt_isnum(ir->t) ? RSET_FPR : RSET_GPR);
  right = (left >> 8); left &= 255;
  if (irt_isnum(ir->t)) {
    asm_guard(as, (ir->o & 1) ? MIPSI_BC1T : MIPSI_BC1F, 0, 0);
    emit_fgh(as, MIPSI_C_EQ_D, 0, left, right);
  } else {
    asm_guard(as, (ir->o & 1) ? MIPSI_BEQ : MIPSI_BNE, left, right);
  }
}

#if LJ_HASFFI
/* 64 bit integer comparisons. */
static void asm_comp64(ASMState *as, IRIns *ir)
{
  /* ORDER IR: LT GE LE GT  ULT UGE ULE UGT. */
  IROp op = (ir-1)->o;
  MCLabel l_end;
  Reg rightlo, leftlo, righthi, lefthi = ra_alloc2(as, ir, RSET_GPR);
  righthi = (lefthi >> 8); lefthi &= 255;
  leftlo = ra_alloc2(as, ir-1,
                     rset_exclude(rset_exclude(RSET_GPR, lefthi), righthi));
  rightlo = (leftlo >> 8); leftlo &= 255;
  asm_guard(as, ((op^(op>>1))&1) ? MIPSI_BNE : MIPSI_BEQ, RID_TMP, RID_ZERO);
  l_end = emit_label(as);
  if (lefthi != righthi)
    emit_dst(as, (op&4) ? MIPSI_SLTU : MIPSI_SLT, RID_TMP,
             (op&2) ? righthi : lefthi, (op&2) ? lefthi : righthi);
  emit_dst(as, MIPSI_SLTU, RID_TMP,
           (op&2) ? rightlo : leftlo, (op&2) ? leftlo : rightlo);
  if (lefthi != righthi)
    emit_branch(as, MIPSI_BEQ, lefthi, righthi, l_end);
}

static void asm_comp64eq(ASMState *as, IRIns *ir)
{
  Reg tmp, right, left = ra_alloc2(as, ir, RSET_GPR);
  right = (left >> 8); left &= 255;
  asm_guard(as, ((ir-1)->o & 1) ? MIPSI_BEQ : MIPSI_BNE, RID_TMP, RID_ZERO);
  tmp = ra_scratch(as, rset_exclude(rset_exclude(RSET_GPR, left), right));
  emit_dst(as, MIPSI_OR, RID_TMP, RID_TMP, tmp);
  emit_dst(as, MIPSI_XOR, tmp, left, right);
  left = ra_alloc2(as, ir-1, RSET_GPR);
  right = (left >> 8); left &= 255;
  emit_dst(as, MIPSI_XOR, RID_TMP, left, right);
}
#endif

/* -- Support for 64 bit ops in 32 bit mode ------------------------------ */

```

```
/* Hiword op of a split 64 bit op. Previous op must be the loword op. */
static void asm_hiop(ASMState *as, IRIns *ir)
{
#if LJ_HASFFI
  /* HIOP is marked as a store because it needs its own DCE logic. */
  int uselo = ra_used(ir-1), usehi = ra_used(ir);  /* Loword/hiword used? */
  if (LJ_UNLIKELY(!(as->flags & JIT_F_OPT_DCE))) uselo = usehi = 1;
  if ((ir-1)->o == IR_CONV) {  /* Conversions to/from 64 bit. */
    as->curins--;  /* Always skip the CONV. */
    if (usehi || uselo)
      asm_conv64(as, ir);
    return;
  } else if ((ir-1)->o < IR_EQ) {  /* 64 bit integer comparisons. ORDER IR. */
    as->curins--;  /* Always skip the loword comparison. */
    asm_comp64(as, ir);
    return;
  } else if ((ir-1)->o <= IR_NE) {  /* 64 bit integer comparisons. ORDER IR. */
    as->curins--;  /* Always skip the loword comparison. */
    asm_comp64eq(as, ir);
    return;
  } else if ((ir-1)->o == IR_XSTORE) {
    as->curins--;  /* Handle both stores here. */
    if ((ir-1)->r != RID_SINK) {
      asm_xstore_(as, ir, LJ_LE ? 4 : 0);
      asm_xstore_(as, ir-1, LJ_LE ? 0 : 4);
    }
    return;
  }
  if (!usehi) return;  /* Skip unused hiword op for all remaining ops. */
  switch ((ir-1)->o) {
  case IR_ADD: as->curins--; asm_add64(as, ir); break;
  case IR_SUB: as->curins--; asm_sub64(as, ir); break;
  case IR_NEG: as->curins--; asm_neg64(as, ir); break;
  case IR_CALLN:
  case IR_CALLXS:
    if (!uselo)
      ra_allocref(as, ir->op1, RID2RSET(RID_RETLO));  /* Mark lo op as used. */
    break;
  case IR_CNEWI:
    /* Nothing to do here. Handled by lo op itself. */
    break;
  default: lua_assert(0); break;
  }
#else
  UNUSED(as); UNUSED(ir); lua_assert(0);  /* Unused without FFI. */
#endif
}

/* -- Profiling ----------------------------------------------------------- */

static void asm_prof(ASMState *as, IRIns *ir)
{
  UNUSED(ir);
  asm_guard(as, MIPSI_BNE, RID_TMP, RID_ZERO);
  emit_tsi(as, MIPSI_ANDI, RID_TMP, RID_TMP, HOOK_PROFILE);
  emit_lsglptr(as, MIPSI_LBU, RID_TMP,
               (int32_t)offsetof(global_State, hookmask));
}

/* -- Stack handling ------------------------------------------------------ */

/* Check Lua stack size for overflow. Use exit handler as fallback. */
static void asm_stack_check(ASMState *as, BCReg topslot,
                            IRIns *irp, RegSet allow, ExitNo exitno)
{
  /* Try to get an unused temp. register, otherwise spill/restore RID_RET*. */
  Reg tmp, pbase = irp ? (ra_hasreg(irp->r) ? irp->r : RID_TMP) : RID_BASE;
  ExitNo oldsnap = as->snapno;
  rset_clear(allow, pbase);
  tmp = allow ? rset_pickbot(allow) :
                (pbase == RID_RETHI ? RID_RETLO : RID_RETHI);
  as->snapno = exitno;
  asm_guard(as, MIPSI_BNE, RID_TMP, RID_ZERO);
  as->snapno = oldsnap;
  if (allow == RSET_EMPTY)  /* Restore temp. register. */
    emit_tsi(as, MIPSI_LW, tmp, RID_SP, 0);
```

```
  else
    ra_modified(as, tmp);
  emit_tsi(as, MIPSI_SLTIU, RID_TMP, RID_TMP, (int32_t)(8*topslot));
  emit_dst(as, MIPSI_SUBU, RID_TMP, tmp, pbase);
  emit_tsi(as, MIPSI_LW, tmp, tmp, offsetof(lua_State, maxstack));
  if (pbase == RID_TMP)
    emit_getgl(as, RID_TMP, jit_base);
  emit_getgl(as, tmp, cur_L);
  if (allow == RSET_EMPTY)  /* Spill temp. register. */
    emit_tsi(as, MIPSI_SW, tmp, RID_SP, 0);
}

/* Restore Lua stack from on-trace state. */
static void asm_stack_restore(ASMState *as, SnapShot *snap)
{
  SnapEntry *map = &as->T->snapmap[snap->mapofs];
  SnapEntry *flinks = &as->T->snapmap[snap_nextofs(as->T, snap)-1];
  MSize n, nent = snap->nent;
  /* Store the value of all modified slots to the Lua stack. */
  for (n = 0; n < nent; n++) {
    SnapEntry sn = map[n];
    BCReg s = snap_slot(sn);
    int32_t ofs = 8*((int32_t)s-1);
    IRRef ref = snap_ref(sn);
    IRIns *ir = IR(ref);
    if ((sn & SNAP_NORESTORE))
      continue;
    if (irt_isnum(ir->t)) {
      Reg src = ra_alloc1(as, ref, RSET_FPR);
      emit_hsi(as, MIPSI_SDC1, src, RID_BASE, ofs);
    } else {
      Reg type;
      RegSet allow = rset_exclude(RSET_GPR, RID_BASE);
      lua_assert(irt_ispri(ir->t) || irt_isaddr(ir->t) || irt_isinteger(ir->t));
      if (!irt_ispri(ir->t)) {
        Reg src = ra_alloc1(as, ref, allow);
        rset_clear(allow, src);
        emit_tsi(as, MIPSI_SW, src, RID_BASE, ofs+(LJ_BE?4:0));
      }
      if ((sn & (SNAP_CONT|SNAP_FRAME))) {
        if (s == 0) continue;  /* Do not overwrite link to previous frame. */
        type = ra_allock(as, (int32_t)(*flinks--), allow);
      } else {
        type = ra_allock(as, (int32_t)irt_toitype(ir->t), allow);
      }
      emit_tsi(as, MIPSI_SW, type, RID_BASE, ofs+(LJ_BE?0:4));
    }
    checkmclim(as);
  }
  lua_assert(map + nent == flinks);
}

/* -- GC handling --------------------------------------------------------- */

/* Check GC threshold and do one or more GC steps. */
static void asm_gc_check(ASMState *as)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_gc_step_jit];
  IRRef args[2];
  MCLabel l_end;
  Reg tmp;
  ra_evictset(as, RSET_SCRATCH);
  l_end = emit_label(as);
  /* Exit trace if in GCSatomic or GCSfinalize. Avoids syncing GC objects. */
  /* Assumes asm_snap_prep() already done. */
  asm_guard(as, MIPSI_BNE, RID_RET, RID_ZERO);
  args[0] = ASMREF_TMP1;  /* global_State *g */
  args[1] = ASMREF_TMP2;  /* MSize steps     */
  asm_gencall(as, ci, args);
  emit_tsi(as, MIPSI_ADDIU, ra_releasetmp(as, ASMREF_TMP1), RID_JGL, -32768);
  tmp = ra_releasetmp(as, ASMREF_TMP2);
  emit_loadi(as, tmp, as->gcsteps);
  /* Jump around GC step if GC total < GC threshold. */
  emit_branch(as, MIPSI_BNE, RID_TMP, RID_ZERO, l_end);
  emit_dst(as, MIPSI_SLTU, RID_TMP, RID_TMP, tmp);
  emit_getgl(as, tmp, gc.threshold);
```

```
  emit_getgl(as, RID_TMP, gc.total);
  as->gcsteps = 0;
  checkmclim(as);
}

/* -- Loop handling ------------------------------------------------------- */

/* Fixup the loop branch. */
static void asm_loop_fixup(ASMState *as)
{
  MCode *p = as->mctop;
  MCode *target = as->mcp;
  p[-1] = MIPSI_NOP;
  if (as->loopinv) {  /* Inverted loop branch? */
    /* asm_guard already inverted the cond branch. Only patch the target. */
    p[-3] |= ((target-p+2) & 0x0000ffffu);
  } else {
    p[-2] = MIPSI_J|(((uintptr_t)target>>2)&0x03ffffffu);
  }
}

/* -- Head of trace ------------------------------------------------------- */

/* Coalesce BASE register for a root trace. */
static void asm_head_root_base(ASMState *as)
{
  IRIns *ir = IR(REF_BASE);
  Reg r = ir->r;
  if (as->loopinv) as->mctop--;
  if (ra_hasreg(r)) {
    ra_free(as, r);
    if (rset_test(as->modset, r) || irt_ismarked(ir->t))
      ir->r = RID_INIT;  /* No inheritance for modified BASE register. */
    if (r != RID_BASE)
      emit_move(as, r, RID_BASE);
  }
}

/* Coalesce BASE register for a side trace. */
static RegSet asm_head_side_base(ASMState *as, IRIns *irp, RegSet allow)
{
  IRIns *ir = IR(REF_BASE);
  Reg r = ir->r;
  if (as->loopinv) as->mctop--;
  if (ra_hasreg(r)) {
    ra_free(as, r);
    if (rset_test(as->modset, r) || irt_ismarked(ir->t))
      ir->r = RID_INIT;  /* No inheritance for modified BASE register. */
    if (irp->r == r) {
      rset_clear(allow, r);  /* Mark same BASE register as coalesced. */
    } else if (ra_hasreg(irp->r) && rset_test(as->freeset, irp->r)) {
      rset_clear(allow, irp->r);
      emit_move(as, r, irp->r);  /* Move from coalesced parent reg. */
    } else {
      emit_getgl(as, r, jit_base);  /* Otherwise reload BASE. */
    }
  }
  return allow;
}

/* -- Tail of trace ------------------------------------------------------- */

/* Fixup the tail code. */
static void asm_tail_fixup(ASMState *as, TraceNo lnk)
{
  MCode *target = lnk ? traceref(as->J,lnk)->mcode : (MCode *)lj_vm_exit_interp;
  int32_t spadj = as->T->spadjust;
  MCode *p = as->mctop-1;
  *p = spadj ? (MIPSI_ADDIU|MIPSF_T(RID_SP)|MIPSF_S(RID_SP)|spadj) : MIPSI_NOP;
  p[-1] = MIPSI_J|(((uintptr_t)target>>2)&0x03ffffffu);
}

/* Prepare tail of code. */
static void asm_tail_prep(ASMState *as)
{
  as->mcp = as->mctop-2;  /* Leave room for branch plus nop or stack adj. */
```

```
  as->invmcp = as->loopref ? as->mcp : NULL;
}

/* -- Trace setup ----------------------------------------------------------- */

/* Ensure there are enough stack slots for call arguments. */
static Reg asm_setup_call_slots(ASMState *as, IRIns *ir, const CCallInfo *ci)
{
  IRRef args[CCI_NARGS_MAX*2];
  uint32_t i, nargs = CCI_XNARGS(ci);
  int nslots = 4, ngpr = REGARG_NUMGPR, nfpr = REGARG_NUMFPR;
  asm_collectargs(as, ir, ci, args);
  for (i = 0; i < nargs; i++) {
    if (args[i] && irt_isfp(IR(args[i])->t) &&
        nfpr > 0 && !(ci->flags & CCI_VARARG)) {
      nfpr--;
      ngpr -= irt_isnum(IR(args[i])->t) ? 2 : 1;
    } else if (args[i] && irt_isnum(IR(args[i])->t)) {
      nfpr = 0;
      ngpr = ngpr & ~1;
      if (ngpr > 0) ngpr -= 2; else nslots = (nslots+3) & ~1;
    } else {
      nfpr = 0;
      if (ngpr > 0) ngpr--; else nslots++;
    }
  }
  if (nslots > as->evenspill)  /* Leave room for args in stack slots. */
    as->evenspill = nslots;
  return irt_isfp(ir->t) ? REGSP_HINT(RID_FPRET) : REGSP_HINT(RID_RET);
}

static void asm_setup_target(ASMState *as)
{
  asm_sparejump_setup(as);
  asm_exitstub_setup(as);
}

/* -- Trace patching -------------------------------------------------------- */

/* Patch exit jumps of existing machine code to a new target. */
void lj_asm_patchexit(jit_State *J, GCtrace *T, ExitNo exitno, MCode *target)
{
  MCode *p = T->mcode;
  MCode *pe = (MCode *)((char *)p + T->szmcode);
  MCode *px = exitstub_trace_addr(T, exitno);
  MCode *cstart = NULL, *cstop = NULL;
  MCode *mcarea = lj_mcode_patch(J, p, 0);
  MCode exitload = MIPSI_LI | MIPSF_T(RID_TMP) | exitno;
  MCode tjump = MIPSI_J|(((uintptr_t)target>>2)&0x03ffffffu);
  for (p++; p < pe; p++) {
    if (*p == exitload) {  /* Look for load of exit number. */
      if (((p[-1] ^ (px-p)) & 0xffffu) == 0) {  /* Look for exitstub branch. */
        ptrdiff_t delta = target - p;
        if (((delta + 0x8000) >> 16) == 0) {  /* Patch in-range branch. */
        patchbranch:
          p[-1] = (p[-1] & 0xffff0000u) | (delta & 0xffffu);
          *p = MIPSI_NOP;  /* Replace the load of the exit number. */
          cstop = p;
          if (!cstart) cstart = p-1;
        } else {  /* Branch out of range. Use spare jump slot in mcarea. */
          int i;
          for (i = 2; i < 2+MIPS_SPAREJUMP*2; i += 2) {
            if (mcarea[i] == tjump) {
              delta = mcarea+i - p;
              goto patchbranch;
            } else if (mcarea[i] == MIPSI_NOP) {
              mcarea[i] = tjump;
              cstart = mcarea+i;
              delta = mcarea+i - p;
              goto patchbranch;
            }
          }
          /* Ignore jump slot overflow. Child trace is simply not attached. */
        }
      } else if (p+1 == pe) {
        /* Patch NOP after code for inverted loop branch. Use of J is ok. */
```

```
1822          lua_assert(p[1] == MIPSI_NOP);
1823          p[1] = tjump;
1824          *p = MIPSI_NOP;  /* Replace the load of the exit number. */
1825          cstop = p+2;
1826          if (!cstart) cstart = p+1;
1827        }
1828      }
1829    }
1830    if (cstart) lj_mcode_sync(cstart, cstop);
1831    lj_mcode_patch(J, mcarea, 1);
1832  }
1833
```

One Level Up Top Level

src/lj_asm_ppc.h - luajit-2.0-src

Global variables defined

- asm_compmap

Functions defined

- asm_add
- asm_add64
- asm_ahustore
- asm_ahuvload
- asm_aref
- asm_arithov
- asm_band
- asm_bitop
- asm_bitshift
- asm_bnot
- asm_bswap
- asm_callx
- asm_cnew
- asm_comp
- asm_comp64
- asm_conv
- asm_exitstub_addr
- asm_exitstub_setup
- asm_fload
- asm_fparith
- asm_fpmath
- asm_fpunary
- asm_fref
- asm_fstore
- asm_fuseabase
- asm_fuseahuref
- asm_fuseandsh

- asm_fusemadd
- asm_fusexref
- asm_fusexrefx
- asm_fxloadins
- asm_fxstoreins
- asm_gc_check
- asm_gencall
- asm_guardcc
- asm_head_root_base
- asm_head_side_base
- asm_hiop
- asm_href
- asm_hrefk
- asm_intcomp_
- asm_loop_fixup
- asm_min_max
- asm_mul
- asm_neg
- asm_neg64
- asm_obar
- asm_prof
- asm_retf
- asm_setup_call_slots
- asm_setup_target
- asm_setupresult
- asm_sload
- asm_stack_check
- asm_stack_restore
- asm_strref
- asm_strto
- asm_sub
- asm_sub64
- asm_tail_fixup

- asm_tail_prep
- asm_tbar
- asm_tobit
- asm_tointg
- asm_tvptr
- asm_uref
- asm_xload
- asm_xstore_
- lj_asm_patchexit
- noconflict
- ra_alloc2
- ra_hintalloc

Macros defined

- AHUREF_LSX
- CC_TWO
- CC_UNSIGNED
- CONFLICT_SEARCH_LIM
- asm_abs
- asm_addov
- asm_atan2
- asm_bor
- asm_brol
- asm_bror
- asm_bsar
- asm_bshl
- asm_bshr
- asm_bxor
- asm_cnew
- asm_div
- asm_equal
- asm_ldexp
- asm_max
- asm_min

- asm_mod
- asm_mulov
- asm_pow
- asm_subov
- asm_xstore

Source code

```
/*
** PPC IR assembler (SSA IR -> machine code).
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

/* -- Register allocator extensions --------------------------------------- */

/* Allocate a register with a hint. */
static Reg ra_hintalloc(ASMState *as, IRRef ref, Reg hint, RegSet allow)
{
  Reg r = IR(ref)->r;
  if (ra_noreg(r)) {
    if (!ra_hashint(r) && !iscrossref(as, ref))
      ra_sethint(IR(ref)->r, hint);  /* Propagate register hint. */
    r = ra_allocref(as, ref, allow);
  }
  ra_noweak(as, r);
  return r;
}

/* Allocate two source registers for three-operand instructions. */
static Reg ra_alloc2(ASMState *as, IRIns *ir, RegSet allow)
{
  IRIns *irl = IR(ir->op1), *irr = IR(ir->op2);
  Reg left = irl->r, right = irr->r;
  if (ra_hasreg(left)) {
    ra_noweak(as, left);
    if (ra_noreg(right))
      right = ra_allocref(as, ir->op2, rset_exclude(allow, left));
    else
      ra_noweak(as, right);
  } else if (ra_hasreg(right)) {
    ra_noweak(as, right);
    left = ra_allocref(as, ir->op1, rset_exclude(allow, right));
  } else if (ra_hashint(right)) {
    right = ra_allocref(as, ir->op2, allow);
    left = ra_alloc1(as, ir->op1, rset_exclude(allow, right));
  } else {
    left = ra_allocref(as, ir->op1, allow);
    right = ra_alloc1(as, ir->op2, rset_exclude(allow, left));
  }
  return left | (right << 8);
}

/* -- Guard handling ------------------------------------------------------ */

/* Setup exit stubs after the end of each trace. */
static void asm_exitstub_setup(ASMState *as, ExitNo nexits)
{
  ExitNo i;
  MCode *mxp = as->mctop;
  if (mxp - (nexits + 3 + MCLIM_REDZONE) < as->mclim)
    asm_mclimit(as);
  /* 1: mflr r0; bl ->vm_exit_handler; li r0, traceno; bl <1; bl <1; ... */
  for (i = nexits-1; (int32_t)i >= 0; i--)
    *--mxp = PPCI_BL|(((-3-i)&0x00ffffffu)<<2);
  *--mxp = PPCI_LI|PPCF_T(RID_TMP)|as->T->traceno;  /* Read by exit handler. */
  mxp--;
  *mxp = PPCI_BL|((((MCode *)(void *)lj_vm_exit_handler-mxp)&0x00ffffffu)<<2);
  *--mxp = PPCI_MFLR|PPCF_T(RID_TMP);
  as->mctop = mxp;
```

```
}

static MCode *asm_exitstub_addr(ASMState *as, ExitNo exitno)
{
  /* Keep this in-sync with exitstub_trace_addr(). */
  return as->mctop + exitno + 3;
}

/* Emit conditional branch to exit for guard. */
static void asm_guardcc(ASMState *as, PPCCC cc)
{
  MCode *target = asm_exitstub_addr(as, as->snapno);
  MCode *p = as->mcp;
  if (LJ_UNLIKELY(p == as->invmcp)) {
    as->loopinv = 1;
    *p = PPCI_B | (((target-p) & 0x00ffffffu) << 2);
    emit_condbranch(as, PPCI_BC, cc^4, p);
    return;
  }
  emit_condbranch(as, PPCI_BC, cc, target);
}

/* -- Operand fusion ------------------------------------------------------ */

/* Limit linear search to this distance. Avoids O(n^2) behavior. */
#define CONFLICT_SEARCH_LIM        31

/* Check if there's no conflicting instruction between curins and ref. */
static int noconflict(ASMState *as, IRRef ref, IROp conflict)
{
  IRIns *ir = as->ir;
  IRRef i = as->curins;
  if (i > ref + CONFLICT_SEARCH_LIM)
    return 0;  /* Give up, ref is too far away. */
  while (--i > ref)
    if (ir[i].o == conflict)
      return 0;  /* Conflict found. */
  return 1;  /* Ok, no conflict. */
}

/* Fuse the array base of colocated arrays. */
static int32_t asm_fuseabase(ASMState *as, IRRef ref)
{
  IRIns *ir = IR(ref);
  if (ir->o == IR_TNEW && ir->op1 <= LJ_MAX_COLOSIZE &&
      !neverfuse(as) && noconflict(as, ref, IR_NEWREF))
    return (int32_t)sizeof(GCtab);
  return 0;
}

/* Indicates load/store indexed is ok. */
#define AHUREF_LSX        ((int32_t)0x80000000)

/* Fuse array/hash/upvalue reference into register+offset operand. */
static Reg asm_fuseahuref(ASMState *as, IRRef ref, int32_t *ofsp, RegSet allow)
{
  IRIns *ir = IR(ref);
  if (ra_noreg(ir->r)) {
    if (ir->o == IR_AREF) {
      if (mayfuse(as, ref)) {
        if (irref_isk(ir->op2)) {
          IRRef tab = IR(ir->op1)->op1;
          int32_t ofs = asm_fuseabase(as, tab);
          IRRef refa = ofs ? tab : ir->op1;
          ofs += 8*IR(ir->op2)->i;
          if (checki16(ofs)) {
            *ofsp = ofs;
            return ra_alloc1(as, refa, allow);
          }
        }
        if (*ofsp == AHUREF_LSX) {
          Reg base = ra_alloc1(as, ir->op1, allow);
          Reg idx = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, base));
          return base | (idx << 8);
        }
      }
```

```
    } else if (ir->o == IR_HREFK) {
      if (mayfuse(as, ref)) {
        int32_t ofs = (int32_t)(IR(ir->op2)->op2 * sizeof(Node));
        if (checki16(ofs)) {
          *ofsp = ofs;
          return ra_alloc1(as, ir->op1, allow);
        }
      }
    } else if (ir->o == IR_UREFC) {
      if (irref_isk(ir->op1)) {
        GCfunc *fn = ir_kfunc(IR(ir->op1));
        int32_t ofs = i32ptr(&gcref(fn->l.uvptr[(ir->op2 >> 8)])->uv.tv);
        int32_t jgl = (intptr_t)J2G(as->J);
        if ((uint32_t)(ofs-jgl) < 65536) {
          *ofsp = ofs-jgl-32768;
          return RID_JGL;
        } else {
          *ofsp = (int16_t)ofs;
          return ra_allock(as, ofs-(int16_t)ofs, allow);
        }
      }
    }
  }
  *ofsp = 0;
  return ra_alloc1(as, ref, allow);
}

/* Fuse XLOAD/XSTORE reference into load/store operand. */
static void asm_fusexref(ASMState *as, PPCIns pi, Reg rt, IRRef ref,
                         RegSet allow, int32_t ofs)
{
  IRIns *ir = IR(ref);
  Reg base;
  if (ra_noreg(ir->r) && canfuse(as, ir)) {
    if (ir->o == IR_ADD) {
      int32_t ofs2;
      if (irref_isk(ir->op2) && (ofs2 = ofs + IR(ir->op2)->i, checki16(ofs2))) {
        ofs = ofs2;
        ref = ir->op1;
      } else if (ofs == 0) {
        Reg right, left = ra_alloc2(as, ir, allow);
        right = (left >> 8); left &= 255;
        emit_fab(as, PPCI_LWZX | ((pi >> 20) & 0x780), rt, left, right);
        return;
      }
    } else if (ir->o == IR_STRREF) {
      lua_assert(ofs == 0);
      ofs = (int32_t)sizeof(GCstr);
      if (irref_isk(ir->op2)) {
        ofs += IR(ir->op2)->i;
        ref = ir->op1;
      } else if (irref_isk(ir->op1)) {
        ofs += IR(ir->op1)->i;
        ref = ir->op2;
      } else {
        /* NYI: Fuse ADD with constant. */
        Reg tmp, right, left = ra_alloc2(as, ir, allow);
        right = (left >> 8); left &= 255;
        tmp = ra_scratch(as, rset_exclude(rset_exclude(allow, left), right));
        emit_fai(as, pi, rt, tmp, ofs);
        emit_tab(as, PPCI_ADD, tmp, left, right);
        return;
      }
      if (!checki16(ofs)) {
        Reg left = ra_alloc1(as, ref, allow);
        Reg right = ra_allock(as, ofs, rset_exclude(allow, left));
        emit_fab(as, PPCI_LWZX | ((pi >> 20) & 0x780), rt, left, right);
        return;
      }
    }
  }
  base = ra_alloc1(as, ref, allow);
  emit_fai(as, pi, rt, base, ofs);
}

/* Fuse XLOAD/XSTORE reference into indexed-only load/store operand. */
```

```
static void asm_fusexrefx(ASMState *as, PPCIns pi, Reg rt, IRRef ref,
                          RegSet allow)
{
  IRIns *ira = IR(ref);
  Reg right, left;
  if (canfuse(as, ira) && ira->o == IR_ADD && ra_noreg(ira->r)) {
    left = ra_alloc2(as, ira, allow);
    right = (left >> 8); left &= 255;
  } else {
    right = ra_alloc1(as, ref, allow);
    left = RID_R0;
  }
  emit_tab(as, pi, rt, left, right);
}

/* Fuse to multiply-add/sub instruction. */
static int asm_fusemadd(ASMState *as, IRIns *ir, PPCIns pi, PPCIns pir)
{
  IRRef lref = ir->op1, rref = ir->op2;
  IRIns *irm;
  if (lref != rref &&
      ((mayfuse(as, lref) && (irm = IR(lref), irm->o == IR_MUL) &&
        ra_noreg(irm->r)) ||
       (mayfuse(as, rref) && (irm = IR(rref), irm->o == IR_MUL) &&
        (rref = lref, pi = pir, ra_noreg(irm->r))))) {
    Reg dest = ra_dest(as, ir, RSET_FPR);
    Reg add = ra_alloc1(as, rref, RSET_FPR);
    Reg right, left = ra_alloc2(as, irm, rset_exclude(RSET_FPR, add));
    right = (left >> 8); left &= 255;
    emit_facb(as, pi, dest, left, right, add);
    return 1;
  }
  return 0;
}

/* -- Calls ------------------------------------------------------------ */

/* Generate a call to a C function. */
static void asm_gencall(ASMState *as, const CCallInfo *ci, IRRef *args)
{
  uint32_t n, nargs = CCI_XNARGS(ci);
  int32_t ofs = 8;
  Reg gpr = REGARG_FIRSTGPR, fpr = REGARG_FIRSTFPR;
  if ((void *)ci->func)
    emit_call(as, (void *)ci->func);
  for (n = 0; n < nargs; n++) {  /* Setup args. */
    IRRef ref = args[n];
    if (ref) {
      IRIns *ir = IR(ref);
      if (irt_isfp(ir->t)) {
        if (fpr <= REGARG_LASTFPR) {
          lua_assert(rset_test(as->freeset, fpr));  /* Already evicted. */
          ra_leftov(as, fpr, ref);
          fpr++;
        } else {
          Reg r = ra_alloc1(as, ref, RSET_FPR);
          if (irt_isnum(ir->t)) ofs = (ofs + 4) & ~4;
          emit_spstore(as, ir, r, ofs);
          ofs += irt_isnum(ir->t) ? 8 : 4;
        }
      } else {
        if (gpr <= REGARG_LASTGPR) {
          lua_assert(rset_test(as->freeset, gpr));  /* Already evicted. */
          ra_leftov(as, gpr, ref);
          gpr++;
        } else {
          Reg r = ra_alloc1(as, ref, RSET_GPR);
          emit_spstore(as, ir, r, ofs);
          ofs += 4;
        }
      }
    } else {
      if (gpr <= REGARG_LASTGPR)
        gpr++;
      else
        ofs += 4;
```

```
290     }
291     checkmclim(as);
292   }
293   if ((ci->flags & CCI_VARARG))  /* Vararg calls need to know about FPR use. */
294     emit_tab(as, fpr == REGARG_FIRSTFPR ? PPCI_CRXOR : PPCI_CREQV, 6, 6, 6);
295 }
296
297 /* Setup result reg/sp for call. Evict scratch regs. */
298 static void asm_setupresult(ASMState *as, IRIns *ir, const CCallInfo *ci)
299 {
300   RegSet drop = RSET_SCRATCH;
301   int hiop = ((ir+1)->o == IR_HIOP);
302   if ((ci->flags & CCI_NOFPRCLOBBER))
303     drop &= ~RSET_FPR;
304   if (ra_hasreg(ir->r))
305     rset_clear(drop, ir->r);  /* Dest reg handled below. */
306   if (hiop && ra_hasreg((ir+1)->r))
307     rset_clear(drop, (ir+1)->r);  /* Dest reg handled below. */
308   ra_evictset(as, drop);  /* Evictions must be performed first. */
309   if (ra_used(ir)) {
310     lua_assert(!irt_ispri(ir->t));
311     if (irt_isfp(ir->t)) {
312       if ((ci->flags & CCI_CASTU64)) {
313         /* Use spill slot or temp slots. */
314         int32_t ofs = ir->s ? sps_scale(ir->s) : SPOFS_TMP;
315         Reg dest = ir->r;
316         if (ra_hasreg(dest)) {
317           ra_free(as, dest);
318           ra_modified(as, dest);
319           emit_fai(as, PPCI_LFD, dest, RID_SP, ofs);
320         }
321         emit_tai(as, PPCI_STW, RID_RETHI, RID_SP, ofs);
322         emit_tai(as, PPCI_STW, RID_RETLO, RID_SP, ofs+4);
323       } else {
324         ra_destreg(as, ir, RID_FPRET);
325       }
326 #if LJ_32
327     } else if (hiop) {
328       ra_destpair(as, ir);
329 #endif
330     } else {
331       ra_destreg(as, ir, RID_RET);
332     }
333   }
334 }
335
336 static void asm_callx(ASMState *as, IRIns *ir)
337 {
338   IRRef args[CCI_NARGS_MAX*2];
339   CCallInfo ci;
340   IRRef func;
341   IRIns *irf;
342   ci.flags = asm_callx_flags(as, ir);
343   asm_collectargs(as, ir, &ci, args);
344   asm_setupresult(as, ir, &ci);
345   func = ir->op2; irf = IR(func);
346   if (irf->o == IR_CARG) { func = irf->op1; irf = IR(func); }
347   if (irref_isk(func)) {  /* Call to constant address. */
348     ci.func = (ASMFunction)(void *)(intptr_t)(irf->i);
349   } else {  /* Need a non-argument register for indirect calls. */
350     RegSet allow = RSET_GPR & ~RSET_RANGE(RID_R0, REGARG_LASTGPR+1);
351     Reg freg = ra_alloc1(as, func, allow);
352     *--as->mcp = PPCI_BCTRL;
353     *--as->mcp = PPCI_MTCTR | PPCF_T(freg);
354     ci.func = (ASMFunction)(void *)0;
355   }
356   asm_gencall(as, &ci, args);
357 }
358
359 /* -- Returns ----------------------------------------------------------- */
360
361 /* Return to lower frame. Guard that it goes to the right spot. */
362 static void asm_retf(ASMState *as, IRIns *ir)
363 {
364   Reg base = ra_alloc1(as, REF_BASE, RSET_GPR);
365   void *pc = ir_kptr(IR(ir->op2));
```

```
  int32_t delta = 1+LJ_FR2+bc_a(*((const BCIns *)pc - 1));
  as->topslot -= (BCReg)delta;
  if ((int32_t)as->topslot < 0) as->topslot = 0;
  irt_setmark(IR(REF_BASE)->t);  /* Children must not coalesce with BASE reg. */
  emit_setgl(as, base, jit_base);
  emit_addptr(as, base, -8*delta);
  asm_guardcc(as, CC_NE);
  emit_ab(as, PPCI_CMPW, RID_TMP,
          ra_allock(as, i32ptr(pc), rset_exclude(RSET_GPR, base)));
  emit_tai(as, PPCI_LWZ, RID_TMP, base, -8);
}

/* -- Type conversions ---------------------------------------------------- */

static void asm_tointg(ASMState *as, IRIns *ir, Reg left)
{
  RegSet allow = RSET_FPR;
  Reg tmp = ra_scratch(as, rset_clear(allow, left));
  Reg fbias = ra_scratch(as, rset_clear(allow, tmp));
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg hibias = ra_allock(as, 0x43300000, rset_exclude(RSET_GPR, dest));
  asm_guardcc(as, CC_NE);
  emit_fab(as, PPCI_FCMPU, 0, tmp, left);
  emit_fab(as, PPCI_FSUB, tmp, tmp, fbias);
  emit_fai(as, PPCI_LFD, tmp, RID_SP, SPOFS_TMP);
  emit_tai(as, PPCI_STW, RID_TMP, RID_SP, SPOFS_TMPLO);
  emit_tai(as, PPCI_STW, hibias, RID_SP, SPOFS_TMPHI);
  emit_asi(as, PPCI_XORIS, RID_TMP, dest, 0x8000);
  emit_tai(as, PPCI_LWZ, dest, RID_SP, SPOFS_TMPLO);
  emit_lsptr(as, PPCI_LFS, (fbias & 31),
             (void *)lj_ir_k64_find(as->J, U64x(59800004,59800000)),
             RSET_GPR);
  emit_fai(as, PPCI_STFD, tmp, RID_SP, SPOFS_TMP);
  emit_fb(as, PPCI_FCTIWZ, tmp, left);
}

static void asm_tobit(ASMState *as, IRIns *ir)
{
  RegSet allow = RSET_FPR;
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg left = ra_alloc1(as, ir->op1, allow);
  Reg right = ra_alloc1(as, ir->op2, rset_clear(allow, left));
  Reg tmp = ra_scratch(as, rset_clear(allow, right));
  emit_tai(as, PPCI_LWZ, dest, RID_SP, SPOFS_TMPLO);
  emit_fai(as, PPCI_STFD, tmp, RID_SP, SPOFS_TMP);
  emit_fab(as, PPCI_FADD, tmp, left, right);
}

static void asm_conv(ASMState *as, IRIns *ir)
{
  IRType st = (IRType)(ir->op2 & IRCONV_SRCMASK);
  int stfp = (st == IRT_NUM || st == IRT_FLOAT);
  IRRef lref = ir->op1;
  lua_assert(irt_type(ir->t) != st);
  lua_assert(!(irt_isint64(ir->t) ||
               (st == IRT_I64 || st == IRT_U64)));  /* Handled by SPLIT. */
  if (irt_isfp(ir->t)) {
    Reg dest = ra_dest(as, ir, RSET_FPR);
    if (stfp) {  /* FP to FP conversion. */
      if (st == IRT_NUM)  /* double -> float conversion. */
        emit_fb(as, PPCI_FRSP, dest, ra_alloc1(as, lref, RSET_FPR));
      else  /* float -> double conversion is a no-op on PPC. */
        ra_leftov(as, dest, lref);  /* Do nothing, but may need to move regs. */
    } else {  /* Integer to FP conversion. */
      /* IRT_INT: Flip hibit, bias with 2^52, subtract 2^52+2^31. */
      /* IRT_U32: Bias with 2^52, subtract 2^52. */
      RegSet allow = RSET_GPR;
      Reg left = ra_alloc1(as, lref, allow);
      Reg hibias = ra_allock(as, 0x43300000, rset_clear(allow, left));
      Reg fbias = ra_scratch(as, rset_exclude(RSET_FPR, dest));
      const float *kbias;
      if (irt_isfloat(ir->t)) emit_fb(as, PPCI_FRSP, dest, dest);
      emit_fab(as, PPCI_FSUB, dest, dest, fbias);
      emit_fai(as, PPCI_LFD, dest, RID_SP, SPOFS_TMP);
      kbias = (const float *)lj_ir_k64_find(as->J, U64x(59800004,59800000));
      if (st == IRT_U32) kbias++;
```

```
      emit_lsptr(as, PPCI_LFS, (fbias & 31), (void *)kbias,
                 rset_clear(allow, hibias));
      emit_tai(as, PPCI_STW, st == IRT_U32 ? left : RID_TMP,
               RID_SP, SPOFS_TMPLO);
      emit_tai(as, PPCI_STW, hibias, RID_SP, SPOFS_TMPHI);
      if (st != IRT_U32) emit_asi(as, PPCI_XORIS, RID_TMP, left, 0x8000);
    }
  } else if (stfp) {  /* FP to integer conversion. */
    if (irt_isguard(ir->t)) {
      /* Checked conversions are only supported from number to int. */
      lua_assert(irt_isint(ir->t) && st == IRT_NUM);
      asm_tointg(as, ir, ra_alloc1(as, lref, RSET_FPR));
    } else {
      Reg dest = ra_dest(as, ir, RSET_GPR);
      Reg left = ra_alloc1(as, lref, RSET_FPR);
      Reg tmp = ra_scratch(as, rset_exclude(RSET_FPR, left));
      if (irt_isu32(ir->t)) {
        /* Convert both x and x-2^31 to int and merge results. */
        Reg tmpi = ra_scratch(as, rset_exclude(RSET_GPR, dest));
        emit_asb(as, PPCI_OR, dest, dest, tmpi);  /* Select with mask idiom. */
        emit_asb(as, PPCI_AND, tmpi, tmpi, RID_TMP);
        emit_asb(as, PPCI_ANDC, dest, dest, RID_TMP);
        emit_tai(as, PPCI_LWZ, tmpi, RID_SP, SPOFS_TMPLO);  /* tmp = (int)(x) */
        emit_tai(as, PPCI_ADDIS, dest, dest, 0x8000);  /* dest += 2^31 */
        emit_asb(as, PPCI_SRAWI, RID_TMP, dest, 31);  /* mask = -(dest < 0) */
        emit_fai(as, PPCI_STFD, tmp, RID_SP, SPOFS_TMP);
        emit_tai(as, PPCI_LWZ, dest,
                 RID_SP, SPOFS_TMPLO);  /* dest = (int)(x-2^31) */
        emit_fb(as, PPCI_FCTIWZ, tmp, left);
        emit_fai(as, PPCI_STFD, tmp, RID_SP, SPOFS_TMP);
        emit_fb(as, PPCI_FCTIWZ, tmp, tmp);
        emit_fab(as, PPCI_FSUB, tmp, left, tmp);
        emit_lsptr(as, PPCI_LFS, (tmp & 31),
                   (void *)lj_ir_k64_find(as->J, U64x(4f000000,00000000)),
                   RSET_GPR);
      } else {
        emit_tai(as, PPCI_LWZ, dest, RID_SP, SPOFS_TMPLO);
        emit_fai(as, PPCI_STFD, tmp, RID_SP, SPOFS_TMP);
        emit_fb(as, PPCI_FCTIWZ, tmp, left);
      }
    }
  } else {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    if (st >= IRT_I8 && st <= IRT_U16) {  /* Extend to 32 bit integer. */
      Reg left = ra_alloc1(as, ir->op1, RSET_GPR);
      lua_assert(irt_isint(ir->t) || irt_isu32(ir->t));
      if ((ir->op2 & IRCONV_SEXT))
        emit_as(as, st == IRT_I8 ? PPCI_EXTSB : PPCI_EXTSH, dest, left);
      else
        emit_rot(as, PPCI_RLWINM, dest, left, 0, st == IRT_U8 ? 24 : 16, 31);
    } else {  /* 32/64 bit integer conversions. */
      /* Only need to handle 32/32 bit no-op (cast) on 32 bit archs. */
      ra_leftov(as, dest, lref);  /* Do nothing, but may need to move regs. */
    }
  }
}

static void asm_strto(ASMState *as, IRIns *ir)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_strscan_num];
  IRRef args[2];
  int32_t ofs;
  RegSet drop = RSET_SCRATCH;
  if (ra_hasreg(ir->r)) rset_set(drop, ir->r);  /* Spill dest reg (if any). */
  ra_evictset(as, drop);
  asm_guardcc(as, CC_EQ);
  emit_ai(as, PPCI_CMPWI, RID_RET, 0);  /* Test return status. */
  args[0] = ir->op1;      /* GCstr *str */
  args[1] = ASMREF_TMP1;  /* TValue *n  */
  asm_gencall(as, ci, args);
  /* Store the result to the spill slot or temp slots. */
  ofs = ir->s ? sps_scale(ir->s) : SPOFS_TMP;
  emit_tai(as, PPCI_ADDI, ra_releasetmp(as, ASMREF_TMP1), RID_SP, ofs);
}

/* -- Memory references --------------------------------------------------- */
```

```
518
519 /* Get pointer to TValue. */
520 static void asm_tvptr(ASMState *as, Reg dest, IRRef ref)
521 {
522   IRIns *ir = IR(ref);
523   if (irt_isnum(ir->t)) {
524     if (irref_isk(ref))  /* Use the number constant itself as a TValue. */
525       ra_allockreg(as, i32ptr(ir_knum(ir)), dest);
526     else  /* Otherwise force a spill and use the spill slot. */
527       emit_tai(as, PPCI_ADDI, dest, RID_SP, ra_spill(as, ir));
528   } else {
529     /* Otherwise use g->tmptv to hold the TValue. */
530     RegSet allow = rset_exclude(RSET_GPR, dest);
531     Reg type;
532     emit_tai(as, PPCI_ADDI, dest, RID_JGL, (int32_t)offsetof(global_State, tmptv)-32768);
533     if (!irt_ispri(ir->t)) {
534       Reg src = ra_alloc1(as, ref, allow);
535       emit_setgl(as, src, tmptv.gcr);
536     }
537     type = ra_allock(as, irt_toitype(ir->t), allow);
538     emit_setgl(as, type, tmptv.it);
539   }
540 }
541
542 static void asm_aref(ASMState *as, IRIns *ir)
543 {
544   Reg dest = ra_dest(as, ir, RSET_GPR);
545   Reg idx, base;
546   if (irref_isk(ir->op2)) {
547     IRRef tab = IR(ir->op1)->op1;
548     int32_t ofs = asm_fuseabase(as, tab);
549     IRRef refa = ofs ? tab : ir->op1;
550     ofs += 8*IR(ir->op2)->i;
551     if (checki16(ofs)) {
552       base = ra_alloc1(as, refa, RSET_GPR);
553       emit_tai(as, PPCI_ADDI, dest, base, ofs);
554       return;
555     }
556   }
557   base = ra_alloc1(as, ir->op1, RSET_GPR);
558   idx = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, base));
559   emit_tab(as, PPCI_ADD, dest, RID_TMP, base);
560   emit_slwi(as, RID_TMP, idx, 3);
561 }
562
563 /* Inlined hash lookup. Specialized for key type and for const keys.
564 ** The equivalent C code is:
565 **   Node *n = hashkey(t, key);
566 **   do {
567 **     if (lj_obj_equal(&n->key, key)) return &n->val;
568 **   } while ((n = nextnode(n)));
569 **   return niltv(L);
570 */
571 static void asm_href(ASMState *as, IRIns *ir, IROp merge)
572 {
573   RegSet allow = RSET_GPR;
574   int destused = ra_used(ir);
575   Reg dest = ra_dest(as, ir, allow);
576   Reg tab = ra_alloc1(as, ir->op1, rset_clear(allow, dest));
577   Reg key = RID_NONE, tmp1 = RID_TMP, tmp2;
578   Reg tisnum = RID_NONE, tmpnum = RID_NONE;
579   IRRef refkey = ir->op2;
580   IRIns *irkey = IR(refkey);
581   IRType1 kt = irkey->t;
582   uint32_t khash;
583   MCLabel l_end, l_loop, l_next;
584
585   rset_clear(allow, tab);
586   if (irt_isnum(kt)) {
587     key = ra_alloc1(as, refkey, RSET_FPR);
588     tmpnum = ra_scratch(as, rset_exclude(RSET_FPR, key));
589     tisnum = ra_allock(as, (int32_t)LJ_TISNUM, allow);
590     rset_clear(allow, tisnum);
591   } else if (!irt_ispri(kt)) {
592     key = ra_alloc1(as, refkey, allow);
593     rset_clear(allow, key);
```

```
  }
  tmp2 = ra_scratch(as, allow);
  rset_clear(allow, tmp2);

  /* Key not found in chain: jump to exit (if merged) or load niltv. */
  l_end = emit_label(as);
  as->invmcp = NULL;
  if (merge == IR_NE)
    asm_guardcc(as, CC_EQ);
  else if (destused)
    emit_loada(as, dest, niltvg(J2G(as->J)));

  /* Follow hash chain until the end. */
  l_loop = --as->mcp;
  emit_ai(as, PPCI_CMPWI, dest, 0);
  emit_tai(as, PPCI_LWZ, dest, dest, (int32_t)offsetof(Node, next));
  l_next = emit_label(as);

  /* Type and value comparison. */
  if (merge == IR_EQ)
    asm_guardcc(as, CC_EQ);
  else
    emit_condbranch(as, PPCI_BC|PPCF_Y, CC_EQ, l_end);
  if (irt_isnum(kt)) {
    emit_fab(as, PPCI_FCMPU, 0, tmpnum, key);
    emit_condbranch(as, PPCI_BC, CC_GE, l_next);
    emit_ab(as, PPCI_CMPLW, tmp1, tisnum);
    emit_fai(as, PPCI_LFD, tmpnum, dest, (int32_t)offsetof(Node, key.n));
  } else {
    if (!irt_ispri(kt)) {
      emit_ab(as, PPCI_CMPW, tmp2, key);
      emit_condbranch(as, PPCI_BC, CC_NE, l_next);
    }
    emit_ai(as, PPCI_CMPWI, tmp1, irt_toitype(irkey->t));
    if (!irt_ispri(kt))
      emit_tai(as, PPCI_LWZ, tmp2, dest, (int32_t)offsetof(Node, key.gcr));
  }
  emit_tai(as, PPCI_LWZ, tmp1, dest, (int32_t)offsetof(Node, key.it));
  *l_loop = PPCI_BC | PPCF_Y | PPCF_CC(CC_NE) |
            (((char *)as->mcp-(char *)l_loop) & 0xffffu);

  /* Load main position relative to tab->node into dest. */
  khash = irref_isk(refkey) ? ir_khash(irkey) : 1;
  if (khash == 0) {
    emit_tai(as, PPCI_LWZ, dest, tab, (int32_t)offsetof(GCtab, node));
  } else {
    Reg tmphash = tmp1;
    if (irref_isk(refkey))
      tmphash = ra_allock(as, khash, allow);
    emit_tab(as, PPCI_ADD, dest, dest, tmp1);
    emit_tai(as, PPCI_MULLI, tmp1, tmp1, sizeof(Node));
    emit_asb(as, PPCI_AND, tmp1, tmp2, tmphash);
    emit_tai(as, PPCI_LWZ, dest, tab, (int32_t)offsetof(GCtab, node));
    emit_tai(as, PPCI_LWZ, tmp2, tab, (int32_t)offsetof(GCtab, hmask));
    if (irref_isk(refkey)) {
      /* Nothing to do. */
    } else if (irt_isstr(kt)) {
      emit_tai(as, PPCI_LWZ, tmp1, key, (int32_t)offsetof(GCstr, hash));
    } else {  /* Must match with hash*() in lj_tab.c. */
      emit_tab(as, PPCI_SUBF, tmp1, tmp2, tmp1);
      emit_rotlwi(as, tmp2, tmp2, HASH_ROT3);
      emit_asb(as, PPCI_XOR, tmp1, tmp1, tmp2);
      emit_rotlwi(as, tmp1, tmp1, (HASH_ROT2+HASH_ROT1)&31);
      emit_tab(as, PPCI_SUBF, tmp2, dest, tmp2);
      if (irt_isnum(kt)) {
        int32_t ofs = ra_spill(as, irkey);
        emit_asb(as, PPCI_XOR, tmp2, tmp2, tmp1);
        emit_rotlwi(as, dest, tmp1, HASH_ROT1);
        emit_tab(as, PPCI_ADD, tmp1, tmp1, tmp1);
        emit_tai(as, PPCI_LWZ, tmp2, RID_SP, ofs+4);
        emit_tai(as, PPCI_LWZ, tmp1, RID_SP, ofs);
      } else {
        emit_asb(as, PPCI_XOR, tmp2, key, tmp1);
        emit_rotlwi(as, dest, tmp1, HASH_ROT1);
        emit_tai(as, PPCI_ADDI, tmp1, tmp2, HASH_BIAS);
        emit_tai(as, PPCI_ADDIS, tmp2, key, (HASH_BIAS + 32768)>>16);
```

```
      }
    }
  }
}

static void asm_hrefk(ASMState *as, IRIns *ir)
{
  IRIns *kslot = IR(ir->op2);
  IRIns *irkey = IR(kslot->op1);
  int32_t ofs = (int32_t)(kslot->op2 * sizeof(Node));
  int32_t kofs = ofs + (int32_t)offsetof(Node, key);
  Reg dest = (ra_used(ir)||ofs > 32736) ? ra_dest(as, ir, RSET_GPR) : RID_NONE;
  Reg node = ra_alloc1(as, ir->op1, RSET_GPR);
  Reg key = RID_NONE, type = RID_TMP, idx = node;
  RegSet allow = rset_exclude(RSET_GPR, node);
  lua_assert(ofs % sizeof(Node) == 0);
  if (ofs > 32736) {
    idx = dest;
    rset_clear(allow, dest);
    kofs = (int32_t)offsetof(Node, key);
  } else if (ra_hasreg(dest)) {
    emit_tai(as, PPCI_ADDI, dest, node, ofs);
  }
  asm_guardcc(as, CC_NE);
  if (!irt_ispri(irkey->t)) {
    key = ra_scratch(as, allow);
    rset_clear(allow, key);
  }
  rset_clear(allow, type);
  if (irt_isnum(irkey->t)) {
    emit_cmpi(as, key, (int32_t)ir_knum(irkey)->u32.lo);
    asm_guardcc(as, CC_NE);
    emit_cmpi(as, type, (int32_t)ir_knum(irkey)->u32.hi);
  } else {
    if (ra_hasreg(key)) {
      emit_cmpi(as, key, irkey->i);  /* May use RID_TMP, i.e. type. */
      asm_guardcc(as, CC_NE);
    }
    emit_ai(as, PPCI_CMPWI, type, irt_toitype(irkey->t));
  }
  if (ra_hasreg(key)) emit_tai(as, PPCI_LWZ, key, idx, kofs+4);
  emit_tai(as, PPCI_LWZ, type, idx, kofs);
  if (ofs > 32736) {
    emit_tai(as, PPCI_ADDIS, dest, dest, (ofs + 32768) >> 16);
    emit_tai(as, PPCI_ADDI, dest, node, ofs);
  }
}

static void asm_uref(ASMState *as, IRIns *ir)
{
  /* NYI: Check that UREFO is still open and not aliasing a slot. */
  Reg dest = ra_dest(as, ir, RSET_GPR);
  if (irref_isk(ir->op1)) {
    GCfunc *fn = ir_kfunc(IR(ir->op1));
    MRef *v = &gcref(fn->l.uvptr[(ir->op2 >> 8)])->uv.v;
    emit_lsptr(as, PPCI_LWZ, dest, v, RSET_GPR);
  } else {
    Reg uv = ra_scratch(as, RSET_GPR);
    Reg func = ra_alloc1(as, ir->op1, RSET_GPR);
    if (ir->o == IR_UREFC) {
      asm_guardcc(as, CC_NE);
      emit_ai(as, PPCI_CMPWI, RID_TMP, 1);
      emit_tai(as, PPCI_ADDI, dest, uv, (int32_t)offsetof(GCupval, tv));
      emit_tai(as, PPCI_LBZ, RID_TMP, uv, (int32_t)offsetof(GCupval, closed));
    } else {
      emit_tai(as, PPCI_LWZ, dest, uv, (int32_t)offsetof(GCupval, v));
    }
    emit_tai(as, PPCI_LWZ, uv, func,
             (int32_t)offsetof(GCfuncL, uvptr) + 4*(int32_t)(ir->op2 >> 8));
  }
}

static void asm_fref(ASMState *as, IRIns *ir)
{
  UNUSED(as); UNUSED(ir);
  lua_assert(!ra_used(ir));
```

```
}

static void asm_strref(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  IRRef ref = ir->op2, refk = ir->op1;
  int32_t ofs = (int32_t)sizeof(GCstr);
  Reg r;
  if (irref_isk(ref)) {
    IRRef tmp = refk; refk = ref; ref = tmp;
  } else if (!irref_isk(refk)) {
    Reg right, left = ra_alloc1(as, ir->op1, RSET_GPR);
    IRIns *irr = IR(ir->op2);
    if (ra_hasreg(irr->r)) {
      ra_noweak(as, irr->r);
      right = irr->r;
    } else if (mayfuse(as, irr->op2) &&
               irr->o == IR_ADD && irref_isk(irr->op2) &&
               checki16(ofs + IR(irr->op2)->i)) {
      ofs += IR(irr->op2)->i;
      right = ra_alloc1(as, irr->op1, rset_exclude(RSET_GPR, left));
    } else {
      right = ra_allocref(as, ir->op2, rset_exclude(RSET_GPR, left));
    }
    emit_tai(as, PPCI_ADDI, dest, dest, ofs);
    emit_tab(as, PPCI_ADD, dest, left, right);
    return;
  }
  r = ra_alloc1(as, ref, RSET_GPR);
  ofs += IR(refk)->i;
  if (checki16(ofs))
    emit_tai(as, PPCI_ADDI, dest, r, ofs);
  else
    emit_tab(as, PPCI_ADD, dest, r,
             ra_allock(as, ofs, rset_exclude(RSET_GPR, r)));
}

/* -- Loads and stores ---------------------------------------------------- */

static PPCIns asm_fxloadins(IRIns *ir)
{
  switch (irt_type(ir->t)) {
  case IRT_I8: return PPCI_LBZ;  /* Needs sign-extension. */
  case IRT_U8: return PPCI_LBZ;
  case IRT_I16: return PPCI_LHA;
  case IRT_U16: return PPCI_LHZ;
  case IRT_NUM: return PPCI_LFD;
  case IRT_FLOAT: return PPCI_LFS;
  default: return PPCI_LWZ;
  }
}

static PPCIns asm_fxstoreins(IRIns *ir)
{
  switch (irt_type(ir->t)) {
  case IRT_I8: case IRT_U8: return PPCI_STB;
  case IRT_I16: case IRT_U16: return PPCI_STH;
  case IRT_NUM: return PPCI_STFD;
  case IRT_FLOAT: return PPCI_STFS;
  default: return PPCI_STW;
  }
}

static void asm_fload(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg idx = ra_alloc1(as, ir->op1, RSET_GPR);
  PPCIns pi = asm_fxloadins(ir);
  int32_t ofs;
  if (ir->op2 == IRFL_TAB_ARRAY) {
    ofs = asm_fuseabase(as, ir->op1);
    if (ofs) {  /* Turn the t->array load into an add for colocated arrays. */
      emit_tai(as, PPCI_ADDI, dest, idx, ofs);
      return;
    }
  }
```

```
  ofs = field_ofs[ir->op2];
  lua_assert(!irt_isi8(ir->t));
  emit_tai(as, pi, dest, idx, ofs);
}

static void asm_fstore(ASMState *as, IRIns *ir)
{
  if (ir->r != RID_SINK) {
    Reg src = ra_alloc1(as, ir->op2, RSET_GPR);
    IRIns *irf = IR(ir->op1);
    Reg idx = ra_alloc1(as, irf->op1, rset_exclude(RSET_GPR, src));
    int32_t ofs = field_ofs[irf->op2];
    PPCIns pi = asm_fxstoreins(ir);
    emit_tai(as, pi, src, idx, ofs);
  }
}

static void asm_xload(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, irt_isfp(ir->t) ? RSET_FPR : RSET_GPR);
  lua_assert(!(ir->op2 & IRXLOAD_UNALIGNED));
  if (irt_isi8(ir->t))
    emit_as(as, PPCI_EXTSB, dest, dest);
  asm_fusexref(as, asm_fxloadins(ir), dest, ir->op1, RSET_GPR, 0);
}

static void asm_xstore_(ASMState *as, IRIns *ir, int32_t ofs)
{
  IRIns *irb;
  if (ir->r == RID_SINK)
    return;
  if (ofs == 0 && mayfuse(as, ir->op2) && (irb = IR(ir->op2))->o == IR_BSWAP &&
      ra_noreg(irb->r) && (irt_isint(ir->t) || irt_isu32(ir->t))) {
    /* Fuse BSWAP with XSTORE to stwbrx. */
    Reg src = ra_alloc1(as, irb->op1, RSET_GPR);
    asm_fusexrefx(as, PPCI_STWBRX, src, ir->op1, rset_exclude(RSET_GPR, src));
  } else {
    Reg src = ra_alloc1(as, ir->op2, irt_isfp(ir->t) ? RSET_FPR : RSET_GPR);
    asm_fusexref(as, asm_fxstoreins(ir), src, ir->op1,
                 rset_exclude(RSET_GPR, src), ofs);
  }
}

#define asm_xstore(as, ir)        asm_xstore_(as, ir, 0)

static void asm_ahuvload(ASMState *as, IRIns *ir)
{
  IRType1 t = ir->t;
  Reg dest = RID_NONE, type = RID_TMP, tmp = RID_TMP, idx;
  RegSet allow = RSET_GPR;
  int32_t ofs = AHUREF_LSX;
  if (ra_used(ir)) {
    lua_assert(irt_isnum(t) || irt_isint(t) || irt_isaddr(t));
    if (!irt_isnum(t)) ofs = 0;
    dest = ra_dest(as, ir, irt_isnum(t) ? RSET_FPR : RSET_GPR);
    rset_clear(allow, dest);
  }
  idx = asm_fuseahuref(as, ir->op1, &ofs, allow);
  if (irt_isnum(t)) {
    Reg tisnum = ra_allock(as, (int32_t)LJ_TISNUM, rset_exclude(allow, idx));
    asm_guardcc(as, CC_GE);
    emit_ab(as, PPCI_CMPLW, type, tisnum);
    if (ra_hasreg(dest)) {
      if (ofs == AHUREF_LSX) {
        tmp = ra_scratch(as, rset_exclude(rset_exclude(RSET_GPR,
                                                       (idx&255)), (idx>>8)));
        emit_fab(as, PPCI_LFDX, dest, (idx&255), tmp);
      } else {
        emit_fai(as, PPCI_LFD, dest, idx, ofs);
      }
    }
  } else {
    asm_guardcc(as, CC_NE);
    emit_ai(as, PPCI_CMPWI, type, irt_toitype(t));
    if (ra_hasreg(dest)) emit_tai(as, PPCI_LWZ, dest, idx, ofs+4);
  }
```

```
  if (ofs == AHUREF_LSX) {
    emit_tab(as, PPCI_LWZX, type, (idx&255), tmp);
    emit_slwi(as, tmp, (idx>>8), 3);
  } else {
    emit_tai(as, PPCI_LWZ, type, idx, ofs);
  }
}

static void asm_ahustore(ASMState *as, IRIns *ir)
{
  RegSet allow = RSET_GPR;
  Reg idx, src = RID_NONE, type = RID_NONE;
  int32_t ofs = AHUREF_LSX;
  if (ir->r == RID_SINK)
    return;
  if (irt_isnum(ir->t)) {
    src = ra_alloc1(as, ir->op2, RSET_FPR);
  } else {
    if (!irt_ispri(ir->t)) {
      src = ra_alloc1(as, ir->op2, allow);
      rset_clear(allow, src);
      ofs = 0;
    }
    type = ra_allock(as, (int32_t)irt_toitype(ir->t), allow);
    rset_clear(allow, type);
  }
  idx = asm_fuseahuref(as, ir->op1, &ofs, allow);
  if (irt_isnum(ir->t)) {
    if (ofs == AHUREF_LSX) {
      emit_fab(as, PPCI_STFDX, src, (idx&255), RID_TMP);
      emit_slwi(as, RID_TMP, (idx>>8), 3);
    } else {
      emit_fai(as, PPCI_STFD, src, idx, ofs);
    }
  } else {
    if (ra_hasreg(src))
      emit_tai(as, PPCI_STW, src, idx, ofs+4);
    if (ofs == AHUREF_LSX) {
      emit_tab(as, PPCI_STWX, type, (idx&255), RID_TMP);
      emit_slwi(as, RID_TMP, (idx>>8), 3);
    } else {
      emit_tai(as, PPCI_STW, type, idx, ofs);
    }
  }
}

static void asm_sload(ASMState *as, IRIns *ir)
{
  int32_t ofs = 8*((int32_t)ir->op1-1) + ((ir->op2 & IRSLOAD_FRAME) ? 0 : 4);
  IRType1 t = ir->t;
  Reg dest = RID_NONE, type = RID_NONE, base;
  RegSet allow = RSET_GPR;
  lua_assert(!(ir->op2 & IRSLOAD_PARENT));  /* Handled by asm_head_side(). */
  lua_assert(irt_isguard(t) || !(ir->op2 & IRSLOAD_TYPECHECK));
  lua_assert(LJ_DUALNUM ||
             !irt_isint(t) || (ir->op2 & (IRSLOAD_CONVERT|IRSLOAD_FRAME)));
  if ((ir->op2 & IRSLOAD_CONVERT) && irt_isguard(t) && irt_isint(t)) {
    dest = ra_scratch(as, RSET_FPR);
    asm_tointg(as, ir, dest);
    t.irt = IRT_NUM;  /* Continue with a regular number type check. */
  } else if (ra_used(ir)) {
    lua_assert(irt_isnum(t) || irt_isint(t) || irt_isaddr(t));
    dest = ra_dest(as, ir, irt_isnum(t) ? RSET_FPR : RSET_GPR);
    rset_clear(allow, dest);
    base = ra_alloc1(as, REF_BASE, allow);
    rset_clear(allow, base);
    if ((ir->op2 & IRSLOAD_CONVERT)) {
      if (irt_isint(t)) {
        emit_tai(as, PPCI_LWZ, dest, RID_SP, SPOFS_TMPLO);
        dest = ra_scratch(as, RSET_FPR);
        emit_fai(as, PPCI_STFD, dest, RID_SP, SPOFS_TMP);
        emit_fb(as, PPCI_FCTIWZ, dest, dest);
        t.irt = IRT_NUM;  /* Check for original type. */
      } else {
        Reg tmp = ra_scratch(as, allow);
        Reg hibias = ra_allock(as, 0x43300000, rset_clear(allow, tmp));
```

```
        Reg fbias = ra_scratch(as, rset_exclude(RSET_FPR, dest));
        emit_fab(as, PPCI_FSUB, dest, dest, fbias);
        emit_fai(as, PPCI_LFD, dest, RID_SP, SPOFS_TMP);
        emit_lsptr(as, PPCI_LFS, (fbias & 31),
                   (void *)lj_ir_k64_find(as->J, U64x(59800004,59800000)),
                   rset_clear(allow, hibias));
        emit_tai(as, PPCI_STW, tmp, RID_SP, SPOFS_TMPLO);
        emit_tai(as, PPCI_STW, hibias, RID_SP, SPOFS_TMPHI);
        emit_asi(as, PPCI_XORIS, tmp, tmp, 0x8000);
        dest = tmp;
        t.irt = IRT_INT;  /* Check for original type. */
      }
    }
    goto dotypecheck;
  }
  base = ra_alloc1(as, REF_BASE, allow);
  rset_clear(allow, base);
dotypecheck:
  if (irt_isnum(t)) {
    if ((ir->op2 & IRSLOAD_TYPECHECK)) {
      Reg tisnum = ra_allock(as, (int32_t)LJ_TISNUM, allow);
      asm_guardcc(as, CC_GE);
      emit_ab(as, PPCI_CMPLW, RID_TMP, tisnum);
      type = RID_TMP;
    }
    if (ra_hasreg(dest)) emit_fai(as, PPCI_LFD, dest, base, ofs-4);
  } else {
    if ((ir->op2 & IRSLOAD_TYPECHECK)) {
      asm_guardcc(as, CC_NE);
      emit_ai(as, PPCI_CMPWI, RID_TMP, irt_toitype(t));
      type = RID_TMP;
    }
    if (ra_hasreg(dest)) emit_tai(as, PPCI_LWZ, dest, base, ofs);
  }
  if (ra_hasreg(type)) emit_tai(as, PPCI_LWZ, type, base, ofs-4);
}

/* -- Allocations --------------------------------------------------------- */

#if LJ_HASFFI
static void asm_cnew(ASMState *as, IRIns *ir)
{
  CTState *cts = ctype_ctsG(J2G(as->J));
  CTypeID id = (CTypeID)IR(ir->op1)->i;
  CTSize sz;
  CTInfo info = lj_ctype_info(cts, id, &sz);
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_mem_newgco];
  IRRef args[4];
  RegSet drop = RSET_SCRATCH;
  lua_assert(sz != CTSIZE_INVALID || (ir->o == IR_CNEW && ir->op2 != REF_NIL));

  as->gcsteps++;
  if (ra_hasreg(ir->r))
    rset_clear(drop, ir->r);  /* Dest reg handled below. */
  ra_evictset(as, drop);
  if (ra_used(ir))
    ra_destreg(as, ir, RID_RET);  /* GCcdata * */

  /* Initialize immutable cdata object. */
  if (ir->o == IR_CNEWI) {
    RegSet allow = (RSET_GPR & ~RSET_SCRATCH);
    int32_t ofs = sizeof(GCcdata);
    lua_assert(sz == 4 || sz == 8);
    if (sz == 8) {
      ofs += 4;
      lua_assert((ir+1)->o == IR_HIOP);
    }
    for (;;) {
      Reg r = ra_alloc1(as, ir->op2, allow);
      emit_tai(as, PPCI_STW, r, RID_RET, ofs);
      rset_clear(allow, r);
      if (ofs == sizeof(GCcdata)) break;
      ofs -= 4; ir++;
    }
  } else if (ir->op2 != REF_NIL) {  /* Create VLA/VLS/aligned cdata. */
    ci = &lj_ir_callinfo[IRCALL_lj_cdata_newv];
```

```
    args[0] = ASMREF_L;     /* lua_State *L */
    args[1] = ir->op1;      /* CTypeID id   */
    args[2] = ir->op2;      /* CTSize sz    */
    args[3] = ASMREF_TMP1;  /* CTSize align */
    asm_gencall(as, ci, args);
    emit_loadi(as, ra_releasetmp(as, ASMREF_TMP1), (int32_t)ctype_align(info));
    return;
  }

  /* Initialize gct and ctypeid. lj_mem_newgco() already sets marked. */
  emit_tai(as, PPCI_STB, RID_RET+1, RID_RET, offsetof(GCcdata, gct));
  emit_tai(as, PPCI_STH, RID_TMP, RID_RET, offsetof(GCcdata, ctypeid));
  emit_ti(as, PPCI_LI, RID_RET+1, ~LJ_TCDATA);
  emit_ti(as, PPCI_LI, RID_TMP, id);  /* Lower 16 bit used. Sign-ext ok. */
  args[0] = ASMREF_L;     /* lua_State *L */
  args[1] = ASMREF_TMP1;  /* MSize size   */
  asm_gencall(as, ci, args);
  ra_allockreg(as, (int32_t)(sz+sizeof(GCcdata)),
               ra_releasetmp(as, ASMREF_TMP1));
}
#else
#define asm_cnew(as, ir)        ((void)0)
#endif

/* -- Write barriers ------------------------------------------------------ */

static void asm_tbar(ASMState *as, IRIns *ir)
{
  Reg tab = ra_alloc1(as, ir->op1, RSET_GPR);
  Reg mark = ra_scratch(as, rset_exclude(RSET_GPR, tab));
  Reg link = RID_TMP;
  MCLabel l_end = emit_label(as);
  emit_tai(as, PPCI_STW, link, tab, (int32_t)offsetof(GCtab, gclist));
  emit_tai(as, PPCI_STB, mark, tab, (int32_t)offsetof(GCtab, marked));
  emit_setgl(as, tab, gc.grayagain);
  lua_assert(LJ_GC_BLACK == 0x04);
  emit_rot(as, PPCI_RLWINM, mark, mark, 0, 30, 28);  /* Clear black bit. */
  emit_getgl(as, link, gc.grayagain);
  emit_condbranch(as, PPCI_BC|PPCF_Y, CC_EQ, l_end);
  emit_asi(as, PPCI_ANDIDOT, RID_TMP, mark, LJ_GC_BLACK);
  emit_tai(as, PPCI_LBZ, mark, tab, (int32_t)offsetof(GCtab, marked));
}

static void asm_obar(ASMState *as, IRIns *ir)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_gc_barrieruv];
  IRRef args[2];
  MCLabel l_end;
  Reg obj, val, tmp;
  /* No need for other object barriers (yet). */
  lua_assert(IR(ir->op1)->o == IR_UREFC);
  ra_evictset(as, RSET_SCRATCH);
  l_end = emit_label(as);
  args[0] = ASMREF_TMP1;  /* global_State *g */
  args[1] = ir->op1;      /* TValue *tv      */
  asm_gencall(as, ci, args);
  emit_tai(as, PPCI_ADDI, ra_releasetmp(as, ASMREF_TMP1), RID_JGL, -32768);
  obj = IR(ir->op1)->r;
  tmp = ra_scratch(as, rset_exclude(RSET_GPR, obj));
  emit_condbranch(as, PPCI_BC|PPCF_Y, CC_EQ, l_end);
  emit_asi(as, PPCI_ANDIDOT, tmp, tmp, LJ_GC_BLACK);
  emit_condbranch(as, PPCI_BC, CC_EQ, l_end);
  emit_asi(as, PPCI_ANDIDOT, RID_TMP, RID_TMP, LJ_GC_WHITES);
  val = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, obj));
  emit_tai(as, PPCI_LBZ, tmp, obj,
           (int32_t)offsetof(GCupval, marked)-(int32_t)offsetof(GCupval, tv));
  emit_tai(as, PPCI_LBZ, RID_TMP, val, (int32_t)offsetof(GChead, marked));
}

/* -- Arithmetic and logic operations ------------------------------------- */

static void asm_fparith(ASMState *as, IRIns *ir, PPCIns pi)
{
  Reg dest = ra_dest(as, ir, RSET_FPR);
  Reg right, left = ra_alloc2(as, ir, RSET_FPR);
  right = (left >> 8); left &= 255;
```

```
  if (pi == PPCI_FMUL)
    emit_fac(as, pi, dest, left, right);
  else
    emit_fab(as, pi, dest, left, right);
}

static void asm_fpunary(ASMState *as, IRIns *ir, PPCIns pi)
{
  Reg dest = ra_dest(as, ir, RSET_FPR);
  Reg left = ra_hintalloc(as, ir->op1, dest, RSET_FPR);
  emit_fb(as, pi, dest, left);
}

static void asm_fpmath(ASMState *as, IRIns *ir)
{
  if (ir->op2 == IRFPM_EXP2 && asm_fpjoin_pow(as, ir))
    return;
  if (ir->op2 == IRFPM_SQRT && (as->flags & JIT_F_SQRT))
    asm_fpunary(as, ir, PPCI_FSQRT);
  else
    asm_callid(as, ir, IRCALL_lj_vm_floor + ir->op2);
}

static void asm_add(ASMState *as, IRIns *ir)
{
  if (irt_isnum(ir->t)) {
    if (!asm_fusemadd(as, ir, PPCI_FMADD, PPCI_FMADD))
      asm_fparith(as, ir, PPCI_FADD);
  } else {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    Reg right, left = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
    PPCIns pi;
    if (irref_isk(ir->op2)) {
      int32_t k = IR(ir->op2)->i;
      if (checki16(k)) {
        pi = PPCI_ADDI;
        /* May fail due to spills/restores above, but simplifies the logic. */
        if (as->flagmcp == as->mcp) {
          as->flagmcp = NULL;
          as->mcp++;
          pi = PPCI_ADDICDOT;
        }
        emit_tai(as, pi, dest, left, k);
        return;
      } else if ((k & 0xffff) == 0) {
        emit_tai(as, PPCI_ADDIS, dest, left, (k >> 16));
        return;
      } else if (!as->sectref) {
        emit_tai(as, PPCI_ADDIS, dest, dest, (k + 32768) >> 16);
        emit_tai(as, PPCI_ADDI, dest, left, k);
        return;
      }
    }
    pi = PPCI_ADD;
    /* May fail due to spills/restores above, but simplifies the logic. */
    if (as->flagmcp == as->mcp) {
      as->flagmcp = NULL;
      as->mcp++;
      pi |= PPCF_DOT;
    }
    right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
    emit_tab(as, pi, dest, left, right);
  }
}

static void asm_sub(ASMState *as, IRIns *ir)
{
  if (irt_isnum(ir->t)) {
    if (!asm_fusemadd(as, ir, PPCI_FMSUB, PPCI_FNMSUB))
      asm_fparith(as, ir, PPCI_FSUB);
  } else {
    PPCIns pi = PPCI_SUBF;
    Reg dest = ra_dest(as, ir, RSET_GPR);
    Reg left, right;
    if (irref_isk(ir->op1)) {
      int32_t k = IR(ir->op1)->i;
```

```
      if (checki16(k)) {
        right = ra_alloc1(as, ir->op2, RSET_GPR);
        emit_tai(as, PPCI_SUBFIC, dest, right, k);
        return;
      }
    }
    /* May fail due to spills/restores above, but simplifies the logic. */
    if (as->flagmcp == as->mcp) {
      as->flagmcp = NULL;
      as->mcp++;
      pi |= PPCF_DOT;
    }
    left = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
    right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
    emit_tab(as, pi, dest, right, left);  /* Subtract right _from_ left. */
  }
}

static void asm_mul(ASMState *as, IRIns *ir)
{
  if (irt_isnum(ir->t)) {
    asm_fparith(as, ir, PPCI_FMUL);
  } else {
    PPCIns pi = PPCI_MULLW;
    Reg dest = ra_dest(as, ir, RSET_GPR);
    Reg right, left = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
    if (irref_isk(ir->op2)) {
      int32_t k = IR(ir->op2)->i;
      if (checki16(k)) {
        emit_tai(as, PPCI_MULLI, dest, left, k);
        return;
      }
    }
    /* May fail due to spills/restores above, but simplifies the logic. */
    if (as->flagmcp == as->mcp) {
      as->flagmcp = NULL;
      as->mcp++;
      pi |= PPCF_DOT;
    }
    right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
    emit_tab(as, pi, dest, left, right);
  }
}

#define asm_div(as, ir)                asm_fparith(as, ir, PPCI_FDIV)
#define asm_mod(as, ir)                asm_callid(as, ir, IRCALL_lj_vm_modi)
#define asm_pow(as, ir)                asm_callid(as, ir, IRCALL_lj_vm_powi)

static void asm_neg(ASMState *as, IRIns *ir)
{
  if (irt_isnum(ir->t)) {
    asm_fpunary(as, ir, PPCI_FNEG);
  } else {
    Reg dest, left;
    PPCIns pi = PPCI_NEG;
    if (as->flagmcp == as->mcp) {
      as->flagmcp = NULL;
      as->mcp++;
      pi |= PPCF_DOT;
    }
    dest = ra_dest(as, ir, RSET_GPR);
    left = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
    emit_tab(as, pi, dest, left, 0);
  }
}

#define asm_abs(as, ir)                asm_fpunary(as, ir, PPCI_FABS)
#define asm_atan2(as, ir)        asm_callid(as, ir, IRCALL_atan2)
#define asm_ldexp(as, ir)        asm_callid(as, ir, IRCALL_ldexp)

static void asm_arithov(ASMState *as, IRIns *ir, PPCIns pi)
{
  Reg dest, left, right;
  if (as->flagmcp == as->mcp) {
    as->flagmcp = NULL;
    as->mcp++;
```

```
  }
  asm_guardcc(as, CC_SO);
  dest = ra_dest(as, ir, RSET_GPR);
  left = ra_alloc2(as, ir, RSET_GPR);
  right = (left >> 8); left &= 255;
  if (pi == PPCI_SUBFO) { Reg tmp = left; left = right; right = tmp; }
  emit_tab(as, pi|PPCF_DOT, dest, left, right);
}

#define asm_addov(as, ir)		asm_arithov(as, ir, PPCI_ADDO)
#define asm_subov(as, ir)		asm_arithov(as, ir, PPCI_SUBFO)
#define asm_mulov(as, ir)		asm_arithov(as, ir, PPCI_MULLWO)

#if LJ_HASFFI
static void asm_add64(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg right, left = ra_alloc1(as, ir->op1, RSET_GPR);
  PPCIns pi = PPCI_ADDE;
  if (irref_isk(ir->op2)) {
    int32_t k = IR(ir->op2)->i;
    if (k == 0)
      pi = PPCI_ADDZE;
    else if (k == -1)
      pi = PPCI_ADDME;
    else
      goto needright;
    right = 0;
  } else {
  needright:
    right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
  }
  emit_tab(as, pi, dest, left, right);
  ir--;
  dest = ra_dest(as, ir, RSET_GPR);
  left = ra_alloc1(as, ir->op1, RSET_GPR);
  if (irref_isk(ir->op2)) {
    int32_t k = IR(ir->op2)->i;
    if (checki16(k)) {
      emit_tai(as, PPCI_ADDIC, dest, left, k);
      return;
    }
  }
  right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
  emit_tab(as, PPCI_ADDC, dest, left, right);
}

static void asm_sub64(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg left, right = ra_alloc1(as, ir->op2, RSET_GPR);
  PPCIns pi = PPCI_SUBFE;
  if (irref_isk(ir->op1)) {
    int32_t k = IR(ir->op1)->i;
    if (k == 0)
      pi = PPCI_SUBFZE;
    else if (k == -1)
      pi = PPCI_SUBFME;
    else
      goto needleft;
    left = 0;
  } else {
  needleft:
    left = ra_alloc1(as, ir->op1, rset_exclude(RSET_GPR, right));
  }
  emit_tab(as, pi, dest, right, left);  /* Subtract right _from_ left. */
  ir--;
  dest = ra_dest(as, ir, RSET_GPR);
  right = ra_alloc1(as, ir->op2, RSET_GPR);
  if (irref_isk(ir->op1)) {
    int32_t k = IR(ir->op1)->i;
    if (checki16(k)) {
      emit_tai(as, PPCI_SUBFIC, dest, right, k);
      return;
    }
  }
```

```
  left = ra_alloc1(as, ir->op1, rset_exclude(RSET_GPR, right));
  emit_tab(as, PPCI_SUBFC, dest, right, left);
}

static void asm_neg64(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg left = ra_alloc1(as, ir->op1, RSET_GPR);
  emit_tab(as, PPCI_SUBFZE, dest, left, 0);
  ir--;
  dest = ra_dest(as, ir, RSET_GPR);
  left = ra_alloc1(as, ir->op1, RSET_GPR);
  emit_tai(as, PPCI_SUBFIC, dest, left, 0);
}
#endif

static void asm_bnot(ASMState *as, IRIns *ir)
{
  Reg dest, left, right;
  PPCIns pi = PPCI_NOR;
  if (as->flagmcp == as->mcp) {
    as->flagmcp = NULL;
    as->mcp++;
    pi |= PPCF_DOT;
  }
  dest = ra_dest(as, ir, RSET_GPR);
  if (mayfuse(as, ir->op1)) {
    IRIns *irl = IR(ir->op1);
    if (irl->o == IR_BAND)
      pi ^= (PPCI_NOR ^ PPCI_NAND);
    else if (irl->o == IR_BXOR)
      pi ^= (PPCI_NOR ^ PPCI_EQV);
    else if (irl->o != IR_BOR)
      goto nofuse;
    left = ra_hintalloc(as, irl->op1, dest, RSET_GPR);
    right = ra_alloc1(as, irl->op2, rset_exclude(RSET_GPR, left));
  } else {
nofuse:
    left = right = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
  }
  emit_asb(as, pi, dest, left, right);
}

static void asm_bswap(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  IRIns *irx;
  if (mayfuse(as, ir->op1) && (irx = IR(ir->op1))->o == IR_XLOAD &&
      ra_noreg(irx->r) && (irt_isint(irx->t) || irt_isu32(irx->t))) {
    /* Fuse BSWAP with XLOAD to lwbrx. */
    asm_fusexrefx(as, PPCI_LWBRX, dest, irx->op1, RSET_GPR);
  } else {
    Reg left = ra_alloc1(as, ir->op1, RSET_GPR);
    Reg tmp = dest;
    if (tmp == left) {
      tmp = RID_TMP;
      emit_mr(as, dest, RID_TMP);
    }
    emit_rot(as, PPCI_RLWIMI, tmp, left, 24, 16, 23);
    emit_rot(as, PPCI_RLWIMI, tmp, left, 24, 0, 7);
    emit_rotlwi(as, tmp, left, 8);
  }
}

/* Fuse BAND with contiguous bitmask and a shift to rlwinm. */
static void asm_fuseandsh(ASMState *as, PPCIns pi, int32_t mask, IRRef ref)
{
  IRIns *ir;
  Reg left;
  if (mayfuse(as, ref) && (ir = IR(ref), ra_noreg(ir->r)) &&
      irref_isk(ir->op2) && ir->o >= IR_BSHL && ir->o <= IR_BROR) {
    int32_t sh = (IR(ir->op2)->i & 31);
    switch (ir->o) {
    case IR_BSHL:
      if ((mask & ((1u<<sh)-1))) goto nofuse;
      break;
```

```
    case IR_BSHR:
      if ((mask & ~((~0u)>>sh))) goto nofuse;
      sh = ((32-sh)&31);
      break;
    case IR_BROL:
      break;
    default:
      goto nofuse;
    }
    left = ra_alloc1(as, ir->op1, RSET_GPR);
    *--as->mcp = pi | PPCF_T(left) | PPCF_B(sh);
    return;
  }
nofuse:
  left = ra_alloc1(as, ref, RSET_GPR);
  *--as->mcp = pi | PPCF_T(left);
}

static void asm_band(ASMState *as, IRIns *ir)
{
  Reg dest, left, right;
  IRRef lref = ir->op1;
  PPCIns dot = 0;
  IRRef op2;
  if (as->flagmcp == as->mcp) {
    as->flagmcp = NULL;
    as->mcp++;
    dot = PPCF_DOT;
  }
  dest = ra_dest(as, ir, RSET_GPR);
  if (irref_isk(ir->op2)) {
    int32_t k = IR(ir->op2)->i;
    if (k) {
      /* First check for a contiguous bitmask as used by rlwinm. */
      uint32_t s1 = lj_ffs((uint32_t)k);
      uint32_t k1 = ((uint32_t)k >> s1);
      if ((k1 & (k1+1)) == 0) {
        asm_fuseandsh(as, PPCI_RLWINM|dot | PPCF_A(dest) |
                          PPCF_MB(31-lj_fls((uint32_t)k)) | PPCF_ME(31-s1),
                          k, lref);
        return;
      }
      if (~(uint32_t)k) {
        uint32_t s2 = lj_ffs(~(uint32_t)k);
        uint32_t k2 = (~(uint32_t)k >> s2);
        if ((k2 & (k2+1)) == 0) {
          asm_fuseandsh(as, PPCI_RLWINM|dot | PPCF_A(dest) |
                            PPCF_MB(32-s2) | PPCF_ME(30-lj_fls(~(uint32_t)k)),
                            k, lref);
          return;
        }
      }
    }
    if (checku16(k)) {
      left = ra_alloc1(as, lref, RSET_GPR);
      emit_asi(as, PPCI_ANDIDOT, dest, left, k);
      return;
    } else if ((k & 0xffff) == 0) {
      left = ra_alloc1(as, lref, RSET_GPR);
      emit_asi(as, PPCI_ANDISDOT, dest, left, (k >> 16));
      return;
    }
  }
  op2 = ir->op2;
  if (mayfuse(as, op2) && IR(op2)->o == IR_BNOT && ra_noreg(IR(op2)->r)) {
    dot ^= (PPCI_AND ^ PPCI_ANDC);
    op2 = IR(op2)->op1;
  }
  left = ra_hintalloc(as, lref, dest, RSET_GPR);
  right = ra_alloc1(as, op2, rset_exclude(RSET_GPR, left));
  emit_asb(as, PPCI_AND ^ dot, dest, left, right);
}

static void asm_bitop(ASMState *as, IRIns *ir, PPCIns pi, PPCIns pik)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
```

```
  Reg right, left = ra_hintalloc(as, ir->op1, dest, RSET_GPR);
  if (irref_isk(ir->op2)) {
    int32_t k = IR(ir->op2)->i;
    Reg tmp = left;
    if ((checku16(k) || (k & 0xffff) == 0) || (tmp = dest, !as->sectref)) {
      if (!checku16(k)) {
        emit_asi(as, pik ^ (PPCI_ORI ^ PPCI_ORIS), dest, tmp, (k >> 16));
        if ((k & 0xffff) == 0) return;
      }
      emit_asi(as, pik, dest, left, k);
      return;
    }
  }
  /* May fail due to spills/restores above, but simplifies the logic. */
  if (as->flagmcp == as->mcp) {
    as->flagmcp = NULL;
    as->mcp++;
    pi |= PPCF_DOT;
  }
  right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
  emit_asb(as, pi, dest, left, right);
}

#define asm_bor(as, ir)                asm_bitop(as, ir, PPCI_OR, PPCI_ORI)
#define asm_bxor(as, ir)        asm_bitop(as, ir, PPCI_XOR, PPCI_XORI)

static void asm_bitshift(ASMState *as, IRIns *ir, PPCIns pi, PPCIns pik)
{
  Reg dest, left;
  Reg dot = 0;
  if (as->flagmcp == as->mcp) {
    as->flagmcp = NULL;
    as->mcp++;
    dot = PPCF_DOT;
  }
  dest = ra_dest(as, ir, RSET_GPR);
  left = ra_alloc1(as, ir->op1, RSET_GPR);
  if (irref_isk(ir->op2)) {  /* Constant shifts. */
    int32_t shift = (IR(ir->op2)->i & 31);
    if (pik == 0)  /* SLWI */
      emit_rot(as, PPCI_RLWINM|dot, dest, left, shift, 0, 31-shift);
    else if (pik == 1)  /* SRWI */
      emit_rot(as, PPCI_RLWINM|dot, dest, left, (32-shift)&31, shift, 31);
    else
      emit_asb(as, pik|dot, dest, left, shift);
  } else {
    Reg right = ra_alloc1(as, ir->op2, rset_exclude(RSET_GPR, left));
    emit_asb(as, pi|dot, dest, left, right);
  }
}

#define asm_bshl(as, ir)        asm_bitshift(as, ir, PPCI_SLW, 0)
#define asm_bshr(as, ir)        asm_bitshift(as, ir, PPCI_SRW, 1)
#define asm_bsar(as, ir)        asm_bitshift(as, ir, PPCI_SRAW, PPCI_SRAWI)
#define asm_brol(as, ir) \
  asm_bitshift(as, ir, PPCI_RLWNM|PPCF_MB(0)|PPCF_ME(31), \
                       PPCI_RLWINM|PPCF_MB(0)|PPCF_ME(31))
#define asm_bror(as, ir)        lua_assert(0)

static void asm_min_max(ASMState *as, IRIns *ir, int ismax)
{
  if (irt_isnum(ir->t)) {
    Reg dest = ra_dest(as, ir, RSET_FPR);
    Reg tmp = dest;
    Reg right, left = ra_alloc2(as, ir, RSET_FPR);
    right = (left >> 8); left &= 255;
    if (tmp == left || tmp == right)
      tmp = ra_scratch(as, rset_exclude(rset_exclude(rset_exclude(RSET_FPR,
                                        dest), left), right));
    emit_facb(as, PPCI_FSEL, dest, tmp,
              ismax ? left : right, ismax ? right : left);
    emit_fab(as, PPCI_FSUB, tmp, left, right);
  } else {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    Reg tmp1 = RID_TMP, tmp2 = dest;
    Reg right, left = ra_alloc2(as, ir, RSET_GPR);
```

```
    right = (left >> 8); left &= 255;
    if (tmp2 == left || tmp2 == right)
      tmp2 = ra_scratch(as, rset_exclude(rset_exclude(rset_exclude(RSET_GPR,
                                        dest), left), right));
    emit_tab(as, PPCI_ADD, dest, tmp2, right);
    emit_asb(as, ismax ? PPCI_ANDC : PPCI_AND, tmp2, tmp2, tmp1);
    emit_tab(as, PPCI_SUBFE, tmp1, tmp1, tmp1);
    emit_tab(as, PPCI_SUBFC, tmp2, tmp2, tmp1);
    emit_asi(as, PPCI_XORIS, tmp2, right, 0x8000);
    emit_asi(as, PPCI_XORIS, tmp1, left, 0x8000);
  }
}

#define asm_min(as, ir)               asm_min_max(as, ir, 0)
#define asm_max(as, ir)               asm_min_max(as, ir, 1)

/* -- Comparisons ---------------------------------------------------------- */

#define CC_UNSIGNED        0x08        /* Unsigned integer comparison. */
#define CC_TWO                0x80        /* Check two flags for FP comparison. */

/* Map of comparisons to flags. ORDER IR. */
static const uint8_t asm_compmap[IR_ABC+1] = {
  /* op     int cc                FP cc */
  /* LT  */ CC_GE                + (CC_GE<<4),
  /* GE  */ CC_LT                + (CC_LE<<4) + CC_TWO,
  /* LE  */ CC_GT                + (CC_GE<<4) + CC_TWO,
  /* GT  */ CC_LE                + (CC_LE<<4),
  /* ULT */ CC_GE + CC_UNSIGNED + (CC_GT<<4) + CC_TWO,
  /* UGE */ CC_LT + CC_UNSIGNED + (CC_LT<<4),
  /* ULE */ CC_GT + CC_UNSIGNED + (CC_GT<<4),
  /* UGT */ CC_LE + CC_UNSIGNED + (CC_LT<<4) + CC_TWO,
  /* EQ  */ CC_NE                + (CC_NE<<4),
  /* NE  */ CC_EQ                + (CC_EQ<<4),
  /* ABC */ CC_LE + CC_UNSIGNED + (CC_LT<<4) + CC_TWO  /* Same as UGT. */
};

static void asm_intcomp_(ASMState *as, IRRef lref, IRRef rref, Reg cr, PPCCC cc)
{
  Reg right, left = ra_alloc1(as, lref, RSET_GPR);
  if (irref_isk(rref)) {
    int32_t k = IR(rref)->i;
    if ((cc & CC_UNSIGNED) == 0) {  /* Signed comparison with constant. */
      if (checki16(k)) {
        emit_tai(as, PPCI_CMPWI, cr, left, k);
        /* Signed comparison with zero and referencing previous ins? */
        if (k == 0 && lref == as->curins-1)
          as->flagmcp = as->mcp;  /* Allow elimination of the compare. */
        return;
      } else if ((cc & 3) == (CC_EQ & 3)) {  /* Use CMPLWI for EQ or NE. */
        if (checku16(k)) {
          emit_tai(as, PPCI_CMPLWI, cr, left, k);
          return;
        } else if (!as->sectref && ra_noreg(IR(rref)->r)) {
          emit_tai(as, PPCI_CMPLWI, cr, RID_TMP, k);
          emit_asi(as, PPCI_XORIS, RID_TMP, left, (k >> 16));
          return;
        }
      }
    } else {  /* Unsigned comparison with constant. */
      if (checku16(k)) {
        emit_tai(as, PPCI_CMPLWI, cr, left, k);
        return;
      }
    }
  }
  right = ra_alloc1(as, rref, rset_exclude(RSET_GPR, left));
  emit_tab(as, (cc & CC_UNSIGNED) ? PPCI_CMPLW : PPCI_CMPW, cr, left, right);
}

static void asm_comp(ASMState *as, IRIns *ir)
{
  PPCCC cc = asm_compmap[ir->o];
  if (irt_isnum(ir->t)) {
    Reg right, left = ra_alloc2(as, ir, RSET_FPR);
    right = (left >> 8); left &= 255;
```

```
    asm_guardcc(as, (cc >> 4));
    if ((cc & CC_TWO))
      emit_tab(as, PPCI_CROR, ((cc>>4)&3), ((cc>>4)&3), (CC_EQ&3));
    emit_fab(as, PPCI_FCMPU, 0, left, right);
  } else {
    IRRef lref = ir->op1, rref = ir->op2;
    if (irref_isk(lref) && !irref_isk(rref)) {
      /* Swap constants to the right (only for ABC). */
      IRRef tmp = lref; lref = rref; rref = tmp;
      if ((cc & 2) == 0) cc ^= 1;  /* LT <-> GT, LE <-> GE */
    }
    asm_guardcc(as, cc);
    asm_intcomp_(as, lref, rref, 0, cc);
  }
}

#define asm_equal(as, ir)        asm_comp(as, ir)

#if LJ_HASFFI
/* 64 bit integer comparisons. */
static void asm_comp64(ASMState *as, IRIns *ir)
{
  PPCCC cc = asm_compmap[(ir-1)->o];
  if ((cc&3) == (CC_EQ&3)) {
    asm_guardcc(as, cc);
    emit_tab(as, (cc&4) ? PPCI_CRAND : PPCI_CROR,
             (CC_EQ&3), (CC_EQ&3), 4+(CC_EQ&3));
  } else {
    asm_guardcc(as, CC_EQ);
    emit_tab(as, PPCI_CROR, (CC_EQ&3), (CC_EQ&3), ((cc^~(cc>>2))&1));
    emit_tab(as, (cc&4) ? PPCI_CRAND : PPCI_CRANDC,
             (CC_EQ&3), (CC_EQ&3), 4+(cc&3));
  }
  /* Loword comparison sets cr1 and is unsigned, except for equality. */
  asm_intcomp_(as, (ir-1)->op1, (ir-1)->op2, 4,
               cc | ((cc&3) == (CC_EQ&3) ? 0 : CC_UNSIGNED));
  /* Hiword comparison sets cr0. */
  asm_intcomp_(as, ir->op1, ir->op2, 0, cc);
  as->flagmcp = NULL;  /* Doesn't work here. */
}
#endif

/* -- Support for 64 bit ops in 32 bit mode ------------------------------ */

/* Hiword op of a split 64 bit op. Previous op must be the loword op. */
static void asm_hiop(ASMState *as, IRIns *ir)
{
#if LJ_HASFFI
  /* HIOP is marked as a store because it needs its own DCE logic. */
  int uselo = ra_used(ir-1), usehi = ra_used(ir);  /* Loword/hiword used? */
  if (LJ_UNLIKELY(!(as->flags & JIT_F_OPT_DCE))) uselo = usehi = 1;
  if ((ir-1)->o == IR_CONV) {  /* Conversions to/from 64 bit. */
    as->curins--;  /* Always skip the CONV. */
    if (usehi || uselo)
      asm_conv64(as, ir);
    return;
  } else if ((ir-1)->o <= IR_NE) {  /* 64 bit integer comparisons. ORDER IR. */
    as->curins--;  /* Always skip the loword comparison. */
    asm_comp64(as, ir);
    return;
  } else if ((ir-1)->o == IR_XSTORE) {
    as->curins--;  /* Handle both stores here. */
    if ((ir-1)->r != RID_SINK) {
      asm_xstore_(as, ir, 0);
      asm_xstore_(as, ir-1, 4);
    }
    return;
  }
  if (!usehi) return;  /* Skip unused hiword op for all remaining ops. */
  switch ((ir-1)->o) {
  case IR_ADD: as->curins--; asm_add64(as, ir); break;
  case IR_SUB: as->curins--; asm_sub64(as, ir); break;
  case IR_NEG: as->curins--; asm_neg64(as, ir); break;
  case IR_CALLN:
  case IR_CALLXS:
    if (!uselo)
```

```
      ra_allocref(as, ir->op1, RID2RSET(RID_RETLO));  /* Mark lo op as used. */
      break;
    case IR_CNEWI:
      /* Nothing to do here. Handled by lo op itself. */
      break;
    default: lua_assert(0); break;
    }
#else
  UNUSED(as); UNUSED(ir); lua_assert(0);  /* Unused without FFI. */
#endif
}

/* -- Profiling ----------------------------------------------------------- */

static void asm_prof(ASMState *as, IRIns *ir)
{
  UNUSED(ir);
  asm_guardcc(as, CC_NE);
  emit_asi(as, PPCI_ANDIDOT, RID_TMP, RID_TMP, HOOK_PROFILE);
  emit_lsglptr(as, PPCI_LBZ, RID_TMP,
               (int32_t)offsetof(global_State, hookmask));
}

/* -- Stack handling ------------------------------------------------------ */

/* Check Lua stack size for overflow. Use exit handler as fallback. */
static void asm_stack_check(ASMState *as, BCReg topslot,
                            IRIns *irp, RegSet allow, ExitNo exitno)
{
  /* Try to get an unused temp. register, otherwise spill/restore RID_RET*. */
  Reg tmp, pbase = irp ? (ra_hasreg(irp->r) ? irp->r : RID_TMP) : RID_BASE;
  rset_clear(allow, pbase);
  tmp = allow ? rset_pickbot(allow) :
                (pbase == RID_RETHI ? RID_RETLO : RID_RETHI);
  emit_condbranch(as, PPCI_BC, CC_LT, asm_exitstub_addr(as, exitno));
  if (allow == RSET_EMPTY)  /* Restore temp. register. */
    emit_tai(as, PPCI_LWZ, tmp, RID_SP, SPOFS_TMPW);
  else
    ra_modified(as, tmp);
  emit_ai(as, PPCI_CMPLWI, RID_TMP, (int32_t)(8*topslot));
  emit_tab(as, PPCI_SUBF, RID_TMP, pbase, tmp);
  emit_tai(as, PPCI_LWZ, tmp, tmp, offsetof(lua_State, maxstack));
  if (pbase == RID_TMP)
    emit_getgl(as, RID_TMP, jit_base);
  emit_getgl(as, tmp, cur_L);
  if (allow == RSET_EMPTY)  /* Spill temp. register. */
    emit_tai(as, PPCI_STW, tmp, RID_SP, SPOFS_TMPW);
}

/* Restore Lua stack from on-trace state. */
static void asm_stack_restore(ASMState *as, SnapShot *snap)
{
  SnapEntry *map = &as->T->snapmap[snap->mapofs];
  SnapEntry *flinks = &as->T->snapmap[snap_nextofs(as->T, snap)-1];
  MSize n, nent = snap->nent;
  /* Store the value of all modified slots to the Lua stack. */
  for (n = 0; n < nent; n++) {
    SnapEntry sn = map[n];
    BCReg s = snap_slot(sn);
    int32_t ofs = 8*((int32_t)s-1);
    IRRef ref = snap_ref(sn);
    IRIns *ir = IR(ref);
    if ((sn & SNAP_NORESTORE))
      continue;
    if (irt_isnum(ir->t)) {
      Reg src = ra_alloc1(as, ref, RSET_FPR);
      emit_fai(as, PPCI_STFD, src, RID_BASE, ofs);
    } else {
      Reg type;
      RegSet allow = rset_exclude(RSET_GPR, RID_BASE);
      lua_assert(irt_ispri(ir->t) || irt_isaddr(ir->t) || irt_isinteger(ir->t));
      if (!irt_ispri(ir->t)) {
        Reg src = ra_alloc1(as, ref, allow);
        rset_clear(allow, src);
        emit_tai(as, PPCI_STW, src, RID_BASE, ofs+4);
      }
```

```
1810        if ((sn & (SNAP_CONT|SNAP_FRAME))) {
1811          if (s == 0) continue;  /* Do not overwrite link to previous frame. */
1812          type = ra_allock(as, (int32_t)(*flinks--), allow);
1813        } else {
1814          type = ra_allock(as, (int32_t)irt_toitype(ir->t), allow);
1815        }
1816        emit_tai(as, PPCI_STW, type, RID_BASE, ofs);
1817      }
1818      checkmclim(as);
1819    }
1820    lua_assert(map + nent == flinks);
1821  }
1822
1823  /* -- GC handling ---------------------------------------------------------- */
1824
1825  /* Check GC threshold and do one or more GC steps. */
1826  static void asm_gc_check(ASMState *as)
1827  {
1828    const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_gc_step_jit];
1829    IRRef args[2];
1830    MCLabel l_end;
1831    Reg tmp;
1832    ra_evictset(as, RSET_SCRATCH);
1833    l_end = emit_label(as);
1834    /* Exit trace if in GCSatomic or GCSfinalize. Avoids syncing GC objects. */
1835    asm_guardcc(as, CC_NE);  /* Assumes asm_snap_prep() already done. */
1836    emit_ai(as, PPCI_CMPWI, RID_RET, 0);
1837    args[0] = ASMREF_TMP1;  /* global_State *g */
1838    args[1] = ASMREF_TMP2;  /* MSize steps     */
1839    asm_gencall(as, ci, args);
1840    emit_tai(as, PPCI_ADDI, ra_releasetmp(as, ASMREF_TMP1), RID_JGL, -32768);
1841    tmp = ra_releasetmp(as, ASMREF_TMP2);
1842    emit_loadi(as, tmp, as->gcsteps);
1843    /* Jump around GC step if GC total < GC threshold. */
1844    emit_condbranch(as, PPCI_BC|PPCF_Y, CC_LT, l_end);
1845    emit_ab(as, PPCI_CMPLW, RID_TMP, tmp);
1846    emit_getgl(as, tmp, gc.threshold);
1847    emit_getgl(as, RID_TMP, gc.total);
1848    as->gcsteps = 0;
1849    checkmclim(as);
1850  }
1851
1852  /* -- Loop handling -------------------------------------------------------- */
1853
1854  /* Fixup the loop branch. */
1855  static void asm_loop_fixup(ASMState *as)
1856  {
1857    MCode *p = as->mctop;
1858    MCode *target = as->mcp;
1859    if (as->loopinv) {  /* Inverted loop branch? */
1860      /* asm_guardcc already inverted the cond branch and patched the final b. */
1861      p[-2] = (p[-2] & (0xffff0000u & ~PPCF_Y)) | (((target-p+2) & 0x3fffu) << 2);
1862    } else {
1863      p[-1] = PPCI_B|(((target-p+1)&0x00ffffffu)<<2);
1864    }
1865  }
1866
1867  /* -- Head of trace -------------------------------------------------------- */
1868
1869  /* Coalesce BASE register for a root trace. */
1870  static void asm_head_root_base(ASMState *as)
1871  {
1872    IRIns *ir = IR(REF_BASE);
1873    Reg r = ir->r;
1874    if (ra_hasreg(r)) {
1875      ra_free(as, r);
1876      if (rset_test(as->modset, r) || irt_ismarked(ir->t))
1877        ir->r = RID_INIT;  /* No inheritance for modified BASE register. */
1878      if (r != RID_BASE)
1879        emit_mr(as, r, RID_BASE);
1880    }
1881  }
1882
1883  /* Coalesce BASE register for a side trace. */
1884  static RegSet asm_head_side_base(ASMState *as, IRIns *irp, RegSet allow)
1885  {
```

```
  IRIns *ir = IR(REF_BASE);
  Reg r = ir->r;
  if (ra_hasreg(r)) {
    ra_free(as, r);
    if (rset_test(as->modset, r) || irt_ismarked(ir->t))
      ir->r = RID_INIT;  /* No inheritance for modified BASE register. */
    if (irp->r == r) {
      rset_clear(allow, r);  /* Mark same BASE register as coalesced. */
    } else if (ra_hasreg(irp->r) && rset_test(as->freeset, irp->r)) {
      rset_clear(allow, irp->r);
      emit_mr(as, r, irp->r);  /* Move from coalesced parent reg. */
    } else {
      emit_getgl(as, r, jit_base);  /* Otherwise reload BASE. */
    }
  }
  return allow;
}

/* -- Tail of trace ------------------------------------------------------ */

/* Fixup the tail code. */
static void asm_tail_fixup(ASMState *as, TraceNo lnk)
{
  MCode *p = as->mctop;
  MCode *target;
  int32_t spadj = as->T->spadjust;
  if (spadj == 0) {
    *--p = PPCI_NOP;
    *--p = PPCI_NOP;
    as->mctop = p;
  } else {
    /* Patch stack adjustment. */
    lua_assert(checki16(CFRAME_SIZE+spadj));
    p[-3] = PPCI_ADDI | PPCF_T(RID_TMP) | PPCF_A(RID_SP) | (CFRAME_SIZE+spadj);
    p[-2] = PPCI_STWU | PPCF_T(RID_TMP) | PPCF_A(RID_SP) | spadj;
  }
  /* Patch exit branch. */
  target = lnk ? traceref(as->J, lnk)->mcode : (MCode *)lj_vm_exit_interp;
  p[-1] = PPCI_B|(((target-p+1)&0x00ffffffu)<<2);
}

/* Prepare tail of code. */
static void asm_tail_prep(ASMState *as)
{
  MCode *p = as->mctop - 1;  /* Leave room for exit branch. */
  if (as->loopref) {
    as->invmcp = as->mcp = p;
  } else {
    as->mcp = p-2;  /* Leave room for stack pointer adjustment. */
    as->invmcp = NULL;
  }
}

/* -- Trace setup -------------------------------------------------------- */

/* Ensure there are enough stack slots for call arguments. */
static Reg asm_setup_call_slots(ASMState *as, IRIns *ir, const CCallInfo *ci)
{
  IRRef args[CCI_NARGS_MAX*2];
  uint32_t i, nargs = CCI_XNARGS(ci);
  int nslots = 2, ngpr = REGARG_NUMGPR, nfpr = REGARG_NUMFPR;
  asm_collectargs(as, ir, ci, args);
  for (i = 0; i < nargs; i++)
    if (args[i] && irt_isfp(IR(args[i])->t)) {
      if (nfpr > 0) nfpr--; else nslots = (nslots+3) & ~1;
    } else {
      if (ngpr > 0) ngpr--; else nslots++;
    }
  if (nslots > as->evenspill)  /* Leave room for args in stack slots. */
    as->evenspill = nslots;
  return irt_isfp(ir->t) ? REGSP_HINT(RID_FPRET) : REGSP_HINT(RID_RET);
}

static void asm_setup_target(ASMState *as)
{
  asm_exitstub_setup(as, as->T->nsnap + (as->parent ? 1 : 0));
```

```
1962 }
1963
1964 /* -- Trace patching ------------------------------------------------------ */
1965
1966 /* Patch exit jumps of existing machine code to a new target. */
1967 void lj_asm_patchexit(jit_State *J, GCtrace *T, ExitNo exitno, MCode *target)
1968 {
1969   MCode *p = T->mcode;
1970   MCode *pe = (MCode *)((char *)p + T->szmcode);
1971   MCode *px = exitstub_trace_addr(T, exitno);
1972   MCode *cstart = NULL;
1973   MCode *mcarea = lj_mcode_patch(J, p, 0);
1974   int clearso = 0;
1975   for (; p < pe; p++) {
1976     /* Look for exitstub branch, try to replace with branch to target. */
1977     uint32_t ins = *p;
1978     if ((ins & 0xfc000000u) == 0x40000000u &&
1979         ((ins ^ ((char *)px-(char *)p)) & 0xffffu) == 0) {
1980       ptrdiff_t delta = (char *)target - (char *)p;
1981       if (((ins >> 16) & 3) == (CC_SO&3)) {
1982         clearso = sizeof(MCode);
1983         delta -= sizeof(MCode);
1984       }
1985       /* Many, but not all short-range branches can be patched directly. */
1986       if (((delta + 0x8000) >> 16) == 0) {
1987         *p = (ins & 0xffdf0000u) | ((uint32_t)delta & 0xffffu) |
1988              ((delta & 0x8000) * (PPCF_Y/0x8000));
1989         if (!cstart) cstart = p;
1990       }
1991     } else if ((ins & 0xfc000000u) == PPCI_B &&
1992                ((ins ^ ((char *)px-(char *)p)) & 0x03ffffffu) == 0) {
1993       ptrdiff_t delta = (char *)target - (char *)p;
1994       lua_assert(((delta + 0x02000000) >> 26) == 0);
1995       *p = PPCI_B | ((uint32_t)delta & 0x03ffffffu);
1996       if (!cstart) cstart = p;
1997     }
1998   }
1999   {  /* Always patch long-range branch in exit stub itself. */
2000     ptrdiff_t delta = (char *)target - (char *)px - clearso;
2001     lua_assert(((delta + 0x02000000) >> 26) == 0);
2002     *px = PPCI_B | ((uint32_t)delta & 0x03ffffffu);
2003   }
2004   if (!cstart) cstart = px;
2005   lj_mcode_sync(cstart, px+1);
2006   if (clearso) {  /* Extend the current trace. Ugly workaround. */
2007     MCode *pp = J->cur.mcode;
2008     J->cur.szmcode += sizeof(MCode);
2009     *--pp = PPCI_MCRXR;  /* Clear SO flag. */
2010     J->cur.mcode = pp;
2011     lj_mcode_sync(pp, pp+1);
2012   }
2013   lj_mcode_patch(J, mcarea, 1);
2014 }
2015
```

One Level Up Top Level

src/lj_emit_ppc.h - luajit-2.0-src

Data types defined

- MCLabel

Functions defined

- emit_addptr
- emit_call
- emit_cmpi
- emit_condbranch
- emit_jmp
- emit_kdelta1
- emit_loadi
- emit_loadofs
- emit_lsglptr
- emit_lsptr
- emit_movrr
- emit_rot
- emit_rotlwi
- emit_slwi
- emit_spsub
- emit_storeofs
- emit_tab
- emit_tai

Macros defined

- emit_ab
- emit_ai
- emit_as
- emit_asb
- emit_asi
- emit_canremat
- emit_fab
- emit_fac

- emit_facb
- emit_fai
- emit_fb
- emit_getgl
- emit_label
- emit_loada
- emit_loadn
- emit_mr
- emit_setgl
- emit_setvmstate
- emit_ti

Source code

```
/*
** PPC instruction emitter.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

/* -- Emit basic instructions --------------------------------------------- */

static void emit_tab(ASMState *as, PPCIns pi, Reg rt, Reg ra, Reg rb)
{
  *--as->mcp = pi | PPCF_T(rt) | PPCF_A(ra) | PPCF_B(rb);
}

#define emit_asb(as, pi, ra, rs, rb)        emit_tab(as, (pi), (rs), (ra), (rb))
#define emit_as(as, pi, ra, rs)                emit_tab(as, (pi), (rs), (ra), 0)
#define emit_ab(as, pi, ra, rb)                emit_tab(as, (pi), 0, (ra), (rb))

static void emit_tai(ASMState *as, PPCIns pi, Reg rt, Reg ra, int32_t i)
{
  *--as->mcp = pi | PPCF_T(rt) | PPCF_A(ra) | (i & 0xffff);
}

#define emit_ti(as, pi, rt, i)                emit_tai(as, (pi), (rt), 0, (i))
#define emit_ai(as, pi, ra, i)                emit_tai(as, (pi), 0, (ra), (i))
#define emit_asi(as, pi, ra, rs, i)        emit_tai(as, (pi), (rs), (ra), (i))

#define emit_fab(as, pi, rf, ra, rb) \
  emit_tab(as, (pi), (rf)&31, (ra)&31, (rb)&31)
#define emit_fb(as, pi, rf, rb)                emit_tab(as, (pi), (rf)&31, 0, (rb)&31)
#define emit_fac(as, pi, rf, ra, rc) \
  emit_tab(as, (pi) | PPCF_C((rc) & 31), (rf)&31, (ra)&31, 0)
#define emit_facb(as, pi, rf, ra, rc, rb) \
  emit_tab(as, (pi) | PPCF_C((rc) & 31), (rf)&31, (ra)&31, (rb)&31)
#define emit_fai(as, pi, rf, ra, i)        emit_tai(as, (pi), (rf)&31, (ra), (i))

static void emit_rot(ASMState *as, PPCIns pi, Reg ra, Reg rs,
                     int32_t n, int32_t b, int32_t e)
{
  *--as->mcp = pi | PPCF_T(rs) | PPCF_A(ra) | PPCF_B(n) |
               PPCF_MB(b) | PPCF_ME(e);
}

static void emit_slwi(ASMState *as, Reg ra, Reg rs, int32_t n)
{
  lua_assert(n >= 0 && n < 32);
  emit_rot(as, PPCI_RLWINM, ra, rs, n, 0, 31-n);
}

```

```
static void emit_rotlwi(ASMState *as, Reg ra, Reg rs, int32_t n)
{
  lua_assert(n >= 0 && n < 32);
  emit_rot(as, PPCI_RLWINM, ra, rs, n, 0, 31);
}

/* -- Emit loads/stores ---------------------------------------------- */

/* Prefer rematerialization of BASE/L from global_State over spills. */
#define emit_canremat(ref)        ((ref) <= REF_BASE)

/* Try to find a one step delta relative to another constant. */
static int emit_kdelta1(ASMState *as, Reg t, int32_t i)
{
  RegSet work = ~as->freeset & RSET_GPR;
  while (work) {
    Reg r = rset_picktop(work);
    IRRef ref = regcost_ref(as->cost[r]);
    lua_assert(r != t);
    if (ref < ASMREF_L) {
      int32_t delta = i - (ra_iskref(ref) ? ra_krefk(as, ref) : IR(ref)->i);
      if (checki16(delta)) {
        emit_tai(as, PPCI_ADDI, t, r, delta);
        return 1;
      }
    }
    rset_clear(work, r);
  }
  return 0;  /* Failed. */
}

/* Load a 32 bit constant into a GPR. */
static void emit_loadi(ASMState *as, Reg r, int32_t i)
{
  if (checki16(i)) {
    emit_ti(as, PPCI_LI, r, i);
  } else {
    if ((i & 0xffff)) {
      int32_t jgl = i32ptr(J2G(as->J));
      if ((uint32_t)(i-jgl) < 65536) {
        emit_tai(as, PPCI_ADDI, r, RID_JGL, i-jgl-32768);
        return;
      } else if (emit_kdelta1(as, r, i)) {
        return;
      }
      emit_asi(as, PPCI_ORI, r, r, i);
    }
    emit_ti(as, PPCI_LIS, r, (i >> 16));
  }
}

#define emit_loada(as, r, addr)               emit_loadi(as, (r), i32ptr((addr)))

static Reg ra_allock(ASMState *as, int32_t k, RegSet allow);

/* Get/set from constant pointer. */
static void emit_lsptr(ASMState *as, PPCIns pi, Reg r, void *p, RegSet allow)
{
  int32_t jgl = i32ptr(J2G(as->J));
  int32_t i = i32ptr(p);
  Reg base;
  if ((uint32_t)(i-jgl) < 65536) {
    i = i-jgl-32768;
    base = RID_JGL;
  } else {
    base = ra_allock(as, i-(int16_t)i, allow);
  }
  emit_tai(as, pi, r, base, i);
}

#define emit_loadn(as, r, tv) \
  emit_lsptr(as, PPCI_LFD, ((r) & 31), (void *)(tv), RSET_GPR)

/* Get/set global_State fields. */
static void emit_lsglptr(ASMState *as, PPCIns pi, Reg r, int32_t ofs)
{
```

```
  emit_tai(as, pi, r, RID_JGL, ofs-32768);
}

#define emit_getgl(as, r, field) \
  emit_lsglptr(as, PPCI_LWZ, (r), (int32_t)offsetof(global_State, field))
#define emit_setgl(as, r, field) \
  emit_lsglptr(as, PPCI_STW, (r), (int32_t)offsetof(global_State, field))

/* Trace number is determined from per-trace exit stubs. */
#define emit_setvmstate(as, i)                UNUSED(i)

/* -- Emit control-flow instructions ------------------------------------- */

/* Label for internal jumps. */
typedef MCode *MCLabel;

/* Return label pointing to current PC. */
#define emit_label(as)                ((as)->mcp)

static void emit_condbranch(ASMState *as, PPCIns pi, PPCCC cc, MCode *target)
{
  MCode *p = --as->mcp;
  ptrdiff_t delta = (char *)target - (char *)p;
  lua_assert(((delta + 0x8000) >> 16) == 0);
  pi ^= (delta & 0x8000) * (PPCF_Y/0x8000);
  *p = pi | PPCF_CC(cc) | ((uint32_t)delta & 0xffffu);
}

static void emit_jmp(ASMState *as, MCode *target)
{
  MCode *p = --as->mcp;
  ptrdiff_t delta = (char *)target - (char *)p;
  *p = PPCI_B | (delta & 0x03fffffcu);
}

static void emit_call(ASMState *as, void *target)
{
  MCode *p = --as->mcp;
  ptrdiff_t delta = (char *)target - (char *)p;
  if ((((delta>>2) + 0x00800000) >> 24) == 0) {
    *p = PPCI_BL | (delta & 0x03fffffcu);
  } else {  /* Target out of range: need indirect call. Don't use arg reg. */
    RegSet allow = RSET_GPR & ~RSET_RANGE(RID_R0, REGARG_LASTGPR+1);
    Reg r = ra_allock(as, i32ptr(target), allow);
    *p = PPCI_BCTRL;
    p[-1] = PPCI_MTCTR | PPCF_T(r);
    as->mcp = p-1;
  }
}

/* -- Emit generic operations -------------------------------------------- */

#define emit_mr(as, dst, src) \
  emit_asb(as, PPCI_MR, (dst), (src), (src))

/* Generic move between two regs. */
static void emit_movrr(ASMState *as, IRIns *ir, Reg dst, Reg src)
{
  UNUSED(ir);
  if (dst < RID_MAX_GPR)
    emit_mr(as, dst, src);
  else
    emit_fb(as, PPCI_FMR, dst, src);
}

/* Generic load of register with base and (small) offset address. */
static void emit_loadofs(ASMState *as, IRIns *ir, Reg r, Reg base, int32_t ofs)
{
  if (r < RID_MAX_GPR)
    emit_tai(as, PPCI_LWZ, r, base, ofs);
  else
    emit_fai(as, irt_isnum(ir->t) ? PPCI_LFD : PPCI_LFS, r, base, ofs);
}

/* Generic store of register with base and (small) offset address. */
static void emit_storeofs(ASMState *as, IRIns *ir, Reg r, Reg base, int32_t ofs)
```

```
{
  if (r < RID_MAX_GPR)
    emit_tai(as, PPCI_STW, r, base, ofs);
  else
    emit_fai(as, irt_isnum(ir->t) ? PPCI_STFD : PPCI_STFS, r, base, ofs);
}

/* Emit a compare (for equality) with a constant operand. */
static void emit_cmpi(ASMState *as, Reg r, int32_t k)
{
  if (checki16(k)) {
    emit_ai(as, PPCI_CMPWI, r, k);
  } else if (checku16(k)) {
    emit_ai(as, PPCI_CMPLWI, r, k);
  } else {
    emit_ai(as, PPCI_CMPLWI, RID_TMP, k);
    emit_asi(as, PPCI_XORIS, RID_TMP, r, (k >> 16));
  }
}

/* Add offset to pointer. */
static void emit_addptr(ASMState *as, Reg r, int32_t ofs)
{
  if (ofs) {
    emit_tai(as, PPCI_ADDI, r, r, ofs);
    if (!checki16(ofs))
      emit_tai(as, PPCI_ADDIS, r, r, (ofs + 32768) >> 16);
  }
}

static void emit_spsub(ASMState *as, int32_t ofs)
{
  if (ofs) {
    emit_tai(as, PPCI_STWU, RID_TMP, RID_SP, -ofs);
    emit_tai(as, PPCI_ADDI, RID_TMP, RID_SP,
             CFRAME_SIZE + (as->parent ? as->parent->spadjust : 0));
  }
}

```

One Level Up Top Level

src/lj_asm_x86.h - luajit-2.0-src

Global variables defined

- asm_compmap

Functions defined

- asm_add
- asm_ahustore
- asm_ahuvload
- asm_aref
- asm_bitshift
- asm_bswap
- asm_callx
- asm_callx_func
- asm_cnew
- asm_comp
- asm_comp_int64
- asm_conv
- asm_conv64
- asm_conv_fp_int64
- asm_conv_int64_fp
- asm_count_call_slots
- asm_div
- asm_exitstub_gen
- asm_exitstub_setup
- asm_fparith
- asm_fpmath
- asm_fppowi
- asm_fref
- asm_fuseabase
- asm_fuseahuref
- asm_fusearef
- asm_fusefref

- asm_fuseload
- asm_fuseloadm
- asm_fusestrref
- asm_fusexref
- asm_fxload
- asm_fxstore
- asm_gc_check
- asm_gencall
- asm_guardcc
- asm_head_root_base
- asm_head_side_base
- asm_hiop
- asm_href
- asm_hrefk
- asm_intarith
- asm_intmin_max
- asm_isk32
- asm_ldexp
- asm_lea
- asm_load_lightud64
- asm_loop_fixup
- asm_max
- asm_min
- asm_mod
- asm_mul
- asm_neg
- asm_neg_not
- asm_obar
- asm_pow
- asm_prof
- asm_retf
- asm_setup_call_slots
- asm_setup_target

- asm_setupresult
- asm_sload
- asm_stack_check
- asm_stack_restore
- asm_strref
- asm_strto
- asm_sub
- asm_swapops
- asm_tail_fixup
- asm_tail_prep
- asm_tbar
- asm_tobit
- asm_tointg
- asm_tvptr
- asm_uref
- asm_x87load
- lj_asm_patchexit
- noconflict

Macros defined

- COMPFLAGS
- CONFLICT_SEARCH_LIM
- VCC_P
- VCC_PS
- VCC_S
- VCC_U
- asm_abs
- asm_addov
- asm_atan2
- asm_band
- asm_bnot
- asm_bor
- asm_brol
- asm_bror

- asm_bsar
- asm_bshl
- asm_bshr
- asm_bxor
- asm_cnew
- asm_equal
- asm_fload
- asm_fstore
- asm_fuseloadm
- asm_mulov
- asm_subov
- asm_xload
- asm_xstore

Source code

```
/*
** x86/x64 IR assembler (SSA IR -> machine code).
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

/* -- Guard handling ------------------------------------------------------ */

/* Generate an exit stub group at the bottom of the reserved MCode memory. */
static MCode *asm_exitstub_gen(ASMState *as, ExitNo group)
{
  ExitNo i, groupofs = (group*EXITSTUBS_PER_GROUP) & 0xff;
  MCode *mxp = as->mcbot;
  MCode *mxpstart = mxp;
  if (mxp + (2+2)*EXITSTUBS_PER_GROUP+8+5 >= as->mctop)
    asm_mclimit(as);
  /* Push low byte of exitno for each exit stub. */
  *mxp++ = XI_PUSHi8; *mxp++ = (MCode)groupofs;
  for (i = 1; i < EXITSTUBS_PER_GROUP; i++) {
    *mxp++ = XI_JMPs; *mxp++ = (MCode)((2+2)*(EXITSTUBS_PER_GROUP - i) - 2);
    *mxp++ = XI_PUSHi8; *mxp++ = (MCode)(groupofs + i);
  }
  /* Push the high byte of the exitno for each exit stub group. */
  *mxp++ = XI_PUSHi8; *mxp++ = (MCode)((group*EXITSTUBS_PER_GROUP)>>8);
  /* Store DISPATCH at original stack slot 0. Account for the two push ops. */
  *mxp++ = XI_MOVmi;
  *mxp++ = MODRM(XM_OFS8, 0, RID_ESP);
  *mxp++ = MODRM(XM_SCALE1, RID_ESP, RID_ESP);
  *mxp++ = 2*sizeof(void *);
  *(int32_t *)mxp = ptr2addr(J2GG(as->J)->dispatch); mxp += 4;
  /* Jump to exit handler which fills in the ExitState. */
  *mxp++ = XI_JMP; mxp += 4;
  *((int32_t *)(mxp-4)) = jmprel(mxp, (MCode *)(void *)lj_vm_exit_handler);
  /* Commit the code for this group (even if assembly fails later on). */
  lj_mcode_commitbot(as->J, mxp);
  as->mcbot = mxp;
  as->mclim = as->mcbot + MCLIM_REDZONE;
  return mxpstart;
}

/* Setup all needed exit stubs. */
static void asm_exitstub_setup(ASMState *as, ExitNo nexits)
{
  ExitNo i;
```

```
  if (nexits >= EXITSTUBS_PER_GROUP*LJ_MAX_EXITSTUBGR)
    lj_trace_err(as->J, LJ_TRERR_SNAPOV);
  for (i = 0; i < (nexits+EXITSTUBS_PER_GROUP-1)/EXITSTUBS_PER_GROUP; i++)
    if (as->J->exitstubgroup[i] == NULL)
      as->J->exitstubgroup[i] = asm_exitstub_gen(as, i);
}

/* Emit conditional branch to exit for guard.
** It's important to emit this *after* all registers have been allocated,
** because rematerializations may invalidate the flags.
*/
static void asm_guardcc(ASMState *as, int cc)
{
  MCode *target = exitstub_addr(as->J, as->snapno);
  MCode *p = as->mcp;
  if (LJ_UNLIKELY(p == as->invmcp)) {
    as->loopinv = 1;
    *(int32_t *)(p+1) = jmprel(p+5, target);
    target = p;
    cc ^= 1;
    if (as->realign) {
      emit_sjcc(as, cc, target);
      return;
    }
  }
  emit_jcc(as, cc, target);
}

/* -- Memory operand fusion --------------------------------------------- */

/* Limit linear search to this distance. Avoids O(n^2) behavior. */
#define CONFLICT_SEARCH_LIM        31

/* Check if a reference is a signed 32 bit constant. */
static int asm_isk32(ASMState *as, IRRef ref, int32_t *k)
{
  if (irref_isk(ref)) {
    IRIns *ir = IR(ref);
    if (ir->o != IR_KINT64) {
      *k = ir->i;
      return 1;
    } else if (checki32((int64_t)ir_kint64(ir)->u64)) {
      *k = (int32_t)ir_kint64(ir)->u64;
      return 1;
    }
  }
  return 0;
}

/* Check if there's no conflicting instruction between curins and ref.
** Also avoid fusing loads if there are multiple references.
*/
static int noconflict(ASMState *as, IRRef ref, IROp conflict, int noload)
{
  IRIns *ir = as->ir;
  IRRef i = as->curins;
   if (i > ref + CONFLICT_SEARCH_LIM)
     return 0;  /* Give up, ref is too far away. */
   while (--i > ref) {
     if (ir[i].o == conflict)
       return 0;  /* Conflict found. */
     else if (!noload && (ir[i].op1 == ref || ir[i].op2 == ref))
       return 0;
   }
   return 1;  /* Ok, no conflict. */
 }

 /* Fuse array base into memory operand. */
 static IRRef asm_fuseabase(ASMState *as, IRRef ref)
 {
   IRIns *irb = IR(ref);
   as->mrm.ofs = 0;
   if (irb->o == IR_FLOAD) {
     IRIns *ira = IR(irb->op1);
     lua_assert(irb->op2 == IRFL_TAB_ARRAY);
     /* We can avoid the FLOAD of t->array for colocated arrays. */
```

```
    if (ira->o == IR_TNEW && ira->op1 <= LJ_MAX_COLOSIZE &&
        !neverfuse(as) && noconflict(as, irb->op1, IR_NEWREF, 1)) {
      as->mrm.ofs = (int32_t)sizeof(GCtab);  /* Ofs to colocated array. */
      return irb->op1;  /* Table obj. */
    }
  } else if (irb->o == IR_ADD && irref_isk(irb->op2)) {
    /* Fuse base offset (vararg load). */
    as->mrm.ofs = IR(irb->op2)->i;
    return irb->op1;
  }
  return ref;  /* Otherwise use the given array base. */
}

/* Fuse array reference into memory operand. */
static void asm_fusearef(ASMState *as, IRIns *ir, RegSet allow)
{
  IRIns *irx;
  lua_assert(ir->o == IR_AREF);
  as->mrm.base = (uint8_t)ra_alloc1(as, asm_fuseabase(as, ir->op1), allow);
  irx = IR(ir->op2);
  if (irref_isk(ir->op2)) {
    as->mrm.ofs += 8*irx->i;
    as->mrm.idx = RID_NONE;
  } else {
    rset_clear(allow, as->mrm.base);
    as->mrm.scale = XM_SCALE8;
    /* Fuse a constant ADD (e.g. t[i+1]) into the offset.
    ** Doesn't help much without ABCelim, but reduces register pressure.
    */
    if (!LJ_64 &&  /* Has bad effects with negative index on x64. */
        mayfuse(as, ir->op2) && ra_noreg(irx->r) &&
        irx->o == IR_ADD && irref_isk(irx->op2)) {
      as->mrm.ofs += 8*IR(irx->op2)->i;
      as->mrm.idx = (uint8_t)ra_alloc1(as, irx->op1, allow);
    } else {
      as->mrm.idx = (uint8_t)ra_alloc1(as, ir->op2, allow);
    }
  }
}

/* Fuse array/hash/upvalue reference into memory operand.
** Caveat: this may allocate GPRs for the base/idx registers. Be sure to
** pass the final allow mask, excluding any GPRs used for other inputs.
** In particular: 2-operand GPR instructions need to call ra_dest() first!
*/
static void asm_fuseahuref(ASMState *as, IRRef ref, RegSet allow)
{
  IRIns *ir = IR(ref);
  if (ra_noreg(ir->r)) {
    switch ((IROp)ir->o) {
    case IR_AREF:
      if (mayfuse(as, ref)) {
        asm_fusearef(as, ir, allow);
        return;
      }
      break;
    case IR_HREFK:
      if (mayfuse(as, ref)) {
        as->mrm.base = (uint8_t)ra_alloc1(as, ir->op1, allow);
        as->mrm.ofs = (int32_t)(IR(ir->op2)->op2 * sizeof(Node));
        as->mrm.idx = RID_NONE;
        return;
      }
      break;
    case IR_UREFC:
      if (irref_isk(ir->op1)) {
        GCfunc *fn = ir_kfunc(IR(ir->op1));
        GCupval *uv = &gcref(fn->l.uvptr[(ir->op2 >> 8)])->uv;
        as->mrm.ofs = ptr2addr(&uv->tv);
        as->mrm.base = as->mrm.idx = RID_NONE;
        return;
      }
      break;
    default:
      lua_assert(ir->o == IR_HREF || ir->o == IR_NEWREF || ir->o == IR_UREFO ||
                 ir->o == IR_KKPTR);
```

```
      break;
    }
  }
  as->mrm.base = (uint8_t)ra_alloc1(as, ref, allow);
  as->mrm.ofs = 0;
  as->mrm.idx = RID_NONE;
}

/* Fuse FLOAD/FREF reference into memory operand. */
static void asm_fusefref(ASMState *as, IRIns *ir, RegSet allow)
{
  lua_assert(ir->o == IR_FLOAD || ir->o == IR_FREF);
  as->mrm.ofs = field_ofs[ir->op2];
  as->mrm.idx = RID_NONE;
  if (irref_isk(ir->op1)) {
    as->mrm.ofs += IR(ir->op1)->i;
    as->mrm.base = RID_NONE;
  } else {
    as->mrm.base = (uint8_t)ra_alloc1(as, ir->op1, allow);
  }
}

/* Fuse string reference into memory operand. */
static void asm_fusestrref(ASMState *as, IRIns *ir, RegSet allow)
{
  IRIns *irr;
  lua_assert(ir->o == IR_STRREF);
  as->mrm.base = as->mrm.idx = RID_NONE;
  as->mrm.scale = XM_SCALE1;
  as->mrm.ofs = sizeof(GCstr);
  if (irref_isk(ir->op1)) {
    as->mrm.ofs += IR(ir->op1)->i;
  } else {
    Reg r = ra_alloc1(as, ir->op1, allow);
    rset_clear(allow, r);
    as->mrm.base = (uint8_t)r;
  }
  irr = IR(ir->op2);
  if (irref_isk(ir->op2)) {
    as->mrm.ofs += irr->i;
  } else {
    Reg r;
    /* Fuse a constant add into the offset, e.g. string.sub(s, i+10). */
    if (!LJ_64 &&  /* Has bad effects with negative index on x64. */
        mayfuse(as, ir->op2) && irr->o == IR_ADD && irref_isk(irr->op2)) {
      as->mrm.ofs += IR(irr->op2)->i;
      r = ra_alloc1(as, irr->op1, allow);
    } else {
      r = ra_alloc1(as, ir->op2, allow);
    }
    if (as->mrm.base == RID_NONE)
      as->mrm.base = (uint8_t)r;
    else
      as->mrm.idx = (uint8_t)r;
  }
}

static void asm_fusexref(ASMState *as, IRRef ref, RegSet allow)
{
  IRIns *ir = IR(ref);
  as->mrm.idx = RID_NONE;
  if (ir->o == IR_KPTR || ir->o == IR_KKPTR) {
    as->mrm.ofs = ir->i;
    as->mrm.base = RID_NONE;
  } else if (ir->o == IR_STRREF) {
    asm_fusestrref(as, ir, allow);
  } else {
    as->mrm.ofs = 0;
    if (canfuse(as, ir) && ir->o == IR_ADD && ra_noreg(ir->r)) {
      /* Gather (base+idx*sz)+ofs as emitted by cdata ptr/array indexing. */
      IRIns *irx;
      IRRef idx;
      Reg r;
      if (asm_isk32(as, ir->op2, &as->mrm.ofs)) {  /* Recognize x+ofs. */
        ref = ir->op1;
        ir = IR(ref);
```

```
        if (!(ir->o == IR_ADD && canfuse(as, ir) && ra_noreg(ir->r)))
          goto noadd;
      }
      as->mrm.scale = XM_SCALE1;
      idx = ir->op1;
      ref = ir->op2;
      irx = IR(idx);
      if (!(irx->o == IR_BSHL || irx->o == IR_ADD)) {  /* Try other operand. */
        idx = ir->op2;
        ref = ir->op1;
        irx = IR(idx);
      }
      if (canfuse(as, irx) && ra_noreg(irx->r)) {
        if (irx->o == IR_BSHL && irref_isk(irx->op2) && IR(irx->op2)->i <= 3) {
          /* Recognize idx<<b with b = 0-3, corresponding to sz = (1),2,4,8. */
          idx = irx->op1;
          as->mrm.scale = (uint8_t)(IR(irx->op2)->i << 6);
        } else if (irx->o == IR_ADD && irx->op1 == irx->op2) {
          /* FOLD does idx*2 ==> idx<<1 ==> idx+idx. */
          idx = irx->op1;
          as->mrm.scale = XM_SCALE2;
        }
      }
      r = ra_alloc1(as, idx, allow);
      rset_clear(allow, r);
      as->mrm.idx = (uint8_t)r;
    }
  noadd:
    as->mrm.base = (uint8_t)ra_alloc1(as, ref, allow);
  }
}

/* Fuse load into memory operand. */
static Reg asm_fuseload(ASMState *as, IRRef ref, RegSet allow)
{
  IRIns *ir = IR(ref);
  if (ra_hasreg(ir->r)) {
    if (allow != RSET_EMPTY) {  /* Fast path. */
      ra_noweak(as, ir->r);
      return ir->r;
    }
  fusespill:
    /* Force a spill if only memory operands are allowed (asm_x87load). */
    as->mrm.base = RID_ESP;
    as->mrm.ofs = ra_spill(as, ir);
    as->mrm.idx = RID_NONE;
    return RID_MRM;
  }
  if (ir->o == IR_KNUM) {
    RegSet avail = as->freeset & ~as->modset & RSET_FPR;
    lua_assert(allow != RSET_EMPTY);
    if (!(avail & (avail-1))) {  /* Fuse if less than two regs available. */
      as->mrm.ofs = ptr2addr(ir_knum(ir));
      as->mrm.base = as->mrm.idx = RID_NONE;
      return RID_MRM;
    }
  } else if (ir->o == IR_KINT64) {
    RegSet avail = as->freeset & ~as->modset & RSET_GPR;
    lua_assert(allow != RSET_EMPTY);
    if (!(avail & (avail-1))) {  /* Fuse if less than two regs available. */
      as->mrm.ofs = ptr2addr(ir_kint64(ir));
      as->mrm.base = as->mrm.idx = RID_NONE;
      return RID_MRM;
    }
  } else if (mayfuse(as, ref)) {
    RegSet xallow = (allow & RSET_GPR) ? allow : RSET_GPR;
    if (ir->o == IR_SLOAD) {
      if (!(ir->op2 & (IRSLOAD_PARENT|IRSLOAD_CONVERT)) &&
          noconflict(as, ref, IR_RETF, 0)) {
        as->mrm.base = (uint8_t)ra_alloc1(as, REF_BASE, xallow);
        as->mrm.ofs = 8*((int32_t)ir->op1-1) + ((ir->op2&IRSLOAD_FRAME)?4:0);
        as->mrm.idx = RID_NONE;
        return RID_MRM;
      }
    } else if (ir->o == IR_FLOAD) {
      /* Generic fusion is only ok for 32 bit operand (but see asm_comp). */
```

```
      if ((irt_isint(ir->t) || irt_isu32(ir->t) || irt_isaddr(ir->t)) &&
          noconflict(as, ref, IR_FSTORE, 0)) {
        asm_fusefref(as, ir, xallow);
        return RID_MRM;
      }
    } else if (ir->o == IR_ALOAD || ir->o == IR_HLOAD || ir->o == IR_ULOAD) {
      if (noconflict(as, ref, ir->o + IRDELTA_L2S, 0)) {
        asm_fuseahuref(as, ir->op1, xallow);
        return RID_MRM;
      }
    } else if (ir->o == IR_XLOAD) {
      /* Generic fusion is not ok for 8/16 bit operands (but see asm_comp).
      ** Fusing unaligned memory operands is ok on x86 (except for SIMD types).
      */
      if ((!irt_typerange(ir->t, IRT_I8, IRT_U16)) &&
          noconflict(as, ref, IR_XSTORE, 0)) {
        asm_fusexref(as, ir->op1, xallow);
        return RID_MRM;
      }
    } else if (ir->o == IR_VLOAD) {
      asm_fuseahuref(as, ir->op1, xallow);
      return RID_MRM;
    }
  }
  if (!(as->freeset & allow) && !irref_isk(ref) &&
      (allow == RSET_EMPTY || ra_hasspill(ir->s) || iscrossref(as, ref)))
    goto fusespill;
  return ra_allocref(as, ref, allow);
}

#if LJ_64
/* Don't fuse a 32 bit load into a 64 bit operation. */
static Reg asm_fuseloadm(ASMState *as, IRRef ref, RegSet allow, int is64)
{
  if (is64 && !irt_is64(IR(ref)->t))
    return ra_alloc1(as, ref, allow);
  return asm_fuseload(as, ref, allow);
}
#else
#define asm_fuseloadm(as, ref, allow, is64)  asm_fuseload(as, (ref), (allow))
#endif

/* -- Calls ------------------------------------------------------------- */

/* Count the required number of stack slots for a call. */
static int asm_count_call_slots(ASMState *as, const CCallInfo *ci, IRRef *args)
{
  uint32_t i, nargs = CCI_XNARGS(ci);
  int nslots = 0;
#if LJ_64
  if (LJ_ABI_WIN) {
    nslots = (int)(nargs*2);  /* Only matters for more than four args. */
  } else {
    int ngpr = REGARG_NUMGPR, nfpr = REGARG_NUMFPR;
    for (i = 0; i < nargs; i++)
      if (args[i] && irt_isfp(IR(args[i])->t)) {
        if (nfpr > 0) nfpr--; else nslots += 2;
      } else {
        if (ngpr > 0) ngpr--; else nslots += 2;
      }
  }
#else
  int ngpr = 0;
  if ((ci->flags & CCI_CC_MASK) == CCI_CC_FASTCALL)
    ngpr = 2;
  else if ((ci->flags & CCI_CC_MASK) == CCI_CC_THISCALL)
    ngpr = 1;
  for (i = 0; i < nargs; i++)
    if (args[i] && irt_isfp(IR(args[i])->t)) {
      nslots += irt_isnum(IR(args[i])->t) ? 2 : 1;
    } else {
      if (ngpr > 0) ngpr--; else nslots++;
    }
#endif
  return nslots;
}
```

```
424
425 /* Generate a call to a C function. */
426 static void asm_gencall(ASMState *as, const CCallInfo *ci, IRRef *args)
427 {
428   uint32_t n, nargs = CCI_XNARGS(ci);
429   int32_t ofs = STACKARG_OFS;
430 #if LJ_64
431   uint32_t gprs = REGARG_GPRS;
432   Reg fpr = REGARG_FIRSTFPR;
433 #if !LJ_ABI_WIN
434   MCode *patchnfpr = NULL;
435 #endif
436 #else
437   uint32_t gprs = 0;
438   if ((ci->flags & CCI_CC_MASK) != CCI_CC_CDECL) {
439     if ((ci->flags & CCI_CC_MASK) == CCI_CC_THISCALL)
440       gprs = (REGARG_GPRS & 31);
441     else if ((ci->flags & CCI_CC_MASK) == CCI_CC_FASTCALL)
442       gprs = REGARG_GPRS;
443   }
444 #endif
445   if ((void *)ci->func)
446     emit_call(as, ci->func);
447 #if LJ_64
448   if ((ci->flags & CCI_VARARG)) {  /* Special handling for vararg calls. */
449 #if LJ_ABI_WIN
450     for (n = 0; n < 4 && n < nargs; n++) {
451       IRIns *ir = IR(args[n]);
452       if (irt_isfp(ir->t))  /* Duplicate FPRs in GPRs. */
453         emit_rr(as, XO_MOVDto, (irt_isnum(ir->t) ? REX_64 : 0) | (fpr+n),
454                 ((gprs >> (n*5)) & 31));  /* Either MOVD or MOVQ. */
455     }
456 #else
457     patchnfpr = --as->mcp;  /* Indicate number of used FPRs in register al. */
458     *--as->mcp = XI_MOVrib | RID_EAX;
459 #endif
460   }
461 #endif
462   for (n = 0; n < nargs; n++) {  /* Setup args. */
463     IRRef ref = args[n];
464     IRIns *ir = IR(ref);
465     Reg r;
466 #if LJ_64 && LJ_ABI_WIN
467     /* Windows/x64 argument registers are strictly positional. */
468     r = irt_isfp(ir->t) ? (fpr <= REGARG_LASTFPR ? fpr : 0) : (gprs & 31);
469     fpr++; gprs >>= 5;
470 #elif LJ_64
471     /* POSIX/x64 argument registers are used in order of appearance. */
472     if (irt_isfp(ir->t)) {
473       r = fpr <= REGARG_LASTFPR ? fpr++ : 0;
474     } else {
475       r = gprs & 31; gprs >>= 5;
476     }
477 #else
478     if (ref && irt_isfp(ir->t)) {
479       r = 0;
480     } else {
481       r = gprs & 31; gprs >>= 5;
482       if (!ref) continue;
483     }
484 #endif
485     if (r) {  /* Argument is in a register. */
486       if (r < RID_MAX_GPR && ref < ASMREF_TMP1) {
487 #if LJ_64
488         if (ir->o == IR_KINT64)
489           emit_loadu64(as, r, ir_kint64(ir)->u64);
490         else
491 #endif
492           emit_loadi(as, r, ir->i);
493       } else {
494         lua_assert(rset_test(as->freeset, r));  /* Must have been evicted. */
495         if (ra_hasreg(ir->r)) {
496           ra_noweak(as, ir->r);
497           emit_movrr(as, ir, r, ir->r);
498         } else {
499           ra_allocref(as, ref, RID2RSET(r));
```

```
500         }
501       }
502     } else if (irt_isfp(ir->t)) {  /* FP argument is on stack. */
503       lua_assert(!(irt_isfloat(ir->t) && irref_isk(ref)));  /* No float k. */
504       if (LJ_32 && (ofs & 4) && irref_isk(ref)) {
505         /* Split stores for unaligned FP consts. */
506         emit_movmroi(as, RID_ESP, ofs, (int32_t)ir_knum(ir)->u32.lo);
507         emit_movmroi(as, RID_ESP, ofs+4, (int32_t)ir_knum(ir)->u32.hi);
508       } else {
509         r = ra_alloc1(as, ref, RSET_FPR);
510         emit_rmro(as, irt_isnum(ir->t) ? XO_MOVSDto : XO_MOVSSto,
511                   r, RID_ESP, ofs);
512       }
513       ofs += (LJ_32 && irt_isfloat(ir->t)) ? 4 : 8;
514     } else {  /* Non-FP argument is on stack. */
515       if (LJ_32 && ref < ASMREF_TMP1) {
516         emit_movmroi(as, RID_ESP, ofs, ir->i);
517       } else {
518         r = ra_alloc1(as, ref, RSET_GPR);
519         emit_movtomro(as, REX_64 + r, RID_ESP, ofs);
520       }
521       ofs += sizeof(intptr_t);
522     }
523     checkmclim(as);
524   }
525 #if LJ_64 && !LJ_ABI_WIN
526   if (patchnfpr) *patchnfpr = fpr - REGARG_FIRSTFPR;
527 #endif
528 }
529
530 /* Setup result reg/sp for call. Evict scratch regs. */
531 static void asm_setupresult(ASMState *as, IRIns *ir, const CCallInfo *ci)
532 {
533   RegSet drop = RSET_SCRATCH;
534   int hiop = (LJ_32 && (ir+1)->o == IR_HIOP);
535   if ((ci->flags & CCI_NOFPRCLOBBER))
536     drop &= ~RSET_FPR;
537   if (ra_hasreg(ir->r))
538     rset_clear(drop, ir->r);  /* Dest reg handled below. */
539   if (hiop && ra_hasreg((ir+1)->r))
540     rset_clear(drop, (ir+1)->r);  /* Dest reg handled below. */
541   ra_evictset(as, drop);  /* Evictions must be performed first. */
542   if (ra_used(ir)) {
543     if (irt_isfp(ir->t)) {
544       int32_t ofs = sps_scale(ir->s);  /* Use spill slot or temp slots. */
545 #if LJ_64
546       if ((ci->flags & CCI_CASTU64)) {
547         Reg dest = ir->r;
548         if (ra_hasreg(dest)) {
549           ra_free(as, dest);
550           ra_modified(as, dest);
551           emit_rr(as, XO_MOVD, dest|REX_64, RID_RET);  /* Really MOVQ. */
552         }
553         if (ofs) emit_movtomro(as, RID_RET|REX_64, RID_ESP, ofs);
554       } else {
555         ra_destreg(as, ir, RID_FPRET);
556       }
557 #else
558       /* Number result is in x87 st0 for x86 calling convention. */
559       Reg dest = ir->r;
560       if (ra_hasreg(dest)) {
561         ra_free(as, dest);
562         ra_modified(as, dest);
563         emit_rmro(as, irt_isnum(ir->t) ? XO_MOVSD : XO_MOVSS,
564                   dest, RID_ESP, ofs);
565       }
566       if ((ci->flags & CCI_CASTU64)) {
567         emit_movtomro(as, RID_RETLO, RID_ESP, ofs);
568         emit_movtomro(as, RID_RETHI, RID_ESP, ofs+4);
569       } else {
570         emit_rmro(as, irt_isnum(ir->t) ? XO_FSTPq : XO_FSTPd,
571                   irt_isnum(ir->t) ? XOg_FSTPq : XOg_FSTPd, RID_ESP, ofs);
572       }
573 #endif
574 #if LJ_32
575     } else if (hiop) {
```

```
      ra_destpair(as, ir);
#endif
    } else {
      lua_assert(!irt_ispri(ir->t));
      ra_destreg(as, ir, RID_RET);
    }
  } else if (LJ_32 && irt_isfp(ir->t) && !(ci->flags & CCI_CASTU64)) {
    emit_x87op(as, XI_FPOP);  /* Pop unused result from x87 st0. */
  }
}

/* Return a constant function pointer or NULL for indirect calls. */
static void *asm_callx_func(ASMState *as, IRIns *irf, IRRef func)
{
#if LJ_32
  UNUSED(as);
  if (irref_isk(func))
    return (void *)irf->i;
#else
  if (irref_isk(func)) {
    MCode *p;
    if (irf->o == IR_KINT64)
      p = (MCode *)(void *)ir_k64(irf)->u64;
    else
      p = (MCode *)(void *)(uintptr_t)(uint32_t)irf->i;
    if (p - as->mcp == (int32_t)(p - as->mcp))
      return p;  /* Call target is still in +-2GB range. */
    /* Avoid the indirect case of emit_call(). Try to hoist func addr. */
  }
#endif
  return NULL;
}

static void asm_callx(ASMState *as, IRIns *ir)
{
  IRRef args[CCI_NARGS_MAX*2];
  CCallInfo ci;
  IRRef func;
  IRIns *irf;
  int32_t spadj = 0;
  ci.flags = asm_callx_flags(as, ir);
  asm_collectargs(as, ir, &ci, args);
  asm_setupresult(as, ir, &ci);
#if LJ_32
  /* Have to readjust stack after non-cdecl calls due to callee cleanup. */
  if ((ci.flags & CCI_CC_MASK) != CCI_CC_CDECL)
    spadj = 4 * asm_count_call_slots(as, &ci, args);
#endif
  func = ir->op2; irf = IR(func);
  if (irf->o == IR_CARG) { func = irf->op1; irf = IR(func); }
  ci.func = (ASMFunction)asm_callx_func(as, irf, func);
  if (!(void *)ci.func) {
    /* Use a (hoistable) non-scratch register for indirect calls. */
    RegSet allow = (RSET_GPR & ~RSET_SCRATCH);
    Reg r = ra_alloc1(as, func, allow);
    if (LJ_32) emit_spsub(as, spadj);  /* Above code may cause restores! */
    emit_rr(as, XO_GROUP5, XOg_CALL, r);
  } else if (LJ_32) {
    emit_spsub(as, spadj);
  }
  asm_gencall(as, &ci, args);
}

/* -- Returns ----------------------------------------------------------- */

/* Return to lower frame. Guard that it goes to the right spot. */
static void asm_retf(ASMState *as, IRIns *ir)
{
  Reg base = ra_alloc1(as, REF_BASE, RSET_GPR);
  void *pc = ir_kptr(IR(ir->op2));
  int32_t delta = 1+LJ_FR2+bc_a(*((const BCIns *)pc - 1));
  as->topslot -= (BCReg)delta;
  if ((int32_t)as->topslot < 0) as->topslot = 0;
  irt_setmark(IR(REF_BASE)->t);  /* Children must not coalesce with BASE reg. */
  emit_setgl(as, base, jit_base);
  emit_addptr(as, base, -8*delta);
```

```
  asm_guardcc(as, CC_NE);
  emit_gmroi(as, XG_ARITHi(XOg_CMP), base, -4, ptr2addr(pc));
}

/* -- Type conversions ---------------------------------------------------- */

static void asm_tointg(ASMState *as, IRIns *ir, Reg left)
{
  Reg tmp = ra_scratch(as, rset_exclude(RSET_FPR, left));
  Reg dest = ra_dest(as, ir, RSET_GPR);
  asm_guardcc(as, CC_P);
  asm_guardcc(as, CC_NE);
  emit_rr(as, XO_UCOMISD, left, tmp);
  emit_rr(as, XO_CVTSI2SD, tmp, dest);
  emit_rr(as, XO_XORPS, tmp, tmp);  /* Avoid partial register stall. */
  emit_rr(as, XO_CVTTSD2SI, dest, left);
  /* Can't fuse since left is needed twice. */
}

static void asm_tobit(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  Reg tmp = ra_noreg(IR(ir->op1)->r) ?
              ra_alloc1(as, ir->op1, RSET_FPR) :
              ra_scratch(as, RSET_FPR);
  Reg right = asm_fuseload(as, ir->op2, rset_exclude(RSET_FPR, tmp));
  emit_rr(as, XO_MOVDto, tmp, dest);
  emit_mrm(as, XO_ADDSD, tmp, right);
  ra_left(as, tmp, ir->op1);
}

static void asm_conv(ASMState *as, IRIns *ir)
{
  IRType st = (IRType)(ir->op2 & IRCONV_SRCMASK);
  int st64 = (st == IRT_I64 || st == IRT_U64 || (LJ_64 && st == IRT_P64));
  int stfp = (st == IRT_NUM || st == IRT_FLOAT);
  IRRef lref = ir->op1;
  lua_assert(irt_type(ir->t) != st);
  lua_assert(!(LJ_32 && (irt_isint64(ir->t) || st64)));  /* Handled by SPLIT. */
  if (irt_isfp(ir->t)) {
    Reg dest = ra_dest(as, ir, RSET_FPR);
    if (stfp) {  /* FP to FP conversion. */
      Reg left = asm_fuseload(as, lref, RSET_FPR);
      emit_mrm(as, st == IRT_NUM ? XO_CVTSD2SS : XO_CVTSS2SD, dest, left);
      if (left == dest) return;  /* Avoid the XO_XORPS. */
    } else if (LJ_32 && st == IRT_U32) {  /* U32 to FP conversion on x86. */
      /* number = (2^52+2^51 .. u32) - (2^52+2^51) */
      cTValue *k = lj_ir_k64_find(as->J, U64x(43380000,00000000));
      Reg bias = ra_scratch(as, rset_exclude(RSET_FPR, dest));
      if (irt_isfloat(ir->t))
        emit_rr(as, XO_CVTSD2SS, dest, dest);
      emit_rr(as, XO_SUBSD, dest, bias);  /* Subtract 2^52+2^51 bias. */
      emit_rr(as, XO_XORPS, dest, bias);  /* Merge bias and integer. */
      emit_loadn(as, bias, k);
      emit_mrm(as, XO_MOVD, dest, asm_fuseload(as, lref, RSET_GPR));
      return;
    } else {  /* Integer to FP conversion. */
      Reg left = (LJ_64 && (st == IRT_U32 || st == IRT_U64)) ?
                 ra_alloc1(as, lref, RSET_GPR) :
                 asm_fuseloadm(as, lref, RSET_GPR, st64);
      if (LJ_64 && st == IRT_U64) {
        MCLabel l_end = emit_label(as);
        const void *k = lj_ir_k64_find(as->J, U64x(43f00000,00000000));
        emit_rma(as, XO_ADDSD, dest, k);  /* Add 2^64 to compensate. */
        emit_sjcc(as, CC_NS, l_end);
        emit_rr(as, XO_TEST, left|REX_64, left);  /* Check if u64 >= 2^63. */
      }
      emit_mrm(as, irt_isnum(ir->t) ? XO_CVTSI2SD : XO_CVTSI2SS,
               dest|((LJ_64 && (st64 || st == IRT_U32)) ? REX_64 : 0), left);
    }
    emit_rr(as, XO_XORPS, dest, dest);  /* Avoid partial register stall. */
  } else if (stfp) {  /* FP to integer conversion. */
    if (irt_isguard(ir->t)) {
      /* Checked conversions are only supported from number to int. */
      lua_assert(irt_isint(ir->t) && st == IRT_NUM);
      asm_tointg(as, ir, ra_alloc1(as, lref, RSET_FPR));
```

```
    } else {
      Reg dest = ra_dest(as, ir, RSET_GPR);
      x86Op op = st == IRT_NUM ? XO_CVTTSD2SI : XO_CVTTSS2SI;
      if (LJ_64 ? irt_isu64(ir->t) : irt_isu32(ir->t)) {
        /* LJ_64: For inputs >= 2^63 add -2^64, convert again. */
        /* LJ_32: For inputs >= 2^31 add -2^31, convert again and add 2^31. */
        Reg tmp = ra_noreg(IR(lref)->r) ? ra_alloc1(as, lref, RSET_FPR) :
                                          ra_scratch(as, RSET_FPR);
        MCLabel l_end = emit_label(as);
        if (LJ_32)
          emit_gri(as, XG_ARITHi(XOg_ADD), dest, (int32_t)0x80000000);
        emit_rr(as, op, dest|REX_64, tmp);
        if (st == IRT_NUM)
          emit_rma(as, XO_ADDSD, tmp, lj_ir_k64_find(as->J,
                   LJ_64 ? U64x(c3f00000,00000000) : U64x(c1e00000,00000000)));
        else
          emit_rma(as, XO_ADDSS, tmp, lj_ir_k64_find(as->J,
                   LJ_64 ? U64x(00000000,df800000) : U64x(00000000,cf000000)));
        emit_sjcc(as, CC_NS, l_end);
        emit_rr(as, XO_TEST, dest|REX_64, dest);  /* Check if dest negative. */
        emit_rr(as, op, dest|REX_64, tmp);
        ra_left(as, tmp, lref);
      } else {
        Reg left = asm_fuseload(as, lref, RSET_FPR);
        if (LJ_64 && irt_isu32(ir->t))
          emit_rr(as, XO_MOV, dest, dest);  /* Zero hiword. */
        emit_mrm(as, op,
                 dest|((LJ_64 &&
                        (irt_is64(ir->t) || irt_isu32(ir->t))) ? REX_64 : 0),
                 left);
      }
    }
  } else if (st >= IRT_I8 && st <= IRT_U16) {  /* Extend to 32 bit integer. */
    Reg left, dest = ra_dest(as, ir, RSET_GPR);
    RegSet allow = RSET_GPR;
    x86Op op;
    lua_assert(irt_isint(ir->t) || irt_isu32(ir->t));
    if (st == IRT_I8) {
      op = XO_MOVSXb; allow = RSET_GPR8; dest |= FORCE_REX;
    } else if (st == IRT_U8) {
      op = XO_MOVZXb; allow = RSET_GPR8; dest |= FORCE_REX;
    } else if (st == IRT_I16) {
      op = XO_MOVSXw;
    } else {
      op = XO_MOVZXw;
    }
    left = asm_fuseload(as, lref, allow);
    /* Add extra MOV if source is already in wrong register. */
    if (!LJ_64 && left != RID_MRM && !rset_test(allow, left)) {
      Reg tmp = ra_scratch(as, allow);
      emit_rr(as, op, dest, tmp);
      emit_rr(as, XO_MOV, tmp, left);
    } else {
      emit_mrm(as, op, dest, left);
    }
  } else {  /* 32/64 bit integer conversions. */
    if (LJ_32) {  /* Only need to handle 32/32 bit no-op (cast) on x86. */
      Reg dest = ra_dest(as, ir, RSET_GPR);
      ra_left(as, dest, lref);  /* Do nothing, but may need to move regs. */
    } else if (irt_is64(ir->t)) {
      Reg dest = ra_dest(as, ir, RSET_GPR);
      if (st64 || !(ir->op2 & IRCONV_SEXT)) {
        /* 64/64 bit no-op (cast) or 32 to 64 bit zero extension. */
        ra_left(as, dest, lref);  /* Do nothing, but may need to move regs. */
      } else {  /* 32 to 64 bit sign extension. */
        Reg left = asm_fuseload(as, lref, RSET_GPR);
        emit_mrm(as, XO_MOVSXd, dest|REX_64, left);
      }
    } else {
      Reg dest = ra_dest(as, ir, RSET_GPR);
      if (st64) {
        Reg left = asm_fuseload(as, lref, RSET_GPR);
        /* This is either a 32 bit reg/reg mov which zeroes the hiword
        ** or a load of the loword from a 64 bit address.
        */
        emit_mrm(as, XO_MOV, dest, left);
```

```
      } else {  /* 32/32 bit no-op (cast). */
        ra_left(as, dest, lref);  /* Do nothing, but may need to move regs. */
      }
    }
  }
}

#if LJ_32 && LJ_HASFFI
/* No SSE conversions to/from 64 bit on x86, so resort to ugly x87 code. */

/* 64 bit integer to FP conversion in 32 bit mode. */
static void asm_conv_fp_int64(ASMState *as, IRIns *ir)
{
  Reg hi = ra_alloc1(as, ir->op1, RSET_GPR);
  Reg lo = ra_alloc1(as, (ir-1)->op1, rset_exclude(RSET_GPR, hi));
  int32_t ofs = sps_scale(ir->s);  /* Use spill slot or temp slots. */
  Reg dest = ir->r;
  if (ra_hasreg(dest)) {
    ra_free(as, dest);
    ra_modified(as, dest);
    emit_rmro(as, irt_isnum(ir->t) ? XO_MOVSD : XO_MOVSS, dest, RID_ESP, ofs);
  }
  emit_rmro(as, irt_isnum(ir->t) ? XO_FSTPq : XO_FSTPd,
            irt_isnum(ir->t) ? XOg_FSTPq : XOg_FSTPd, RID_ESP, ofs);
  if (((ir-1)->op2 & IRCONV_SRCMASK) == IRT_U64) {
    /* For inputs in [2^63,2^64-1] add 2^64 to compensate. */
    MCLabel l_end = emit_label(as);
    emit_rma(as, XO_FADDq, XOg_FADDq,
             lj_ir_k64_find(as->J, U64x(43f00000,00000000)));
    emit_sjcc(as, CC_NS, l_end);
    emit_rr(as, XO_TEST, hi, hi);  /* Check if u64 >= 2^63. */
  } else {
    lua_assert(((ir-1)->op2 & IRCONV_SRCMASK) == IRT_I64);
  }
  emit_rmro(as, XO_FILDq, XOg_FILDq, RID_ESP, 0);
  /* NYI: Avoid narrow-to-wide store-to-load forwarding stall. */
  emit_rmro(as, XO_MOVto, hi, RID_ESP, 4);
  emit_rmro(as, XO_MOVto, lo, RID_ESP, 0);
}

/* FP to 64 bit integer conversion in 32 bit mode. */
static void asm_conv_int64_fp(ASMState *as, IRIns *ir)
{
  IRType st = (IRType)((ir-1)->op2 & IRCONV_SRCMASK);
  IRType dt = (((ir-1)->op2 & IRCONV_DSTMASK) >> IRCONV_DSH);
  Reg lo, hi;
  lua_assert(st == IRT_NUM || st == IRT_FLOAT);
  lua_assert(dt == IRT_I64 || dt == IRT_U64);
  hi = ra_dest(as, ir, RSET_GPR);
  lo = ra_dest(as, ir-1, rset_exclude(RSET_GPR, hi));
  if (ra_used(ir-1)) emit_rmro(as, XO_MOV, lo, RID_ESP, 0);
  /* NYI: Avoid wide-to-narrow store-to-load forwarding stall. */
  if (!(as->flags & JIT_F_SSE3)) {  /* Set FPU rounding mode to default. */
    emit_rmro(as, XO_FLDCW, XOg_FLDCW, RID_ESP, 4);
    emit_rmro(as, XO_MOVto, lo, RID_ESP, 4);
    emit_gri(as, XG_ARITHi(XOg_AND), lo, 0xf3ff);
  }
  if (dt == IRT_U64) {
    /* For inputs in [2^63,2^64-1] add -2^64 and convert again. */
    MCLabel l_pop, l_end = emit_label(as);
    emit_x87op(as, XI_FPOP);
    l_pop = emit_label(as);
    emit_sjmp(as, l_end);
    emit_rmro(as, XO_MOV, hi, RID_ESP, 4);
    if ((as->flags & JIT_F_SSE3))
      emit_rmro(as, XO_FISTTPq, XOg_FISTTPq, RID_ESP, 0);
    else
      emit_rmro(as, XO_FISTPq, XOg_FISTPq, RID_ESP, 0);
    emit_rma(as, XO_FADDq, XOg_FADDq,
             lj_ir_k64_find(as->J, U64x(c3f00000,00000000)));
    emit_sjcc(as, CC_NS, l_pop);
    emit_rr(as, XO_TEST, hi, hi);  /* Check if out-of-range (2^63). */
  }
  emit_rmro(as, XO_MOV, hi, RID_ESP, 4);
  if ((as->flags & JIT_F_SSE3)) {  /* Truncation is easy with SSE3. */
    emit_rmro(as, XO_FISTTPq, XOg_FISTTPq, RID_ESP, 0);
```

```
  } else {  /* Otherwise set FPU rounding mode to truncate before the store. */
    emit_rmro(as, XO_FISTPq, XOg_FISTPq, RID_ESP, 0);
    emit_rmro(as, XO_FLDCW, XOg_FLDCW, RID_ESP, 0);
    emit_rmro(as, XO_MOVtow, lo, RID_ESP, 0);
    emit_rmro(as, XO_ARITHw(XOg_OR), lo, RID_ESP, 0);
    emit_loadi(as, lo, 0xc00);
    emit_rmro(as, XO_FNSTCW, XOg_FNSTCW, RID_ESP, 0);
  }
  if (dt == IRT_U64)
    emit_x87op(as, XI_FDUP);
  emit_mrm(as, st == IRT_NUM ? XO_FLDq : XO_FLDd,
           st == IRT_NUM ? XOg_FLDq: XOg_FLDd,
           asm_fuseload(as, ir->op1, RSET_EMPTY));
}

static void asm_conv64(ASMState *as, IRIns *ir)
{
  if (irt_isfp(ir->t))
    asm_conv_fp_int64(as, ir);
  else
    asm_conv_int64_fp(as, ir);
}
#endif

static void asm_strto(ASMState *as, IRIns *ir)
{
  /* Force a spill slot for the destination register (if any). */
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_strscan_num];
  IRRef args[2];
  RegSet drop = RSET_SCRATCH;
  if ((drop & RSET_FPR) != RSET_FPR && ra_hasreg(ir->r))
    rset_set(drop, ir->r);  /* WIN64 doesn't spill all FPRs. */
  ra_evictset(as, drop);
  asm_guardcc(as, CC_E);
  emit_rr(as, XO_TEST, RID_RET, RID_RET);  /* Test return status. */
  args[0] = ir->op1;      /* GCstr *str */
  args[1] = ASMREF_TMP1;  /* TValue *n  */
  asm_gencall(as, ci, args);
  /* Store the result to the spill slot or temp slots. */
  emit_rmro(as, XO_LEA, ra_releasetmp(as, ASMREF_TMP1)|REX_64,
            RID_ESP, sps_scale(ir->s));
}

/* -- Memory references --------------------------------------------------- */

/* Get pointer to TValue. */
static void asm_tvptr(ASMState *as, Reg dest, IRRef ref)
{
  IRIns *ir = IR(ref);
  if (irt_isnum(ir->t)) {
    /* For numbers use the constant itself or a spill slot as a TValue. */
    if (irref_isk(ref))
      emit_loada(as, dest, ir_knum(ir));
    else
      emit_rmro(as, XO_LEA, dest|REX_64, RID_ESP, ra_spill(as, ir));
  } else {
    /* Otherwise use g->tmptv to hold the TValue. */
    if (!irref_isk(ref)) {
      Reg src = ra_alloc1(as, ref, rset_exclude(RSET_GPR, dest));
      emit_movtomro(as, REX_64IR(ir, src), dest, 0);
    } else if (!irt_ispri(ir->t)) {
      emit_movmroi(as, dest, 0, ir->i);
    }
    if (!(LJ_64 && irt_islightud(ir->t)))
      emit_movmroi(as, dest, 4, irt_toitype(ir->t));
    emit_loada(as, dest, &J2G(as->J)->tmptv);
  }
}

static void asm_aref(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  asm_fusearef(as, ir, RSET_GPR);
  if (!(as->mrm.idx == RID_NONE && as->mrm.ofs == 0))
    emit_mrm(as, XO_LEA, dest, RID_MRM);
  else if (as->mrm.base != dest)
```

```
956     emit_rr(as, XO_MOV, dest, as->mrm.base);
957   }
958
959   /* Inlined hash lookup. Specialized for key type and for const keys.
960   ** The equivalent C code is:
961   **   Node *n = hashkey(t, key);
962   **   do {
963   **     if (lj_obj_equal(&n->key, key)) return &n->val;
964   **   } while ((n = nextnode(n)));
965   **   return niltv(L);
966   */
967   static void asm_href(ASMState *as, IRIns *ir, IROp merge)
968   {
969     RegSet allow = RSET_GPR;
970     int destused = ra_used(ir);
971     Reg dest = ra_dest(as, ir, allow);
972     Reg tab = ra_alloc1(as, ir->op1, rset_clear(allow, dest));
973     Reg key = RID_NONE, tmp = RID_NONE;
974     IRIns *irkey = IR(ir->op2);
975     int isk = irref_isk(ir->op2);
976     IRType1 kt = irkey->t;
977     uint32_t khash;
978     MCLabel l_end, l_loop, l_next;
979
980     if (!isk) {
981       rset_clear(allow, tab);
982       key = ra_alloc1(as, ir->op2, irt_isnum(kt) ? RSET_FPR : allow);
983       if (!irt_isstr(kt))
984         tmp = ra_scratch(as, rset_exclude(allow, key));
985     }
986
987     /* Key not found in chain: jump to exit (if merged) or load niltv. */
988     l_end = emit_label(as);
989     if (merge == IR_NE)
990       asm_guardcc(as, CC_E);  /* XI_JMP is not found by lj_asm_patchexit. */
991     else if (destused)
992       emit_loada(as, dest, niltvg(J2G(as->J)));
993
994     /* Follow hash chain until the end. */
995     l_loop = emit_sjcc_label(as, CC_NZ);
996     emit_rr(as, XO_TEST, dest, dest);
997     emit_rmro(as, XO_MOV, dest, dest, offsetof(Node, next));
998     l_next = emit_label(as);
999
1000     /* Type and value comparison. */
1001     if (merge == IR_EQ)
1002       asm_guardcc(as, CC_E);
1003     else
1004       emit_sjcc(as, CC_E, l_end);
1005     if (irt_isnum(kt)) {
1006       if (isk) {
1007         /* Assumes -0.0 is already canonicalized to +0.0. */
1008         emit_gmroi(as, XG_ARITHi(XOg_CMP), dest, offsetof(Node, key.u32.lo),
1009                    (int32_t)ir_knum(irkey)->u32.lo);
1010         emit_sjcc(as, CC_NE, l_next);
1011         emit_gmroi(as, XG_ARITHi(XOg_CMP), dest, offsetof(Node, key.u32.hi),
1012                    (int32_t)ir_knum(irkey)->u32.hi);
1013       } else {
1014         emit_sjcc(as, CC_P, l_next);
1015         emit_rmro(as, XO_UCOMISD, key, dest, offsetof(Node, key.n));
1016         emit_sjcc(as, CC_AE, l_next);
1017         /* The type check avoids NaN penalties and complaints from Valgrind. */
1018   #if LJ_64
1019         emit_u32(as, LJ_TISNUM);
1020         emit_rmro(as, XO_ARITHi, XOg_CMP, dest, offsetof(Node, key.it));
1021   #else
1022         emit_i8(as, LJ_TISNUM);
1023         emit_rmro(as, XO_ARITHi8, XOg_CMP, dest, offsetof(Node, key.it));
1024   #endif
1025       }
1026   #if LJ_64
1027     } else if (irt_islightud(kt)) {
1028       emit_rmro(as, XO_CMP, key|REX_64, dest, offsetof(Node, key.u64));
1029   #endif
1030     } else {
1031       if (!irt_ispri(kt)) {
```

```
      lua_assert(irt_isaddr(kt));
      if (isk)
        emit_gmroi(as, XG_ARITHi(XOg_CMP), dest, offsetof(Node, key.gcr),
                   ptr2addr(ir_kgc(irkey)));
      else
        emit_rmro(as, XO_CMP, key, dest, offsetof(Node, key.gcr));
      emit_sjcc(as, CC_NE, l_next);
    }
    lua_assert(!irt_isnil(kt));
    emit_i8(as, irt_toitype(kt));
    emit_rmro(as, XO_ARITHi8, XOg_CMP, dest, offsetof(Node, key.it));
  }
  emit_sfixup(as, l_loop);
  checkmclim(as);

  /* Load main position relative to tab->node into dest. */
  khash = isk ? ir_khash(irkey) : 1;
  if (khash == 0) {
    emit_rmro(as, XO_MOV, dest, tab, offsetof(GCtab, node));
  } else {
    emit_rmro(as, XO_ARITH(XOg_ADD), dest, tab, offsetof(GCtab, node));
    if ((as->flags & JIT_F_PREFER_IMUL)) {
      emit_i8(as, sizeof(Node));
      emit_rr(as, XO_IMULi8, dest, dest);
    } else {
      emit_shifti(as, XOg_SHL, dest, 3);
      emit_rmrxo(as, XO_LEA, dest, dest, dest, XM_SCALE2, 0);
    }
    if (isk) {
      emit_gri(as, XG_ARITHi(XOg_AND), dest, (int32_t)khash);
      emit_rmro(as, XO_MOV, dest, tab, offsetof(GCtab, hmask));
    } else if (irt_isstr(kt)) {
      emit_rmro(as, XO_ARITH(XOg_AND), dest, key, offsetof(GCstr, hash));
      emit_rmro(as, XO_MOV, dest, tab, offsetof(GCtab, hmask));
    } else {  /* Must match with hashrot() in lj_tab.c. */
      emit_rmro(as, XO_ARITH(XOg_AND), dest, tab, offsetof(GCtab, hmask));
      emit_rr(as, XO_ARITH(XOg_SUB), dest, tmp);
      emit_shifti(as, XOg_ROL, tmp, HASH_ROT3);
      emit_rr(as, XO_ARITH(XOg_XOR), dest, tmp);
      emit_shifti(as, XOg_ROL, dest, HASH_ROT2);
      emit_rr(as, XO_ARITH(XOg_SUB), tmp, dest);
      emit_shifti(as, XOg_ROL, dest, HASH_ROT1);
      emit_rr(as, XO_ARITH(XOg_XOR), tmp, dest);
      if (irt_isnum(kt)) {
        emit_rr(as, XO_ARITH(XOg_ADD), dest, dest);
#if LJ_64
        emit_shifti(as, XOg_SHR|REX_64, dest, 32);
        emit_rr(as, XO_MOV, tmp, dest);
        emit_rr(as, XO_MOVDto, key|REX_64, dest);
#else
        emit_rmro(as, XO_MOV, dest, RID_ESP, ra_spill(as, irkey)+4);
        emit_rr(as, XO_MOVDto, key, tmp);
#endif
      } else {
        emit_rr(as, XO_MOV, tmp, key);
        emit_rmro(as, XO_LEA, dest, key, HASH_BIAS);
      }
    }
  }
}

static void asm_hrefk(ASMState *as, IRIns *ir)
{
  IRIns *kslot = IR(ir->op2);
  IRIns *irkey = IR(kslot->op1);
  int32_t ofs = (int32_t)(kslot->op2 * sizeof(Node));
  Reg dest = ra_used(ir) ? ra_dest(as, ir, RSET_GPR) : RID_NONE;
  Reg node = ra_alloc1(as, ir->op1, RSET_GPR);
#if !LJ_64
  MCLabel l_exit;
#endif
  lua_assert(ofs % sizeof(Node) == 0);
  if (ra_hasreg(dest)) {
    if (ofs != 0) {
      if (dest == node && !(as->flags & JIT_F_LEA_AGU))
        emit_gri(as, XG_ARITHi(XOg_ADD), dest, ofs);
```

```
      else
        emit_rmro(as, XO_LEA, dest, node, ofs);
    } else if (dest != node) {
      emit_rr(as, XO_MOV, dest, node);
    }
  }
  asm_guardcc(as, CC_NE);
#if LJ_64
  if (!irt_ispri(irkey->t)) {
    Reg key = ra_scratch(as, rset_exclude(RSET_GPR, node));
    emit_rmro(as, XO_CMP, key|REX_64, node,
              ofs + (int32_t)offsetof(Node, key.u64));
    lua_assert(irt_isnum(irkey->t) || irt_isgcv(irkey->t));
    /* Assumes -0.0 is already canonicalized to +0.0. */
    emit_loadu64(as, key, irt_isnum(irkey->t) ? ir_knum(irkey)->u64 :
                          ((uint64_t)irt_toitype(irkey->t) << 32) |
                          (uint64_t)(uint32_t)ptr2addr(ir_kgc(irkey)));
  } else {
    lua_assert(!irt_isnil(irkey->t));
    emit_i8(as, irt_toitype(irkey->t));
    emit_rmro(as, XO_ARITHi8, XOg_CMP, node,
              ofs + (int32_t)offsetof(Node, key.it));
  }
#else
  l_exit = emit_label(as);
  if (irt_isnum(irkey->t)) {
    /* Assumes -0.0 is already canonicalized to +0.0. */
    emit_gmroi(as, XG_ARITHi(XOg_CMP), node,
               ofs + (int32_t)offsetof(Node, key.u32.lo),
               (int32_t)ir_knum(irkey)->u32.lo);
    emit_sjcc(as, CC_NE, l_exit);
    emit_gmroi(as, XG_ARITHi(XOg_CMP), node,
               ofs + (int32_t)offsetof(Node, key.u32.hi),
               (int32_t)ir_knum(irkey)->u32.hi);
  } else {
    if (!irt_ispri(irkey->t)) {
      lua_assert(irt_isgcv(irkey->t));
      emit_gmroi(as, XG_ARITHi(XOg_CMP), node,
                 ofs + (int32_t)offsetof(Node, key.gcr),
                 ptr2addr(ir_kgc(irkey)));
      emit_sjcc(as, CC_NE, l_exit);
    }
    lua_assert(!irt_isnil(irkey->t));
    emit_i8(as, irt_toitype(irkey->t));
    emit_rmro(as, XO_ARITHi8, XOg_CMP, node,
              ofs + (int32_t)offsetof(Node, key.it));
  }
#endif
}

static void asm_uref(ASMState *as, IRIns *ir)
{
  /* NYI: Check that UREFO is still open and not aliasing a slot. */
  Reg dest = ra_dest(as, ir, RSET_GPR);
  if (irref_isk(ir->op1)) {
    GCfunc *fn = ir_kfunc(IR(ir->op1));
    MRef *v = &gcref(fn->l.uvptr[(ir->op2 >> 8)])->uv.v;
    emit_rma(as, XO_MOV, dest, v);
  } else {
    Reg uv = ra_scratch(as, RSET_GPR);
    Reg func = ra_alloc1(as, ir->op1, RSET_GPR);
    if (ir->o == IR_UREFC) {
      emit_rmro(as, XO_LEA, dest, uv, offsetof(GCupval, tv));
      asm_guardcc(as, CC_NE);
      emit_i8(as, 1);
      emit_rmro(as, XO_ARITHib, XOg_CMP, uv, offsetof(GCupval, closed));
    } else {
      emit_rmro(as, XO_MOV, dest, uv, offsetof(GCupval, v));
    }
    emit_rmro(as, XO_MOV, uv, func,
              (int32_t)offsetof(GCfuncL, uvptr) + 4*(int32_t)(ir->op2 >> 8));
  }
}

static void asm_fref(ASMState *as, IRIns *ir)
{
```

```
  Reg dest = ra_dest(as, ir, RSET_GPR);
  asm_fusefref(as, ir, RSET_GPR);
  emit_mrm(as, XO_LEA, dest, RID_MRM);
}

static void asm_strref(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, RSET_GPR);
  asm_fusestrref(as, ir, RSET_GPR);
  if (as->mrm.base == RID_NONE)
    emit_loadi(as, dest, as->mrm.ofs);
  else if (as->mrm.base == dest && as->mrm.idx == RID_NONE)
    emit_gri(as, XG_ARITHi(XOg_ADD), dest, as->mrm.ofs);
  else
    emit_mrm(as, XO_LEA, dest, RID_MRM);
}

/* -- Loads and stores ---------------------------------------------------- */

static void asm_fxload(ASMState *as, IRIns *ir)
{
  Reg dest = ra_dest(as, ir, irt_isfp(ir->t) ? RSET_FPR : RSET_GPR);
  x86Op xo;
  if (ir->o == IR_FLOAD)
    asm_fusefref(as, ir, RSET_GPR);
  else
    asm_fusexref(as, ir->op1, RSET_GPR);
  /* ir->op2 is ignored -- unaligned loads are ok on x86. */
  switch (irt_type(ir->t)) {
  case IRT_I8: xo = XO_MOVSXb; break;
  case IRT_U8: xo = XO_MOVZXb; break;
  case IRT_I16: xo = XO_MOVSXw; break;
  case IRT_U16: xo = XO_MOVZXw; break;
  case IRT_NUM: xo = XO_MOVSD; break;
  case IRT_FLOAT: xo = XO_MOVSS; break;
  default:
    if (LJ_64 && irt_is64(ir->t))
      dest |= REX_64;
    else
      lua_assert(irt_isint(ir->t) || irt_isu32(ir->t) || irt_isaddr(ir->t));
    xo = XO_MOV;
    break;
  }
  emit_mrm(as, xo, dest, RID_MRM);
}

#define asm_fload(as, ir)        asm_fxload(as, ir)
#define asm_xload(as, ir)        asm_fxload(as, ir)

static void asm_fxstore(ASMState *as, IRIns *ir)
{
  RegSet allow = RSET_GPR;
  Reg src = RID_NONE, osrc = RID_NONE;
  int32_t k = 0;
  if (ir->r == RID_SINK)
    return;
  /* The IRT_I16/IRT_U16 stores should never be simplified for constant
  ** values since mov word [mem], imm16 has a length-changing prefix.
  */
  if (irt_isi16(ir->t) || irt_isu16(ir->t) || irt_isfp(ir->t) ||
      !asm_isk32(as, ir->op2, &k)) {
    RegSet allow8 = irt_isfp(ir->t) ? RSET_FPR :
                    (irt_isi8(ir->t) || irt_isu8(ir->t)) ? RSET_GPR8 : RSET_GPR;
    src = osrc = ra_alloc1(as, ir->op2, allow8);
    if (!LJ_64 && !rset_test(allow8, src)) {  /* Already in wrong register. */
      rset_clear(allow, osrc);
      src = ra_scratch(as, allow8);
    }
    rset_clear(allow, src);
  }
  if (ir->o == IR_FSTORE) {
    asm_fusefref(as, IR(ir->op1), allow);
  } else {
    asm_fusexref(as, ir->op1, allow);
    if (LJ_32 && ir->o == IR_HIOP) as->mrm.ofs += 4;
  }
```

```
  if (ra_hasreg(src)) {
    x86Op xo;
    switch (irt_type(ir->t)) {
    case IRT_I8: case IRT_U8: xo = XO_MOVtob; src |= FORCE_REX; break;
    case IRT_I16: case IRT_U16: xo = XO_MOVtow; break;
    case IRT_NUM: xo = XO_MOVSDto; break;
    case IRT_FLOAT: xo = XO_MOVSSto; break;
#if LJ_64
    case IRT_LIGHTUD: lua_assert(0);  /* NYI: mask 64 bit lightuserdata. */
#endif
    default:
      if (LJ_64 && irt_is64(ir->t))
        src |= REX_64;
      else
        lua_assert(irt_isint(ir->t) || irt_isu32(ir->t) || irt_isaddr(ir->t));
      xo = XO_MOVto;
      break;
    }
    emit_mrm(as, xo, src, RID_MRM);
    if (!LJ_64 && src != osrc) {
      ra_noweak(as, osrc);
      emit_rr(as, XO_MOV, src, osrc);
    }
  } else {
    if (irt_isi8(ir->t) || irt_isu8(ir->t)) {
      emit_i8(as, k);
      emit_mrm(as, XO_MOVmib, 0, RID_MRM);
    } else {
      lua_assert(irt_is64(ir->t) || irt_isint(ir->t) || irt_isu32(ir->t) ||
                 irt_isaddr(ir->t));
      emit_i32(as, k);
      emit_mrm(as, XO_MOVmi, REX_64IR(ir, 0), RID_MRM);
    }
  }
}

#define asm_fstore(as, ir)        asm_fxstore(as, ir)
#define asm_xstore(as, ir)        asm_fxstore(as, ir)

#if LJ_64
static Reg asm_load_lightud64(ASMState *as, IRIns *ir, int typecheck)
{
  if (ra_used(ir) || typecheck) {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    if (typecheck) {
      Reg tmp = ra_scratch(as, rset_exclude(RSET_GPR, dest));
      asm_guardcc(as, CC_NE);
      emit_i8(as, -2);
      emit_rr(as, XO_ARITHi8, XOg_CMP, tmp);
      emit_shifti(as, XOg_SAR|REX_64, tmp, 47);
      emit_rr(as, XO_MOV, tmp|REX_64, dest);
    }
    return dest;
  } else {
    return RID_NONE;
  }
}
#endif

static void asm_ahuvload(ASMState *as, IRIns *ir)
{
  lua_assert(irt_isnum(ir->t) || irt_ispri(ir->t) || irt_isaddr(ir->t) ||
             (LJ_DUALNUM && irt_isint(ir->t)));
#if LJ_64
  if (irt_islightud(ir->t)) {
    Reg dest = asm_load_lightud64(as, ir, 1);
    if (ra_hasreg(dest)) {
      asm_fuseahuref(as, ir->op1, RSET_GPR);
      emit_mrm(as, XO_MOV, dest|REX_64, RID_MRM);
    }
    return;
  } else
#endif
  if (ra_used(ir)) {
    RegSet allow = irt_isnum(ir->t) ? RSET_FPR : RSET_GPR;
    Reg dest = ra_dest(as, ir, allow);
```

```
    asm_fuseahuref(as, ir->op1, RSET_GPR);
    emit_mrm(as, dest < RID_MAX_GPR ? XO_MOV : XO_MOVSD, dest, RID_MRM);
  } else {
    asm_fuseahuref(as, ir->op1, RSET_GPR);
  }
  /* Always do the type check, even if the load result is unused. */
  as->mrm.ofs += 4;
  asm_guardcc(as, irt_isnum(ir->t) ? CC_AE : CC_NE);
  if (LJ_64 && irt_type(ir->t) >= IRT_NUM) {
    lua_assert(irt_isinteger(ir->t) || irt_isnum(ir->t));
    emit_u32(as, LJ_TISNUM);
    emit_mrm(as, XO_ARITHi, XOg_CMP, RID_MRM);
  } else {
    emit_i8(as, irt_toitype(ir->t));
    emit_mrm(as, XO_ARITHi8, XOg_CMP, RID_MRM);
  }
}

static void asm_ahustore(ASMState *as, IRIns *ir)
{
  if (ir->r == RID_SINK)
    return;
  if (irt_isnum(ir->t)) {
    Reg src = ra_alloc1(as, ir->op2, RSET_FPR);
    asm_fuseahuref(as, ir->op1, RSET_GPR);
    emit_mrm(as, XO_MOVSDto, src, RID_MRM);
#if LJ_64
  } else if (irt_islightud(ir->t)) {
    Reg src = ra_alloc1(as, ir->op2, RSET_GPR);
    asm_fuseahuref(as, ir->op1, rset_exclude(RSET_GPR, src));
    emit_mrm(as, XO_MOVto, src|REX_64, RID_MRM);
#endif
  } else {
    IRIns *irr = IR(ir->op2);
    RegSet allow = RSET_GPR;
    Reg src = RID_NONE;
    if (!irref_isk(ir->op2)) {
      src = ra_alloc1(as, ir->op2, allow);
      rset_clear(allow, src);
    }
    asm_fuseahuref(as, ir->op1, allow);
    if (ra_hasreg(src)) {
      emit_mrm(as, XO_MOVto, src, RID_MRM);
    } else if (!irt_ispri(irr->t)) {
      lua_assert(irt_isaddr(ir->t) || (LJ_DUALNUM && irt_isinteger(ir->t)));
      emit_i32(as, irr->i);
      emit_mrm(as, XO_MOVmi, 0, RID_MRM);
    }
    as->mrm.ofs += 4;
    emit_i32(as, (int32_t)irt_toitype(ir->t));
    emit_mrm(as, XO_MOVmi, 0, RID_MRM);
  }
}

static void asm_sload(ASMState *as, IRIns *ir)
{
  int32_t ofs = 8*((int32_t)ir->op1-1) + ((ir->op2 & IRSLOAD_FRAME) ? 4 : 0);
  IRType1 t = ir->t;
  Reg base;
  lua_assert(!(ir->op2 & IRSLOAD_PARENT));  /* Handled by asm_head_side(). */
  lua_assert(irt_isguard(t) || !(ir->op2 & IRSLOAD_TYPECHECK));
  lua_assert(LJ_DUALNUM ||
             !irt_isint(t) || (ir->op2 & (IRSLOAD_CONVERT|IRSLOAD_FRAME)));
  if ((ir->op2 & IRSLOAD_CONVERT) && irt_isguard(t) && irt_isint(t)) {
    Reg left = ra_scratch(as, RSET_FPR);
    asm_tointg(as, ir, left);  /* Frees dest reg. Do this before base alloc. */
    base = ra_alloc1(as, REF_BASE, RSET_GPR);
    emit_rmro(as, XO_MOVSD, left, base, ofs);
    t.irt = IRT_NUM;  /* Continue with a regular number type check. */
#if LJ_64
  } else if (irt_islightud(t)) {
    Reg dest = asm_load_lightud64(as, ir, (ir->op2 & IRSLOAD_TYPECHECK));
    if (ra_hasreg(dest)) {
      base = ra_alloc1(as, REF_BASE, RSET_GPR);
      emit_rmro(as, XO_MOV, dest|REX_64, base, ofs);
    }
```

```
    return;
#endif
  } else if (ra_used(ir)) {
    RegSet allow = irt_isnum(t) ? RSET_FPR : RSET_GPR;
    Reg dest = ra_dest(as, ir, allow);
    base = ra_alloc1(as, REF_BASE, RSET_GPR);
    lua_assert(irt_isnum(t) || irt_isint(t) || irt_isaddr(t));
    if ((ir->op2 & IRSLOAD_CONVERT)) {
      t.irt = irt_isint(t) ? IRT_NUM : IRT_INT;  /* Check for original type. */
      emit_rmro(as, irt_isint(t) ? XO_CVTSI2SD : XO_CVTTSD2SI, dest, base, ofs);
    } else {
      emit_rmro(as, irt_isnum(t) ? XO_MOVSD : XO_MOV, dest, base, ofs);
    }
  } else {
    if (!(ir->op2 & IRSLOAD_TYPECHECK))
      return;  /* No type check: avoid base alloc. */
    base = ra_alloc1(as, REF_BASE, RSET_GPR);
  }
  if ((ir->op2 & IRSLOAD_TYPECHECK)) {
    /* Need type check, even if the load result is unused. */
    asm_guardcc(as, irt_isnum(t) ? CC_AE : CC_NE);
    if (LJ_64 && irt_type(t) >= IRT_NUM) {
      lua_assert(irt_isinteger(t) || irt_isnum(t));
      emit_u32(as, LJ_TISNUM);
      emit_rmro(as, XO_ARITHi, XOg_CMP, base, ofs+4);
    } else {
      emit_i8(as, irt_toitype(t));
      emit_rmro(as, XO_ARITHi8, XOg_CMP, base, ofs+4);
    }
  }
}

/* -- Allocations --------------------------------------------------------- */

#if LJ_HASFFI
static void asm_cnew(ASMState *as, IRIns *ir)
{
  CTState *cts = ctype_ctsG(J2G(as->J));
  CTypeID id = (CTypeID)IR(ir->op1)->i;
  CTSize sz;
  CTInfo info = lj_ctype_info(cts, id, &sz);
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_mem_newgco];
  IRRef args[4];
  lua_assert(sz != CTSIZE_INVALID || (ir->o == IR_CNEW && ir->op2 != REF_NIL));

  as->gcsteps++;
  asm_setupresult(as, ir, ci);  /* GCcdata * */

  /* Initialize immutable cdata object. */
  if (ir->o == IR_CNEWI) {
    RegSet allow = (RSET_GPR & ~RSET_SCRATCH);
#if LJ_64
    Reg r64 = sz == 8 ? REX_64 : 0;
    if (irref_isk(ir->op2)) {
      IRIns *irk = IR(ir->op2);
      uint64_t k = irk->o == IR_KINT64 ? ir_k64(irk)->u64 :
                                         (uint64_t)(uint32_t)irk->i;
      if (sz == 4 || checki32((int64_t)k)) {
        emit_i32(as, (int32_t)k);
        emit_rmro(as, XO_MOVmi, r64, RID_RET, sizeof(GCcdata));
      } else {
        emit_movtomro(as, RID_ECX + r64, RID_RET, sizeof(GCcdata));
        emit_loadu64(as, RID_ECX, k);
      }
    } else {
      Reg r = ra_alloc1(as, ir->op2, allow);
      emit_movtomro(as, r + r64, RID_RET, sizeof(GCcdata));
    }
#else
    int32_t ofs = sizeof(GCcdata);
    if (sz == 8) {
      ofs += 4; ir++;
      lua_assert(ir->o == IR_HIOP);
    }
    do {
      if (irref_isk(ir->op2)) {
```

```
        emit_movmroi(as, RID_RET, ofs, IR(ir->op2)->i);
      } else {
        Reg r = ra_alloc1(as, ir->op2, allow);
        emit_movtomro(as, r, RID_RET, ofs);
        rset_clear(allow, r);
      }
      if (ofs == sizeof(GCcdata)) break;
      ofs -= 4; ir--;
    } while (1);
#endif
    lua_assert(sz == 4 || sz == 8);
  } else if (ir->op2 != REF_NIL) {  /* Create VLA/VLS/aligned cdata. */
    ci = &lj_ir_callinfo[IRCALL_lj_cdata_newv];
    args[0] = ASMREF_L;     /* lua_State *L */
    args[1] = ir->op1;      /* CTypeID id   */
    args[2] = ir->op2;      /* CTSize sz    */
    args[3] = ASMREF_TMP1;  /* CTSize align */
    asm_gencall(as, ci, args);
    emit_loadi(as, ra_releasetmp(as, ASMREF_TMP1), (int32_t)ctype_align(info));
    return;
  }

  /* Combine initialization of marked, gct and ctypeid. */
  emit_movtomro(as, RID_ECX, RID_RET, offsetof(GCcdata, marked));
  emit_gri(as, XG_ARITHi(XOg_OR), RID_ECX,
           (int32_t)((~LJ_TCDATA<<8)+(id<<16)));
  emit_gri(as, XG_ARITHi(XOg_AND), RID_ECX, LJ_GC_WHITES);
  emit_opgl(as, XO_MOVZXb, RID_ECX, gc.currentwhite);

  args[0] = ASMREF_L;     /* lua_State *L */
  args[1] = ASMREF_TMP1;  /* MSize size   */
  asm_gencall(as, ci, args);
  emit_loadi(as, ra_releasetmp(as, ASMREF_TMP1), (int32_t)(sz+sizeof(GCcdata)));
}
#else
#define asm_cnew(as, ir)        ((void)0)
#endif

/* -- Write barriers ------------------------------------------------------ */

static void asm_tbar(ASMState *as, IRIns *ir)
{
  Reg tab = ra_alloc1(as, ir->op1, RSET_GPR);
  Reg tmp = ra_scratch(as, rset_exclude(RSET_GPR, tab));
  MCLabel l_end = emit_label(as);
  emit_movtomro(as, tmp, tab, offsetof(GCtab, gclist));
  emit_setgl(as, tab, gc.grayagain);
  emit_getgl(as, tmp, gc.grayagain);
  emit_i8(as, ~LJ_GC_BLACK);
  emit_rmro(as, XO_ARITHib, XOg_AND, tab, offsetof(GCtab, marked));
  emit_sjcc(as, CC_Z, l_end);
  emit_i8(as, LJ_GC_BLACK);
  emit_rmro(as, XO_GROUP3b, XOg_TEST, tab, offsetof(GCtab, marked));
}

static void asm_obar(ASMState *as, IRIns *ir)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_gc_barrieruv];
  IRRef args[2];
  MCLabel l_end;
  Reg obj;
  /* No need for other object barriers (yet). */
  lua_assert(IR(ir->op1)->o == IR_UREFC);
  ra_evictset(as, RSET_SCRATCH);
  l_end = emit_label(as);
  args[0] = ASMREF_TMP1;  /* global_State *g */
  args[1] = ir->op1;      /* TValue *tv      */
  asm_gencall(as, ci, args);
  emit_loada(as, ra_releasetmp(as, ASMREF_TMP1), J2G(as->J));
  obj = IR(ir->op1)->r;
  emit_sjcc(as, CC_Z, l_end);
  emit_i8(as, LJ_GC_WHITES);
  if (irref_isk(ir->op2)) {
    GCobj *vp = ir_kgc(IR(ir->op2));
    emit_rma(as, XO_GROUP3b, XOg_TEST, &vp->gch.marked);
  } else {
```

```
    Reg val = ra_alloc1(as, ir->op2, rset_exclude(RSET_SCRATCH&RSET_GPR, obj));
    emit_rmro(as, XO_GROUP3b, XOg_TEST, val, (int32_t)offsetof(GChead, marked));
  }
  emit_sjcc(as, CC_Z, l_end);
  emit_i8(as, LJ_GC_BLACK);
  emit_rmro(as, XO_GROUP3b, XOg_TEST, obj,
            (int32_t)offsetof(GCupval, marked)-(int32_t)offsetof(GCupval, tv));
}

/* -- FP/int arithmetic and logic operations ------------------------------ */

/* Load reference onto x87 stack. Force a spill to memory if needed. */
static void asm_x87load(ASMState *as, IRRef ref)
{
  IRIns *ir = IR(ref);
  if (ir->o == IR_KNUM) {
    cTValue *tv = ir_knum(ir);
    if (tvispzero(tv))  /* Use fldz only for +0. */
      emit_x87op(as, XI_FLDZ);
    else if (tvispone(tv))
      emit_x87op(as, XI_FLD1);
    else
      emit_rma(as, XO_FLDq, XOg_FLDq, tv);
  } else if (ir->o == IR_CONV && ir->op2 == IRCONV_NUM_INT && !ra_used(ir) &&
             !irref_isk(ir->op1) && mayfuse(as, ir->op1)) {
    IRIns *iri = IR(ir->op1);
    emit_rmro(as, XO_FILDd, XOg_FILDd, RID_ESP, ra_spill(as, iri));
  } else {
    emit_mrm(as, XO_FLDq, XOg_FLDq, asm_fuseload(as, ref, RSET_EMPTY));
  }
}

static void asm_fpmath(ASMState *as, IRIns *ir)
{
  IRFPMathOp fpm = (IRFPMathOp)ir->op2;
  if (fpm == IRFPM_SQRT) {
    Reg dest = ra_dest(as, ir, RSET_FPR);
    Reg left = asm_fuseload(as, ir->op1, RSET_FPR);
    emit_mrm(as, XO_SQRTSD, dest, left);
  } else if (fpm <= IRFPM_TRUNC) {
    if (as->flags & JIT_F_SSE4_1) {  /* SSE4.1 has a rounding instruction. */
      Reg dest = ra_dest(as, ir, RSET_FPR);
      Reg left = asm_fuseload(as, ir->op1, RSET_FPR);
      /* ROUNDSD has a 4-byte opcode which doesn't fit in x86Op.
      ** Let's pretend it's a 3-byte opcode, and compensate afterwards.
      ** This is atrocious, but the alternatives are much worse.
      */
      /* Round down/up/trunc == 1001/1010/1011. */
      emit_i8(as, 0x09 + fpm);
      emit_mrm(as, XO_ROUNDSD, dest, left);
      if (LJ_64 && as->mcp[1] != (MCode)(XO_ROUNDSD >> 16)) {
        as->mcp[0] = as->mcp[1]; as->mcp[1] = 0x0f;  /* Swap 0F and REX. */
      }
      *--as->mcp = 0x66;  /* 1st byte of ROUNDSD opcode. */
    } else {  /* Call helper functions for SSE2 variant. */
      /* The modified regs must match with the *.dasc implementation. */
      RegSet drop = RSET_RANGE(RID_XMM0, RID_XMM3+1)|RID2RSET(RID_EAX);
      if (ra_hasreg(ir->r))
        rset_clear(drop, ir->r);  /* Dest reg handled below. */
      ra_evictset(as, drop);
      ra_destreg(as, ir, RID_XMM0);
      emit_call(as, fpm == IRFPM_FLOOR ? lj_vm_floor_sse :
                    fpm == IRFPM_CEIL ? lj_vm_ceil_sse : lj_vm_trunc_sse);
      ra_left(as, RID_XMM0, ir->op1);
    }
  } else if (fpm == IRFPM_EXP2 && asm_fpjoin_pow(as, ir)) {
    /* Rejoined to pow(). */
  } else {
    asm_callid(as, ir, IRCALL_lj_vm_floor + fpm);
  }
}

#define asm_atan2(as, ir)        asm_callid(as, ir, IRCALL_atan2)

static void asm_ldexp(ASMState *as, IRIns *ir)
{
```

```
  int32_t ofs = sps_scale(ir->s);  /* Use spill slot or temp slots. */
  Reg dest = ir->r;
  if (ra_hasreg(dest)) {
    ra_free(as, dest);
    ra_modified(as, dest);
    emit_rmro(as, XO_MOVSD, dest, RID_ESP, ofs);
  }
  emit_rmro(as, XO_FSTPq, XOg_FSTPq, RID_ESP, ofs);
  emit_x87op(as, XI_FPOP1);
  emit_x87op(as, XI_FSCALE);
  asm_x87load(as, ir->op1);
  asm_x87load(as, ir->op2);
}

static void asm_fppowi(ASMState *as, IRIns *ir)
{
  /* The modified regs must match with the *.dasc implementation. */
  RegSet drop = RSET_RANGE(RID_XMM0, RID_XMM1+1)|RID2RSET(RID_EAX);
  if (ra_hasreg(ir->r))
    rset_clear(drop, ir->r);  /* Dest reg handled below. */
  ra_evictset(as, drop);
  ra_destreg(as, ir, RID_XMM0);
  emit_call(as, lj_vm_powi_sse);
  ra_left(as, RID_XMM0, ir->op1);
  ra_left(as, RID_EAX, ir->op2);
}

static void asm_pow(ASMState *as, IRIns *ir)
{
#if LJ_64 && LJ_HASFFI
  if (!irt_isnum(ir->t))
    asm_callid(as, ir, irt_isi64(ir->t) ? IRCALL_lj_carith_powi64 :
                                          IRCALL_lj_carith_powu64);
  else
#endif
    asm_fppowi(as, ir);
}

static int asm_swapops(ASMState *as, IRIns *ir)
{
  IRIns *irl = IR(ir->op1);
  IRIns *irr = IR(ir->op2);
  lua_assert(ra_noreg(irr->r));
  if (!irm_iscomm(lj_ir_mode[ir->o]))
    return 0;  /* Can't swap non-commutative operations. */
  if (irref_isk(ir->op2))
    return 0;  /* Don't swap constants to the left. */
  if (ra_hasreg(irl->r))
    return 1;  /* Swap if left already has a register. */
  if (ra_samehint(ir->r, irr->r))
    return 1;  /* Swap if dest and right have matching hints. */
  if (as->curins > as->loopref) {  /* In variant part? */
    if (ir->op2 < as->loopref && !irt_isphi(irr->t))
      return 0;  /* Keep invariants on the right. */
    if (ir->op1 < as->loopref && !irt_isphi(irl->t))
      return 1;  /* Swap invariants to the right. */
  }
  if (opisfusableload(irl->o))
    return 1;  /* Swap fusable loads to the right. */
  return 0;  /* Otherwise don't swap. */
}

static void asm_fparith(ASMState *as, IRIns *ir, x86Op xo)
{
  IRRef lref = ir->op1;
  IRRef rref = ir->op2;
  RegSet allow = RSET_FPR;
  Reg dest;
  Reg right = IR(rref)->r;
  if (ra_hasreg(right)) {
    rset_clear(allow, right);
    ra_noweak(as, right);
  }
  dest = ra_dest(as, ir, allow);
  if (lref == rref) {
    right = dest;
```

```
  } else if (ra_noreg(right)) {
    if (asm_swapops(as, ir)) {
      IRRef tmp = lref; lref = rref; rref = tmp;
    }
    right = asm_fuseload(as, rref, rset_clear(allow, dest));
  }
  emit_mrm(as, xo, dest, right);
  ra_left(as, dest, lref);
}

static void asm_intarith(ASMState *as, IRIns *ir, x86Arith xa)
{
  IRRef lref = ir->op1;
  IRRef rref = ir->op2;
  RegSet allow = RSET_GPR;
  Reg dest, right;
  int32_t k = 0;
  if (as->flagmcp == as->mcp) {  /* Drop test r,r instruction. */
    MCode *p = as->mcp + ((LJ_64 && *as->mcp < XI_TESTb) ? 3 : 2);
    if ((p[1] & 15) < 14) {
      if ((p[1] & 15) >= 12) p[1] -= 4;  /* L <->S, NL <-> NS */
      as->flagmcp = NULL;
      as->mcp = p;
    }  /* else: cannot transform LE/NLE to cc without use of OF. */
  }
  right = IR(rref)->r;
  if (ra_hasreg(right)) {
    rset_clear(allow, right);
    ra_noweak(as, right);
  }
  dest = ra_dest(as, ir, allow);
  if (lref == rref) {
    right = dest;
  } else if (ra_noreg(right) && !asm_isk32(as, rref, &k)) {
    if (asm_swapops(as, ir)) {
      IRRef tmp = lref; lref = rref; rref = tmp;
    }
    right = asm_fuseloadm(as, rref, rset_clear(allow, dest), irt_is64(ir->t));
  }
  if (irt_isguard(ir->t))  /* For IR_ADDOV etc. */
    asm_guardcc(as, CC_O);
  if (xa != XOg_X_IMUL) {
    if (ra_hasreg(right))
      emit_mrm(as, XO_ARITH(xa), REX_64IR(ir, dest), right);
    else
      emit_gri(as, XG_ARITHi(xa), REX_64IR(ir, dest), k);
  } else if (ra_hasreg(right)) {  /* IMUL r, mrm. */
    emit_mrm(as, XO_IMUL, REX_64IR(ir, dest), right);
  } else {  /* IMUL r, r, k. */
    /* NYI: use lea/shl/add/sub (FOLD only does 2^k) depending on CPU. */
    Reg left = asm_fuseloadm(as, lref, RSET_GPR, irt_is64(ir->t));
    x86Op xo;
    if (checki8(k)) { emit_i8(as, k); xo = XO_IMULi8;
    } else { emit_i32(as, k); xo = XO_IMULi; }
    emit_mrm(as, xo, REX_64IR(ir, dest), left);
    return;
  }
  ra_left(as, dest, lref);
}

/* LEA is really a 4-operand ADD with an independent destination register,
** up to two source registers and an immediate. One register can be scaled
** by 1, 2, 4 or 8. This can be used to avoid moves or to fuse several
** instructions.
**
** Currently only a few common cases are supported:
** - 3-operand ADD:    y = a+b; y = a+k   with a and b already allocated
** - Left ADD fusion:  y = (a+b)+k; y = (a+k)+b
** - Right ADD fusion: y = a+(b+k)
** The ommited variants have already been reduced by FOLD.
**
** There are more fusion opportunities, like gathering shifts or joining
** common references. But these are probably not worth the trouble, since
** array indexing is not decomposed and already makes use of all fields
** of the ModRM operand.
*/
```

```
1792 static int asm_lea(ASMState *as, IRIns *ir)
1793 {
1794   IRIns *irl = IR(ir->op1);
1795   IRIns *irr = IR(ir->op2);
1796   RegSet allow = RSET_GPR;
1797   Reg dest;
1798   as->mrm.base = as->mrm.idx = RID_NONE;
1799   as->mrm.scale = XM_SCALE1;
1800   as->mrm.ofs = 0;
1801   if (ra_hasreg(irl->r)) {
1802     rset_clear(allow, irl->r);
1803     ra_noweak(as, irl->r);
1804     as->mrm.base = irl->r;
1805     if (irref_isk(ir->op2) || ra_hasreg(irr->r)) {
1806       /* The PHI renaming logic does a better job in some cases. */
1807       if (ra_hasreg(ir->r) &&
1808           ((irt_isphi(irl->t) && as->phireg[ir->r] == ir->op1) ||
1809            (irt_isphi(irr->t) && as->phireg[ir->r] == ir->op2)))
1810         return 0;
1811       if (irref_isk(ir->op2)) {
1812         as->mrm.ofs = irr->i;
1813       } else {
1814         rset_clear(allow, irr->r);
1815         ra_noweak(as, irr->r);
1816         as->mrm.idx = irr->r;
1817       }
1818     } else if (irr->o == IR_ADD && mayfuse(as, ir->op2) &&
1819                irref_isk(irr->op2)) {
1820       Reg idx = ra_alloc1(as, irr->op1, allow);
1821       rset_clear(allow, idx);
1822       as->mrm.idx = (uint8_t)idx;
1823       as->mrm.ofs = IR(irr->op2)->i;
1824     } else {
1825       return 0;
1826     }
1827   } else if (ir->op1 != ir->op2 && irl->o == IR_ADD && mayfuse(as, ir->op1) &&
1828              (irref_isk(ir->op2) || irref_isk(irl->op2))) {
1829     Reg idx, base = ra_alloc1(as, irl->op1, allow);
1830     rset_clear(allow, base);
1831     as->mrm.base = (uint8_t)base;
1832     if (irref_isk(ir->op2)) {
1833       as->mrm.ofs = irr->i;
1834       idx = ra_alloc1(as, irl->op2, allow);
1835     } else {
1836       as->mrm.ofs = IR(irl->op2)->i;
1837       idx = ra_alloc1(as, ir->op2, allow);
1838     }
1839     rset_clear(allow, idx);
1840     as->mrm.idx = (uint8_t)idx;
1841   } else {
1842     return 0;
1843   }
1844   dest = ra_dest(as, ir, allow);
1845   emit_mrm(as, XO_LEA, dest, RID_MRM);
1846   return 1;  /* Success. */
1847 }
1848
1849 static void asm_add(ASMState *as, IRIns *ir)
1850 {
1851   if (irt_isnum(ir->t))
1852     asm_fparith(as, ir, XO_ADDSD);
1853   else if ((as->flags & JIT_F_LEA_AGU) || as->flagmcp == as->mcp ||
1854            irt_is64(ir->t) || !asm_lea(as, ir))
1855     asm_intarith(as, ir, XOg_ADD);
1856 }
1857
1858 static void asm_sub(ASMState *as, IRIns *ir)
1859 {
1860   if (irt_isnum(ir->t))
1861     asm_fparith(as, ir, XO_SUBSD);
1862   else  /* Note: no need for LEA trick here. i-k is encoded as i+(-k). */
1863     asm_intarith(as, ir, XOg_SUB);
1864 }
1865
1866 static void asm_mul(ASMState *as, IRIns *ir)
1867 {
```

```
1868   if (irt_isnum(ir->t))
1869     asm_fparith(as, ir, XO_MULSD);
1870   else
1871     asm_intarith(as, ir, XOg_X_IMUL);
1872 }
1873
1874 static void asm_div(ASMState *as, IRIns *ir)
1875 {
1876 #if LJ_64 && LJ_HASFFI
1877   if (!irt_isnum(ir->t))
1878     asm_callid(as, ir, irt_isi64(ir->t) ? IRCALL_lj_carith_divi64 :
1879                                           IRCALL_lj_carith_divu64);
1880   else
1881 #endif
1882     asm_fparith(as, ir, XO_DIVSD);
1883 }
1884
1885 static void asm_mod(ASMState *as, IRIns *ir)
1886 {
1887 #if LJ_64 && LJ_HASFFI
1888   if (!irt_isint(ir->t))
1889     asm_callid(as, ir, irt_isi64(ir->t) ? IRCALL_lj_carith_modi64 :
1890                                           IRCALL_lj_carith_modu64);
1891   else
1892 #endif
1893     asm_callid(as, ir, IRCALL_lj_vm_modi);
1894 }
1895
1896 static void asm_neg_not(ASMState *as, IRIns *ir, x86Group3 xg)
1897 {
1898   Reg dest = ra_dest(as, ir, RSET_GPR);
1899   emit_rr(as, XO_GROUP3, REX_64IR(ir, xg), dest);
1900   ra_left(as, dest, ir->op1);
1901 }
1902
1903 static void asm_neg(ASMState *as, IRIns *ir)
1904 {
1905   if (irt_isnum(ir->t))
1906     asm_fparith(as, ir, XO_XORPS);
1907   else
1908     asm_neg_not(as, ir, XOg_NEG);
1909 }
1910
1911 #define asm_abs(as, ir)               asm_fparith(as, ir, XO_ANDPS)
1912
1913 static void asm_intmin_max(ASMState *as, IRIns *ir, int cc)
1914 {
1915   Reg right, dest = ra_dest(as, ir, RSET_GPR);
1916   IRRef lref = ir->op1, rref = ir->op2;
1917   if (irref_isk(rref)) { lref = rref; rref = ir->op1; }
1918   right = ra_alloc1(as, rref, rset_exclude(RSET_GPR, dest));
1919   emit_rr(as, XO_CMOV + (cc<<24), REX_64IR(ir, dest), right);
1920   emit_rr(as, XO_CMP, REX_64IR(ir, dest), right);
1921   ra_left(as, dest, lref);
1922 }
1923
1924 static void asm_min(ASMState *as, IRIns *ir)
1925 {
1926   if (irt_isnum(ir->t))
1927     asm_fparith(as, ir, XO_MINSD);
1928   else
1929     asm_intmin_max(as, ir, CC_G);
1930 }
1931
1932 static void asm_max(ASMState *as, IRIns *ir)
1933 {
1934   if (irt_isnum(ir->t))
1935     asm_fparith(as, ir, XO_MAXSD);
1936   else
1937     asm_intmin_max(as, ir, CC_L);
1938 }
1939
1940 /* Note: don't use LEA for overflow-checking arithmetic! */
1941 #define asm_addov(as, ir)        asm_intarith(as, ir, XOg_ADD)
1942 #define asm_subov(as, ir)        asm_intarith(as, ir, XOg_SUB)
1943 #define asm_mulov(as, ir)        asm_intarith(as, ir, XOg_X_IMUL)
```

```
1944
1945  #define asm_bnot(as, ir)        asm_neg_not(as, ir, XOg_NOT)
1946
1947  static void asm_bswap(ASMState *as, IRIns *ir)
1948  {
1949    Reg dest = ra_dest(as, ir, RSET_GPR);
1950    as->mcp = emit_op(XO_BSWAP + ((dest&7) << 24),
1951                      REX_64IR(ir, 0), dest, 0, as->mcp, 1);
1952    ra_left(as, dest, ir->op1);
1953  }
1954
1955  #define asm_band(as, ir)        asm_intarith(as, ir, XOg_AND)
1956  #define asm_bor(as, ir)                asm_intarith(as, ir, XOg_OR)
1957  #define asm_bxor(as, ir)        asm_intarith(as, ir, XOg_XOR)
1958
1959  static void asm_bitshift(ASMState *as, IRIns *ir, x86Shift xs)
1960  {
1961    IRRef rref = ir->op2;
1962    IRIns *irr = IR(rref);
1963    Reg dest;
1964    if (irref_isk(rref)) {  /* Constant shifts. */
1965      int shift;
1966      dest = ra_dest(as, ir, RSET_GPR);
1967      shift = irr->i & (irt_is64(ir->t) ? 63 : 31);
1968      switch (shift) {
1969      case 0: break;
1970      case 1: emit_rr(as, XO_SHIFT1, REX_64IR(ir, xs), dest); break;
1971      default: emit_shifti(as, REX_64IR(ir, xs), dest, shift); break;
1972      }
1973    } else {  /* Variable shifts implicitly use register cl (i.e. ecx). */
1974      Reg right;
1975      dest = ra_dest(as, ir, rset_exclude(RSET_GPR, RID_ECX));
1976      if (dest == RID_ECX) {
1977        dest = ra_scratch(as, rset_exclude(RSET_GPR, RID_ECX));
1978        emit_rr(as, XO_MOV, RID_ECX, dest);
1979      }
1980      right = irr->r;
1981      if (ra_noreg(right))
1982        right = ra_allocref(as, rref, RID2RSET(RID_ECX));
1983      else if (right != RID_ECX)
1984        ra_scratch(as, RID2RSET(RID_ECX));
1985      emit_rr(as, XO_SHIFTcl, REX_64IR(ir, xs), dest);
1986      ra_noweak(as, right);
1987      if (right != RID_ECX)
1988        emit_rr(as, XO_MOV, RID_ECX, right);
1989    }
1990    ra_left(as, dest, ir->op1);
1991    /*
1992    ** Note: avoid using the flags resulting from a shift or rotate!
1993    ** All of them cause a partial flag stall, except for r,1 shifts
1994    ** (but not rotates). And a shift count of 0 leaves the flags unmodified.
1995    */
1996  }
1997
1998  #define asm_bshl(as, ir)        asm_bitshift(as, ir, XOg_SHL)
1999  #define asm_bshr(as, ir)        asm_bitshift(as, ir, XOg_SHR)
2000  #define asm_bsar(as, ir)        asm_bitshift(as, ir, XOg_SAR)
2001  #define asm_brol(as, ir)        asm_bitshift(as, ir, XOg_ROL)
2002  #define asm_bror(as, ir)        asm_bitshift(as, ir, XOg_ROR)
2003
2004  /* -- Comparisons ---------------------------------------------------------- */
2005
2006  /* Virtual flags for unordered FP comparisons. */
2007  #define VCC_U        0x1000                /* Unordered. */
2008  #define VCC_P        0x2000                /* Needs extra CC_P branch. */
2009  #define VCC_S        0x4000                /* Swap avoids CC_P branch. */
2010  #define VCC_PS        (VCC_P|VCC_S)
2011
2012  /* Map of comparisons to flags. ORDER IR. */
2013  #define COMPFLAGS(ci, cin, cu, cf)        ((ci)+((cu)<<4)+((cin)<<8)+(cf))
2014  static const uint16_t asm_compmap[IR_ABC+1] = {
2015    /*                 signed non-eq unsigned flags */
2016    /* LT  */ COMPFLAGS(CC_GE, CC_G,  CC_AE, VCC_PS),
2017    /* GE  */ COMPFLAGS(CC_L,  CC_L,  CC_B,  0),
2018    /* LE  */ COMPFLAGS(CC_G,  CC_G,  CC_A,  VCC_PS),
2019    /* GT  */ COMPFLAGS(CC_LE, CC_L,  CC_BE, 0),
```

```
  /* ULT */ COMPFLAGS(CC_AE, CC_A,  CC_AE, VCC_U),
  /* UGE */ COMPFLAGS(CC_B,  CC_B,  CC_B,  VCC_U|VCC_PS),
  /* ULE */ COMPFLAGS(CC_A,  CC_A,  CC_A,  VCC_U),
  /* UGT */ COMPFLAGS(CC_BE, CC_B,  CC_BE, VCC_U|VCC_PS),
  /* EQ  */ COMPFLAGS(CC_NE, CC_NE, CC_NE, VCC_P),
  /* NE  */ COMPFLAGS(CC_E,  CC_E,  CC_E,  VCC_U|VCC_P),
  /* ABC */ COMPFLAGS(CC_BE, CC_B,  CC_BE, VCC_U|VCC_PS)  /* Same as UGT. */
};

/* FP and integer comparisons. */
static void asm_comp(ASMState *as, IRIns *ir)
{
  uint32_t cc = asm_compmap[ir->o];
  if (irt_isnum(ir->t)) {
    IRRef lref = ir->op1;
    IRRef rref = ir->op2;
    Reg left, right;
    MCLabel l_around;
    /*
    ** An extra CC_P branch is required to preserve ordered/unordered
    ** semantics for FP comparisons. This can be avoided by swapping
    ** the operands and inverting the condition (except for EQ and UNE).
    ** So always try to swap if possible.
    **
    ** Another option would be to swap operands to achieve better memory
    ** operand fusion. But it's unlikely that this outweighs the cost
    ** of the extra branches.
    */
    if (cc & VCC_S) {  /* Swap? */
      IRRef tmp = lref; lref = rref; rref = tmp;
      cc ^= (VCC_PS|(5<<4));  /* A <-> B, AE <-> BE, PS <-> none */
    }
    left = ra_alloc1(as, lref, RSET_FPR);
    right = asm_fuseload(as, rref, rset_exclude(RSET_FPR, left));
    l_around = emit_label(as);
    asm_guardcc(as, cc >> 4);
    if (cc & VCC_P) {  /* Extra CC_P branch required? */
      if (!(cc & VCC_U)) {
        asm_guardcc(as, CC_P);  /* Branch to exit for ordered comparisons. */
      } else if (l_around != as->invmcp) {
        emit_sjcc(as, CC_P, l_around);  /* Branch around for unordered. */
      } else {
        /* Patched to mcloop by asm_loop_fixup. */
        as->loopinv = 2;
        if (as->realign)
          emit_sjcc(as, CC_P, as->mcp);
        else
          emit_jcc(as, CC_P, as->mcp);
      }
    }
    emit_mrm(as, XO_UCOMISD, left, right);
  } else {
    IRRef lref = ir->op1, rref = ir->op2;
    IROp leftop = (IROp)(IR(lref)->o);
    Reg r64 = REX_64IR(ir, 0);
    int32_t imm = 0;
    lua_assert(irt_is64(ir->t) || irt_isint(ir->t) ||
               irt_isu32(ir->t) || irt_isaddr(ir->t) || irt_isu8(ir->t));
    /* Swap constants (only for ABC) and fusable loads to the right. */
    if (irref_isk(lref) || (!irref_isk(rref) && opisfusableload(leftop))) {
      if ((cc & 0xc) == 0xc) cc ^= 0x53;  /* L <-> G, LE <-> GE */
      else if ((cc & 0xa) == 0x2) cc ^= 0x55;  /* A <-> B, AE <-> BE */
      lref = ir->op2; rref = ir->op1;
    }
    if (asm_isk32(as, rref, &imm)) {
      IRIns *irl = IR(lref);
      /* Check wether we can use test ins. Not for unsigned, since CF=0. */
      int usetest = (imm == 0 && (cc & 0xa) != 0x2);
      if (usetest && irl->o == IR_BAND && irl+1 == ir && !ra_used(irl)) {
        /* Combine comp(BAND(ref, r/imm), 0) into test mrm, r/imm. */
        Reg right, left = RID_NONE;
        RegSet allow = RSET_GPR;
        if (!asm_isk32(as, irl->op2, &imm)) {
          left = ra_alloc1(as, irl->op2, allow);
          rset_clear(allow, left);
        } else {  /* Try to Fuse IRT_I8/IRT_U8 loads, too. See below. */
```

```
2096            IRIns *irll = IR(irl->op1);
2097            if (opisfusableload((IROp)irll->o) &&
2098                (irt_isi8(irll->t) || irt_isu8(irll->t))) {
2099              IRType1 origt = irll->t;  /* Temporarily flip types. */
2100              irll->t.irt = (irll->t.irt & ~IRT_TYPE) | IRT_INT;
2101              as->curins--;  /* Skip to BAND to avoid failing in noconflict(). */
2102              right = asm_fuseload(as, irl->op1, RSET_GPR);
2103              as->curins++;
2104              irll->t = origt;
2105              if (right != RID_MRM) goto test_nofuse;
2106              /* Fusion succeeded, emit test byte mrm, imm8. */
2107              asm_guardcc(as, cc);
2108              emit_i8(as, (imm & 0xff));
2109              emit_mrm(as, XO_GROUP3b, XOg_TEST, RID_MRM);
2110              return;
2111            }
2112          }
2113          as->curins--;  /* Skip to BAND to avoid failing in noconflict(). */
2114          right = asm_fuseloadm(as, irl->op1, allow, r64);
2115          as->curins++;  /* Undo the above. */
2116        test_nofuse:
2117          asm_guardcc(as, cc);
2118          if (ra_noreg(left)) {
2119            emit_i32(as, imm);
2120            emit_mrm(as, XO_GROUP3, r64 + XOg_TEST, right);
2121          } else {
2122            emit_mrm(as, XO_TEST, r64 + left, right);
2123          }
2124        } else {
2125          Reg left;
2126          if (opisfusableload((IROp)irl->o) &&
2127              ((irt_isu8(irl->t) && checku8(imm)) ||
2128               ((irt_isi8(irl->t) || irt_isi16(irl->t)) && checki8(imm)) ||
2129               (irt_isu16(irl->t) && checku16(imm) && checki8((int16_t)imm)))) {
2130            /* Only the IRT_INT case is fused by asm_fuseload.
2131            ** The IRT_I8/IRT_U8 loads and some IRT_I16/IRT_U16 loads
2132            ** are handled here.
2133            ** Note that cmp word [mem], imm16 should not be generated,
2134            ** since it has a length-changing prefix. Compares of a word
2135            ** against a sign-extended imm8 are ok, however.
2136            */
2137            IRType1 origt = irl->t;  /* Temporarily flip types. */
2138            irl->t.irt = (irl->t.irt & ~IRT_TYPE) | IRT_INT;
2139            left = asm_fuseload(as, lref, RSET_GPR);
2140            irl->t = origt;
2141            if (left == RID_MRM) {  /* Fusion succeeded? */
2142              if (irt_isu8(irl->t) || irt_isu16(irl->t))
2143                cc >>= 4;  /* Need unsigned compare. */
2144              asm_guardcc(as, cc);
2145              emit_i8(as, imm);
2146              emit_mrm(as, (irt_isi8(origt) || irt_isu8(origt)) ?
2147                           XO_ARITHib : XO_ARITHiw8, r64 + XOg_CMP, RID_MRM);
2148              return;
2149            }  /* Otherwise handle register case as usual. */
2150          } else {
2151            left = asm_fuseloadm(as, lref,
2152                                 irt_isu8(ir->t) ? RSET_GPR8 : RSET_GPR, r64);
2153          }
2154          asm_guardcc(as, cc);
2155          if (usetest && left != RID_MRM) {
2156            /* Use test r,r instead of cmp r,0. */
2157            x86Op xo = XO_TEST;
2158            if (irt_isu8(ir->t)) {
2159              lua_assert(ir->o == IR_EQ || ir->o == IR_NE);
2160              xo = XO_TESTb;
2161              if (!rset_test(RSET_RANGE(RID_EAX, RID_EBX+1), left)) {
2162                if (LJ_64) {
2163                  left |= FORCE_REX;
2164                } else {
2165                  emit_i32(as, 0xff);
2166                  emit_mrm(as, XO_GROUP3, XOg_TEST, left);
2167                  return;
2168                }
2169              }
2170            }
2171            emit_rr(as, xo, r64 + left, left);
```

```
2172          if (irl+1 == ir)  /* Referencing previous ins? */
2173            as->flagmcp = as->mcp;  /* Set flag to drop test r,r if possible. */
2174        } else {
2175          emit_gmrmi(as, XG_ARITHi(XOg_CMP), r64 + left, imm);
2176        }
2177      }
2178    } else {
2179      Reg left = ra_alloc1(as, lref, RSET_GPR);
2180      Reg right = asm_fuseloadm(as, rref, rset_exclude(RSET_GPR, left), r64);
2181      asm_guardcc(as, cc);
2182      emit_mrm(as, XO_CMP, r64 + left, right);
2183    }
2184  }
2185  }
2186
2187  #define asm_equal(as, ir)        asm_comp(as, ir)
2188
2189  #if LJ_32 && LJ_HASFFI
2190  /* 64 bit integer comparisons in 32 bit mode. */
2191  static void asm_comp_int64(ASMState *as, IRIns *ir)
2192  {
2193    uint32_t cc = asm_compmap[(ir-1)->o];
2194    RegSet allow = RSET_GPR;
2195    Reg lefthi = RID_NONE, leftlo = RID_NONE;
2196    Reg righthi = RID_NONE, rightlo = RID_NONE;
2197    MCLabel l_around;
2198    x86ModRM mrm;
2199
2200    as->curins--;  /* Skip loword ins. Avoids failing in noconflict(), too. */
2201
2202    /* Allocate/fuse hiword operands. */
2203    if (irref_isk(ir->op2)) {
2204      lefthi = asm_fuseload(as, ir->op1, allow);
2205    } else {
2206      lefthi = ra_alloc1(as, ir->op1, allow);
2207      rset_clear(allow, lefthi);
2208      righthi = asm_fuseload(as, ir->op2, allow);
2209      if (righthi == RID_MRM) {
2210        if (as->mrm.base != RID_NONE) rset_clear(allow, as->mrm.base);
2211        if (as->mrm.idx != RID_NONE) rset_clear(allow, as->mrm.idx);
2212      } else {
2213        rset_clear(allow, righthi);
2214      }
2215    }
2216    mrm = as->mrm;  /* Save state for hiword instruction. */
2217
2218    /* Allocate/fuse loword operands. */
2219    if (irref_isk((ir-1)->op2)) {
2220      leftlo = asm_fuseload(as, (ir-1)->op1, allow);
2221    } else {
2222      leftlo = ra_alloc1(as, (ir-1)->op1, allow);
2223      rset_clear(allow, leftlo);
2224      rightlo = asm_fuseload(as, (ir-1)->op2, allow);
2225    }
2226
2227    /* All register allocations must be performed _before_ this point. */
2228    l_around = emit_label(as);
2229    as->invmcp = as->flagmcp = NULL;  /* Cannot use these optimizations. */
2230
2231    /* Loword comparison and branch. */
2232    asm_guardcc(as, cc >> 4);  /* Always use unsigned compare for loword. */
2233    if (ra_noreg(rightlo)) {
2234      int32_t imm = IR((ir-1)->op2)->i;
2235      if (imm == 0 && ((cc >> 4) & 0xa) != 0x2 && leftlo != RID_MRM)
2236        emit_rr(as, XO_TEST, leftlo, leftlo);
2237      else
2238        emit_gmrmi(as, XG_ARITHi(XOg_CMP), leftlo, imm);
2239    } else {
2240      emit_mrm(as, XO_CMP, leftlo, rightlo);
2241    }
2242
2243    /* Hiword comparison and branches. */
2244    if ((cc & 15) != CC_NE)
2245      emit_sjcc(as, CC_NE, l_around);  /* Hiword unequal: skip loword compare. */
2246    if ((cc & 15) != CC_E)
2247      asm_guardcc(as, cc >> 8);  /* Hiword compare without equality check. */
```

```
  as->mrm = mrm;  /* Restore state. */
  if (ra_noreg(righthi)) {
    int32_t imm = IR(ir->op2)->i;
    if (imm == 0 && (cc & 0xa) != 0x2 && lefthi != RID_MRM)
      emit_rr(as, XO_TEST, lefthi, lefthi);
    else
      emit_gmrmi(as, XG_ARITHi(XOg_CMP), lefthi, imm);
  } else {
    emit_mrm(as, XO_CMP, lefthi, righthi);
  }
}
#endif

/* -- Support for 64 bit ops in 32 bit mode ------------------------------ */

/* Hiword op of a split 64 bit op. Previous op must be the loword op. */
static void asm_hiop(ASMState *as, IRIns *ir)
{
#if LJ_32 && LJ_HASFFI
  /* HIOP is marked as a store because it needs its own DCE logic. */
  int uselo = ra_used(ir-1), usehi = ra_used(ir);  /* Loword/hiword used? */
  if (LJ_UNLIKELY(!(as->flags & JIT_F_OPT_DCE))) uselo = usehi = 1;
  if ((ir-1)->o == IR_CONV) {  /* Conversions to/from 64 bit. */
    as->curins--;  /* Always skip the CONV. */
    if (usehi || uselo)
      asm_conv64(as, ir);
    return;
  } else if ((ir-1)->o <= IR_NE) {  /* 64 bit integer comparisons. ORDER IR. */
    asm_comp_int64(as, ir);
    return;
  } else if ((ir-1)->o == IR_XSTORE) {
    if ((ir-1)->r != RID_SINK)
      asm_fxstore(as, ir);
    return;
  }
  if (!usehi) return;  /* Skip unused hiword op for all remaining ops. */
  switch ((ir-1)->o) {
  case IR_ADD:
    as->flagmcp = NULL;
    as->curins--;
    asm_intarith(as, ir, XOg_ADC);
    asm_intarith(as, ir-1, XOg_ADD);
    break;
  case IR_SUB:
    as->flagmcp = NULL;
    as->curins--;
    asm_intarith(as, ir, XOg_SBB);
    asm_intarith(as, ir-1, XOg_SUB);
    break;
  case IR_NEG: {
    Reg dest = ra_dest(as, ir, RSET_GPR);
    emit_rr(as, XO_GROUP3, XOg_NEG, dest);
    emit_i8(as, 0);
    emit_rr(as, XO_ARITHi8, XOg_ADC, dest);
    ra_left(as, dest, ir->op1);
    as->curins--;
    asm_neg_not(as, ir-1, XOg_NEG);
    break;
    }
  case IR_CALLN:
  case IR_CALLXS:
    if (!uselo)
      ra_allocref(as, ir->op1, RID2RSET(RID_RETLO));  /* Mark lo op as used. */
    break;
  case IR_CNEWI:
    /* Nothing to do here. Handled by CNEWI itself. */
    break;
  default: lua_assert(0); break;
  }
#else
  UNUSED(as); UNUSED(ir); lua_assert(0);  /* Unused on x64 or without FFI. */
#endif
}

/* -- Profiling ---------------------------------------------------------- */

```

```
2324 static void asm_prof(ASMState *as, IRIns *ir)
2325 {
2326   UNUSED(ir);
2327   asm_guardcc(as, CC_NE);
2328   emit_i8(as, HOOK_PROFILE);
2329   emit_rma(as, XO_GROUP3b, XOg_TEST, &J2G(as->J)->hookmask);
2330 }
2331
2332 /* -- Stack handling ------------------------------------------------------ */
2333
2334 /* Check Lua stack size for overflow. Use exit handler as fallback. */
2335 static void asm_stack_check(ASMState *as, BCReg topslot,
2336                             IRIns *irp, RegSet allow, ExitNo exitno)
2337 {
2338   /* Try to get an unused temp. register, otherwise spill/restore eax. */
2339   Reg pbase = irp ? irp->r : RID_BASE;
2340   Reg r = allow ? rset_pickbot(allow) : RID_EAX;
2341   emit_jcc(as, CC_B, exitstub_addr(as->J, exitno));
2342   if (allow == RSET_EMPTY)  /* Restore temp. register. */
2343     emit_rmro(as, XO_MOV, r|REX_64, RID_ESP, 0);
2344   else
2345     ra_modified(as, r);
2346   emit_gri(as, XG_ARITHi(XOg_CMP), r, (int32_t)(8*topslot));
2347   if (ra_hasreg(pbase) && pbase != r)
2348     emit_rr(as, XO_ARITH(XOg_SUB), r, pbase);
2349   else
2350     emit_rmro(as, XO_ARITH(XOg_SUB), r, RID_NONE,
2351               ptr2addr(&J2G(as->J)->jit_base));
2352   emit_rmro(as, XO_MOV, r, r, offsetof(lua_State, maxstack));
2353   emit_getgl(as, r, cur_L);
2354   if (allow == RSET_EMPTY)  /* Spill temp. register. */
2355     emit_rmro(as, XO_MOVto, r|REX_64, RID_ESP, 0);
2356 }
2357
2358 /* Restore Lua stack from on-trace state. */
2359 static void asm_stack_restore(ASMState *as, SnapShot *snap)
2360 {
2361   SnapEntry *map = &as->T->snapmap[snap->mapofs];
2362   SnapEntry *flinks = &as->T->snapmap[snap_nextofs(as->T, snap)-1];
2363   MSize n, nent = snap->nent;
2364   /* Store the value of all modified slots to the Lua stack. */
2365   for (n = 0; n < nent; n++) {
2366     SnapEntry sn = map[n];
2367     BCReg s = snap_slot(sn);
2368     int32_t ofs = 8*((int32_t)s-1);
2369     IRRef ref = snap_ref(sn);
2370     IRIns *ir = IR(ref);
2371     if ((sn & SNAP_NORESTORE))
2372       continue;
2373     if (irt_isnum(ir->t)) {
2374       Reg src = ra_alloc1(as, ref, RSET_FPR);
2375       emit_rmro(as, XO_MOVSDto, src, RID_BASE, ofs);
2376     } else {
2377       lua_assert(irt_ispri(ir->t) || irt_isaddr(ir->t) ||
2378                  (LJ_DUALNUM && irt_isinteger(ir->t)));
2379       if (!irref_isk(ref)) {
2380         Reg src = ra_alloc1(as, ref, rset_exclude(RSET_GPR, RID_BASE));
2381         emit_movtomro(as, REX_64IR(ir, src), RID_BASE, ofs);
2382       } else if (!irt_ispri(ir->t)) {
2383         emit_movmroi(as, RID_BASE, ofs, ir->i);
2384       }
2385       if ((sn & (SNAP_CONT|SNAP_FRAME))) {
2386         if (s != 0)  /* Do not overwrite link to previous frame. */
2387           emit_movmroi(as, RID_BASE, ofs+4, (int32_t)(*flinks--));
2388       } else {
2389         if (!(LJ_64 && irt_islightud(ir->t)))
2390           emit_movmroi(as, RID_BASE, ofs+4, irt_toitype(ir->t));
2391       }
2392     }
2393     checkmclim(as);
2394   }
2395   lua_assert(map + nent == flinks);
2396 }
2397
2398 /* -- GC handling --------------------------------------------------------- */
2399
```

```
/* Check GC threshold and do one or more GC steps. */
static void asm_gc_check(ASMState *as)
{
  const CCallInfo *ci = &lj_ir_callinfo[IRCALL_lj_gc_step_jit];
  IRRef args[2];
  MCLabel l_end;
  Reg tmp;
  ra_evictset(as, RSET_SCRATCH);
  l_end = emit_label(as);
  /* Exit trace if in GCSatomic or GCSfinalize. Avoids syncing GC objects. */
  asm_guardcc(as, CC_NE);  /* Assumes asm_snap_prep() already done. */
  emit_rr(as, XO_TEST, RID_RET, RID_RET);
  args[0] = ASMREF_TMP1;  /* global_State *g */
  args[1] = ASMREF_TMP2;  /* MSize steps     */
  asm_gencall(as, ci, args);
  tmp = ra_releasetmp(as, ASMREF_TMP1);
  emit_loada(as, tmp, J2G(as->J));
  emit_loadi(as, ra_releasetmp(as, ASMREF_TMP2), as->gcsteps);
  /* Jump around GC step if GC total < GC threshold. */
  emit_sjcc(as, CC_B, l_end);
  emit_opgl(as, XO_ARITH(XOg_CMP), tmp, gc.threshold);
  emit_getgl(as, tmp, gc.total);
  as->gcsteps = 0;
  checkmclim(as);
}

/* -- Loop handling ------------------------------------------------------- */

/* Fixup the loop branch. */
static void asm_loop_fixup(ASMState *as)
{
  MCode *p = as->mctop;
  MCode *target = as->mcp;
  if (as->realign) {  /* Realigned loops use short jumps. */
    as->realign = NULL;  /* Stop another retry. */
    lua_assert(((intptr_t)target & 15) == 0);
    if (as->loopinv) {  /* Inverted loop branch? */
      p -= 5;
      p[0] = XI_JMP;
      lua_assert(target - p >= -128);
      p[-1] = (MCode)(target - p);  /* Patch sjcc. */
      if (as->loopinv == 2)
        p[-3] = (MCode)(target - p + 2);  /* Patch opt. short jp. */
    } else {
      lua_assert(target - p >= -128);
      p[-1] = (MCode)(int8_t)(target - p);  /* Patch short jmp. */
      p[-2] = XI_JMPs;
    }
  } else {
    MCode *newloop;
    p[-5] = XI_JMP;
    if (as->loopinv) {  /* Inverted loop branch? */
      /* asm_guardcc already inverted the jcc and patched the jmp. */
      p -= 5;
      newloop = target+4;
      *(int32_t *)(p-4) = (int32_t)(target - p);  /* Patch jcc. */
      if (as->loopinv == 2) {
        *(int32_t *)(p-10) = (int32_t)(target - p + 6);  /* Patch opt. jp. */
        newloop = target+8;
      }
    } else {  /* Otherwise just patch jmp. */
      *(int32_t *)(p-4) = (int32_t)(target - p);
      newloop = target+3;
    }
    /* Realign small loops and shorten the loop branch. */
    if (newloop >= p - 128) {
      as->realign = newloop;  /* Force a retry and remember alignment. */
      as->curins = as->stopins;  /* Abort asm_trace now. */
      as->T->nins = as->orignins;  /* Remove any added renames. */
    }
  }
}

/* -- Head of trace ------------------------------------------------------- */

/* Coalesce BASE register for a root trace. */
```

```
2476 static void asm_head_root_base(ASMState *as)
2477 {
2478   IRIns *ir = IR(REF_BASE);
2479   Reg r = ir->r;
2480   if (ra_hasreg(r)) {
2481     ra_free(as, r);
2482     if (rset_test(as->modset, r) || irt_ismarked(ir->t))
2483       ir->r = RID_INIT;  /* No inheritance for modified BASE register. */
2484     if (r != RID_BASE)
2485       emit_rr(as, XO_MOV, r, RID_BASE);
2486   }
2487 }
2488
2489 /* Coalesce or reload BASE register for a side trace. */
2490 static RegSet asm_head_side_base(ASMState *as, IRIns *irp, RegSet allow)
2491 {
2492   IRIns *ir = IR(REF_BASE);
2493   Reg r = ir->r;
2494   if (ra_hasreg(r)) {
2495     ra_free(as, r);
2496     if (rset_test(as->modset, r) || irt_ismarked(ir->t))
2497       ir->r = RID_INIT;  /* No inheritance for modified BASE register. */
2498     if (irp->r == r) {
2499       rset_clear(allow, r);  /* Mark same BASE register as coalesced. */
2500     } else if (ra_hasreg(irp->r) && rset_test(as->freeset, irp->r)) {
2501       rset_clear(allow, irp->r);
2502       emit_rr(as, XO_MOV, r, irp->r);  /* Move from coalesced parent reg. */
2503     } else {
2504       emit_getgl(as, r, jit_base);  /* Otherwise reload BASE. */
2505     }
2506   }
2507   return allow;
2508 }
2509
2510 /* -- Tail of trace ------------------------------------------------------ */
2511
2512 /* Fixup the tail code. */
2513 static void asm_tail_fixup(ASMState *as, TraceNo lnk)
2514 {
2515   /* Note: don't use as->mcp swap + emit_*: emit_op overwrites more bytes. */
2516   MCode *p = as->mctop;
2517   MCode *target, *q;
2518   int32_t spadj = as->T->spadjust;
2519   if (spadj == 0) {
2520     p -= ((as->flags & JIT_F_LEA_AGU) ? 7 : 6) + (LJ_64 ? 1 : 0);
2521   } else {
2522     MCode *p1;
2523     /* Patch stack adjustment. */
2524     if (checki8(spadj)) {
2525       p -= 3;
2526       p1 = p-6;
2527       *p1 = (MCode)spadj;
2528     } else {
2529       p1 = p-9;
2530       *(int32_t *)p1 = spadj;
2531     }
2532     if ((as->flags & JIT_F_LEA_AGU)) {
2533 #if LJ_64
2534       p1[-4] = 0x48;
2535 #endif
2536       p1[-3] = (MCode)XI_LEA;
2537       p1[-2] = MODRM(checki8(spadj) ? XM_OFS8 : XM_OFS32, RID_ESP, RID_ESP);
2538       p1[-1] = MODRM(XM_SCALE1, RID_ESP, RID_ESP);
2539     } else {
2540 #if LJ_64
2541       p1[-3] = 0x48;
2542 #endif
2543       p1[-2] = (MCode)(checki8(spadj) ? XI_ARITHi8 : XI_ARITHi);
2544       p1[-1] = MODRM(XM_REG, XOg_ADD, RID_ESP);
2545     }
2546   }
2547   /* Patch exit branch. */
2548   target = lnk ? traceref(as->J, lnk)->mcode : (MCode *)lj_vm_exit_interp;
2549   *(int32_t *)(p-4) = jmprel(p, target);
2550   p[-5] = XI_JMP;
2551   /* Drop unused mcode tail. Fill with NOPs to make the prefetcher happy. */
```

```
  for (q = as->mctop-1; q >= p; q--)
    *q = XI_NOP;
  as->mctop = p;
}

/* Prepare tail of code. */
static void asm_tail_prep(ASMState *as)
{
  MCode *p = as->mctop;
  /* Realign and leave room for backwards loop branch or exit branch. */
  if (as->realign) {
    int i = ((int)(intptr_t)as->realign) & 15;
    /* Fill unused mcode tail with NOPs to make the prefetcher happy. */
    while (i-- > 0)
      *--p = XI_NOP;
    as->mctop = p;
    p -= (as->loopinv ? 5 : 2);  /* Space for short/near jmp. */
  } else {
    p -= 5;  /* Space for exit branch (near jmp). */
  }
  if (as->loopref) {
    as->invmcp = as->mcp = p;
  } else {
    /* Leave room for ESP adjustment: add esp, imm or lea esp, [esp+imm] */
    as->mcp = p - (((as->flags & JIT_F_LEA_AGU) ? 7 : 6)  + (LJ_64 ? 1 : 0));
    as->invmcp = NULL;
  }
}

/* -- Trace setup ----------------------------------------------------- */

/* Ensure there are enough stack slots for call arguments. */
static Reg asm_setup_call_slots(ASMState *as, IRIns *ir, const CCallInfo *ci)
{
  IRRef args[CCI_NARGS_MAX*2];
  int nslots;
  asm_collectargs(as, ir, ci, args);
  nslots = asm_count_call_slots(as, ci, args);
  if (nslots > as->evenspill)  /* Leave room for args in stack slots. */
    as->evenspill = nslots;
#if LJ_64
  return irt_isfp(ir->t) ? REGSP_HINT(RID_FPRET) : REGSP_HINT(RID_RET);
#else
  return irt_isfp(ir->t) ? REGSP_INIT : REGSP_HINT(RID_RET);
#endif
}

/* Target-specific setup. */
static void asm_setup_target(ASMState *as)
{
  asm_exitstub_setup(as, as->T->nsnap);
}

/* -- Trace patching -------------------------------------------------- */

/* Patch exit jumps of existing machine code to a new target. */
void lj_asm_patchexit(jit_State *J, GCtrace *T, ExitNo exitno, MCode *target)
{
  MCode *p = T->mcode;
  MCode *mcarea = lj_mcode_patch(J, p, 0);
  MSize len = T->szmcode;
  MCode *px = exitstub_addr(J, exitno) - 6;
  MCode *pe = p+len-6;
  uint32_t stateaddr = u32ptr(&J2G(J)->vmstate);
  if (len > 5 && p[len-5] == XI_JMP && p+len-6 + *(int32_t *)(p+len-4) == px)
    *(int32_t *)(p+len-4) = jmprel(p+len, target);
  /* Do not patch parent exit for a stack check. Skip beyond vmstate update. */
  for (; p < pe; p++)
    if (*(uint32_t *)(p+(LJ_64 ? 3 : 2)) == stateaddr && p[0] == XI_MOVmi) {
      p += LJ_64 ? 11 : 10;
      break;
    }
  lua_assert(p < pe);
  for (; p < pe; p++) {
    if ((*(uint16_t *)p & 0xf0ff) == 0x800f && p + *(int32_t *)(p+2) == px) {
      *(int32_t *)(p+2) = jmprel(p+6, target);
```

```
2628        p += 5;
2629      }
2630    }
2631    lj_mcode_sync(T->mcode, T->mcode + T->szmcode);
2632    lj_mcode_patch(J, mcarea, 1);
2633  }
2634
```

One Level Up Top Level

src/lj_bc.c - luajit-2.0-src

Macros defined

- LUA_CORE
- lj_bc_c

Source code

```
1  /*
2  ** Bytecode instruction modes.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lj_bc_c
7  #define LUA_CORE
8
9  #include "lj_obj.h"
10  #include "lj_bc.h"
11
12  /* Bytecode offsets and bytecode instruction modes. */
13  #include "lj_bcdef.h"
14
```

src/lj_bcdef.h - luajit-2.0-src

Global variables defined

- lj_bc_mode
- lj_bc_ofs

Source code

```
/* This is a generated file. DO NOT EDIT! */

LJ_DATADEF const uint16_t lj_bc_ofs[] = {
0,
133,
266,
399,
532,
727,
 925,
 992,
 1055,
 1191,
 1327,
 1380,
 1432,
 1483,
 1534,
 1575,
 1616,
 1644,
 1676,
 1702,
 1734,
 1833,
 1904,
 1981,
 2058,
 2136,
 2205,
 2279,
 2356,
 2433,
 2511,
 2580,
 2631,
 2706,
 2781,
 2857,
 2924,
 2973,
 3053,
 3129,
 3165,
 3201,
 3233,
 3262,
 3287,
 3330,
 3366,
 3453,
 3535,
 3573,
 3607,
 3658,
 3722,
 3831,
 3924,
 3942,
 3960,
```

```
  4088,
  4212,
  4311,
  4363,
  4516,
  4745,
  4869,
  5011,
  5096,
  5142,
  5184,
  5188,
  5336,
  5404,
  5571,
  5762,
  5850,
  5854,
  5990,
  6082,
  6187,
  6354,
  6559,
  6579,
  6763,
  6952,
  6972,
  7016,
  7055,
  7075,
  7093,
  7140,
  7165,
  7185,
  7248,
  7302,
  7302,
  7427,
  7428,
   7517,
   9333,
   9400,
   9920,
   10023,
   10080,
   10194,
   9466,
   9628,
   9720,
   9772,
   9812,
   10258,
   10299,
   10921,
   10354,
   10664,
   10973,
   11147,
   11219,
   11291,
   11362,
   11402,
   11442,
   11482,
   11522,
   11562,
   11602,
   11642,
   11682,
   11722,
   11762,
   12025,
   12173,
   11322,
   11860,
   11802,
```

```
11918,
11976,
12272,
12389,
13089,
13550,
13488,
13621,
13699,
13777,
13855,
13929,
13152,
13264,
13376,
12506,
12549,
12669,
12834,
12919,
13004
};

LJ_DATADEF const uint16_t lj_bc_mode[] = {
BCDEF(BCMODE)
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
BCMODE_FF,
```

```
213 BCMODE_FF,
214 BCMODE_FF,
215 BCMODE_FF,
216 BCMODE_FF,
217 BCMODE_FF,
218 BCMODE_FF
219 };
220
```

One Level Up Top Level

src/lj_carith.h - luajit-2.0-src

Macros defined

- _LJ_CARITH_H

Source code

```
1  /*
2  ** C data arithmetic.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_CARITH_H
7  #define _LJ_CARITH_H
8
9  #include "lj_obj.h"
10
11  #if LJ_HASFFI
12
13  LJ_FUNC int lj_carith_op(lua_State *L, MMS mm);
14
15  #if LJ_32
16  LJ_FUNC uint64_t lj_carith_shl64(uint64_t x, int32_t sh);
17  LJ_FUNC uint64_t lj_carith_shr64(uint64_t x, int32_t sh);
18  LJ_FUNC uint64_t lj_carith_sar64(uint64_t x, int32_t sh);
19  LJ_FUNC uint64_t lj_carith_rol64(uint64_t x, int32_t sh);
20  LJ_FUNC uint64_t lj_carith_ror64(uint64_t x, int32_t sh);
21  #endif
22  LJ_FUNC uint64_t lj_carith_shift64(uint64_t x, int32_t sh, int op);
23  LJ_FUNC uint64_t lj_carith_check64(lua_State *L, int narg, CTypeID *id);
24
25  #if LJ_32 && LJ_HASJIT
26  LJ_FUNC int64_t lj_carith_mul64(int64_t x, int64_t k);
27  #endif
28  LJ_FUNC uint64_t lj_carith_divu64(uint64_t a, uint64_t b);
29  LJ_FUNC int64_t lj_carith_divi64(int64_t a, int64_t b);
30  LJ_FUNC uint64_t lj_carith_modu64(uint64_t a, uint64_t b);
31  LJ_FUNC int64_t lj_carith_modi64(int64_t a, int64_t b);
32  LJ_FUNC uint64_t lj_carith_powu64(uint64_t x, uint64_t k);
33  LJ_FUNC int64_t lj_carith_powi64(int64_t x, int64_t k);
34
35  #endif
36
37  #endif
```

src/lj_ccallback.h - luajit-2.0-src

Macros defined

- _LJ_CCALLBACK_H

Source code

```
/*
** FFI C callback handling.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_CCALLBACK_H
#define _LJ_CCALLBACK_H

#include "lj_obj.h"
 #include "lj_ctype.h"

 #if LJ_HASFFI

 /* Really belongs to lj_vm.h. */
 LJ_ASMF void lj_vm_ffi_callback(void);

 LJ_FUNC MSize lj_ccallback_ptr2slot(CTState *cts, void *p);
 LJ_FUNCA lua_State * LJ_FASTCALL lj_ccallback_enter(CTState *cts, void *cf);
 LJ_FUNCA void LJ_FASTCALL lj_ccallback_leave(CTState *cts, TValue *o);
 LJ_FUNC void *lj_ccallback_new(CTState *cts, CType *ct, GCfunc *fn);
 LJ_FUNC void lj_ccallback_mcode_free(CTState *cts);

 #endif

 #endif
```

src/lj_char.c - luajit-2.0-src

Global variables defined

- lj_char_bits

Macros defined

- LUA_CORE
- lj_char_c

Source code

```
/*
** Character types.
** Donated to the public domain.
**
** This is intended to replace the problematic libc single-byte NLS functions.
** These just don't make sense anymore with UTF-8 locales becoming the norm
** on POSIX systems. It never worked too well on Windows systems since hardly
** anyone bothered to call setlocale().
**
 ** This table is hardcoded for ASCII. Identifiers include the characters
 ** 128-255, too. This allows for the use of all non-ASCII chars as identifiers
 ** in the lexer. This is a broad definition, but works well in practice
 ** for both UTF-8 locales and most single-byte locales (such as ISO-8859-*).
 **
 ** If you really need proper character types for UTF-8 strings, please use
 ** an add-on library such as slnunicode: http://luaforge.net/projects/sln/
 */

#define lj_char_c
#define LUA_CORE

#include "lj_char.h"

LJ_DATADEF const uint8_t lj_char_bits[257] = {
    0,
    1,  1,  1,  1,  1,  1,  1,  1,  1,  3,  3,  3,  3,  3,  1,  1,
    1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
    2,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,
  152,152,152,152,152,152,152,152,152,152,  4,  4,  4,  4,  4,  4,
    4,176,176,176,176,176,176,160,160,160,160,160,160,160,160,160,
  160,160,160,160,160,160,160,160,160,160,160,  4,  4,  4,  4,132,
    4,208,208,208,208,208,208,192,192,192,192,192,192,192,192,192,
  192,192,192,192,192,192,192,192,192,192,192,  4,  4,  4,  4,  1,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,
  128,128,128,128,128,128,128,128,128,128,128,128,128,128,128,128
};

```

src/lj_crecord.h - luajit-2.0-src

Macros defined

- _LJ_CRECORD_H

Source code

```
/*
** Trace recorder for C data operations.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_CRECORD_H
#define _LJ_CRECORD_H

#include "lj_obj.h"
 #include "lj_jit.h"
 #include "lj_ffrecord.h"

 #if LJ_HASJIT && LJ_HASFFI
 LJ_FUNC void LJ_FASTCALL recff_cdata_index(jit_State *J, RecordFFData *rd);
 LJ_FUNC void LJ_FASTCALL recff_cdata_call(jit_State *J, RecordFFData *rd);
 LJ_FUNC void LJ_FASTCALL recff_cdata_arith(jit_State *J, RecordFFData *rd);
 LJ_FUNC void LJ_FASTCALL recff_clib_index(jit_State *J, RecordFFData *rd);
 LJ_FUNC void LJ_FASTCALL recff_ffi_new(jit_State *J, RecordFFData *rd);
 LJ_FUNC void LJ_FASTCALL recff_ffi_errno(jit_State *J, RecordFFData *rd);
 LJ_FUNC void LJ_FASTCALL recff_ffi_string(jit_State *J, RecordFFData *rd);
 LJ_FUNC void LJ_FASTCALL recff_ffi_copy(jit_State *J, RecordFFData *rd);
 LJ_FUNC void LJ_FASTCALL recff_ffi_fill(jit_State *J, RecordFFData *rd);
 LJ_FUNC void LJ_FASTCALL recff_ffi_typeof(jit_State *J, RecordFFData *rd);
 LJ_FUNC void LJ_FASTCALL recff_ffi_istype(jit_State *J, RecordFFData *rd);
 LJ_FUNC void LJ_FASTCALL recff_ffi_abi(jit_State *J, RecordFFData *rd);
 LJ_FUNC void LJ_FASTCALL recff_ffi_xof(jit_State *J, RecordFFData *rd);
 LJ_FUNC void LJ_FASTCALL recff_ffi_gc(jit_State *J, RecordFFData *rd);

 LJ_FUNC void LJ_FASTCALL recff_bit64_tobit(jit_State *J, RecordFFData *rd);
 LJ_FUNC int LJ_FASTCALL recff_bit64_unary(jit_State *J, RecordFFData *rd);
 LJ_FUNC int LJ_FASTCALL recff_bit64_nary(jit_State *J, RecordFFData *rd);
 LJ_FUNC int LJ_FASTCALL recff_bit64_shift(jit_State *J, RecordFFData *rd);
 LJ_FUNC TRef recff_bit64_tohex(jit_State *J, RecordFFData *rd, TRef hdr);

 LJ_FUNC void LJ_FASTCALL lj_crecord_tonumber(jit_State *J, RecordFFData *rd);
 #endif

 #endif
```

src/lj_errmsg.h - luajit-2.0-src

Macros defined

- ERRDEF

Source code

```
1  /*
2  ** VM error messages.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  /* This file may be included multiple times with different ERRDEF macros. */
7
8  /* Basic error handling. */
9  ERRDEF(ERRMEM,        "not enough memory")
10 ERRDEF(ERRERR,         "error in error handling")
11 ERRDEF(ERRCPP,         "C++ exception")
12
13 /* Allocations. */
14 ERRDEF(STROV,         "string length overflow")
15 ERRDEF(UDATAOV,         "userdata length overflow")
16 ERRDEF(STKOV,         "stack overflow")
17 ERRDEF(STKOVM,         "stack overflow (%s)")
18 ERRDEF(TABOV,         "table overflow")
19
20 /* Table indexing. */
21 ERRDEF(NANIDX,         "table index is NaN")
22 ERRDEF(NILIDX,         "table index is nil")
23 ERRDEF(NEXTIDX,         "invalid key to " LUA_QL("next"))
24
25 /* Metamethod resolving. */
26 ERRDEF(BADCALL,         "attempt to call a %s value")
27 ERRDEF(BADOPRT,         "attempt to %s %s " LUA_QS " (a %s value)")
28 ERRDEF(BADOPRV,         "attempt to %s a %s value")
29 ERRDEF(BADCMPT,         "attempt to compare %s with %s")
30 ERRDEF(BADCMPV,         "attempt to compare two %s values")
31 ERRDEF(GETLOOP,         "loop in gettable")
32 ERRDEF(SETLOOP,         "loop in settable")
33 ERRDEF(OPCALL,         "call")
34 ERRDEF(OPINDEX,         "index")
35 ERRDEF(OPARITH,         "perform arithmetic on")
36 ERRDEF(OPCAT,         "concatenate")
37 ERRDEF(OPLEN,         "get length of")
38
39 /* Type checks. */
40 ERRDEF(BADSELF,         "calling " LUA_QS " on bad self (%s)")
41 ERRDEF(BADARG,         "bad argument #%d to " LUA_QS " (%s)")
42 ERRDEF(BADTYPE,         "%s expected, got %s")
43 ERRDEF(BADVAL,         "invalid value")
44 ERRDEF(NOVAL,         "value expected")
45 ERRDEF(NOCORO,         "coroutine expected")
46 ERRDEF(NOTABN,         "nil or table expected")
47 ERRDEF(NOLFUNC,         "Lua function expected")
48 ERRDEF(NOFUNCL,         "function or level expected")
49 ERRDEF(NOSFT,         "string/function/table expected")
50 ERRDEF(NOPROXY,         "boolean or proxy expected")
51 ERRDEF(FORINIT,         LUA_QL("for") " initial value must be a number")
52 ERRDEF(FORLIM,         LUA_QL("for") " limit must be a number")
53 ERRDEF(FORSTEP,         LUA_QL("for") " step must be a number")
54
55 /* C API checks. */
56 ERRDEF(NOENV,         "no calling environment")
57 ERRDEF(CYIELD,         "attempt to yield across C-call boundary")
58 ERRDEF(BADLU,         "bad light userdata pointer")
59 ERRDEF(NOGCMM,         "bad action while in __gc metamethod")
60 #if LJ_TARGET_WINDOWS
61 ERRDEF(BADFPU,         "bad FPU precision (use D3DCREATE_FPU_PRESERVE with DirectX)")
62 #endif
```

```
63
64  /* Standard library function errors. */
65  ERRDEF(ASSERT,        "assertion failed!")
66  ERRDEF(PROTMT,        "cannot change a protected metatable")
67  ERRDEF(UNPACK,        "too many results to unpack")
68  ERRDEF(RDRSTR,        "reader function must return a string")
69  ERRDEF(PRTOSTR,        LUA_QL("tostring") " must return a string to " LUA_QL("print"))
70  ERRDEF(IDXRNG,        "index out of range")
71  ERRDEF(BASERNG,        "base out of range")
72  ERRDEF(LVLRNG,        "level out of range")
73  ERRDEF(INVLVL,        "invalid level")
74  ERRDEF(INVOPT,        "invalid option")
75  ERRDEF(INVOPTM,        "invalid option " LUA_QS)
76  ERRDEF(INVFMT,        "invalid format")
77  ERRDEF(SETFENV,        LUA_QL("setfenv") " cannot change environment of given object")
78  ERRDEF(CORUN,        "cannot resume running coroutine")
79  ERRDEF(CODEAD,        "cannot resume dead coroutine")
80  ERRDEF(COSUSP,        "cannot resume non-suspended coroutine")
81  ERRDEF(TABINS,        "wrong number of arguments to " LUA_QL("insert"))
82  ERRDEF(TABCAT,        "invalid value (%s) at index %d in table for " LUA_QL("concat"))
83  ERRDEF(TABSORT,        "invalid order function for sorting")
84  ERRDEF(IOCLFL,        "attempt to use a closed file")
85  ERRDEF(IOSTDCL,        "standard file is closed")
86  ERRDEF(OSUNIQF,        "unable to generate a unique filename")
87  ERRDEF(OSDATEF,        "field " LUA_QS " missing in date table")
88  ERRDEF(STRDUMP,        "unable to dump given function")
89  ERRDEF(STRSLC,        "string slice too long")
90  ERRDEF(STRPATB,        "missing " LUA_QL("[") " after " LUA_QL("%f") " in pattern")
91  ERRDEF(STRPATC,        "invalid pattern capture")
92  ERRDEF(STRPATE,        "malformed pattern (ends with " LUA_QL("%") ")")
93  ERRDEF(STRPATM,        "malformed pattern (missing " LUA_QL("]") ")")
94  ERRDEF(STRPATU,        "unbalanced pattern")
95  ERRDEF(STRPATX,        "pattern too complex")
96  ERRDEF(STRCAPI,        "invalid capture index")
97  ERRDEF(STRCAPN,        "too many captures")
98  ERRDEF(STRCAPU,        "unfinished capture")
99  ERRDEF(STRFMT,        "invalid option " LUA_QS " to " LUA_QL("format"))
100 ERRDEF(STRGSRV,        "invalid replacement value (a %s)")
101 ERRDEF(BADMODN,        "name conflict for module " LUA_QS)
102 #if LJ_HASJIT
103 ERRDEF(JITPROT,        "runtime code generation failed, restricted kernel?")
104 #if LJ_TARGET_X86ORX64
105 ERRDEF(NOJIT,        "JIT compiler disabled, CPU does not support SSE2")
106 #else
107 ERRDEF(NOJIT,        "JIT compiler disabled")
108 #endif
109 #elif defined(LJ_ARCH_NOJIT)
110 ERRDEF(NOJIT,        "no JIT compiler for this architecture (yet)")
111 #else
112 ERRDEF(NOJIT,        "JIT compiler permanently disabled by build option")
113 #endif
114 ERRDEF(JITOPT,        "unknown or malformed optimization flag " LUA_QS)
115
116 /* Lexer/parser errors. */
117 ERRDEF(XMODE,        "attempt to load chunk with wrong mode")
118 ERRDEF(XNEAR,        "%s near " LUA_QS)
119 ERRDEF(XLINES,        "chunk has too many lines")
120 ERRDEF(XLEVELS,        "chunk has too many syntax levels")
121 ERRDEF(XNUMBER,        "malformed number")
122 ERRDEF(XLSTR,        "unfinished long string")
123 ERRDEF(XLCOM,        "unfinished long comment")
124 ERRDEF(XSTR,        "unfinished string")
125 ERRDEF(XESC,        "invalid escape sequence")
126 ERRDEF(XLDELIM,        "invalid long string delimiter")
127 ERRDEF(XTOKEN,        LUA_QS " expected")
128 ERRDEF(XJUMP,        "control structure too long")
129 ERRDEF(XSLOTS,        "function or expression too complex")
130 ERRDEF(XLIMC,        "chunk has more than %d local variables")
131 ERRDEF(XLIMM,        "main function has more than %d %s")
132 ERRDEF(XLIMF,        "function at line %d has more than %d %s")
133 ERRDEF(XMATCH,        LUA_QS " expected (to close " LUA_QS " at line %d)")
134 ERRDEF(XFIXUP,        "function too long for return fixup")
135 ERRDEF(XPARAM,        "<name> or " LUA_QL("...") " expected")
136 #if !LJ_52
137 ERRDEF(XAMBIG,        "ambiguous syntax (function call x new statement)")
138 #endif
```

```
139 ERRDEF(XFUNARG,        "function arguments expected")
140 ERRDEF(XSYMBOL,        "unexpected symbol")
141 ERRDEF(XDOTS,        "cannot use " LUA_QL("...") " outside a vararg function")
142 ERRDEF(XSYNTAX,        "syntax error")
143 ERRDEF(XFOR,        LUA_QL("=") " or " LUA_QL("in") " expected")
144 ERRDEF(XBREAK,        "no loop to break")
145 ERRDEF(XLUNDEF,        "undefined label " LUA_QS)
146 ERRDEF(XLDUP,        "duplicate label " LUA_QS)
147 ERRDEF(XGSCOPE,        "<goto %s> jumps into the scope of local " LUA_QS)
148
149 /* Bytecode reader errors. */
150 ERRDEF(BCFMT,        "cannot load incompatible bytecode")
151 ERRDEF(BCBAD,        "cannot load malformed bytecode")
152
153 #if LJ_HASFFI
154 /* FFI errors. */
155 ERRDEF(FFI_INVTYPE,        "invalid C type")
156 ERRDEF(FFI_INVSIZE,        "size of C type is unknown or too large")
157 ERRDEF(FFI_BADSCL,        "bad storage class")
158 ERRDEF(FFI_DECLSPEC,        "declaration specifier expected")
159 ERRDEF(FFI_BADTAG,        "undeclared or implicit tag " LUA_QS)
160 ERRDEF(FFI_REDEF,        "attempt to redefine " LUA_QS)
161 ERRDEF(FFI_NUMPARAM,        "wrong number of type parameters")
162 ERRDEF(FFI_INITOV,        "too many initializers for " LUA_QS)
163 ERRDEF(FFI_BADCONV,        "cannot convert " LUA_QS " to " LUA_QS)
164 ERRDEF(FFI_BADLEN,        "attempt to get length of " LUA_QS)
165 ERRDEF(FFI_BADCONCAT,        "attempt to concatenate " LUA_QS " and " LUA_QS)
166 ERRDEF(FFI_BADARITH,        "attempt to perform arithmetic on " LUA_QS " and " LUA_QS)
167 ERRDEF(FFI_BADCOMP,        "attempt to compare " LUA_QS " with " LUA_QS)
168 ERRDEF(FFI_BADCALL,        LUA_QS " is not callable")
169 ERRDEF(FFI_NUMARG,        "wrong number of arguments for function call")
170 ERRDEF(FFI_BADMEMBER,        LUA_QS " has no member named " LUA_QS)
171 ERRDEF(FFI_BADIDX,        LUA_QS " cannot be indexed")
172 ERRDEF(FFI_BADIDXW,        LUA_QS " cannot be indexed with " LUA_QS)
173 ERRDEF(FFI_BADMM,        LUA_QS " has no " LUA_QS " metamethod")
174 ERRDEF(FFI_WRCONST,        "attempt to write to constant location")
175 ERRDEF(FFI_NODECL,        "missing declaration for symbol " LUA_QS)
176 ERRDEF(FFI_BADCBACK,        "bad callback")
177 #if LJ_OS_NOJIT
178 ERRDEF(FFI_CBACKOV,        "no support for callbacks on this OS")
179 #else
180 ERRDEF(FFI_CBACKOV,        "too many callbacks")
181 #endif
182 ERRDEF(FFI_NYIPACKBIT,        "NYI: packed bit fields")
183 ERRDEF(FFI_NYICALL,        "NYI: cannot call this C function (yet)")
184 #endif
185
186 #undef ERRDEF
187
188 /* Detecting unused error messages:
189    awk -F, '/^ERRDEF/ { gsub(/ERRDEF./, ""); printf "grep -q LJ_ERR_%s *.[ch] || echo %s\n", $1, $1}'
lj_errmsg.h | sh
190 */
```

One Level Up Top Level

src/lj_func.h - luajit-2.0-src

Macros defined

- _LJ_FUNC_H

Source code

```
1  /*
2  ** Function handling (prototypes, functions and upvalues).
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_FUNC_H
7  #define _LJ_FUNC_H
8
9  #include "lj_obj.h"
10
11  /* Prototypes. */
12  LJ_FUNC void LJ_FASTCALL lj_func_freeproto(global_State *g, GCproto *pt);
13
14  /* Upvalues. */
15  LJ_FUNCA void LJ_FASTCALL lj_func_closeuv(lua_State *L, TValue *level);
16  LJ_FUNC void LJ_FASTCALL lj_func_freeuv(global_State *g, GCupval *uv);
17
18  /* Functions (closures). */
19  LJ_FUNC GCfunc *lj_func_newC(lua_State *L, MSize nelems, GCtab *env);
20  LJ_FUNC GCfunc *lj_func_newL_empty(lua_State *L, GCproto *pt, GCtab *env);
21  LJ_FUNCA GCfunc *lj_func_newL_gc(lua_State *L, GCproto *pt, GCfuncL *parent);
22  LJ_FUNC void LJ_FASTCALL lj_func_free(global_State *g, GCfunc *c);
23
24  #endif
```

src/lj_gdbjit.c - luajit-2.0-src

Global variables defined

- __jit_debug_descriptor
- elfhdr_template

Data types defined

- ELFheader
- ELFheader
- ELFsectheader
- ELFsectheader
- ELFsymbol
- ELFsymbol
- GDBJITctx
- GDBJITctx
- GDBJITdesc
- GDBJITdesc
- GDBJITentry
- GDBJITentry
- GDBJITentryobj
- GDBJITentryobj
- GDBJITinitf
- GDBJITobj
- GDBJITobj

Functions defined

- __jit_debug_register_code
- gdbjit_buildobj
- gdbjit_catnum
- gdbjit_debugabbrev
- gdbjit_debuginfo
- gdbjit_debugline
- gdbjit_ehframe
- gdbjit_initsect

- gdbjit_newentry
- gdbjit_secthdr
- gdbjit_sleb128
- gdbjit_strz
- gdbjit_symtab
- lj_gdbjit_addtrace
- lj_gdbjit_deltrace

Macros defined

- DADDR
- DADDR
- DALIGNNOP
- DALIGNNOP
- DB
- DB
- DI8
- DI8
- DLNE
- DLNE
- DSECT
- DSECT
- DSTR
- DSTR
- DSV
- DSV
- DU16
- DU16
- DU32
- DU32
- DUV
- DUV
- DW_CIE_VERSION
- ELFSECT_FLAGS_ALLOC
- ELFSECT_FLAGS_EXEC

- ELFSECT_FLAGS_WRITE
- ELFSECT_IDX_ABS
- LUA_CORE
- SECTALIGN
- SECTALIGN
- SECTDEF
- SECTDEF
- lj_gdbjit_c

Source code

```
/*
** Client for the GDB JIT API.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#define lj_gdbjit_c
#define LUA_CORE

#include "lj_obj.h"

#if LJ_HASJIT

#include "lj_gc.h"
#include "lj_err.h"
#include "lj_debug.h"
#include "lj_frame.h"
#include "lj_buf.h"
#include "lj_strfmt.h"
#include "lj_jit.h"
#include "lj_dispatch.h"

/* This is not compiled in by default.
** Enable with -DLUAJIT_USE_GDBJIT in the Makefile and recompile everything.
*/
#ifdef LUAJIT_USE_GDBJIT

/* The GDB JIT API allows JIT compilers to pass debug information about
** JIT-compiled code back to GDB. You need at least GDB 7.0 or higher
** to see it in action.
**
** This is a passive API, so it works even when not running under GDB
** or when attaching to an already running process. Alas, this implies
** enabling it always has a non-negligible overhead -- do not use in
** release mode!
**
** The LuaJIT GDB JIT client is rather minimal at the moment. It gives
** each trace a symbol name and adds a source location and frame unwind
** information. Obviously LuaJIT itself and any embedding C application
** should be compiled with debug symbols, too (see the Makefile).
**
** Traces are named TRACE_1, TRACE_2, ... these correspond to the trace
** numbers from -jv or -jdump. Use "break TRACE_1" or "tbreak TRACE_1" etc.
** to set breakpoints on specific traces (even ahead of their creation).
**
** The source location for each trace allows listing the corresponding
** source lines with the GDB command "list" (but only if the Lua source
** has been loaded from a file). Currently this is always set to the
** location where the trace has been started.
**
** Frame unwind information can be inspected with the GDB command
** "info frame". This also allows proper backtraces across JIT-compiled
** code with the GDB command "bt".
**
** You probably want to add the following settings to a .gdbinit file
```

```
** (or add them to ~/.gdbinit):
**   set disassembly-flavor intel
**   set breakpoint pending on
**
** Here's a sample GDB session:
** ------------------------------------------------------------------------

$ cat >x.lua
for outer=1,100 do
  for inner=1,100 do end
end
^D

$ luajit -jv x.lua
[TRACE   1 x.lua:2]
[TRACE   2 (1/3) x.lua:1 -> 1]

$ gdb --quiet --args luajit x.lua
(gdb) tbreak TRACE_1
Function "TRACE_1" not defined.
Temporary breakpoint 1 (TRACE_1) pending.
(gdb) run
Starting program: luajit x.lua

Temporary breakpoint 1, TRACE_1 () at x.lua:2
2           for inner=1,100 do end
(gdb) list
1         for outer=1,100 do
2           for inner=1,100 do end
3         end
(gdb) bt
#0  TRACE_1 () at x.lua:2
#1  0x08053690 in lua_pcall [...]
[...]
#7  0x0806ff90 in main [...]
(gdb) disass TRACE_1
Dump of assembler code for function TRACE_1:
0xf7fd9fba <TRACE_1+0>:         mov    DWORD PTR ds:0xf7e0e2a0,0x1
0xf7fd9fc4 <TRACE_1+10>:         movsd  xmm7,QWORD PTR [edx+0x20]
[...]
0xf7fd9ff8 <TRACE_1+62>:         jmp    0xf7fd2014
End of assembler dump.
(gdb) tbreak TRACE_2
Function "TRACE_2" not defined.
Temporary breakpoint 2 (TRACE_2) pending.
(gdb) cont
Continuing.

Temporary breakpoint 2, TRACE_2 () at x.lua:1
1         for outer=1,100 do
(gdb) info frame
Stack level 0, frame at 0xffffd7c0:
eip = 0xf7fd9f60 in TRACE_2 (x.lua:1); saved eip 0x8053690
called by frame at 0xffffd7e0
source language unknown.
Arglist at 0xffffd78c, args:
Locals at 0xffffd78c, Previous frame's sp is 0xffffd7c0
Saved registers:
  ebx at 0xffffd7ac, ebp at 0xffffd7b8, esi at 0xffffd7b0, edi at 0xffffd7b4,
  eip at 0xffffd7bc
(gdb)

** ------------------------------------------------------------------------
*/

/* -- GDB JIT API ---------------------------------------------------------- */

/* GDB JIT actions. */
enum {
  GDBJIT_NOACTION = 0,
  GDBJIT_REGISTER,
  GDBJIT_UNREGISTER
};

/* GDB JIT entry. */
typedef struct GDBJITentry {
```

```
  struct GDBJITentry *next_entry;
  struct GDBJITentry *prev_entry;
  const char *symfile_addr;
  uint64_t symfile_size;
} GDBJITentry;

/* GDB JIT descriptor. */
typedef struct GDBJITdesc {
  uint32_t version;
  uint32_t action_flag;
  GDBJITentry *relevant_entry;
  GDBJITentry *first_entry;
} GDBJITdesc;

GDBJITdesc __jit_debug_descriptor = {
  1, GDBJIT_NOACTION, NULL, NULL
};

/* GDB sets a breakpoint at this function. */
void LJ_NOINLINE __jit_debug_register_code()
{
  __asm__ __volatile__("");
};

/* -- In-memory ELF object definitions ----------------------------------- */

/* ELF definitions. */
typedef struct ELFheader {
  uint8_t emagic[4];
  uint8_t eclass;
  uint8_t eendian;
  uint8_t eversion;
  uint8_t eosabi;
  uint8_t eabiversion;
  uint8_t epad[7];
  uint16_t type;
  uint16_t machine;
  uint32_t version;
  uintptr_t entry;
  uintptr_t phofs;
  uintptr_t shofs;
  uint32_t flags;
  uint16_t ehsize;
  uint16_t phentsize;
  uint16_t phnum;
  uint16_t shentsize;
  uint16_t shnum;
  uint16_t shstridx;
} ELFheader;

typedef struct ELFsectheader {
  uint32_t name;
  uint32_t type;
  uintptr_t flags;
  uintptr_t addr;
  uintptr_t ofs;
  uintptr_t size;
  uint32_t link;
  uint32_t info;
  uintptr_t align;
  uintptr_t entsize;
} ELFsectheader;

#define ELFSECT_IDX_ABS                0xfff1

enum {
  ELFSECT_TYPE_PROGBITS = 1,
  ELFSECT_TYPE_SYMTAB = 2,
  ELFSECT_TYPE_STRTAB = 3,
  ELFSECT_TYPE_NOBITS = 8
};

#define ELFSECT_FLAGS_WRITE         1
#define ELFSECT_FLAGS_ALLOC         2
#define ELFSECT_FLAGS_EXEC          4

```

```
typedef struct ELFsymbol {
#if LJ_64
  uint32_t name;
  uint8_t info;
  uint8_t other;
  uint16_t sectidx;
  uintptr_t value;
  uint64_t size;
#else
  uint32_t name;
  uintptr_t value;
  uint32_t size;
  uint8_t info;
  uint8_t other;
  uint16_t sectidx;
#endif
} ELFsymbol;

enum {
  ELFSYM_TYPE_FUNC = 2,
  ELFSYM_TYPE_FILE = 4,
  ELFSYM_BIND_LOCAL = 0 << 4,
  ELFSYM_BIND_GLOBAL = 1 << 4,
};

/* DWARF definitions. */
#define DW_CIE_VERSION        1

enum {
  DW_CFA_nop = 0x0,
  DW_CFA_offset_extended = 0x5,
  DW_CFA_def_cfa = 0xc,
  DW_CFA_def_cfa_offset = 0xe,
  DW_CFA_offset_extended_sf = 0x11,
  DW_CFA_advance_loc = 0x40,
  DW_CFA_offset = 0x80
};

enum {
  DW_EH_PE_udata4 = 3,
  DW_EH_PE_textrel = 0x20
};

enum {
  DW_TAG_compile_unit = 0x11
};

enum {
  DW_children_no = 0,
  DW_children_yes = 1
};

enum {
  DW_AT_name = 0x03,
  DW_AT_stmt_list = 0x10,
  DW_AT_low_pc = 0x11,
  DW_AT_high_pc = 0x12
};

enum {
  DW_FORM_addr = 0x01,
  DW_FORM_data4 = 0x06,
  DW_FORM_string = 0x08
};

enum {
  DW_LNS_extended_op = 0,
  DW_LNS_copy = 1,
  DW_LNS_advance_pc = 2,
  DW_LNS_advance_line = 3
};

enum {
  DW_LNE_end_sequence = 1,
  DW_LNE_set_address = 2
};
```

```
283
284 enum {
285 #if LJ_TARGET_X86
286   DW_REG_AX, DW_REG_CX, DW_REG_DX, DW_REG_BX,
287   DW_REG_SP, DW_REG_BP, DW_REG_SI, DW_REG_DI,
288   DW_REG_RA,
289 #elif LJ_TARGET_X64
290   /* Yes, the order is strange, but correct. */
291   DW_REG_AX, DW_REG_DX, DW_REG_CX, DW_REG_BX,
292   DW_REG_SI, DW_REG_DI, DW_REG_BP, DW_REG_SP,
293   DW_REG_8, DW_REG_9, DW_REG_10, DW_REG_11,
294   DW_REG_12, DW_REG_13, DW_REG_14, DW_REG_15,
295   DW_REG_RA,
296 #elif LJ_TARGET_ARM
297   DW_REG_SP = 13,
298   DW_REG_RA = 14,
299 #elif LJ_TARGET_PPC
300   DW_REG_SP = 1,
301   DW_REG_RA = 65,
302   DW_REG_CR = 70,
303 #elif LJ_TARGET_MIPS
304   DW_REG_SP = 29,
305   DW_REG_RA = 31,
306 #else
307 #error "Unsupported target architecture"
308 #endif
309 };
310
311 /* Minimal list of sections for the in-memory ELF object. */
312 enum {
313   GDBJIT_SECT_NULL,
314   GDBJIT_SECT_text,
315   GDBJIT_SECT_eh_frame,
316   GDBJIT_SECT_shstrtab,
317   GDBJIT_SECT_strtab,
318   GDBJIT_SECT_symtab,
319   GDBJIT_SECT_debug_info,
320   GDBJIT_SECT_debug_abbrev,
321   GDBJIT_SECT_debug_line,
322   GDBJIT_SECT__MAX
323 };
324
325 enum {
326   GDBJIT_SYM_UNDEF,
327   GDBJIT_SYM_FILE,
328   GDBJIT_SYM_FUNC,
329   GDBJIT_SYM__MAX
330 };
331
332 /* In-memory ELF object. */
333 typedef struct GDBJITobj {
334   ELFheader hdr;                        /* ELF header. */
335   ELFsectheader sect[GDBJIT_SECT__MAX];        /* ELF sections. */
336   ELFsymbol sym[GDBJIT_SYM__MAX];        /* ELF symbol table. */
337   uint8_t space[4096];                        /* Space for various section data. */
338 } GDBJITobj;
339
340 /* Combined structure for GDB JIT entry and ELF object. */
341 typedef struct GDBJITentryobj {
342   GDBJITentry entry;
343   size_t sz;
344   GDBJITobj obj;
345 } GDBJITentryobj;
346
347 /* Template for in-memory ELF header. */
348 static const ELFheader elfhdr_template = {
349   .emagic = { 0x7f, 'E', 'L', 'F' },
350   .eclass = LJ_64 ? 2 : 1,
351   .eendian = LJ_ENDIAN_SELECT(1, 2),
352   .eversion = 1,
353 #if LJ_TARGET_LINUX
354   .eosabi = 0,  /* Nope, it's not 3. */
355 #elif defined(__FreeBSD__)
356   .eosabi = 9,
357 #elif defined(__NetBSD__)
358   .eosabi = 2,
```

```
#elif defined(__OpenBSD__)
  .eosabi = 12,
#elif defined(__DragonFly__)
  .eosabi = 0,
#elif (defined(__sun__) && defined(__svr4__))
  .eosabi = 6,
#else
  .eosabi = 0,
#endif
  .eabiversion = 0,
  .epad = { 0, 0, 0, 0, 0, 0, 0 },
  .type = 1,
#if LJ_TARGET_X86
  .machine = 3,
#elif LJ_TARGET_X64
  .machine = 62,
#elif LJ_TARGET_ARM
  .machine = 40,
#elif LJ_TARGET_PPC
  .machine = 20,
#elif LJ_TARGET_MIPS
  .machine = 8,
#else
#error "Unsupported target architecture"
#endif
  .version = 1,
  .entry = 0,
  .phofs = 0,
  .shofs = offsetof(GDBJITobj, sect),
  .flags = 0,
  .ehsize = sizeof(ELFheader),
  .phentsize = 0,
  .phnum = 0,
  .shentsize = sizeof(ELFsectheader),
  .shnum = GDBJIT_SECT__MAX,
  .shstridx = GDBJIT_SECT_shstrtab
};

/* -- In-memory ELF object generation ------------------------------------ */

/* Context for generating the ELF object for the GDB JIT API. */
typedef struct GDBJITctx {
  uint8_t *p;               /* Pointer to next address in obj.space. */
  uint8_t *startp;        /* Pointer to start address in obj.space. */
  GCtrace *T;               /* Generate symbols for this trace. */
  uintptr_t mcaddr;        /* Machine code address. */
  MSize szmcode;        /* Size of machine code. */
  MSize spadjp;               /* Stack adjustment for parent trace or interpreter. */
  MSize spadj;               /* Stack adjustment for trace itself. */
  BCLine lineno;        /* Starting line number. */
  const char *filename;        /* Starting file name. */
  size_t objsize;        /* Final size of ELF object. */
  GDBJITobj obj;        /* In-memory ELF object. */
} GDBJITctx;

/* Add a zero-terminated string. */
static uint32_t gdbjit_strz(GDBJITctx *ctx, const char *str)
{
  uint8_t *p = ctx->p;
  uint32_t ofs = (uint32_t)(p - ctx->startp);
  do {
    *p++ = (uint8_t)*str;
  } while (*str++);
  ctx->p = p;
  return ofs;
}

/* Append a decimal number. */
static void gdbjit_catnum(GDBJITctx *ctx, uint32_t n)
{
  if (n >= 10) { uint32_t m = n / 10; n = n % 10; gdbjit_catnum(ctx, m); }
  *ctx->p++ = '0' + n;
}

/* Add a SLEB128 value. */
static void gdbjit_sleb128(GDBJITctx *ctx, int32_t v)
```

```
{
  uint8_t *p = ctx->p;
  for (; (uint32_t)(v+0x40) >= 0x80; v >>= 7)
    *p++ = (uint8_t)((v & 0x7f) | 0x80);
  *p++ = (uint8_t)(v & 0x7f);
  ctx->p = p;
}

/* Shortcuts to generate DWARF structures. */
#define DB(x)                (*p++ = (x))
#define DI8(x)               (*(int8_t *)p = (x), p++)
#define DU16(x)              (*(uint16_t *)p = (x), p += 2)
#define DU32(x)              (*(uint32_t *)p = (x), p += 4)
#define DADDR(x)        (*(uintptr_t *)p = (x), p += sizeof(uintptr_t))
#define DUV(x)               (p = (uint8_t *)lj_strfmt_wuleb128((char *)p, (x)))
#define DSV(x)               (ctx->p = p, gdbjit_sleb128(ctx, (x)), p = ctx->p)
#define DSTR(str)        (ctx->p = p, gdbjit_strz(ctx, (str)), p = ctx->p)
#define DALIGNNOP(s)        while ((uintptr_t)p & ((s)-1)) *p++ = DW_CFA_nop
#define DSECT(name, stmt) \
  { uint32_t *szp_##name = (uint32_t *)p; p += 4; stmt \
    *szp_##name = (uint32_t)((p-(uint8_t *)szp_##name)-4); } \

/* Initialize ELF section headers. */
static void LJ_FASTCALL gdbjit_secthdr(GDBJITctx *ctx)
{
  ELFsectheader *sect;

  *ctx->p++ = '\0';  /* Empty string at start of string table. */

#define SECTDEF(id, tp, al) \
  sect = &ctx->obj.sect[GDBJIT_SECT_##id]; \
  sect->name = gdbjit_strz(ctx, "." #id); \
  sect->type = ELFSECT_TYPE_##tp; \
  sect->align = (al)

  SECTDEF(text, NOBITS, 16);
  sect->flags = ELFSECT_FLAGS_ALLOC|ELFSECT_FLAGS_EXEC;
  sect->addr = ctx->mcaddr;
  sect->ofs = 0;
  sect->size = ctx->szmcode;

  SECTDEF(eh_frame, PROGBITS, sizeof(uintptr_t));
  sect->flags = ELFSECT_FLAGS_ALLOC;

  SECTDEF(shstrtab, STRTAB, 1);
  SECTDEF(strtab, STRTAB, 1);

  SECTDEF(symtab, SYMTAB, sizeof(uintptr_t));
  sect->ofs = offsetof(GDBJITobj, sym);
  sect->size = sizeof(ctx->obj.sym);
  sect->link = GDBJIT_SECT_strtab;
  sect->entsize = sizeof(ELFsymbol);
  sect->info = GDBJIT_SYM_FUNC;

  SECTDEF(debug_info, PROGBITS, 1);
  SECTDEF(debug_abbrev, PROGBITS, 1);
  SECTDEF(debug_line, PROGBITS, 1);

#undef SECTDEF
}

/* Initialize symbol table. */
static void LJ_FASTCALL gdbjit_symtab(GDBJITctx *ctx)
{
  ELFsymbol *sym;

  *ctx->p++ = '\0';  /* Empty string at start of string table. */

  sym = &ctx->obj.sym[GDBJIT_SYM_FILE];
  sym->name = gdbjit_strz(ctx, "JIT mcode");
  sym->sectidx = ELFSECT_IDX_ABS;
  sym->info = ELFSYM_TYPE_FILE|ELFSYM_BIND_LOCAL;

  sym = &ctx->obj.sym[GDBJIT_SYM_FUNC];
  sym->name = gdbjit_strz(ctx, "TRACE_"); ctx->p--;
  gdbjit_catnum(ctx, ctx->T->traceno); *ctx->p++ = '\0';
```

```
  sym->sectidx = GDBJIT_SECT_text;
  sym->value = 0;
  sym->size = ctx->szmcode;
  sym->info = ELFSYM_TYPE_FUNC|ELFSYM_BIND_GLOBAL;
}

/* Initialize .eh_frame section. */
static void LJ_FASTCALL gdbjit_ehframe(GDBJITctx *ctx)
{
  uint8_t *p = ctx->p;
  uint8_t *framep = p;

  /* Emit DWARF EH CIE. */
  DSECT(CIE,
    DU32(0);                          /* Offset to CIE itself. */
    DB(DW_CIE_VERSION);
    DSTR("zR");                          /* Augmentation. */
    DUV(1);                           /* Code alignment factor. */
    DSV(-(int32_t)sizeof(uintptr_t));  /* Data alignment factor. */
    DB(DW_REG_RA);                   /* Return address register. */
    DB(1); DB(DW_EH_PE_textrel|DW_EH_PE_udata4);  /* Augmentation data. */
    DB(DW_CFA_def_cfa); DUV(DW_REG_SP); DUV(sizeof(uintptr_t));
#if LJ_TARGET_PPC
    DB(DW_CFA_offset_extended_sf); DB(DW_REG_RA); DSV(-1);
#else
    DB(DW_CFA_offset|DW_REG_RA); DUV(1);
#endif
    DALIGNNOP(sizeof(uintptr_t));
  )

  /* Emit DWARF EH FDE. */
  DSECT(FDE,
    DU32((uint32_t)(p-framep));        /* Offset to CIE. */
    DU32(0);                          /* Machine code offset relative to .text. */
    DU32(ctx->szmcode);                  /* Machine code length. */
    DB(0);                           /* Augmentation data. */
    /* Registers saved in CFRAME. */
#if LJ_TARGET_X86
    DB(DW_CFA_offset|DW_REG_BP); DUV(2);
    DB(DW_CFA_offset|DW_REG_DI); DUV(3);
    DB(DW_CFA_offset|DW_REG_SI); DUV(4);
    DB(DW_CFA_offset|DW_REG_BX); DUV(5);
#elif LJ_TARGET_X64
    DB(DW_CFA_offset|DW_REG_BP); DUV(2);
    DB(DW_CFA_offset|DW_REG_BX); DUV(3);
    DB(DW_CFA_offset|DW_REG_15); DUV(4);
    DB(DW_CFA_offset|DW_REG_14); DUV(5);
    /* Extra registers saved for JIT-compiled code. */
    DB(DW_CFA_offset|DW_REG_13); DUV(9);
    DB(DW_CFA_offset|DW_REG_12); DUV(10);
#elif LJ_TARGET_ARM
    {
      int i;
      for (i = 11; i >= 4; i--) { DB(DW_CFA_offset|i); DUV(2+(11-i)); }
    }
#elif LJ_TARGET_PPC
    {
      int i;
      DB(DW_CFA_offset_extended); DB(DW_REG_CR); DUV(55);
      for (i = 14; i <= 31; i++) {
        DB(DW_CFA_offset|i); DUV(37+(31-i));
        DB(DW_CFA_offset|32|i); DUV(2+2*(31-i));
      }
    }
#elif LJ_TARGET_MIPS
    {
      int i;
      DB(DW_CFA_offset|30); DUV(2);
      for (i = 23; i >= 16; i--) { DB(DW_CFA_offset|i); DUV(26-i); }
      for (i = 30; i >= 20; i -= 2) { DB(DW_CFA_offset|32|i); DUV(42-i); }
    }
#else
#error "Unsupported target architecture"
#endif
    if (ctx->spadjp != ctx->spadj) {  /* Parent/interpreter stack frame size. */
      DB(DW_CFA_def_cfa_offset); DUV(ctx->spadjp);
```

```
      DB(DW_CFA_advance_loc|1);  /* Only an approximation. */
    }
    DB(DW_CFA_def_cfa_offset); DUV(ctx->spadj);  /* Trace stack frame size. */
    DALIGNNOP(sizeof(uintptr_t));
  )

  ctx->p = p;
}

/* Initialize .debug_info section. */
static void LJ_FASTCALL gdbjit_debuginfo(GDBJITctx *ctx)
{
  uint8_t *p = ctx->p;

  DSECT(info,
    DU16(2);                        /* DWARF version. */
    DU32(0);                        /* Abbrev offset. */
    DB(sizeof(uintptr_t));        /* Pointer size. */

    DUV(1);                         /* Abbrev #1: DW_TAG_compile_unit. */
    DSTR(ctx->filename);        /* DW_AT_name. */
    DADDR(ctx->mcaddr);               /* DW_AT_low_pc. */
    DADDR(ctx->mcaddr + ctx->szmcode);  /* DW_AT_high_pc. */
    DU32(0);                        /* DW_AT_stmt_list. */
  )

  ctx->p = p;
}

/* Initialize .debug_abbrev section. */
static void LJ_FASTCALL gdbjit_debugabbrev(GDBJITctx *ctx)
{
  uint8_t *p = ctx->p;

  /* Abbrev #1: DW_TAG_compile_unit. */
  DUV(1); DUV(DW_TAG_compile_unit);
  DB(DW_children_no);
  DUV(DW_AT_name);        DUV(DW_FORM_string);
  DUV(DW_AT_low_pc);        DUV(DW_FORM_addr);
  DUV(DW_AT_high_pc);        DUV(DW_FORM_addr);
  DUV(DW_AT_stmt_list);        DUV(DW_FORM_data4);
  DB(0); DB(0);

  ctx->p = p;
}

#define DLNE(op, s)        (DB(DW_LNS_extended_op), DUV(1+(s)), DB((op)))

/* Initialize .debug_line section. */
static void LJ_FASTCALL gdbjit_debugline(GDBJITctx *ctx)
{
  uint8_t *p = ctx->p;

  DSECT(line,
    DU16(2);                        /* DWARF version. */
    DSECT(header,
      DB(1);                        /* Minimum instruction length. */
      DB(1);                        /* is_stmt. */
      DI8(0);                        /* Line base for special opcodes. */
      DB(2);                        /* Line range for special opcodes. */
      DB(3+1);                        /* Opcode base at DW_LNS_advance_line+1. */
      DB(0); DB(1); DB(1);        /* Standard opcode lengths. */
      /* Directory table. */
      DB(0);
      /* File name table. */
      DSTR(ctx->filename); DUV(0); DUV(0); DUV(0);
      DB(0);
    )

    DLNE(DW_LNE_set_address, sizeof(uintptr_t)); DADDR(ctx->mcaddr);
    if (ctx->lineno) {
      DB(DW_LNS_advance_line); DSV(ctx->lineno-1);
    }
    DB(DW_LNS_copy);
    DB(DW_LNS_advance_pc); DUV(ctx->szmcode);
    DLNE(DW_LNE_end_sequence, 0);
```

```
  )

  ctx->p = p;
}

#undef DLNE

/* Undef shortcuts. */
#undef DB
#undef DI8
#undef DU16
#undef DU32
#undef DADDR
#undef DUV
#undef DSV
#undef DSTR
#undef DALIGNNOP
#undef DSECT

/* Type of a section initializer callback. */
typedef void (LJ_FASTCALL *GDBJITinitf)(GDBJITctx *ctx);

/* Call section initializer and set the section offset and size. */
static void gdbjit_initsect(GDBJITctx *ctx, int sect, GDBJITinitf initf)
{
  ctx->startp = ctx->p;
  ctx->obj.sect[sect].ofs = (uintptr_t)((char *)ctx->p - (char *)&ctx->obj);
  initf(ctx);
  ctx->obj.sect[sect].size = (uintptr_t)(ctx->p - ctx->startp);
}

#define SECTALIGN(p, a) \
  ((p) = (uint8_t *)(((uintptr_t)(p) + ((a)-1)) & ~(uintptr_t)((a)-1)))

/* Build in-memory ELF object. */
static void gdbjit_buildobj(GDBJITctx *ctx)
{
  GDBJITobj *obj = &ctx->obj;
  /* Fill in ELF header and clear structures. */
  memcpy(&obj->hdr, &elfhdr_template, sizeof(ELFheader));
  memset(&obj->sect, 0, sizeof(ELFsectheader)*GDBJIT_SECT__MAX);
  memset(&obj->sym, 0, sizeof(ELFsymbol)*GDBJIT_SYM__MAX);
  /* Initialize sections. */
  ctx->p = obj->space;
  gdbjit_initsect(ctx, GDBJIT_SECT_shstrtab, gdbjit_secthdr);
  gdbjit_initsect(ctx, GDBJIT_SECT_strtab, gdbjit_symtab);
  gdbjit_initsect(ctx, GDBJIT_SECT_debug_info, gdbjit_debuginfo);
  gdbjit_initsect(ctx, GDBJIT_SECT_debug_abbrev, gdbjit_debugabbrev);
  gdbjit_initsect(ctx, GDBJIT_SECT_debug_line, gdbjit_debugline);
  SECTALIGN(ctx->p, sizeof(uintptr_t));
  gdbjit_initsect(ctx, GDBJIT_SECT_eh_frame, gdbjit_ehframe);
  ctx->objsize = (size_t)((char *)ctx->p - (char *)obj);
  lua_assert(ctx->objsize < sizeof(GDBJITobj));
}

#undef SECTALIGN

/* -- Interface to GDB JIT API ------------------------------------------- */

/* Add new entry to GDB JIT symbol chain. */
static void gdbjit_newentry(lua_State *L, GDBJITctx *ctx)
{
  /* Allocate memory for GDB JIT entry and ELF object. */
  MSize sz = (MSize)(sizeof(GDBJITentryobj) - sizeof(GDBJITobj) + ctx->objsize);
  GDBJITentryobj *eo = lj_mem_newt(L, sz, GDBJITentryobj);
  memcpy(&eo->obj, &ctx->obj, ctx->objsize);  /* Copy ELF object. */
  eo->sz = sz;
  ctx->T->gdbjit_entry = (void *)eo;
  /* Link new entry to chain and register it. */
  eo->entry.prev_entry = NULL;
  eo->entry.next_entry = __jit_debug_descriptor.first_entry;
  if (eo->entry.next_entry)
    eo->entry.next_entry->prev_entry = &eo->entry;
  eo->entry.symfile_addr = (const char *)&eo->obj;
  eo->entry.symfile_size = ctx->objsize;
  __jit_debug_descriptor.first_entry = &eo->entry;
```

```
  __jit_debug_descriptor.relevant_entry = &eo->entry;
  __jit_debug_descriptor.action_flag = GDBJIT_REGISTER;
  __jit_debug_register_code();
}

/* Add debug info for newly compiled trace and notify GDB. */
void lj_gdbjit_addtrace(jit_State *J, GCtrace *T)
{
  GDBJITctx ctx;
  GCproto *pt = &gcref(T->startpt)->pt;
  TraceNo parent = T->ir[REF_BASE].op1;
  const BCIns *startpc = mref(T->startpc, const BCIns);
  ctx.T = T;
  ctx.mcaddr = (uintptr_t)T->mcode;
  ctx.szmcode = T->szmcode;
  ctx.spadjp = CFRAME_SIZE_JIT +
               (MSize)(parent ? traceref(J, parent)->spadjust : 0);
  ctx.spadj = CFRAME_SIZE_JIT + T->spadjust;
  lua_assert(startpc >= proto_bc(pt) && startpc < proto_bc(pt) + pt->sizebc);
  ctx.lineno = lj_debug_line(pt, proto_bcpos(pt, startpc));
  ctx.filename = proto_chunknamestr(pt);
  if (*ctx.filename == '@' || *ctx.filename == '=')
    ctx.filename++;
  else
    ctx.filename = "(string)";
  gdbjit_buildobj(&ctx);
  gdbjit_newentry(J->L, &ctx);
}

/* Delete debug info for trace and notify GDB. */
void lj_gdbjit_deltrace(jit_State *J, GCtrace *T)
{
  GDBJITentryobj *eo = (GDBJITentryobj *)T->gdbjit_entry;
  if (eo) {
    if (eo->entry.prev_entry)
      eo->entry.prev_entry->next_entry = eo->entry.next_entry;
    else
      __jit_debug_descriptor.first_entry = eo->entry.next_entry;
    if (eo->entry.next_entry)
      eo->entry.next_entry->prev_entry = eo->entry.prev_entry;
    __jit_debug_descriptor.relevant_entry = &eo->entry;
    __jit_debug_descriptor.action_flag = GDBJIT_UNREGISTER;
    __jit_debug_register_code();
    lj_mem_free(J2G(J), eo, eo->sz);
  }
}

#endif
#endif
```

One Level Up Top Level

src/lj_opt_split.c - luajit-2.0-src

Functions defined

- cpsplit
- lj_opt_split
- split_bitop
- split_bitshift
- split_call_l
- split_call_li
- split_call_ll
- split_emit
- split_ir
- split_needsplit
- split_num2int
- split_ptr
- split_subst_snap

Macros defined

- IR
- IR
- LUA_CORE
- lj_opt_split_c

Source code

```
1  /*
2  ** SPLIT: Split 64 bit IR instructions into 32 bit IR instructions.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #define lj_opt_split_c
7  #define LUA_CORE
8
9  #include "lj_obj.h"
10
11 #if LJ_HASJIT && (LJ_SOFTFP || (LJ_32 && LJ_HASFFI))
12
13 #include "lj_err.h"
14 #include "lj_buf.h"
15 #include "lj_ir.h"
16 #include "lj_jit.h"
17 #include "lj_ircall.h"
18 #include "lj_iropt.h"
19 #include "lj_vm.h"
20
21 /* SPLIT pass:
22 **
23 ** This pass splits up 64 bit IR instructions into multiple 32 bit IR
```

```
** instructions. It's only active for soft-float targets or for 32 bit CPUs
** which lack native 64 bit integer operations (the FFI is currently the
** only emitter for 64 bit integer instructions).
**
** Splitting the IR in a separate pass keeps each 32 bit IR assembler
** backend simple. Only a small amount of extra functionality needs to be
** implemented. This is much easier than adding support for allocating
** register pairs to each backend (believe me, I tried). A few simple, but
** important optimizations can be performed by the SPLIT pass, which would
** be tedious to do in the backend.
**
** The basic idea is to replace each 64 bit IR instruction with its 32 bit
** equivalent plus an extra HIOP instruction. The splitted IR is not passed
** through FOLD or any other optimizations, so each HIOP is guaranteed to
** immediately follow it's counterpart. The actual functionality of HIOP is
** inferred from the previous instruction.
**
** The operands of HIOP hold the hiword input references. The output of HIOP
** is the hiword output reference, which is also used to hold the hiword
** register or spill slot information. The register allocator treats this
** instruction independently of any other instruction, which improves code
** quality compared to using fixed register pairs.
**
** It's easier to split up some instructions into two regular 32 bit
** instructions. E.g. XLOAD is split up into two XLOADs with two different
** addresses. Obviously 64 bit constants need to be split up into two 32 bit
** constants, too. Some hiword instructions can be entirely omitted, e.g.
** when zero-extending a 32 bit value to 64 bits. 64 bit arguments for calls
** are split up into two 32 bit arguments each.
**
** On soft-float targets, floating-point instructions are directly converted
** to soft-float calls by the SPLIT pass (except for comparisons and MIN/MAX).
** HIOP for number results has the type IRT_SOFTFP ("sfp" in -jdump).
**
** Here's the IR and x64 machine code for 'x.b = x.a + 1' for a struct with
** two int64_t fields:
**
** 0100    p32 ADD    base  +8
** 0101    i64 XLOAD  0100
** 0102    i64 ADD    0101  +1
** 0103    p32 ADD    base  +16
** 0104    i64 XSTORE 0103  0102
**
**         mov rax, [esi+0x8]
**         add rax, +0x01
**         mov [esi+0x10], rax
**
** Here's the transformed IR and the x86 machine code after the SPLIT pass:
**
** 0100    p32 ADD    base  +8
** 0101    int XLOAD  0100
** 0102    p32 ADD    base  +12
** 0103    int XLOAD  0102
** 0104    int ADD    0101  +1
** 0105    int HIOP   0103  +0
** 0106    p32 ADD    base  +16
** 0107    int XSTORE 0106  0104
** 0108    int HIOP   0106  0105
**
**         mov eax, [esi+0x8]
**         mov ecx, [esi+0xc]
**         add eax, +0x01
**         adc ecx, +0x00
**         mov [esi+0x10], eax
**         mov [esi+0x14], ecx
**
** You may notice the reassociated hiword address computation, which is
** later fused into the mov operands by the assembler.
*/

/* Some local macros to save typing. Undef'd at the end. */
#define IR(ref)                (&J->cur.ir[(ref)])

/* Directly emit the transformed IR without updating chains etc. */
static IRRef split_emit(jit_State *J, uint16_t ot, IRRef1 op1, IRRef1 op2)
{
```

```
  IRRef nref = lj_ir_nextins(J);
  IRIns *ir = IR(nref);
  ir->ot = ot;
  ir->op1 = op1;
  ir->op2 = op2;
  return nref;
}

#if LJ_SOFTFP
/* Emit a (checked) number to integer conversion. */
static IRRef split_num2int(jit_State *J, IRRef lo, IRRef hi, int check)
{
  IRRef tmp, res;
#if LJ_LE
  tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), lo, hi);
#else
  tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), hi, lo);
#endif
  res = split_emit(J, IRTI(IR_CALLN), tmp, IRCALL_softfp_d2i);
  if (check) {
    tmp = split_emit(J, IRTI(IR_CALLN), res, IRCALL_softfp_i2d);
    split_emit(J, IRT(IR_HIOP, IRT_SOFTFP), tmp, tmp);
    split_emit(J, IRTGI(IR_EQ), tmp, lo);
    split_emit(J, IRTG(IR_HIOP, IRT_SOFTFP), tmp+1, hi);
  }
  return res;
}

/* Emit a CALLN with one split 64 bit argument. */
static IRRef split_call_l(jit_State *J, IRRef1 *hisubst, IRIns *oir,
                          IRIns *ir, IRCallID id)
{
  IRRef tmp, op1 = ir->op1;
  J->cur.nins--;
#if LJ_LE
  tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), oir[op1].prev, hisubst[op1]);
#else
  tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), hisubst[op1], oir[op1].prev);
#endif
  ir->prev = tmp = split_emit(J, IRTI(IR_CALLN), tmp, id);
  return split_emit(J, IRT(IR_HIOP, IRT_SOFTFP), tmp, tmp);
}
#endif

/* Emit a CALLN with one split 64 bit argument and a 32 bit argument. */
static IRRef split_call_li(jit_State *J, IRRef1 *hisubst, IRIns *oir,
                           IRIns *ir, IRCallID id)
{
  IRRef tmp, op1 = ir->op1, op2 = ir->op2;
  J->cur.nins--;
#if LJ_LE
  tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), oir[op1].prev, hisubst[op1]);
#else
  tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), hisubst[op1], oir[op1].prev);
#endif
  tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), tmp, oir[op2].prev);
  ir->prev = tmp = split_emit(J, IRTI(IR_CALLN), tmp, id);
  return split_emit(J, IRT(IR_HIOP, IRT_SOFTFP), tmp, tmp);
}

/* Emit a CALLN with two split 64 bit arguments. */
static IRRef split_call_ll(jit_State *J, IRRef1 *hisubst, IRIns *oir,
                           IRIns *ir, IRCallID id)
{
  IRRef tmp, op1 = ir->op1, op2 = ir->op2;
  J->cur.nins--;
#if LJ_LE
  tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), oir[op1].prev, hisubst[op1]);
  tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), tmp, oir[op2].prev);
  tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), tmp, hisubst[op2]);
#else
  tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), hisubst[op1], oir[op1].prev);
  tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), tmp, hisubst[op2]);
  tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), tmp, oir[op2].prev);
#endif
  ir->prev = tmp = split_emit(J, IRTI(IR_CALLN), tmp, id);
```

```
  return split_emit(J,
    IRT(IR_HIOP, (LJ_SOFTFP && irt_isnum(ir->t)) ? IRT_SOFTFP : IRT_INT),
    tmp, tmp);
}

/* Get a pointer to the other 32 bit word (LE: hiword, BE: loword). */
static IRRef split_ptr(jit_State *J, IRIns *oir, IRRef ref)
{
  IRRef nref = oir[ref].prev;
  IRIns *ir = IR(nref);
  int32_t ofs = 4;
  if (ir->o == IR_KPTR)
    return lj_ir_kptr(J, (char *)ir_kptr(ir) + ofs);
  if (ir->o == IR_ADD && irref_isk(ir->op2) && !irt_isphi(oir[ref].t)) {
    /* Reassociate address. */
    ofs += IR(ir->op2)->i;
    nref = ir->op1;
    if (ofs == 0) return nref;
  }
  return split_emit(J, IRTI(IR_ADD), nref, lj_ir_kint(J, ofs));
}

#if LJ_HASFFI
static IRRef split_bitshift(jit_State *J, IRRef1 *hisubst,
                            IRIns *oir, IRIns *nir, IRIns *ir)
{
  IROp op = ir->o;
  IRRef kref = nir->op2;
  if (irref_isk(kref)) {  /* Optimize constant shifts. */
    int32_t k = (IR(kref)->i & 63);
    IRRef lo = nir->op1, hi = hisubst[ir->op1];
    if (op == IR_BROL || op == IR_BROR) {
      if (op == IR_BROR) k = (-k & 63);
      if (k >= 32) { IRRef t = lo; lo = hi; hi = t; k -= 32; }
      if (k == 0) {
      passthrough:
        J->cur.nins--;
        ir->prev = lo;
        return hi;
      } else {
        IRef k1, k2;
        IRRef t1, t2, t3, t4;
        J->cur.nins--;
        k1 = lj_ir_kint(J, k);
        k2 = lj_ir_kint(J, (-k & 31));
        t1 = split_emit(J, IRTI(IR_BSHL), lo, k1);
        t2 = split_emit(J, IRTI(IR_BSHL), hi, k1);
        t3 = split_emit(J, IRTI(IR_BSHR), lo, k2);
        t4 = split_emit(J, IRTI(IR_BSHR), hi, k2);
        ir->prev = split_emit(J, IRTI(IR_BOR), t1, t4);
        return split_emit(J, IRTI(IR_BOR), t2, t3);
      }
    } else if (k == 0) {
      goto passthrough;
    } else if (k < 32) {
      if (op == IR_BSHL) {
        IRRef t1 = split_emit(J, IRTI(IR_BSHL), hi, kref);
        IRRef t2 = split_emit(J, IRTI(IR_BSHR), lo, lj_ir_kint(J, (-k&31)));
        return split_emit(J, IRTI(IR_BOR), t1, t2);
      } else {
        IRRef t1 = ir->prev, t2;
        lua_assert(op == IR_BSHR || op == IR_BSAR);
        nir->o = IR_BSHR;
        t2 = split_emit(J, IRTI(IR_BSHL), hi, lj_ir_kint(J, (-k&31)));
        ir->prev = split_emit(J, IRTI(IR_BOR), t1, t2);
        return split_emit(J, IRTI(op), hi, kref);
      }
    } else {
      if (op == IR_BSHL) {
        if (k == 32)
          J->cur.nins--;
        else
          lo = ir->prev;
        ir->prev = lj_ir_kint(J, 0);
        return lo;
      } else {
```

```
        lua_assert(op == IR_BSHR || op == IR_BSAR);
        if (k == 32) {
          J->cur.nins--;
          ir->prev = hi;
        } else {
          nir->op1 = hi;
        }
        if (op == IR_BSHR)
          return lj_ir_kint(J, 0);
        else
          return split_emit(J, IRTI(IR_BSAR), hi, lj_ir_kint(J, 31));
      }
    }
  }
  return split_call_li(J, hisubst, oir, ir,
                       op - IR_BSHL + IRCALL_lj_carith_shl64);
}

static IRRef split_bitop(jit_State *J, IRRef1 *hisubst,
                         IRIns *nir, IRIns *ir)
{
  IROp op = ir->o;
  IRRef hi, kref = nir->op2;
  if (irref_isk(kref)) {  /* Optimize bit operations with lo constant. */
    int32_t k = IR(kref)->i;
    if (k == 0 || k == -1) {
      if (op == IR_BAND) k = ~k;
      if (k == 0) {
        J->cur.nins--;
        ir->prev = nir->op1;
      } else if (op == IR_BXOR) {
        nir->o = IR_BNOT;
        nir->op2 = 0;
      } else {
        J->cur.nins--;
        ir->prev = kref;
      }
    }
  }
  hi = hisubst[ir->op1];
  kref = hisubst[ir->op2];
  if (irref_isk(kref)) {  /* Optimize bit operations with hi constant. */
    int32_t k = IR(kref)->i;
    if (k == 0 || k == -1) {
      if (op == IR_BAND) k = ~k;
      if (k == 0) {
        return hi;
      } else if (op == IR_BXOR) {
        return split_emit(J, IRTI(IR_BNOT), hi, 0);
      } else {
        return kref;
      }
    }
  }
  return split_emit(J, IRTI(op), hi, kref);
}
#endif

/* Substitute references of a snapshot. */
static void split_subst_snap(jit_State *J, SnapShot *snap, IRIns *oir)
{
  SnapEntry *map = &J->cur.snapmap[snap->mapofs];
  MSize n, nent = snap->nent;
  for (n = 0; n < nent; n++) {
    SnapEntry sn = map[n];
    IRIns *ir = &oir[snap_ref(sn)];
    if (!(LJ_SOFTFP && (sn & SNAP_SOFTFPNUM) && irref_isk(snap_ref(sn))))
      map[n] = ((sn & 0xffff0000) | ir->prev);
  }
}

/* Transform the old IR to the new IR. */
static void split_ir(jit_State *J)
{
  IRRef nins = J->cur.nins, nk = J->cur.nk;
  MSize irlen = nins - nk;
```

```
  MSize need = (irlen+1)*(sizeof(IRIns) + sizeof(IRRef1));
  IRIns *oir = (IRIns *)lj_buf_tmp(J->L, need);
  IRRef1 *hisubst;
  IRRef ref, snref;
  SnapShot *snap;

  /* Copy old IR to buffer. */
  memcpy(oir, IR(nk), irlen*sizeof(IRIns));
  /* Bias hiword substitution table and old IR. Loword kept in field prev. */
  hisubst = (IRRef1 *)&oir[irlen] - nk;
  oir -= nk;

  /* Remove all IR instructions, but retain IR constants. */
  J->cur.nins = REF_FIRST;
  J->loopref = 0;

  /* Process constants and fixed references. */
  for (ref = nk; ref <= REF_BASE; ref++) {
    IRIns *ir = &oir[ref];
    if ((LJ_SOFTFP && ir->o == IR_KNUM) || ir->o == IR_KINT64) {
      /* Split up 64 bit constant. */
      TValue tv = *ir_k64(ir);
      ir->prev = lj_ir_kint(J, (int32_t)tv.u32.lo);
      hisubst[ref] = lj_ir_kint(J, (int32_t)tv.u32.hi);
    } else {
      ir->prev = ref;  /* Identity substitution for loword. */
      hisubst[ref] = 0;
    }
  }

  /* Process old IR instructions. */
  snap = J->cur.snap;
  snref = snap->ref;
  for (ref = REF_FIRST; ref < nins; ref++) {
    IRIns *ir = &oir[ref];
    IRRef nref = lj_ir_nextins(J);
    IRIns *nir = IR(nref);
    IRRef hi = 0;

    if (ref >= snref) {
      snap->ref = nref;
      split_subst_snap(J, snap++, oir);
      snref = snap < &J->cur.snap[J->cur.nsnap] ? snap->ref : ~(IRRef)0;
    }

    /* Copy-substitute old instruction to new instruction. */
    nir->op1 = ir->op1 < nk ? ir->op1 : oir[ir->op1].prev;
    nir->op2 = ir->op2 < nk ? ir->op2 : oir[ir->op2].prev;
    ir->prev = nref;  /* Loword substitution. */
    nir->o = ir->o;
    nir->t.irt = ir->t.irt & ~(IRT_MARK|IRT_ISPHI);
    hisubst[ref] = 0;

    /* Split 64 bit instructions. */
#if LJ_SOFTFP
    if (irt_isnum(ir->t)) {
      nir->t.irt = IRT_INT | (nir->t.irt & IRT_GUARD);  /* Turn into INT op. */
      /* Note: hi ref = lo ref + 1! Required for SNAP_SOFTFPNUM logic. */
      switch (ir->o) {
      case IR_ADD:
        hi = split_call_ll(J, hisubst, oir, ir, IRCALL_softfp_add);
        break;
      case IR_SUB:
        hi = split_call_ll(J, hisubst, oir, ir, IRCALL_softfp_sub);
        break;
      case IR_MUL:
        hi = split_call_ll(J, hisubst, oir, ir, IRCALL_softfp_mul);
        break;
      case IR_DIV:
        hi = split_call_ll(J, hisubst, oir, ir, IRCALL_softfp_div);
        break;
      case IR_POW:
        hi = split_call_li(J, hisubst, oir, ir, IRCALL_lj_vm_powi);
        break;
      case IR_FPMATH:
        /* Try to rejoin pow from EXP2, MUL and LOG2. */
```

```
404         if (nir->op2 == IRFPM_EXP2 && nir->op1 > J->loopref) {
405           IRIns *irp = IR(nir->op1);
406           if (irp->o == IR_CALLN && irp->op2 == IRCALL_softfp_mul) {
407             IRIns *irm4 = IR(irp->op1);
408             IRIns *irm3 = IR(irm4->op1);
409             IRIns *irm12 = IR(irm3->op1);
410             IRIns *irl1 = IR(irm12->op1);
411             if (irm12->op1 > J->loopref && irl1->o == IR_CALLN &&
412                 irl1->op2 == IRCALL_lj_vm_log2) {
413               IRRef tmp = irl1->op1;  /* Recycle first two args from LOG2. */
414               IRRef arg3 = irm3->op2, arg4 = irm4->op2;
415               J->cur.nins--;
416               tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), tmp, arg3);
417               tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), tmp, arg4);
418               ir->prev = tmp = split_emit(J, IRTI(IR_CALLN), tmp, IRCALL_pow);
419               hi = split_emit(J, IRT(IR_HIOP, IRT_SOFTFP), tmp, tmp);
420               break;
421             }
422           }
423         }
424         hi = split_call_l(J, hisubst, oir, ir, IRCALL_lj_vm_floor + ir->op2);
425         break;
426       case IR_ATAN2:
427         hi = split_call_ll(J, hisubst, oir, ir, IRCALL_atan2);
428         break;
429       case IR_LDEXP:
430         hi = split_call_li(J, hisubst, oir, ir, IRCALL_ldexp);
431         break;
432       case IR_NEG: case IR_ABS:
433         nir->o = IR_CONV;  /* Pass through loword. */
434         nir->op2 = (IRT_INT << 5) | IRT_INT;
435         hi = split_emit(J, IRT(ir->o == IR_NEG ? IR_BXOR : IR_BAND, IRT_SOFTFP),
436                         hisubst[ir->op1], hisubst[ir->op2]);
437         break;
438       case IR_SLOAD:
439         if ((nir->op2 & IRSLOAD_CONVERT)) {  /* Convert from int to number. */
440           nir->op2 &= ~IRSLOAD_CONVERT;
441           ir->prev = nref = split_emit(J, IRTI(IR_CALLN), nref,
442                                        IRCALL_softfp_i2d);
443           hi = split_emit(J, IRT(IR_HIOP, IRT_SOFTFP), nref, nref);
444           break;
445         }
446         /* fallthrough */
447       case IR_ALOAD: case IR_HLOAD: case IR_ULOAD: case IR_VLOAD:
448       case IR_STRTO:
449         hi = split_emit(J, IRT(IR_HIOP, IRT_SOFTFP), nref, nref);
450         break;
451       case IR_XLOAD: {
452         IRIns inslo = *nir;  /* Save/undo the emit of the lo XLOAD. */
453         J->cur.nins--;
454         hi = split_ptr(J, oir, ir->op1);  /* Insert the hiref ADD. */
455         nref = lj_ir_nextins(J);
456         nir = IR(nref);
457         *nir = inslo;  /* Re-emit lo XLOAD immediately before hi XLOAD. */
458         hi = split_emit(J, IRT(IR_XLOAD, IRT_SOFTFP), hi, ir->op2);
459 #if LJ_LE
460         ir->prev = nref;
461 #else
462         ir->prev = hi; hi = nref;
463 #endif
464         break;
465         }
466       case IR_ASTORE: case IR_HSTORE: case IR_USTORE: case IR_XSTORE:
467         split_emit(J, IRT(IR_HIOP, IRT_SOFTFP), nir->op1, hisubst[ir->op2]);
468         break;
469       case IR_CONV: {  /* Conversion to number. Others handled below. */
470         IRType st = (IRType)(ir->op2 & IRCONV_SRCMASK);
471         UNUSED(st);
472 #if LJ_32 && LJ_HASFFI
473         if (st == IRT_I64 || st == IRT_U64) {
474           hi = split_call_l(J, hisubst, oir, ir,
475                 st == IRT_I64 ? IRCALL_fp64_l2d : IRCALL_fp64_ul2d);
476           break;
477         }
478 #endif
479         lua_assert(st == IRT_INT ||
```

```
                  (LJ_32 && LJ_HASFFI && (st == IRT_U32 || st == IRT_FLOAT)));
        nir->o = IR_CALLN;
#if LJ_32 && LJ_HASFFI
        nir->op2 = st == IRT_INT ? IRCALL_softfp_i2d :
                   st == IRT_FLOAT ? IRCALL_softfp_f2d :
                   IRCALL_softfp_ui2d;
#else
        nir->op2 = IRCALL_softfp_i2d;
#endif
        hi = split_emit(J, IRT(IR_HIOP, IRT_SOFTFP), nref, nref);
        break;
        }
      case IR_CALLN:
      case IR_CALLL:
      case IR_CALLS:
      case IR_CALLXS:
        goto split_call;
      case IR_PHI:
        if (nir->op1 == nir->op2)
          J->cur.nins--;  /* Drop useless PHIs. */
        if (hisubst[ir->op1] != hisubst[ir->op2])
          split_emit(J, IRT(IR_PHI, IRT_SOFTFP),
                     hisubst[ir->op1], hisubst[ir->op2]);
        break;
      case IR_HIOP:
        J->cur.nins--;  /* Drop joining HIOP. */
        ir->prev = nir->op1;
        hi = nir->op2;
        break;
      default:
        lua_assert(ir->o <= IR_NE || ir->o == IR_MIN || ir->o == IR_MAX);
        hi = split_emit(J, IRTG(IR_HIOP, IRT_SOFTFP),
                        hisubst[ir->op1], hisubst[ir->op2]);
        break;
      }
    } else
#endif
#if LJ_32 && LJ_HASFFI
    if (irt_isint64(ir->t)) {
      IRRef hiref = hisubst[ir->op1];
      nir->t.irt = IRT_INT | (nir->t.irt & IRT_GUARD);  /* Turn into INT op. */
      switch (ir->o) {
      case IR_ADD:
      case IR_SUB:
        /* Use plain op for hiword if loword cannot produce a carry/borrow. */
        if (irref_isk(nir->op2) && IR(nir->op2)->i == 0) {
          ir->prev = nir->op1;  /* Pass through loword. */
          nir->op1 = hiref; nir->op2 = hisubst[ir->op2];
          hi = nref;
          break;
        }
        /* fallthrough */
      case IR_NEG:
        hi = split_emit(J, IRTI(IR_HIOP), hiref, hisubst[ir->op2]);
        break;
      case IR_MUL:
        hi = split_call_ll(J, hisubst, oir, ir, IRCALL_lj_carith_mul64);
        break;
      case IR_DIV:
        hi = split_call_ll(J, hisubst, oir, ir,
                           irt_isi64(ir->t) ? IRCALL_lj_carith_divi64 :
                                              IRCALL_lj_carith_divu64);
        break;
      case IR_MOD:
        hi = split_call_ll(J, hisubst, oir, ir,
                           irt_isi64(ir->t) ? IRCALL_lj_carith_modi64 :
                                              IRCALL_lj_carith_modu64);
        break;
      case IR_POW:
        hi = split_call_ll(J, hisubst, oir, ir,
                           irt_isi64(ir->t) ? IRCALL_lj_carith_powi64 :
                                              IRCALL_lj_carith_powu64);
        break;
      case IR_BNOT:
        hi = split_emit(J, IRTI(IR_BNOT), hiref, 0);
        break;
```

```
      case IR_BSWAP:
        ir->prev = split_emit(J, IRTI(IR_BSWAP), hiref, 0);
        hi = nref;
        break;
      case IR_BAND: case IR_BOR: case IR_BXOR:
        hi = split_bitop(J, hisubst, nir, ir);
        break;
      case IR_BSHL: case IR_BSHR: case IR_BSAR: case IR_BROL: case IR_BROR:
        hi = split_bitshift(J, hisubst, oir, nir, ir);
        break;
      case IR_FLOAD:
        lua_assert(ir->op2 == IRFL_CDATA_INT64);
        hi = split_emit(J, IRTI(IR_FLOAD), nir->op1, IRFL_CDATA_INT64_4);
#if LJ_BE
        ir->prev = hi; hi = nref;
#endif
        break;
      case IR_XLOAD:
        hi = split_emit(J, IRTI(IR_XLOAD), split_ptr(J, oir, ir->op1), ir->op2);
#if LJ_BE
        ir->prev = hi; hi = nref;
#endif
        break;
      case IR_XSTORE:
        split_emit(J, IRTI(IR_HIOP), nir->op1, hisubst[ir->op2]);
        break;
      case IR_CONV: {  /* Conversion to 64 bit integer. Others handled below. */
        IRType st = (IRType)(ir->op2 & IRCONV_SRCMASK);
#if LJ_SOFTFP
        if (st == IRT_NUM) {  /* NUM to 64 bit int conv. */
          hi = split_call_l(J, hisubst, oir, ir,
                 irt_isi64(ir->t) ? IRCALL_fp64_d2l : IRCALL_fp64_d2ul);
        } else if (st == IRT_FLOAT) {  /* FLOAT to 64 bit int conv. */
          nir->o = IR_CALLN;
          nir->op2 = irt_isi64(ir->t) ? IRCALL_fp64_f2l : IRCALL_fp64_f2ul;
          hi = split_emit(J, IRTI(IR_HIOP), nref, nref);
        }
#else
        if (st == IRT_NUM || st == IRT_FLOAT) {  /* FP to 64 bit int conv. */
          hi = split_emit(J, IRTI(IR_HIOP), nir->op1, nref);
        }
#endif
        else if (st == IRT_I64 || st == IRT_U64) {  /* 64/64 bit cast. */
          /* Drop cast, since assembler doesn't care. */
          goto fwdlo;
        } else if ((ir->op2 & IRCONV_SEXT)) {  /* Sign-extend to 64 bit. */
          IRRef k31 = lj_ir_kint(J, 31);
          nir = IR(nref);  /* May have been reallocated. */
          ir->prev = nir->op1;  /* Pass through loword. */
          nir->o = IR_BSAR;  /* hi = bsar(lo, 31). */
          nir->op2 = k31;
          hi = nref;
        } else {  /* Zero-extend to 64 bit. */
          hi = lj_ir_kint(J, 0);
          goto fwdlo;
        }
        break;
        }
      case IR_CALLXS:
        goto split_call;
      case IR_PHI: {
        IRRef hiref2;
        if ((irref_isk(nir->op1) && irref_isk(nir->op2)) ||
            nir->op1 == nir->op2)
          J->cur.nins--;  /* Drop useless PHIs. */
        hiref2 = hisubst[ir->op2];
        if (!((irref_isk(hiref) && irref_isk(hiref2)) || hiref == hiref2))
          split_emit(J, IRTI(IR_PHI), hiref, hiref2);
        break;
        }
      case IR_HIOP:
        J->cur.nins--;  /* Drop joining HIOP. */
        ir->prev = nir->op1;
        hi = nir->op2;
        break;
      default:
```

```
632         lua_assert(ir->o <= IR_NE);  /* Comparisons. */
633         split_emit(J, IRTGI(IR_HIOP), hiref, hisubst[ir->op2]);
634         break;
635       }
636     } else
637 #endif
638 #if LJ_SOFTFP
639     if (ir->o == IR_SLOAD) {
640       if ((nir->op2 & IRSLOAD_CONVERT)) {  /* Convert from number to int. */
641         nir->op2 &= ~IRSLOAD_CONVERT;
642         if (!(nir->op2 & IRSLOAD_TYPECHECK))
643           nir->t.irt = IRT_INT;  /* Drop guard. */
644         split_emit(J, IRT(IR_HIOP, IRT_SOFTFP), nref, nref);
645         ir->prev = split_num2int(J, nref, nref+1, irt_isguard(ir->t));
646       }
647     } else if (ir->o == IR_TOBIT) {
648       IRRef tmp, op1 = ir->op1;
649       J->cur.nins--;
650 #if LJ_LE
651       tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), oir[op1].prev, hisubst[op1]);
652 #else
653       tmp = split_emit(J, IRT(IR_CARG, IRT_NIL), hisubst[op1], oir[op1].prev);
654 #endif
655       ir->prev = split_emit(J, IRTI(IR_CALLN), tmp, IRCALL_lj_vm_tobit);
656     } else if (ir->o == IR_TOSTR) {
657       if (hisubst[ir->op1]) {
658         if (irref_isk(ir->op1))
659           nir->op1 = ir->op1;
660         else
661           split_emit(J, IRT(IR_HIOP, IRT_NIL), hisubst[ir->op1], nref);
662       }
663     } else if (ir->o == IR_HREF || ir->o == IR_NEWREF) {
664       if (irref_isk(ir->op2) && hisubst[ir->op2])
665         nir->op2 = ir->op2;
666     } else
667 #endif
668     if (ir->o == IR_CONV) {  /* See above, too. */
669       IRType st = (IRType)(ir->op2 & IRCONV_SRCMASK);
670 #if LJ_32 && LJ_HASFFI
671       if (st == IRT_I64 || st == IRT_U64) {  /* Conversion from 64 bit int. */
672 #if LJ_SOFTFP
673         if (irt_isfloat(ir->t)) {
674           split_call_l(J, hisubst, oir, ir,
675                        st == IRT_I64 ? IRCALL_fp64_l2f : IRCALL_fp64_ul2f);
676           J->cur.nins--;  /* Drop unused HIOP. */
677         }
678 #else
679         if (irt_isfp(ir->t)) {  /* 64 bit integer to FP conversion. */
680           ir->prev = split_emit(J, IRT(IR_HIOP, irt_type(ir->t)),
681                                 hisubst[ir->op1], nref);
682         }
683 #endif
684         else {  /* Truncate to lower 32 bits. */
685         fwdlo:
686           ir->prev = nir->op1;  /* Forward loword. */
687           /* Replace with NOP to avoid messing up the snapshot logic. */
688           nir->ot = IRT(IR_NOP, IRT_NIL);
689           nir->op1 = nir->op2 = 0;
690         }
691       }
692 #endif
693 #if LJ_SOFTFP && LJ_32 && LJ_HASFFI
694       else if (irt_isfloat(ir->t)) {
695         if (st == IRT_NUM) {
696           split_call_l(J, hisubst, oir, ir, IRCALL_softfp_d2f);
697           J->cur.nins--;  /* Drop unused HIOP. */
698         } else {
699           nir->o = IR_CALLN;
700           nir->op2 = st == IRT_INT ? IRCALL_softfp_i2f : IRCALL_softfp_ui2f;
701         }
702       } else if (st == IRT_FLOAT) {
703         nir->o = IR_CALLN;
704         nir->op2 = irt_isint(ir->t) ? IRCALL_softfp_f2i : IRCALL_softfp_f2ui;
705       } else
706 #endif
707 #if LJ_SOFTFP
```

```
      if (st == IRT_NUM || (LJ_32 && LJ_HASFFI && st == IRT_FLOAT)) {
        if (irt_isguard(ir->t)) {
          lua_assert(st == IRT_NUM && irt_isint(ir->t));
          J->cur.nins--;
          ir->prev = split_num2int(J, nir->op1, hisubst[ir->op1], 1);
        } else {
          split_call_l(J, hisubst, oir, ir,
#if LJ_32 && LJ_HASFFI
            st == IRT_NUM ?
              (irt_isint(ir->t) ? IRCALL_softfp_d2i : IRCALL_softfp_d2ui) :
              (irt_isint(ir->t) ? IRCALL_softfp_f2i : IRCALL_softfp_f2ui)
#else
            IRCALL_softfp_d2i
#endif
          );
          J->cur.nins--;  /* Drop unused HIOP. */
        }
      }
#endif
    } else if (ir->o == IR_CALLXS) {
      IRRef hiref;
    split_call:
      hiref = hisubst[ir->op1];
      if (hiref) {
        IROpT ot = nir->ot;
        IRRef op2 = nir->op2;
        nir->ot = IRT(IR_CARG, IRT_NIL);
#if LJ_LE
        nir->op2 = hiref;
#else
        nir->op2 = nir->op1; nir->op1 = hiref;
#endif
        ir->prev = nref = split_emit(J, ot, nref, op2);
      }
      if (LJ_SOFTFP ? irt_is64(ir->t) : irt_isint64(ir->t))
        hi = split_emit(J,
          IRT(IR_HIOP, (LJ_SOFTFP && irt_isnum(ir->t)) ? IRT_SOFTFP : IRT_INT),
          nref, nref);
    } else if (ir->o == IR_CARG) {
      IRRef hiref = hisubst[ir->op1];
      if (hiref) {
        IRRef op2 = nir->op2;
#if LJ_LE
        nir->op2 = hiref;
#else
        nir->op2 = nir->op1; nir->op1 = hiref;
#endif
        ir->prev = nref = split_emit(J, IRT(IR_CARG, IRT_NIL), nref, op2);
        nir = IR(nref);
      }
      hiref = hisubst[ir->op2];
      if (hiref) {
#if !LJ_TARGET_X86
        int carg = 0;
        IRIns *cir;
        for (cir = IR(nir->op1); cir->o == IR_CARG; cir = IR(cir->op1))
          carg++;
        if ((carg & 1) == 0) {  /* Align 64 bit arguments. */
          IRRef op2 = nir->op2;
          nir->op2 = REF_NIL;
          nref = split_emit(J, IRT(IR_CARG, IRT_NIL), nref, op2);
          nir = IR(nref);
        }
#endif
#if LJ_BE
        { IRRef tmp = nir->op2; nir->op2 = hiref; hiref = tmp; }
#endif
        ir->prev = split_emit(J, IRT(IR_CARG, IRT_NIL), nref, hiref);
      }
    } else if (ir->o == IR_CNEWI) {
      if (hisubst[ir->op2])
        split_emit(J, IRT(IR_HIOP, IRT_NIL), nref, hisubst[ir->op2]);
    } else if (ir->o == IR_LOOP) {
      J->loopref = nref;  /* Needed by assembler. */
    }
    hisubst[ref] = hi;  /* Store hiword substitution. */
```

```
  }
  if (snref == nins) {  /* Substitution for last snapshot. */
    snap->ref = J->cur.nins;
    split_subst_snap(J, snap, oir);
  }

  /* Add PHI marks. */
  for (ref = J->cur.nins-1; ref >= REF_FIRST; ref--) {
    IRIns *ir = IR(ref);
    if (ir->o != IR_PHI) break;
    if (!irref_isk(ir->op1)) irt_setphi(IR(ir->op1)->t);
    if (ir->op2 > J->loopref) irt_setphi(IR(ir->op2)->t);
  }
}

/* Protected callback for split pass. */
static TValue *cpsplit(lua_State *L, lua_CFunction dummy, void *ud)
{
  jit_State *J = (jit_State *)ud;
  split_ir(J);
  UNUSED(L); UNUSED(dummy);
  return NULL;
}

#if defined(LUA_USE_ASSERT) || LJ_SOFTFP
/* Slow, but sure way to check whether a SPLIT pass is needed. */
static int split_needsplit(jit_State *J)
{
  IRIns *ir, *irend;
  IRRef ref;
  for (ir = IR(REF_FIRST), irend = IR(J->cur.nins); ir < irend; ir++)
    if (LJ_SOFTFP ? irt_is64orfp(ir->t) : irt_isint64(ir->t))
      return 1;
  if (LJ_SOFTFP) {
    for (ref = J->chain[IR_SLOAD]; ref; ref = IR(ref)->prev)
      if ((IR(ref)->op2 & IRSLOAD_CONVERT))
        return 1;
    if (J->chain[IR_TOBIT])
      return 1;
  }
  for (ref = J->chain[IR_CONV]; ref; ref = IR(ref)->prev) {
    IRType st = (IR(ref)->op2 & IRCONV_SRCMASK);
    if ((LJ_SOFTFP && (st == IRT_NUM || st == IRT_FLOAT)) ||
        st == IRT_I64 || st == IRT_U64)
      return 1;
  }
  return 0;  /* Nope. */
}
#endif

/* SPLIT pass. */
void lj_opt_split(jit_State *J)
{
#if LJ_SOFTFP
  if (!J->needsplit)
    J->needsplit = split_needsplit(J);
#else
  lua_assert(J->needsplit >= split_needsplit(J));  /* Verify flag. */
#endif
  if (J->needsplit) {
    int errcode = lj_vm_cpcall(J->L, NULL, J, cpsplit);
    if (errcode) {
      /* Completely reset the trace to avoid inconsistent dump on abort. */
      J->cur.nins = J->cur.nk = REF_BASE;
      J->cur.nsnap = 0;
      lj_err_throw(J->L, errcode);  /* Propagate errors. */
    }
  }
}

#undef IR

#endif
```

One Level Up Top Level

src/lj_parse.h - luajit-2.0-src

Macros defined

- _LJ_PARSE_H

Source code

```
/*
** Lua parser (source code -> bytecode).
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

#ifndef _LJ_PARSE_H
#define _LJ_PARSE_H

#include "lj_obj.h"
 #include "lj_lex.h"

 LJ_FUNC GCproto *lj_parse(LexState *ls);
 LJ_FUNC GCstr *lj_parse_keepstr(LexState *ls, const char *str, size_t l);
 #if LJ_HASFFI
 LJ_FUNC void lj_parse_keepcdata(LexState *ls, TValue *tv, GCcdata *cd);
 #endif

 #endif
```

src/lj_profile.h - luajit-2.0-src

Macros defined

- _LJ_PROFILE_H

Source code

```
1  /*
2  ** Low-overhead profiling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_PROFILE_H
7  #define _LJ_PROFILE_H
8
9  #include "lj_obj.h"
10
11  #if LJ_HASPROFILE
12
13  LJ_FUNC void LJ_FASTCALL lj_profile_interpreter(lua_State *L);
14  #if !LJ_PROFILE_SIGPROF
15  LJ_FUNC void LJ_FASTCALL lj_profile_hook_enter(global_State *g);
16  LJ_FUNC void LJ_FASTCALL lj_profile_hook_leave(global_State *g);
17  #endif
18
19  #endif
20
21  #endif
```

src/lj_traceerr.h - luajit-2.0-src

Macros defined

- TREDEF

Source code

```
/*
** Trace compiler error messages.
** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
*/

/* This file may be included multiple times with different TREDEF macros. */

/* Recording. */
TREDEF(RECERR,        "error thrown or hook called during recording")
 TREDEF(TRACEUV,        "trace too short")
 TREDEF(TRACEOV,        "trace too long")
 TREDEF(STACKOV,        "trace too deep")
 TREDEF(SNAPOV,        "too many snapshots")
 TREDEF(BLACKL,        "blacklisted")
 TREDEF(NYIBC,        "NYI: bytecode %d")

 /* Recording loop ops. */
 TREDEF(LLEAVE,        "leaving loop in root trace")
 TREDEF(LINNER,        "inner loop in root trace")
 TREDEF(LUNROLL,        "loop unroll limit reached")

 /* Recording calls/returns. */
 TREDEF(BADTYPE,        "bad argument type")
 TREDEF(CJITOFF,        "JIT compilation disabled for function")
 TREDEF(CUNROLL,        "call unroll limit reached")
 TREDEF(DOWNREC,        "down-recursion, restarting")
 TREDEF(NYIFFU,        "NYI: unsupported variant of FastFunc %s")
 TREDEF(NYIRETL,        "NYI: return to lower frame")

 /* Recording indexed load/store. */
 TREDEF(STORENN,        "store with nil or NaN key")
 TREDEF(NOMM,        "missing metamethod")
 TREDEF(IDXLOOP,        "looping index lookup")
 TREDEF(NYITMIX,        "NYI: mixed sparse/dense table")

 /* Recording C data operations. */
 TREDEF(NOCACHE,        "symbol not in cache")
 TREDEF(NYICONV,        "NYI: unsupported C type conversion")
 TREDEF(NYICALL,        "NYI: unsupported C function type")

 /* Optimizations. */
 TREDEF(GFAIL,        "guard would always fail")
 TREDEF(PHIOV,        "too many PHIs")
 TREDEF(TYPEINS,        "persistent type instability")

 /* Assembler. */
 TREDEF(MCODEAL,        "failed to allocate mcode memory")
 TREDEF(MCODEOV,        "machine code too long")
 TREDEF(MCODELM,        "hit mcode limit (retrying)")
 TREDEF(SPILLOV,        "too many spill slots")
 TREDEF(BADRA,        "inconsistent register allocation")
 TREDEF(NYIIR,        "NYI: cannot assemble IR instruction %d")
 TREDEF(NYIPHI,        "NYI: PHI shuffling too complex")
 TREDEF(NYICOAL,        "NYI: register coalescing too complex")

 #undef TREDEF

 /* Detecting unused error messages:
    awk -F, '/^TREDEF/ { gsub(/TREDEF./, ""); printf "grep -q LJ_TRERR_%s *.[ch] || echo %s\n", $1, $1}'
lj_traceerr.h | sh
 */
```

src/lj_udata.h - luajit-2.0-src

Macros defined

- _LJ_UDATA_H

Source code

```
1  /*
2  ** Userdata handling.
3  ** Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
4  */
5
6  #ifndef _LJ_UDATA_H
7  #define _LJ_UDATA_H
8
9  #include "lj_obj.h"
10
11 LJ_FUNC GCudata *lj_udata_new(lua_State *L, MSize sz, GCtab *env);
12 LJ_FUNC void LJ_FASTCALL lj_udata_free(global_State *g, GCudata *ud);
13
14 #endif
```

src/lj_vm.S - luajit-2.0-src

```
1          .file "buildvm_x86.dasc"
2          .text
3          .p2align 4
4
5          .globl lj_vm_asm_begin
6          .hidden lj_vm_asm_begin
7          .type lj_vm_asm_begin, @object
8          .size lj_vm_asm_begin, 0
9  lj_vm_asm_begin:
10  .Lbegin:
11
12           .globl lj_BC_ISLT
13           .hidden lj_BC_ISLT
14           .type lj_BC_ISLT, @function
15           .size lj_BC_ISLT, 133
16  lj_BC_ISLT:
17           .byte 129,124,202,4,255,255,254,255,117,50,129,124,194,4,255,255
18           .byte 254,255,117,69,139,44,202,131,195,4,59,44,194,125,11,15
19           .byte 183,67,254,141,156,131,0,0,254,255,139,3,15,182,204,15
20           .byte 182,232,131,195,4,193,232,16,65,255,36,238,15,135,130,34
21           .byte 0,0,129,124,194,4,255,255,254,255,114,40,15,133,114,34
22           .byte 0,0,242,15,42,4,194,235,32,15,135,101,34,0,0,242
23           .byte 15,42,12,202,242,15,16,4,194,131,195,4,102,15,46,193
24           .byte 118,184,235,171,242,15,16,4,194,131,195,4,102,15,46,4
25           .byte 202,118,167,235,154
26
27           .globl lj_BC_ISGE
28           .hidden lj_BC_ISGE
29           .type lj_BC_ISGE, @function
30           .size lj_BC_ISGE, 133
31  lj_BC_ISGE:
32           .byte 129,124,202,4,255,255,254,255,117,50,129,124,194,4,255,255
33           .byte 254,255,117,69,139,44,202,131,195,4,59,44,194,124,11,15
34           .byte 183,67,254,141,156,131,0,0,254,255,139,3,15,182,204,15
35           .byte 182,232,131,195,4,193,232,16,65,255,36,238,15,135,253,33
36           .byte 0,0,129,124,194,4,255,255,254,255,114,40,15,133,237,33
37           .byte 0,0,242,15,42,4,194,235,32,15,135,224,33,0,0,242
38           .byte 15,42,12,202,242,15,16,4,194,131,195,4,102,15,46,193
39           .byte 119,184,235,171,242,15,16,4,194,131,195,4,102,15,46,4
40           .byte 202,119,167,235,154
41
42           .globl lj_BC_ISLE
43           .hidden lj_BC_ISLE
44           .type lj_BC_ISLE, @function
45           .size lj_BC_ISLE, 133
46  lj_BC_ISLE:
47           .byte 129,124,202,4,255,255,254,255,117,50,129,124,194,4,255,255
48           .byte 254,255,117,69,139,44,202,131,195,4,59,44,194,127,11,15
49           .byte 183,67,254,141,156,131,0,0,254,255,139,3,15,182,204,15
50           .byte 182,232,131,195,4,193,232,16,65,255,36,238,15,135,120,33
51           .byte 0,0,129,124,194,4,255,255,254,255,114,40,15,133,104,33
52           .byte 0,0,242,15,42,4,194,235,32,15,135,91,33,0,0,242
53           .byte 15,42,12,202,242,15,16,4,194,131,195,4,102,15,46,193
54           .byte 114,184,235,171,242,15,16,4,194,131,195,4,102,15,46,4
55           .byte 202,114,167,235,154
56
57           .globl lj_BC_ISGT
58           .hidden lj_BC_ISGT
59           .type lj_BC_ISGT, @function
60           .size lj_BC_ISGT, 133
61  lj_BC_ISGT:
62           .byte 129,124,202,4,255,255,254,255,117,50,129,124,194,4,255,255
63           .byte 254,255,117,69,139,44,202,131,195,4,59,44,194,126,11,15
64           .byte 183,67,254,141,156,131,0,0,254,255,139,3,15,182,204,15
65           .byte 182,232,131,195,4,193,232,16,65,255,36,238,15,135,243,32
66           .byte 0,0,129,124,194,4,255,255,254,255,114,40,15,133,227,32
67           .byte 0,0,242,15,42,4,194,235,32,15,135,214,32,0,0,242
68           .byte 15,42,12,202,242,15,16,4,194,131,195,4,102,15,46,193
69           .byte 115,184,235,171,242,15,16,4,194,131,195,4,102,15,46,4
70           .byte 202,115,167,235,154
71
```

```
        .globl lj_BC_ISEQV
        .hidden lj_BC_ISEQV
        .type lj_BC_ISEQV, @function
        .size lj_BC_ISEQV, 195
lj_BC_ISEQV:
        .byte 139,108,194,4,131,195,4,129,253,255,255,254,255,117,47,129
        .byte 124,202,4,255,255,254,255,117,58,139,44,194,59,44,202,117
        .byte 11,15,183,67,254,141,156,131,0,0,254,255,139,3,15,182
        .byte 204,15,182,232,131,195,4,193,232,16,65,255,36,238,119,60
        .byte 129,124,202,4,255,255,254,255,114,23,117,48,242,15,42,4
        .byte 202,235,19,119,39,242,15,42,4,194,102,15,46,4,202,235
        .byte 10,242,15,16,4,202,102,15,46,4,194,122,13,117,11,15
        .byte 183,67,254,141,156,131,0,0,254,255,235,176,131,253,245,15
        .byte 132,163,32,0,0,131,124,202,4,245,15,132,152,32,0,0
        .byte 57,108,202,4,117,228,131,253,253,115,212,139,12,202,139,4
        .byte 194,57,193,116,202,131,253,244,119,208,131,253,243,114,203,139
        .byte 105,16,133,237,116,196,246,69,6,16,117,190,49,237,233,72
        .byte 32,0,0

        .globl lj_BC_ISNEV
        .hidden lj_BC_ISNEV
        .type lj_BC_ISNEV, @function
        .size lj_BC_ISNEV, 198
lj_BC_ISNEV:
        .byte 139,108,194,4,131,195,4,129,253,255,255,254,255,117,47,129
        .byte 124,202,4,255,255,254,255,117,58,139,44,194,59,44,202,116
        .byte 11,15,183,67,254,141,156,131,0,0,254,255,139,3,15,182
        .byte 204,15,182,232,131,195,4,193,232,16,65,255,36,238,119,60
        .byte 129,124,202,4,255,255,254,255,114,23,117,48,242,15,42,4
        .byte 202,235,19,119,39,242,15,42,4,194,102,15,46,4,202,235
        .byte 10,242,15,16,4,202,102,15,46,4,194,122,2,116,11,15
        .byte 183,67,254,141,156,131,0,0,254,255,235,176,131,253,245,15
        .byte 132,224,31,0,0,131,124,202,4,245,15,132,213,31,0,0
        .byte 57,108,202,4,117,217,131,253,253,115,223,139,12,202,139,4
        .byte 194,57,193,116,213,131,253,244,119,197,131,253,243,114,192,139
        .byte 105,16,133,237,116,185,246,69,6,16,117,179,189,1,0,0
        .byte 0,233,130,31,0,0

        .globl lj_BC_ISEQS
        .hidden lj_BC_ISEQS
        .type lj_BC_ISEQS, @function
        .size lj_BC_ISEQS, 67
lj_BC_ISEQS:
        .byte 72,247,208,139,108,202,4,131,195,4,131,253,251,117,38,139
        .byte 12,202,65,59,12,135,117,11,15,183,67,254,141,156,131,0
        .byte 0,254,255,139,3,15,182,204,15,182,232,131,195,4,193,232
        .byte 16,65,255,36,238,131,253,245,15,133,40,255,255,255,233,92
        .byte 31,0,0

        .globl lj_BC_ISNES
        .hidden lj_BC_ISNES
        .type lj_BC_ISNES, @function
        .size lj_BC_ISNES, 63
lj_BC_ISNES:
        .byte 72,247,208,139,108,202,4,131,195,4,131,253,251,117,38,139
        .byte 12,202,65,59,12,135,116,11,15,183,67,254,141,156,131,0
        .byte 0,254,255,139,3,15,182,204,15,182,232,131,195,4,193,232
        .byte 16,65,255,36,238,131,253,245,117,222,233,29,31,0,0

        .globl lj_BC_ISEQN
        .hidden lj_BC_ISEQN
        .type lj_BC_ISEQN, @function
        .size lj_BC_ISEQN, 136
lj_BC_ISEQN:
        .byte 139,108,202,4,131,195,4,129,253,255,255,254,255,117,49,65
        .byte 129,124,199,4,255,255,254,255,117,59,65,139,44,199,59,44
        .byte 202,117,11,15,183,67,254,141,156,131,0,0,254,255,139,3
        .byte 15,182,204,15,182,232,131,195,4,193,232,16,65,255,36,238
        .byte 119,60,65,129,124,199,4,255,255,254,255,114,21,242,65,15
        .byte 42,4,199,235,19,242,15,42,4,202,102,65,15,46,4,199
        .byte 235,11,242,65,15,16,4,199,102,15,46,4,202,122,13,117
        .byte 11,15,183,67,254,141,156,131,0,0,254,255,235,176,131,253
        .byte 245,117,171,233,149,30,0,0

        .globl lj_BC_ISNEN
        .hidden lj_BC_ISNEN
```

```
        .type lj_BC_ISNEN, @function
        .size lj_BC_ISNEN, 136
lj_BC_ISNEN:
        .byte 139,108,202,4,131,195,4,129,253,255,255,254,255,117,49,65
        .byte 129,124,199,4,255,255,254,255,117,59,65,139,44,199,59,44
        .byte 202,116,11,15,183,67,254,141,156,131,0,0,254,255,139,3
        .byte 15,182,204,15,182,232,131,195,4,193,232,16,65,255,36,238
        .byte 119,60,65,129,124,199,4,255,255,254,255,114,21,242,65,15
        .byte 42,4,199,235,19,242,15,42,4,202,102,65,15,46,4,199
        .byte 235,11,242,65,15,16,4,199,102,15,46,4,202,122,2,116
        .byte 11,15,183,67,254,141,156,131,0,0,254,255,235,176,131,253
        .byte 245,117,238,233,13,30,0,0

        .globl lj_BC_ISEQP
        .hidden lj_BC_ISEQP
        .type lj_BC_ISEQP, @function
        .size lj_BC_ISEQP, 53
lj_BC_ISEQP:
        .byte 72,247,208,139,108,202,4,131,195,4,57,197,117,29,15,183
        .byte 67,254,141,156,131,0,0,254,255,139,3,15,182,204,15,182
        .byte 232,131,195,4,193,232,16,65,255,36,238,131,253,245,117,233
        .byte 233,216,29,0,0

        .globl lj_BC_ISNEP
        .hidden lj_BC_ISNEP
        .type lj_BC_ISNEP, @function
        .size lj_BC_ISNEP, 52
lj_BC_ISNEP:
        .byte 72,247,208,139,108,202,4,131,195,4,57,197,116,20,131,253
        .byte 245,15,132,193,29,0,0,15,183,67,254,141,156,131,0,0
        .byte 254,255,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238

        .globl lj_BC_ISTC
        .hidden lj_BC_ISTC
        .type lj_BC_ISTC, @function
        .size lj_BC_ISTC, 51
lj_BC_ISTC:
        .byte 139,108,194,4,131,195,4,131,253,254,115,21,137,108,202,4
        .byte 139,44,194,137,44,202,15,183,67,254,141,156,131,0,0,254
        .byte 255,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65
        .byte 255,36,238

        .globl lj_BC_ISFC
        .hidden lj_BC_ISFC
        .type lj_BC_ISFC, @function
        .size lj_BC_ISFC, 51
lj_BC_ISFC:
        .byte 139,108,194,4,131,195,4,131,253,254,114,21,137,108,202,4
        .byte 139,44,194,137,44,202,15,183,67,254,141,156,131,0,0,254
        .byte 255,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65
        .byte 255,36,238

        .globl lj_BC_IST
        .hidden lj_BC_IST
        .type lj_BC_IST, @function
        .size lj_BC_IST, 41
lj_BC_IST:
        .byte 139,108,194,4,131,195,4,131,253,254,115,11,15,183,67,254
        .byte 141,156,131,0,0,254,255,139,3,15,182,204,15,182,232,131
        .byte 195,4,193,232,16,65,255,36,238

        .globl lj_BC_ISF
        .hidden lj_BC_ISF
        .type lj_BC_ISF, @function
        .size lj_BC_ISF, 41
lj_BC_ISF:
        .byte 139,108,194,4,131,195,4,131,253,254,114,11,15,183,67,254
        .byte 141,156,131,0,0,254,255,139,3,15,182,204,15,182,232,131
        .byte 195,4,193,232,16,65,255,36,238

        .globl lj_BC_ISTYPE
        .hidden lj_BC_ISTYPE
        .type lj_BC_ISTYPE, @function
        .size lj_BC_ISTYPE, 28
lj_BC_ISTYPE:
```

```
        .byte 3,68,202,4,15,133,252,28,0,0,139,3,15,182,204,15
        .byte 182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_ISNUM
        .hidden lj_BC_ISNUM
        .type lj_BC_ISNUM, @function
        .size lj_BC_ISNUM, 32
lj_BC_ISNUM:
        .byte 129,124,202,4,255,255,254,255,15,131,220,28,0,0,139,3
        .byte 15,182,204,15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_MOV
        .hidden lj_BC_MOV
        .type lj_BC_MOV, @function
        .size lj_BC_MOV, 26
lj_BC_MOV:
        .byte 72,139,44,194,72,137,44,202,139,3,15,182,204,15,182,232
        .byte 131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_NOT
        .hidden lj_BC_NOT
        .type lj_BC_NOT, @function
        .size lj_BC_NOT, 32
lj_BC_NOT:
        .byte 49,237,131,124,194,4,254,131,213,253,137,108,202,4,139,3
        .byte 15,182,204,15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_UNM
        .hidden lj_BC_UNM
        .type lj_BC_UNM, @function
        .size lj_BC_UNM, 99
lj_BC_UNM:
        .byte 129,124,194,4,255,255,254,255,117,53,139,44,194,247,221,112
        .byte 29,199,68,202,4,255,255,254,255,137,44,202,139,3,15,182
        .byte 204,15,182,232,131,195,4,193,232,16,65,255,36,238,199,68
        .byte 202,4,0,0,224,65,199,4,202,0,0,0,0,235,221,15
        .byte 135,128,28,0,0,242,15,16,4,194,72,184,0,0,0,0
        .byte 0,0,0,128,102,72,15,110,200,15,87,193,242,15,17,4
        .byte 202,235,185

        .globl lj_BC_LEN
        .hidden lj_BC_LEN
        .type lj_BC_LEN, @function
        .size lj_BC_LEN, 71
lj_BC_LEN:
        .byte 131,124,194,4,251,117,35,139,4,194,139,64,12,199,68,202
        .byte 4,255,255,254,255,137,4,202,139,3,15,182,204,15,182,232
        .byte 131,195,4,193,232,16,65,255,36,238,131,124,194,4,244,15
        .byte 133,125,28,0,0,139,60,194,137,213
        call lj_tab_len
        .byte 137,234,15,182,75,253,235,198

        .globl lj_BC_ADDVN
        .hidden lj_BC_ADDVN
        .type lj_BC_ADDVN, @function
        .size lj_BC_ADDVN, 77
lj_BC_ADDVN:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,133
        .byte 243,27,0,0,65,129,124,199,4,255,255,254,255,15,133,228
        .byte 27,0,0,139,44,234,65,3,44,199,15,128,211,27,0,0
        .byte 199,68,202,4,255,255,254,255,137,44,202,139,3,15,182,204
        .byte 15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_SUBVN
        .hidden lj_BC_SUBVN
        .type lj_BC_SUBVN, @function
        .size lj_BC_SUBVN, 77
lj_BC_SUBVN:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,133
        .byte 166,27,0,0,65,129,124,199,4,255,255,254,255,15,133,151
        .byte 27,0,0,139,44,234,65,43,44,199,15,128,134,27,0,0
        .byte 199,68,202,4,255,255,254,255,137,44,202,139,3,15,182,204
        .byte 15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_MULVN
        .hidden lj_BC_MULVN
```

```
        .type lj_BC_MULVN, @function
        .size lj_BC_MULVN, 78
lj_BC_MULVN:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,133
        .byte 89,27,0,0,65,129,124,199,4,255,255,254,255,15,133,74
        .byte 27,0,0,139,44,234,65,15,175,44,199,15,128,56,27,0
        .byte 0,199,68,202,4,255,255,254,255,137,44,202,139,3,15,182
        .byte 204,15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_DIVVN
        .hidden lj_BC_DIVVN
        .type lj_BC_DIVVN, @function
        .size lj_BC_DIVVN, 69
lj_BC_DIVVN:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 11,27,0,0,65,129,124,199,4,255,255,254,255,15,131,252
        .byte 26,0,0,242,15,16,4,234,242,65,15,94,4,199,242,15
        .byte 17,4,202,139,3,15,182,204,15,182,232,131,195,4,193,232
        .byte 16,65,255,36,238

        .globl lj_BC_MODVN
        .hidden lj_BC_MODVN
        .type lj_BC_MODVN, @function
        .size lj_BC_MODVN, 74
lj_BC_MODVN:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 198,26,0,0,65,129,124,199,4,255,255,254,255,15,131,183
        .byte 26,0,0,242,15,16,4,234,242,65,15,16,12,199,232,245
        .byte 50,0,0,242,15,17,4,202,139,3,15,182,204,15,182,232
        .byte 131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_ADDNV
        .hidden lj_BC_ADDNV
        .type lj_BC_ADDNV, @function
        .size lj_BC_ADDNV, 77
lj_BC_ADDNV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,133
        .byte 134,26,0,0,65,129,124,199,4,255,255,254,255,15,133,119
        .byte 26,0,0,65,139,4,199,3,4,234,15,128,102,26,0,0
        .byte 199,68,202,4,255,255,254,255,137,4,202,139,3,15,182,204
        .byte 15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_SUBNV
        .hidden lj_BC_SUBNV
        .type lj_BC_SUBNV, @function
        .size lj_BC_SUBNV, 77
lj_BC_SUBNV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,133
        .byte 57,26,0,0,65,129,124,199,4,255,255,254,255,15,133,42
        .byte 26,0,0,65,139,4,199,43,4,234,15,128,25,26,0,0
        .byte 199,68,202,4,255,255,254,255,137,4,202,139,3,15,182,204
        .byte 15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_MULNV
        .hidden lj_BC_MULNV
        .type lj_BC_MULNV, @function
        .size lj_BC_MULNV, 78
lj_BC_MULNV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,133
        .byte 236,25,0,0,65,129,124,199,4,255,255,254,255,15,133,221
        .byte 25,0,0,65,139,4,199,15,175,4,234,15,128,203,25,0
        .byte 0,199,68,202,4,255,255,254,255,137,4,202,139,3,15,182
        .byte 204,15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_DIVNV
        .hidden lj_BC_DIVNV
        .type lj_BC_DIVNV, @function
        .size lj_BC_DIVNV, 69
lj_BC_DIVNV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 158,25,0,0,65,129,124,199,4,255,255,254,255,15,131,143
        .byte 25,0,0,242,65,15,16,4,199,242,15,94,4,234,242,15
        .byte 17,4,202,139,3,15,182,204,15,182,232,131,195,4,193,232
        .byte 16,65,255,36,238

        .globl lj_BC_MODNV
```

```
        .hidden lj_BC_MODNV
        .type lj_BC_MODNV, @function
        .size lj_BC_MODNV, 51
lj_BC_MODNV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 89,25,0,0,65,129,124,199,4,255,255,254,255,15,131,74
        .byte 25,0,0,242,65,15,16,4,199,242,15,16,12,234,233,132
        .byte 254,255,255

        .globl lj_BC_ADDVV
        .hidden lj_BC_ADDVV
        .type lj_BC_ADDVV, @function
        .size lj_BC_ADDVV, 75
lj_BC_ADDVV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,133
        .byte 59,25,0,0,129,124,194,4,255,255,254,255,15,133,45,25
        .byte 0,0,139,44,234,3,44,194,15,128,29,25,0,0,199,68
        .byte 202,4,255,255,254,255,137,44,202,139,3,15,182,204,15,182
        .byte 232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_SUBVV
        .hidden lj_BC_SUBVV
        .type lj_BC_SUBVV, @function
        .size lj_BC_SUBVV, 75
lj_BC_SUBVV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,133
        .byte 240,24,0,0,129,124,194,4,255,255,254,255,15,133,226,24
        .byte 0,0,139,44,234,43,44,194,15,128,210,24,0,0,199,68
        .byte 202,4,255,255,254,255,137,44,202,139,3,15,182,204,15,182
        .byte 232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_MULVV
        .hidden lj_BC_MULVV
        .type lj_BC_MULVV, @function
        .size lj_BC_MULVV, 76
lj_BC_MULVV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,133
        .byte 165,24,0,0,129,124,194,4,255,255,254,255,15,133,151,24
        .byte 0,0,139,44,234,15,175,44,194,15,128,134,24,0,0,199
        .byte 68,202,4,255,255,254,255,137,44,202,139,3,15,182,204,15
        .byte 182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_DIVVV
        .hidden lj_BC_DIVVV
        .type lj_BC_DIVVV, @function
        .size lj_BC_DIVVV, 67
lj_BC_DIVVV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 89,24,0,0,129,124,194,4,255,255,254,255,15,131,75,24
        .byte 0,0,242,15,16,4,234,242,15,94,4,194,242,15,17,4
        .byte 202,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65
        .byte 255,36,238

        .globl lj_BC_MODVV
        .hidden lj_BC_MODVV
        .type lj_BC_MODVV, @function
        .size lj_BC_MODVV, 49
lj_BC_MODVV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 22,24,0,0,129,124,194,4,255,255,254,255,15,131,8,24
        .byte 0,0,242,15,16,4,234,242,15,16,12,194,233,46,253,255
        .byte 255

        .globl lj_BC_POW
        .hidden lj_BC_POW
        .type lj_BC_POW, @function
        .size lj_BC_POW, 80
lj_BC_POW:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 229,23,0,0,129,124,194,4,255,255,254,255,15,131,215,23
        .byte 0,0,242,15,16,4,234,242,15,16,12,194,137,213
        call pow@PLT
        .byte 15,182,75,253,137,234,242,15,17,4,202,139,3,15,182,204
        .byte 15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_CAT
```

```
        .hidden lj_BC_CAT
        .type lj_BC_CAT, @function
        .size lj_BC_CAT, 76
lj_BC_CAT:
        .byte 15,182,236,15,182,192,139,124,36,24,137,87,16,141,52,194
        .byte 137,194,41,234,137,253,137,92,36,28
        call lj_meta_cat
        .byte 139,85,16,133,192,15,133,176,23,0,0,15,182,107,255,15
        .byte 182,75,253,72,139,4,234,72,137,4,202,139,3,15,182,204
        .byte 15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_KSTR
        .hidden lj_BC_KSTR
        .type lj_BC_KSTR, @function
        .size lj_BC_KSTR, 36
lj_BC_KSTR:
        .byte 72,247,208,65,139,4,135,199,68,202,4,251,255,255,255,137
        .byte 4,202,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238

        .globl lj_BC_KCDATA
        .hidden lj_BC_KCDATA
        .type lj_BC_KCDATA, @function
        .size lj_BC_KCDATA, 36
lj_BC_KCDATA:
        .byte 72,247,208,65,139,4,135,199,68,202,4,245,255,255,255,137
        .byte 4,202,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238

        .globl lj_BC_KSHORT
        .hidden lj_BC_KSHORT
        .type lj_BC_KSHORT, @function
        .size lj_BC_KSHORT, 32
lj_BC_KSHORT:
        .byte 15,191,192,199,68,202,4,255,255,254,255,137,4,202,139,3
        .byte 15,182,204,15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_KNUM
        .hidden lj_BC_KNUM
        .type lj_BC_KNUM, @function
        .size lj_BC_KNUM, 29
lj_BC_KNUM:
        .byte 242,65,15,16,4,199,242,15,17,4,202,139,3,15,182,204
        .byte 15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_KPRI
        .hidden lj_BC_KPRI
        .type lj_BC_KPRI, @function
        .size lj_BC_KPRI, 25
lj_BC_KPRI:
        .byte 72,247,208,137,68,202,4,139,3,15,182,204,15,182,232,131
        .byte 195,4,193,232,16,65,255,36,238

        .globl lj_BC_KNIL
        .hidden lj_BC_KNIL
        .type lj_BC_KNIL, @function
        .size lj_BC_KNIL, 43
lj_BC_KNIL:
        .byte 141,76,202,12,141,68,194,4,189,255,255,255,255,137,105,248
        .byte 137,41,131,193,8,57,193,118,247,139,3,15,182,204,15,182
        .byte 232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_UGET
        .hidden lj_BC_UGET
        .type lj_BC_UGET, @function
        .size lj_BC_UGET, 36
lj_BC_UGET:
        .byte 139,106,248,139,108,133,20,139,109,16,72,139,69,0,72,137
        .byte 4,202,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238

        .globl lj_BC_USETV
        .hidden lj_BC_USETV
        .type lj_BC_USETV, @function
        .size lj_BC_USETV, 87
lj_BC_USETV:
```

```
        .byte 139,106,248,139,108,141,20,128,125,6,0,139,109,16,139,12
        .byte 194,139,68,194,4,137,77,0,137,69,4,116,6,246,69,252
        .byte 4,117,18,139,3,15,182,204,15,182,232,131,195,4,193,232
        .byte 16,65,255,36,238,131,232,252,131,248,246,118,230,246,65,4
        .byte 3,116,224,137,238,137,213,65,141,190,32,244,255,255
        call lj_gc_barrieruv
        .byte 137,234,235,204

        .globl lj_BC_USETS
        .hidden lj_BC_USETS
        .type lj_BC_USETS, @function
        .size lj_BC_USETS, 82
lj_BC_USETS:
        .byte 72,247,208,139,106,248,139,108,141,20,65,139,12,135,139,69
        .byte 16,137,8,199,64,4,251,255,255,255,246,69,4,4,117,18
        .byte 139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,255
        .byte 36,238,246,65,4,3,116,232,128,125,6,0,116,226,137,213
        .byte 137,198,65,141,190,32,244,255,255
        call lj_gc_barrieruv
        .byte 137,234,235,206

        .globl lj_BC_USETN
        .hidden lj_BC_USETN
        .type lj_BC_USETN, @function
        .size lj_BC_USETN, 38
lj_BC_USETN:
        .byte 139,106,248,242,65,15,16,4,199,139,108,141,20,139,77,16
        .byte 242,15,17,1,139,3,15,182,204,15,182,232,131,195,4,193
        .byte 232,16,65,255,36,238

        .globl lj_BC_USETP
        .hidden lj_BC_USETP
        .type lj_BC_USETP, @function
        .size lj_BC_USETP, 34
lj_BC_USETP:
        .byte 72,247,208,139,106,248,139,108,141,20,139,77,16,137,65,4
        .byte 139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,255
        .byte 36,238

        .globl lj_BC_UCLO
        .hidden lj_BC_UCLO
        .type lj_BC_UCLO, @function
        .size lj_BC_UCLO, 51
lj_BC_UCLO:
        .byte 141,156,131,0,0,254,255,139,108,36,24,131,125,40,0,116
        .byte 16,137,85,16,141,52,202,137,239
        call lj_func_closeuv
        .byte 139,85,16,139,3,15,182,204,15,182,232,131,195,4,193,232
        .byte 16,65,255,36,238

        .globl lj_BC_FNEW
        .hidden lj_BC_FNEW
        .type lj_BC_FNEW, @function
        .size lj_BC_FNEW, 64
lj_BC_FNEW:
        .byte 72,247,208,139,108,36,24,137,85,16,139,82,248,65,139,52
        .byte 135,137,239,137,92,36,28
        call lj_func_newL_gc
        .byte 139,85,16,15,182,75,253,137,4,202,199,68,202,4,247,255
        .byte 255,255,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238

        .globl lj_BC_TNEW
        .hidden lj_BC_TNEW
        .type lj_BC_TNEW, @function
        .size lj_BC_TNEW, 109
lj_BC_TNEW:
        .byte 139,108,36,24,137,85,16,65,139,142,64,244,255,255,65,59
        .byte 142,68,244,255,255,137,92,36,28,115,69,137,194,37,255,7
        .byte 0,0,193,234,11,61,255,7,0,0,116,45,137,239,137,198
        call lj_tab_new
        .byte 139,85,16,15,182,75,253,137,4,202,199,68,202,4,244,255
        .byte 255,255,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238,184,1,8,0,0,235,204,137,239
        call lj_gc_step_fixtop
        .byte 15,183,67,254,235,174
```

```
        .globl lj_BC_TDUP
        .hidden lj_BC_TDUP
        .type lj_BC_TDUP, @function
        .size lj_BC_TDUP, 93
lj_BC_TDUP:
        .byte 72,247,208,139,108,36,24,65,139,142,64,244,255,255,137,92
        .byte 36,28,65,59,142,68,244,255,255,137,85,16,115,47,65,139
        .byte 52,135,137,239
        call lj_tab_dup
        .byte 139,85,16,15,182,75,253,137,4,202,199,68,202,4,244,255
        .byte 255,255,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238,137,239
        call lj_gc_step_fixtop
        .byte 15,183,67,254,72,247,208,235,193

        .globl lj_BC_GGET
        .hidden lj_BC_GGET
        .type lj_BC_GGET, @function
        .size lj_BC_GGET, 18
lj_BC_GGET:
        .byte 72,247,208,139,106,248,139,109,8,65,139,4,135,233,173,0
        .byte 0,0

        .globl lj_BC_GSET
        .hidden lj_BC_GSET
        .type lj_BC_GSET, @function
        .size lj_BC_GSET, 18
lj_BC_GSET:
        .byte 72,247,208,139,106,248,139,109,8,65,139,4,135,233,71,2
        .byte 0,0

        .globl lj_BC_TGETV
        .hidden lj_BC_TGETV
        .type lj_BC_TGETV, @function
        .size lj_BC_TGETV, 128
lj_BC_TGETV:
        .byte 15,182,236,15,182,192,131,124,234,4,244,15,133,232,17,0
        .byte 0,139,44,234,129,124,194,4,255,255,254,255,117,82,139,4
        .byte 194,59,69,24,15,131,207,17,0,0,193,224,3,3,69,8
        .byte 131,120,4,255,116,25,72,139,40,72,137,44,202,139,3,15
        .byte 182,204,15,182,232,131,195,4,193,232,16,65,255,36,238,131
        .byte 125,16,0,116,17,139,77,16,246,65,6,1,15,132,151,17
        .byte 0,0,15,182,75,253,199,68,202,4,255,255,255,255,235,205
        .byte 131,124,194,4,251,15,133,126,17,0,0,139,4,194,235,27

        .globl lj_BC_TGETS
        .hidden lj_BC_TGETS
        .type lj_BC_TGETS, @function
        .size lj_BC_TGETS, 124
lj_BC_TGETS:
        .byte 15,182,236,15,182,192,72,247,208,65,139,4,135,131,124,234
        .byte 4,244,15,133,35,17,0,0,139,44,234,139,77,28,35,72
        .byte 8,107,201,24,3,77,20,131,121,12,251,117,54,57,65,8
        .byte 117,49,131,121,4,255,116,50,15,182,67,253,72,139,41,72
        .byte 137,44,194,139,3,15,182,204,15,182,232,131,195,4,193,232
        .byte 16,65,255,36,238,15,182,67,253,199,68,194,4,255,255,255
        .byte 255,235,224,139,73,16,133,201,117,189,139,77,16,133,201,116
        .byte 228,246,65,6,1,117,222,233,191,16,0,0

        .globl lj_BC_TGETB
        .hidden lj_BC_TGETB
        .type lj_BC_TGETB, @function
        .size lj_BC_TGETB, 99
lj_BC_TGETB:
        .byte 15,182,236,15,182,192,131,124,234,4,244,15,133,215,16,0
        .byte 0,139,44,234,59,69,24,15,131,203,16,0,0,193,224,3
        .byte 3,69,8,131,120,4,255,116,25,72,139,40,72,137,44,202
        .byte 139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,255
        .byte 36,238,131,125,16,0,116,17,139,77,16,246,65,6,1,15
        .byte 132,147,16,0,0,15,182,75,253,199,68,202,4,255,255,255
        .byte 255,235,205

        .globl lj_BC_TGETR
        .hidden lj_BC_TGETR
        .type lj_BC_TGETR, @function
```

```
        .size lj_BC_TGETR, 52
lj_BC_TGETR:
        .byte 15,182,236,15,182,192,139,44,234,139,4,194,59,69,24,15
        .byte 131,230,16,0,0,193,224,3,3,69,8,72,139,40,72,137
        .byte 44,202,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238

        .globl lj_BC_TSETV
        .hidden lj_BC_TSETV
        .type lj_BC_TSETV, @function
        .size lj_BC_TSETV, 153
lj_BC_TSETV:
        .byte 15,182,236,15,182,192,131,124,234,4,244,15,133,26,17,0
        .byte 0,139,44,234,129,124,194,4,255,255,254,255,117,80,139,4
        .byte 194,59,69,24,15,131,1,17,0,0,193,224,3,3,69,8
        .byte 131,120,4,255,116,31,246,69,4,4,117,66,72,139,44,202
        .byte 72,137,40,139,3,15,182,204,15,182,232,131,195,4,193,232
        .byte 16,65,255,36,238,131,125,16,0,116,219,139,77,16,246,65
        .byte 6,2,15,132,195,16,0,0,15,182,75,253,235,200,131,124
        .byte 194,4,251,15,133,178,16,0,0,139,4,194,235,54,128,101
        .byte 4,251,65,139,142,92,244,255,255,65,137,174,92,244,255,255
        .byte 137,77,12,15,182,75,253,235,163

        .globl lj_BC_TSETS
        .hidden lj_BC_TSETS
        .type lj_BC_TSETS, @function
        .size lj_BC_TSETS, 229
lj_BC_TSETS:
        .byte 15,182,236,15,182,192,72,247,208,65,139,4,135,131,124,234
        .byte 4,244,15,133,60,16,0,0,139,44,234,139,77,28,35,72
        .byte 8,107,201,24,198,69,6,0,3,77,20,131,121,12,251,117
        .byte 77,57,65,8,117,72,131,121,4,255,116,39,246,69,4,4
        .byte 15,133,133,0,0,0,15,182,67,253,72,139,44,194,72,137
        .byte 41,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65
        .byte 255,36,238,131,125,16,0,116,211,137,12,36,139,77,16,246
        .byte 65,6,2,15,132,219,15,0,0,139,12,36,235,190,139,73
        .byte 16,133,201,117,166,139,77,16,133,201,116,10,246,65,6,2
        .byte 15,132,190,15,0,0,137,4,36,199,68,36,4,251,255,255
        .byte 255,137,108,36,8,139,124,36,24,137,87,16,72,141,20,36
        .byte 137,238,137,253,137,92,36,28
        call lj_tab_newkey
        .byte 139,85,16,139,108,36,8,137,193,233,113,255,255,255,128,101
        .byte 4,251,65,139,134,92,244,255,255,65,137,174,92,244,255,255
        .byte 137,69,12,233,97,255,255,255

        .globl lj_BC_TSETB
        .hidden lj_BC_TSETB
        .type lj_BC_TSETB, @function
        .size lj_BC_TSETB, 124
lj_BC_TSETB:
        .byte 15,182,236,15,182,192,131,124,234,4,244,15,133,135,15,0
        .byte 0,139,44,234,59,69,24,15,131,123,15,0,0,193,224,3
        .byte 3,69,8,131,120,4,255,116,31,246,69,4,4,117,50,72
        .byte 139,12,202,72,137,8,139,3,15,182,204,15,182,232,131,195
        .byte 4,193,232,16,65,255,36,238,131,125,16,0,116,219,139,77
        .byte 16,246,65,6,2,15,132,61,15,0,0,15,182,75,253,235
        .byte 200,128,101,4,251,65,139,142,92,244,255,255,65,137,174,92
        .byte 244,255,255,137,77,12,15,182,75,253,235,179

        .globl lj_BC_TSETM
        .hidden lj_BC_TSETM
        .type lj_BC_TSETM, @function
        .size lj_BC_TSETM, 142
lj_BC_TSETM:
        .byte 68,137,60,36,69,139,60,199,141,12,202,139,105,248,246,69
        .byte 4,4,117,99,139,68,36,4,131,232,1,116,37,68,1,248
        .byte 59,69,24,119,51,68,41,248,65,193,231,3,68,3,125,8
        .byte 72,139,41,131,193,8,73,137,47,65,131,199,8,131,232,1
        .byte 117,238,68,139,60,36,139,3,15,182,204,15,182,232,131,195
        .byte 4,193,232,16,65,255,36,238,139,124,36,24,137,87,16,137
        .byte 238,137,194,137,253,137,92,36,28
        call lj_tab_reasize
        .byte 139,85,16,15,182,75,253,235,145,128,101,4,251,65,139,134
        .byte 92,244,255,255,65,137,174,92,244,255,255,137,69,12,235,134

        .globl lj_BC_TSETR
```

```
        .hidden lj_BC_TSETR
        .type lj_BC_TSETR, @function
        .size lj_BC_TSETR, 85
lj_BC_TSETR:
        .byte 15,182,236,15,182,192,139,44,234,139,4,194,246,69,4,4
        .byte 117,40,59,69,24,15,131,245,14,0,0,193,224,3,3,69
        .byte 8,72,139,44,202,72,137,40,139,3,15,182,204,15,182,232
        .byte 131,195,4,193,232,16,65,255,36,238,128,101,4,251,65,139
        .byte 142,92,244,255,255,65,137,174,92,244,255,255,137,77,12,15
        .byte 182,75,253,235,189

        .globl lj_BC_CALLM
        .hidden lj_BC_CALLM
        .type lj_BC_CALLM, @function
        .size lj_BC_CALLM, 46
lj_BC_CALLM:
        .byte 15,182,192,3,68,36,4,131,124,202,4,247,139,44,202,15
        .byte 133,248,15,0,0,141,84,202,8,137,90,252,139,93,16,139
        .byte 11,15,182,233,15,182,205,131,195,4,65,255,36,238

        .globl lj_BC_CALL
        .hidden lj_BC_CALL
        .type lj_BC_CALL, @function
        .size lj_BC_CALL, 42
lj_BC_CALL:
        .byte 15,182,192,131,124,202,4,247,139,44,202,15,133,206,15,0
        .byte 0,141,84,202,8,137,90,252,139,93,16,139,11,15,182,233
        .byte 15,182,205,131,195,4,65,255,36,238

        .globl lj_BC_CALLMT
        .hidden lj_BC_CALLMT
        .type lj_BC_CALLMT, @function
        .size lj_BC_CALLMT, 4
lj_BC_CALLMT:
        .byte 3,68,36,4

        .globl lj_BC_CALLT
        .hidden lj_BC_CALLT
        .type lj_BC_CALLT, @function
        .size lj_BC_CALLT, 148
lj_BC_CALLT:
        .byte 141,76,202,8,65,137,215,139,105,248,131,121,252,247,15,133
        .byte 161,15,0,0,139,90,252,247,195,3,0,0,0,117,91,137
        .byte 106,248,137,68,36,4,131,232,1,116,21,72,139,41,131,193
        .byte 8,73,137,47,65,131,199,8,131,232,1,117,238,139,106,248
        .byte 139,68,36,4,128,125,6,1,119,18,139,93,16,139,11,15
        .byte 182,233,15,182,205,131,195,4,65,255,36,238,247,195,3,0
        .byte 0,0,117,230,15,182,75,253,72,247,209,68,139,124,202,248
        .byte 69,139,127,16,69,139,127,208,235,208,131,235,3,247,195,7
        .byte 0,0,0,117,10,41,218,65,137,215,139,90,252,235,144,131
        .byte 195,3,235,139

        .globl lj_BC_ITERC
        .hidden lj_BC_ITERC
        .type lj_BC_ITERC, @function
        .size lj_BC_ITERC, 68
lj_BC_ITERC:
        .byte 141,76,202,8,72,139,105,232,72,139,65,240,72,137,41,72
        .byte 137,65,8,139,105,224,139,65,228,137,105,248,137,65,252,131
        .byte 248,247,184,3,0,0,0,15,133,244,14,0,0,137,202,137
        .byte 90,252,139,93,16,139,11,15,182,233,15,182,205,131,195,4
        .byte 65,255,36,238

        .globl lj_BC_ITERN
        .hidden lj_BC_ITERN
        .type lj_BC_ITERN, @function
        .size lj_BC_ITERN, 167
lj_BC_ITERN:
        .byte 68,137,60,36,68,137,116,36,4,139,108,202,240,139,68,202
        .byte 248,68,139,117,24,131,195,4,68,139,125,8,68,57,240,115
        .byte 78,65,131,124,199,4,255,116,65,199,68,202,4,255,255,254
        .byte 255,137,4,202,73,139,44,199,72,137,108,202,8,131,192,1
        .byte 137,68,202,248,15,183,67,254,141,156,131,0,0,254,255,68
        .byte 139,116,36,4,68,139,60,36,139,3,15,182,204,15,182,232
        .byte 131,195,4,193,232,16,65,255,36,238,131,192,1,235,173,68
        .byte 41,240,59,69,28,119,216,68,107,248,24,68,3,125,20,65
```

```
        .byte 131,127,4,255,116,28,70,141,116,48,1,73,139,111,8,73
        .byte 139,7,72,137,44,202,72,137,68,202,8,68,137,116,202,248
        .byte 235,162,131,192,1,235,203

        .globl lj_BC_VARG
        .hidden lj_BC_VARG
        .type lj_BC_VARG, @function
        .size lj_BC_VARG, 191
lj_BC_VARG:
        .byte 15,182,236,15,182,192,68,137,60,36,68,141,124,194,11,141
        .byte 12,202,68,43,122,252,133,237,116,68,141,108,233,248,65,57
        .byte 215,115,23,73,139,71,248,65,131,199,8,72,137,1,131,193
        .byte 8,57,233,115,19,65,57,215,114,233,199,65,4,255,255,255
        .byte 255,131,193,8,57,233,114,242,68,139,60,36,139,3,15,182
        .byte 204,15,182,232,131,195,4,193,232,16,65,255,36,238,199,68
        .byte 36,4,1,0,0,0,137,208,68,41,248,118,219,137,197,193
        .byte 237,3,131,197,1,137,108,36,4,139,108,36,24,1,200,59
        .byte 69,32,119,21,73,139,71,248,65,131,199,8,72,137,1,131
        .byte 193,8,65,57,215,114,237,235,175,137,85,16,137,77,24,137
        .byte 92,36,28,65,41,215,139,116,36,4,131,238,1,137,239
        call lj_state_growstack
        .byte 139,85,16,139,77,24,65,1,215,235,197

        .globl lj_BC_ISNEXT
        .hidden lj_BC_ISNEXT
        .type lj_BC_ISNEXT, @function
        .size lj_BC_ISNEXT, 88
lj_BC_ISNEXT:
        .byte 131,124,202,236,247,117,65,139,108,202,232,131,124,202,244,244
        .byte 117,54,131,124,202,252,255,117,47,128,125,6,4,117,41,141
        .byte 156,131,0,0,254,255,199,68,202,248,0,0,0,0,199,68
        .byte 202,252,255,127,254,255,139,3,15,182,204,15,182,232,131,195
        .byte 4,193,232,16,65,255,36,238,198,67,252,88,141,156,131,0
        .byte 0,254,255,198,3,69,235,222

        .globl lj_BC_RETM
        .hidden lj_BC_RETM
        .type lj_BC_RETM, @function
        .size lj_BC_RETM, 4
lj_BC_RETM:
        .byte 3,68,36,4

        .globl lj_BC_RET
        .hidden lj_BC_RET
        .type lj_BC_RET, @function
        .size lj_BC_RET, 136
lj_BC_RET:
        .byte 193,225,3,139,90,252,137,68,36,4,247,195,3,0,0,0
        .byte 117,94,65,137,215,131,232,1,116,17,73,139,44,15,73,137
        .byte 111,248,65,131,199,8,131,232,1,117,239,139,68,36,4,15
        .byte 182,107,255,57,197,119,40,15,182,75,253,72,247,209,141,20
        .byte 202,68,139,122,248,69,139,127,16,69,139,127,208,139,3,15
        .byte 182,204,15,182,232,131,195,4,193,232,16,65,255,36,238,65
        .byte 199,71,252,255,255,255,255,65,131,199,8,131,192,1,235,195
        .byte 141,107,253,247,197,7,0,0,0,15,133,151,6,0,0,41
        .byte 234,1,233,233,123,255,255,255

        .globl lj_BC_RET0
        .hidden lj_BC_RET0
        .type lj_BC_RET0, @function
        .size lj_BC_RET0, 92
lj_BC_RET0:
        .byte 139,90,252,137,68,36,4,247,195,3,0,0,0,117,58,56
        .byte 67,255,119,40,15,182,75,253,72,247,209,141,20,202,68,139
        .byte 122,248,69,139,127,16,69,139,127,208,139,3,15,182,204,15
        .byte 182,232,131,195,4,193,232,16,65,255,36,238,199,68,194,244
        .byte 255,255,255,255,131,192,1,235,198,141,107,253,247,197,7,0
        .byte 0,0,15,133,54,6,0,0,41,234,235,164

        .globl lj_BC_RET1
        .hidden lj_BC_RET1
        .type lj_BC_RET1, @function
        .size lj_BC_RET1, 105
lj_BC_RET1:
        .byte 193,225,3,139,90,252,137,68,36,4,247,195,3,0,0,0
        .byte 117,66,72,139,44,10,72,137,106,248,56,67,255,119,40,15
```

```
        .byte 182,75,253,72,247,209,141,20,202,68,139,122,248,69,139,127
        .byte 16,69,139,127,208,139,3,15,182,204,15,182,232,131,195,4
        .byte 193,232,16,65,255,36,238,199,68,194,244,255,255,255,255,131
        .byte 192,1,235,198,141,107,253,247,197,7,0,0,0,15,133,207
        .byte 5,0,0,41,234,1,233,235,154

        .globl lj_BC_FORI
        .hidden lj_BC_FORI
        .type lj_BC_FORI, @function
        .size lj_BC_FORI, 167
lj_BC_FORI:
        .byte 141,12,202,129,121,4,255,255,254,255,117,91,129,121,12,255
        .byte 255,254,255,15,133,6,12,0,0,129,121,20,255,255,254,255
        .byte 15,133,249,11,0,0,139,41,131,121,16,0,124,40,59,105
        .byte 8,199,65,28,255,255,254,255,137,105,24,126,7,141,156,131
        .byte 0,0,254,255,139,3,15,182,204,15,182,232,131,195,4,193
        .byte 232,16,65,255,36,238,59,105,8,199,65,28,255,255,254,255
        .byte 137,105,24,125,223,235,214,15,131,178,11,0,0,129,121,12
        .byte 255,255,254,255,15,131,165,11,0,0,139,105,20,129,253,255
        .byte 255,254,255,15,131,150,11,0,0,242,15,16,1,242,15,16
        .byte 73,8,124,13,102,15,46,200,242,15,17,65,24,115,165,235
        .byte 156,102,15,46,193,235,241

        .globl lj_BC_JFORI
        .hidden lj_BC_JFORI
        .type lj_BC_JFORI, @function
        .size lj_BC_JFORI, 205
lj_BC_JFORI:
        .byte 141,12,202,129,121,4,255,255,254,255,117,114,129,121,12,255
        .byte 255,254,255,15,133,95,11,0,0,129,121,20,255,255,254,255
        .byte 15,133,82,11,0,0,139,41,131,121,16,0,124,48,59,105
        .byte 8,199,65,28,255,255,254,255,137,105,24,141,156,131,0,0
        .byte 254,255,15,183,67,254,15,142,151,2,0,0,139,3,15,182
        .byte 204,15,182,232,131,195,4,193,232,16,65,255,36,238,59,105
        .byte 8,199,65,28,255,255,254,255,137,105,24,141,156,131,0,0
        .byte 254,255,15,183,67,254,15,141,103,2,0,0,235,206,15,131
        .byte 244,10,0,0,129,121,12,255,255,254,255,15,131,231,10,0
        .byte 0,139,105,20,129,253,255,255,254,255,15,131,216,10,0,0
        .byte 242,15,16,1,242,15,16,73,8,124,28,102,15,46,200,242
        .byte 15,17,65,24,141,156,131,0,0,254,255,15,183,67,254,15
        .byte 131,30,2,0,0,235,133,102,15,46,193,235,226

        .globl lj_BC_FORL
        .hidden lj_BC_FORL
        .type lj_BC_FORL, @function
        .size lj_BC_FORL, 20
lj_BC_FORL:
        .byte 137,221,209,237,131,229,126,102,65,131,108,46,128,2,15,130
        .byte 66,30,0,0

        .globl lj_BC_IFORL
        .hidden lj_BC_IFORL
        .type lj_BC_IFORL, @function
        .size lj_BC_IFORL, 184
lj_BC_IFORL:
        .byte 141,12,202,129,121,4,255,255,254,255,117,102,129,121,12,255
        .byte 255,254,255,15,133,227,34,0,0,129,121,20,255,255,254,255
        .byte 15,133,214,34,0,0,139,105,16,133,237,120,46,3,41,112
        .byte 24,137,41,59,105,8,199,65,28,255,255,254,255,137,105,24
        .byte 127,7,141,156,131,0,0,254,255,139,3,15,182,204,15,182
        .byte 232,131,195,4,193,232,16,65,255,36,238,3,41,112,234,137
        .byte 41,59,105,8,199,65,28,255,255,254,255,137,105,24,124,217
        .byte 235,208,129,121,12,255,255,254,255,15,131,125,34,0,0,129
        .byte 121,20,255,255,254,255,15,131,112,34,0,0,139,105,20,242
        .byte 15,16,1,242,15,16,73,8,242,15,88,65,16,242,15,17
        .byte 1,133,237,120,13,102,15,46,200,242,15,17,65,24,114,153
        .byte 235,144,102,15,46,193,235,241

        .globl lj_BC_JFORL
        .hidden lj_BC_JFORL
        .type lj_BC_JFORL, @function
        .size lj_BC_JFORL, 189
lj_BC_JFORL:
        .byte 141,12,202,129,121,4,255,255,254,255,117,103,129,121,12,255
        .byte 255,254,255,15,133,43,34,0,0,129,121,20,255,255,254,255
        .byte 15,133,30,34,0,0,139,105,16,133,237,120,43,3,41,112
```

```
        .byte 21,137,41,59,105,8,199,65,28,255,255,254,255,137,105,24
        .byte 15,142,4,1,0,0,139,3,15,182,204,15,182,232,131,195
        .byte 4,193,232,16,65,255,36,238,3,41,112,234,137,41,59,105
        .byte 8,199,65,28,255,255,254,255,137,105,24,15,141,217,0,0
        .byte 0,235,211,129,121,12,255,255,254,255,15,131,196,33,0,0
        .byte 129,121,20,255,255,254,255,15,131,183,33,0,0,139,105,20
        .byte 242,15,16,1,242,15,16,73,8,242,15,88,65,16,242,15
        .byte 17,1,133,237,120,17,102,15,46,200,242,15,17,65,24,15
        .byte 131,149,0,0,0,235,143,102,15,46,193,235,237

        .globl lj_BC_ITERL
        .hidden lj_BC_ITERL
        .type lj_BC_ITERL, @function
        .size lj_BC_ITERL, 20
lj_BC_ITERL:
        .byte 137,221,209,237,131,229,126,102,65,131,108,46,128,2,15,130
         .byte 185,28,0,0

         .globl lj_BC_IITERL
         .hidden lj_BC_IITERL
         .type lj_BC_IITERL, @function
         .size lj_BC_IITERL, 44
 lj_BC_IITERL:
         .byte 141,12,202,139,105,4,131,253,255,116,15,141,156,131,0,0
         .byte 254,255,139,1,137,105,252,137,65,248,139,3,15,182,204,15
         .byte 182,232,131,195,4,193,232,16,65,255,36,238

         .globl lj_BC_JITERL
         .hidden lj_BC_JITERL
         .type lj_BC_JITERL, @function
         .size lj_BC_JITERL, 39
 lj_BC_JITERL:
         .byte 141,12,202,139,105,4,131,253,255,116,10,137,105,252,139,41
         .byte 137,105,248,235,56,139,3,15,182,204,15,182,232,131,195,4
         .byte 193,232,16,65,255,36,238

         .globl lj_BC_LOOP
         .hidden lj_BC_LOOP
         .type lj_BC_LOOP, @function
         .size lj_BC_LOOP, 20
 lj_BC_LOOP:
         .byte 137,221,209,237,131,229,126,102,65,131,108,46,128,2,15,130
         .byte 82,28,0,0

         .globl lj_BC_ILOOP
         .hidden lj_BC_ILOOP
         .type lj_BC_ILOOP, @function
         .size lj_BC_ILOOP, 18
 lj_BC_ILOOP:
         .byte 139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,255
         .byte 36,238

         .globl lj_BC_JLOOP
         .hidden lj_BC_JLOOP
         .type lj_BC_JLOOP, @function
         .size lj_BC_JLOOP, 47
 lj_BC_JLOOP:
         .byte 65,139,142,240,246,255,255,139,4,129,72,139,64,64,139,108
         .byte 36,24,65,137,150,28,245,255,255,65,137,174,136,244,255,255
         .byte 76,137,36,36,76,137,108,36,8,72,131,236,16,255,224

         .globl lj_BC_JMP
         .hidden lj_BC_JMP
         .type lj_BC_JMP, @function
         .size lj_BC_JMP, 25
 lj_BC_JMP:
         .byte 141,156,131,0,0,254,255,139,3,15,182,204,15,182,232,131
         .byte 195,4,193,232,16,65,255,36,238

         .globl lj_BC_FUNCF
         .hidden lj_BC_FUNCF
         .type lj_BC_FUNCF, @function
         .size lj_BC_FUNCF, 20
 lj_BC_FUNCF:
         .byte 137,221,209,237,131,229,126,102,65,131,108,46,128,1,15,130
         .byte 28,28,0,0
```

```
1060
1061         .globl lj_BC_IFUNCF
1062         .hidden lj_BC_IFUNCF
1063         .type lj_BC_IFUNCF, @function
1064         .size lj_BC_IFUNCF, 63
1065 lj_BC_IFUNCF:
1066         .byte 68,139,123,204,139,108,36,24,141,12,202,59,77,32,15,135
1067         .byte 223,2,0,0,15,182,75,194,57,200,118,18,139,3,15,182
1068         .byte 204,15,182,232,131,195,4,193,232,16,65,255,36,238,199,68
1069         .byte 194,252,255,255,255,255,131,192,1,57,200,118,241,235,221
1070
1071         .globl lj_BC_JFUNCF
1072         .hidden lj_BC_JFUNCF
1073         .type lj_BC_JFUNCF, @function
1074         .size lj_BC_JFUNCF, 54
1075 lj_BC_JFUNCF:
1076         .byte 68,139,123,204,139,108,36,24,141,12,202,59,77,32,15,135
1077         .byte 160,2,0,0,15,182,75,194,57,200,118,9,15,183,67,254
1078         .byte 233,64,255,255,255,199,68,194,252,255,255,255,255,131,192,1
1079         .byte 57,200,118,241,235,230
1080
1081         .globl lj_BC_FUNCV
1082         .hidden lj_BC_FUNCV
1083         .type lj_BC_FUNCV, @function
1084         .size lj_BC_FUNCV, 0
1085 lj_BC_FUNCV:
1086
1087         .globl lj_BC_IFUNCV
1088         .hidden lj_BC_IFUNCV
1089         .type lj_BC_IFUNCV, @function
1090         .size lj_BC_IFUNCV, 125
1091 lj_BC_IFUNCV:
1092         .byte 141,44,197,3,0,0,0,141,4,194,68,139,122,248,137,104
1093         .byte 252,68,137,120,248,139,108,36,24,141,12,200,59,77,32,15
1094         .byte 135,84,2,0,0,137,209,137,194,15,182,107,194,133,237,116
1095         .byte 37,131,193,8,57,209,115,52,68,139,121,248,68,137,56,68
1096         .byte 139,121,252,68,137,120,4,131,192,8,199,65,252,255,255,255
1097         .byte 255,131,237,1,117,219,68,139,123,204,139,3,15,182,204,15
1098         .byte 182,232,131,195,4,193,232,16,65,255,36,238,199,64,4,255
1099         .byte 255,255,255,131,192,8,131,237,1,117,241,235,217
1100
1101         .globl lj_BC_JFUNCV
1102         .hidden lj_BC_JFUNCV
1103         .type lj_BC_JFUNCV, @function
1104         .size lj_BC_JFUNCV, 1
1105 lj_BC_JFUNCV:
1106         .byte 204
1107
1108         .globl lj_BC_FUNCC
1109         .hidden lj_BC_FUNCC
1110         .type lj_BC_FUNCC, @function
1111         .size lj_BC_FUNCC, 89
1112 lj_BC_FUNCC:
1113         .byte 139,106,248,76,139,125,24,139,108,36,24,141,68,194,248,137
1114         .byte 85,16,141,136,160,0,0,0,59,77,32,137,69,24,137,239
1115         .byte 15,135,206,1,0,0,65,199,134,120,244,255,255,254,255,255
1116         .byte 255,65,255,215,139,85,16,65,137,174,24,245,255,255,65,199
1117         .byte 134,120,244,255,255,255,255,255,255,141,12,194,247,217,3,77
1118         .byte 24,139,90,252,233,126,0,0,0
1119
1120         .globl lj_BC_FUNCCW
1121         .hidden lj_BC_FUNCCW
1122         .type lj_BC_FUNCCW, @function
1123         .size lj_BC_FUNCCW, 93
1124 lj_BC_FUNCCW:
1125         .byte 139,106,248,76,139,125,24,139,108,36,24,141,68,194,248,137
1126         .byte 85,16,141,136,160,0,0,0,59,77,32,137,69,24,76,137
1127         .byte 254,137,239,15,135,114,1,0,0,65,199,134,120,244,255,255
1128         .byte 254,255,255,255,65,255,150,0,245,255,255,139,85,16,65,137
1129         .byte 174,24,245,255,255,65,199,134,120,244,255,255,255,255,255,255
1130         .byte 141,12,194,247,217,3,77,24,139,90,252,235,33
1131
1132         .globl lj_vm_returnp
1133         .hidden lj_vm_returnp
1134         .type lj_vm_returnp, @function
1135         .size lj_vm_returnp, 33
```

```
lj_vm_returnp:
        .byte 247,195,4,0,0,0,15,132,245,2,0,0,131,227,248,41
        .byte 218,72,141,76,25,248,139,90,252,199,68,10,4,253,255,255
        .byte 255

        .globl lj_vm_returnc
        .hidden lj_vm_returnc
        .type lj_vm_returnc, @function
        .size lj_vm_returnc, 25
lj_vm_returnc:
        .byte 131,192,1,15,132,167,0,0,0,137,68,36,4,247,195,3
        .byte 0,0,0,15,132,252,248,255,255

        .globl lj_vm_return
        .hidden lj_vm_return
        .type lj_vm_return, @function
        .size lj_vm_return, 75
lj_vm_return:
        .byte 131,243,1,247,195,3,0,0,0,117,187,65,199,134,120,244
        .byte 255,255,254,255,255,255,131,227,248,41,211,247,219,131,232,1
        .byte 116,16,72,139,44,10,72,137,106,248,131,194,8,131,232,1
        .byte 117,240,139,108,36,24,137,93,16,139,68,36,4,139,76,36
        .byte 16,57,193,117,28,131,234,8,137,85,24

        .globl lj_vm_leave_cp
        .hidden lj_vm_leave_cp
        .type lj_vm_leave_cp, @function
        .size lj_vm_leave_cp, 11
lj_vm_leave_cp:
        .byte 72,139,76,36,32,72,137,77,48,49,192

        .globl lj_vm_leave_unw
        .hidden lj_vm_leave_unw
        .type lj_vm_leave_unw, @function
        .size lj_vm_leave_unw, 65
lj_vm_leave_unw:
        .byte 72,131,196,40,65,94,65,95,91,93,195,114,20,59,85,32
        .byte 119,26,199,66,252,255,255,255,255,131,194,8,131,192,1,235
        .byte 202,133,201,116,202,41,193,141,20,202,235,195,137,85,24,137
        .byte 68,36,4,137,206,137,239
        call lj_state_growstack
        .byte 139,85,24,235,162

        .globl lj_vm_unwind_yield
        .hidden lj_vm_unwind_yield
        .type lj_vm_unwind_yield, @function
        .size lj_vm_unwind_yield, 4
lj_vm_unwind_yield:
        .byte 176,1,235,5

        .globl lj_vm_unwind_c
        .hidden lj_vm_unwind_c
        .type lj_vm_unwind_c, @function
        .size lj_vm_unwind_c, 5
lj_vm_unwind_c:
        .byte 137,240,72,137,252

        .globl lj_vm_unwind_c_eh
        .hidden lj_vm_unwind_c_eh
        .type lj_vm_unwind_c_eh, @function
        .size lj_vm_unwind_c_eh, 16
lj_vm_unwind_c_eh:
        .byte 139,108,36,24,139,109,8,199,69,88,254,255,255,255,235,166

        .globl lj_vm_unwind_rethrow
        .hidden lj_vm_unwind_rethrow
        .type lj_vm_unwind_rethrow, @function
        .size lj_vm_unwind_rethrow, 21
lj_vm_unwind_rethrow:
        .byte 139,124,36,24,137,198,72,131,196,40,65,94,65,95,91,93
        jmp lj_err_throw

        .globl lj_vm_unwind_ff
        .hidden lj_vm_unwind_ff
        .type lj_vm_unwind_ff, @function
        .size lj_vm_unwind_ff, 7
```

```
lj_vm_unwind_ff:
        .byte 72,131,231,252,72,137,252

        .globl lj_vm_unwind_ff_eh
        .hidden lj_vm_unwind_ff_eh
        .type lj_vm_unwind_ff_eh, @function
        .size lj_vm_unwind_ff_eh, 56
lj_vm_unwind_ff_eh:
        .byte 139,108,36,24,72,199,193,248,255,255,255,184,2,0,0,0
        .byte 139,85,16,68,139,117,8,65,129,198,224,11,0,0,139,90
        .byte 252,199,66,252,254,255,255,255,65,199,134,120,244,255,255,255
        .byte 255,255,255,233,227,254,255,255

        .globl lj_vm_growstack_c
        .hidden lj_vm_growstack_c
        .type lj_vm_growstack_c, @function
        .size lj_vm_growstack_c, 7
lj_vm_growstack_c:
        .byte 190,20,0,0,0,235,28

        .globl lj_vm_growstack_v
        .hidden lj_vm_growstack_v
        .type lj_vm_growstack_v, @function
        .size lj_vm_growstack_v, 5
lj_vm_growstack_v:
        .byte 131,232,8,235,4

        .globl lj_vm_growstack_f
        .hidden lj_vm_growstack_f
        .type lj_vm_growstack_f, @function
        .size lj_vm_growstack_f, 65
lj_vm_growstack_f:
        .byte 141,68,194,248,15,182,75,195,131,195,4,137,85,16,137,69
        .byte 24,137,92,36,28,137,206,137,239
        call lj_state_growstack
        .byte 139,85,16,139,69,24,139,106,248,41,208,193,232,3,131,192
        .byte 1,139,93,16,139,11,15,182,233,15,182,205,131,195,4,65
        .byte 255,36,238

        .globl lj_vm_resume
        .hidden lj_vm_resume
        .type lj_vm_resume, @function
        .size lj_vm_resume, 132
lj_vm_resume:
        .byte 85,83,65,87,65,86,72,131,236,40,137,253,137,124,36,24
        .byte 137,241,187,5,0,0,0,49,192,76,141,124,36,1,68,139
        .byte 117,8,65,129,198,224,11,0,0,137,68,36,28,72,137,68
        .byte 36,32,137,68,36,16,137,68,36,20,76,137,125,48,56,69
        .byte 7,15,132,137,0,0,0,65,137,174,24,245,255,255,65,199
        .byte 134,120,244,255,255,255,255,255,255,136,69,7,139,85,16,139
        .byte 69,24,41,200,193,232,3,131,192,1,41,209,139,90,252,137
        .byte 68,36,4,247,195,3,0,0,0,15,132,44,247,255,255,233
        .byte 43,254,255,255

        .globl lj_vm_pcall
        .hidden lj_vm_pcall
        .type lj_vm_pcall, @function
        .size lj_vm_pcall, 21
lj_vm_pcall:
        .byte 85,83,65,87,65,86,72,131,236,40,187,5,0,0,0,137
        .byte 76,36,20,235,15

        .globl lj_vm_call
        .hidden lj_vm_call
        .type lj_vm_call, @function
        .size lj_vm_call, 91
lj_vm_call:
        .byte 85,83,65,87,65,86,72,131,236,40,187,1,0,0,0,137
        .byte 84,36,16,137,253,137,124,36,24,137,241,68,139,117,8,76
        .byte 139,125,48,76,137,124,36,32,137,108,36,28,65,129,198,224
        .byte 11,0,0,72,137,101,48,65,137,174,24,245,255,255,65,199
        .byte 134,120,244,255,255,255,255,255,255,139,85,16,1,203,41,211
        .byte 139,69,24,41,200,193,232,3,131,192,1

        .globl lj_vm_call_dispatch
        .hidden lj_vm_call_dispatch
```

```
        .type lj_vm_call_dispatch, @function
        .size lj_vm_call_dispatch, 13
lj_vm_call_dispatch:
        .byte 139,105,248,131,121,252,247,15,133,179,3,0,0

        .globl lj_vm_call_dispatch_f
        .hidden lj_vm_call_dispatch_f
        .type lj_vm_call_dispatch_f, @function
        .size lj_vm_call_dispatch_f, 23
lj_vm_call_dispatch_f:
        .byte 137,202,137,90,252,139,93,16,139,11,15,182,233,15,182,205
        .byte 131,195,4,65,255,36,238

        .globl lj_vm_cpcall
        .hidden lj_vm_cpcall
        .type lj_vm_cpcall, @function
        .size lj_vm_cpcall, 94
lj_vm_cpcall:
        .byte 85,83,65,87,65,86,72,131,236,40,137,253,137,124,36,24
        .byte 137,108,36,28,68,139,125,36,68,43,125,24,68,139,117,8
        .byte 199,68,36,20,0,0,0,0,68,137,124,36,16,65,129,198
        .byte 224,11,0,0,76,139,125,48,76,137,124,36,32,72,137,101
        .byte 48,65,137,174,24,245,255,255,255,209,133,192,15,132,144,253
        .byte 255,255,137,193,187,5,0,0,0,233,90,255,255,255

        .globl lj_cont_dispatch
        .hidden lj_cont_dispatch
        .type lj_cont_dispatch, @function
        .size lj_cont_dispatch, 74
lj_cont_dispatch:
        .byte 1,209,131,227,248,137,213,41,218,199,68,193,252,255,255,255
        .byte 255,137,200,139,93,244,72,99,77,240,131,249,1,118,24,76
        .byte 141,61,31,223,255,255,76,1,249,68,139,122,248,69,139,127
        .byte 16,69,139,127,208,255,225,15,132,108,28,0,0,41,213,193
        .byte 237,3,141,69,255,233,9,22,0,0

        .globl lj_cont_cat
        .hidden lj_cont_cat
        .type lj_cont_cat, @function
        .size lj_cont_cat, 46
lj_cont_cat:
        .byte 15,182,75,255,131,237,16,141,12,202,41,233,15,132,134,0
        .byte 0,0,247,217,193,233,3,139,124,36,24,137,87,16,137,202
        .byte 72,139,8,72,137,77,0,137,238,233,206,234,255,255

        .globl lj_vmeta_tgets
        .hidden lj_vmeta_tgets
        .type lj_vmeta_tgets, @function
        .size lj_vmeta_tgets, 41
lj_vmeta_tgets:
        .byte 137,4,36,199,68,36,4,251,255,255,255,72,141,4,36,128
        .byte 123,252,54,117,48,65,141,142,176,244,255,255,137,41,199,65
        .byte 4,244,255,255,255,137,205,235,35

        .globl lj_vmeta_tgetb
        .hidden lj_vmeta_tgetb
        .type lj_vmeta_tgetb, @function
        .size lj_vmeta_tgetb, 21
lj_vmeta_tgetb:
        .byte 15,182,67,254,199,68,36,4,255,255,254,255,137,4,36,72
        .byte 141,4,36,235,7

        .globl lj_vmeta_tgetv
        .hidden lj_vmeta_tgetv
        .type lj_vmeta_tgetv, @function
        .size lj_vmeta_tgetv, 44
lj_vmeta_tgetv:
        .byte 15,182,67,254,141,4,194,15,182,107,255,141,44,234,139,124
        .byte 36,24,137,87,16,137,238,72,137,194,137,253,137,92,36,28
        call lj_meta_tget
        .byte 139,85,16,133,192,116,29

        .globl lj_cont_ra
        .hidden lj_cont_ra
        .type lj_cont_ra, @function
        .size lj_cont_ra, 53
```

```
lj_cont_ra:
        .byte 15,182,75,253,72,139,40,72,137,44,202,139,3,15,182,204
        .byte 15,182,232,131,195,4,193,232,16,65,255,36,238,139,77,24
        .byte 137,89,244,141,89,2,41,211,139,105,248,184,3,0,0,0
        .byte 233,116,254,255,255

        .globl lj_vmeta_tgetr
        .hidden lj_vmeta_tgetr
        .type lj_vmeta_tgetr, @function
        .size lj_vmeta_tgetr, 38
lj_vmeta_tgetr:
        .byte 137,239,137,213,137,198
        call lj_tab_getinth
        .byte 15,182,75,253,137,234,133,192,15,133,7,239,255,255,199,68
        .byte 202,4,255,255,255,255,233,1,239,255,255

        .globl lj_vmeta_tsets
        .hidden lj_vmeta_tsets
        .type lj_vmeta_tsets, @function
        .size lj_vmeta_tsets, 41
lj_vmeta_tsets:
        .byte 137,4,36,199,68,36,4,251,255,255,255,72,141,4,36,128
        .byte 123,252,55,117,48,65,141,142,176,244,255,255,137,41,199,65
        .byte 4,244,255,255,255,137,205,235,35

        .globl lj_vmeta_tsetb
        .hidden lj_vmeta_tsetb
        .type lj_vmeta_tsetb, @function
        .size lj_vmeta_tsetb, 21
lj_vmeta_tsetb:
        .byte 15,182,67,254,199,68,36,4,255,255,254,255,137,4,36,72
        .byte 141,4,36,235,7

        .globl lj_vmeta_tsetv
        .hidden lj_vmeta_tsetv
        .type lj_vmeta_tsetv, @function
        .size lj_vmeta_tsetv, 55
lj_vmeta_tsetv:
        .byte 15,182,67,254,141,4,194,15,182,107,255,141,44,234,139,124
        .byte 36,24,137,87,16,137,238,72,137,194,137,253,137,92,36,28
        call lj_meta_tset
        .byte 139,85,16,133,192,116,29,15,182,75,253,72,139,44,202,72
        .byte 137,40

        .globl lj_cont_nop
        .hidden lj_cont_nop
        .type lj_cont_nop, @function
        .size lj_cont_nop, 54
lj_cont_nop:
        .byte 139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,255
        .byte 36,238,139,77,24,137,89,244,15,182,67,253,72,139,44,194
        .byte 72,137,105,16,141,89,2,41,211,139,105,248,184,4,0,0
        .byte 0,233,163,253,255,255

        .globl lj_vmeta_tsetr
        .hidden lj_vmeta_tsetr
        .type lj_vmeta_tsetr, @function
        .size lj_vmeta_tsetr, 33
lj_vmeta_tsetr:
        .byte 139,124,36,24,137,238,137,87,16,137,213,137,194,137,92,36
        .byte 28
        call lj_tab_setinth
        .byte 15,182,75,253,137,234,233,240,240,255,255

        .globl lj_vmeta_comp
        .hidden lj_vmeta_comp
        .type lj_vmeta_comp, @function
        .size lj_vmeta_comp, 74
lj_vmeta_comp:
        .byte 139,108,36,24,137,85,16,141,52,202,141,20,194,137,239,15
        .byte 182,75,252,137,92,36,28
        call lj_meta_comp
        .byte 139,85,16,131,248,1,15,135,219,0,0,0,141,91,4,114
        .byte 11,15,183,67,254,141,156,131,0,0,254,255,139,3,15,182
        .byte 204,15,182,232,131,195,4,193,232,16,65,255,36,238

```

```
        .globl lj_cont_condt
        .hidden lj_cont_condt
        .type lj_cont_condt, @function
        .size lj_cont_condt, 11
lj_cont_condt:
        .byte 131,195,4,131,120,4,254,114,218,235,227

        .globl lj_cont_condf
        .hidden lj_cont_condf
        .type lj_cont_condf, @function
        .size lj_cont_condf, 6
lj_cont_condf:
        .byte 131,120,4,254,235,205

        .globl lj_vmeta_equal
        .hidden lj_vmeta_equal
        .type lj_vmeta_equal, @function
        .size lj_vmeta_equal, 29
lj_vmeta_equal:
        .byte 131,235,4,137,206,137,233,139,108,36,24,137,85,16,137,194
        .byte 137,239,137,92,36,28
        call lj_meta_equal
        .byte 235,164

        .globl lj_vmeta_equal_cd
        .hidden lj_vmeta_equal_cd
        .type lj_vmeta_equal_cd, @function
        .size lj_vmeta_equal_cd, 26
lj_vmeta_equal_cd:
        .byte 131,235,4,139,108,36,24,137,85,16,137,239,139,115,252,137
        .byte 92,36,28
        call lj_meta_equal_cd
        .byte 235,138

        .globl lj_vmeta_istype
        .hidden lj_vmeta_istype
        .type lj_vmeta_istype, @function
        .size lj_vmeta_istype, 29
lj_vmeta_istype:
        .byte 139,108,36,24,137,85,16,137,206,15,183,83,254,137,239,137
        .byte 92,36,28
        call lj_meta_istype
        .byte 139,85,16,235,137

        .globl lj_vmeta_arith_vno
        .hidden lj_vmeta_arith_vno
        .type lj_vmeta_arith_vno, @function
        .size lj_vmeta_arith_vno, 4
lj_vmeta_arith_vno:
        .byte 15,182,107,255

        .globl lj_vmeta_arith_vn
        .hidden lj_vmeta_arith_vn
        .type lj_vmeta_arith_vn, @function
        .size lj_vmeta_arith_vn, 6
lj_vmeta_arith_vn:
        .byte 65,141,4,199,235,28

        .globl lj_vmeta_arith_nvo
        .hidden lj_vmeta_arith_nvo
        .type lj_vmeta_arith_nvo, @function
        .size lj_vmeta_arith_nvo, 4
lj_vmeta_arith_nvo:
        .byte 15,182,67,254

        .globl lj_vmeta_arith_nv
        .hidden lj_vmeta_arith_nv
        .type lj_vmeta_arith_nv, @function
        .size lj_vmeta_arith_nv, 10
lj_vmeta_arith_nv:
        .byte 65,141,4,199,141,44,234,149,235,17

        .globl lj_vmeta_unm
        .hidden lj_vmeta_unm
        .type lj_vmeta_unm, @function
        .size lj_vmeta_unm, 7
```

```
lj_vmeta_unm:
        .byte 141,4,194,137,197,235,10

        .globl lj_vmeta_arith_vvo
        .hidden lj_vmeta_arith_vvo
        .type lj_vmeta_arith_vvo, @function
        .size lj_vmeta_arith_vvo, 4
lj_vmeta_arith_vvo:
        .byte 15,182,107,255

        .globl lj_vmeta_arith_vv
        .hidden lj_vmeta_arith_vv
        .type lj_vmeta_arith_vv, @function
        .size lj_vmeta_arith_vv, 49
lj_vmeta_arith_vv:
        .byte 141,4,194,141,44,234,141,12,202,68,15,182,67,252,137,206
        .byte 137,193,139,124,36,24,137,87,16,137,234,137,253,137,92,36
        .byte 28
        call lj_meta_arith
        .byte 139,85,16,133,192,15,132,166,254,255,255

        .globl lj_vmeta_binop
        .hidden lj_vmeta_binop
        .type lj_vmeta_binop, @function
        .size lj_vmeta_binop, 20
lj_vmeta_binop:
        .byte 137,193,41,208,137,89,244,141,88,2,184,3,0,0,0,233
        .byte 94,252,255,255

        .globl lj_vmeta_len
        .hidden lj_vmeta_len
        .type lj_vmeta_len, @function
        .size lj_vmeta_len, 26
lj_vmeta_len:
        .byte 139,108,36,24,137,85,16,141,52,194,137,239,137,92,36,28
        call lj_meta_len
        .byte 139,85,16,235,210

        .globl lj_vmeta_call_ra
        .hidden lj_vmeta_call_ra
        .type lj_vmeta_call_ra, @function
        .size lj_vmeta_call_ra, 4
lj_vmeta_call_ra:
        .byte 141,76,202,8

        .globl lj_vmeta_call
        .hidden lj_vmeta_call
        .type lj_vmeta_call, @function
        .size lj_vmeta_call, 81
lj_vmeta_call:
        .byte 137,76,36,4,137,4,36,131,233,8,139,108,36,24,137,85
        .byte 16,137,206,141,20,193,137,239,137,92,36,28
        call lj_meta_call
        .byte 139,85,16,139,76,36,4,139,4,36,139,105,248,131,192,1
        .byte 65,57,215,15,132,37,240,255,255,137,202,137,90,252,139,93
        .byte 16,139,11,15,182,233,15,182,205,131,195,4,65,255,36,238

        .globl lj_vmeta_for
        .hidden lj_vmeta_for
        .type lj_vmeta_for, @function
        .size lj_vmeta_for, 43
lj_vmeta_for:
        .byte 139,108,36,24,137,85,16,137,206,137,239,137,92,36,28
        call lj_meta_for
        .byte 139,85,16,139,67,252,15,182,204,15,182,232,193,232,16,65
        .byte 255,164,238,208,4,0,0

        .globl lj_ff_assert
        .hidden lj_ff_assert
        .type lj_ff_assert, @function
        .size lj_ff_assert, 67
lj_ff_assert:
        .byte 131,248,2,15,130,57,18,0,0,139,106,4,131,253,254,15
        .byte 131,45,18,0,0,139,90,252,137,68,36,4,137,106,252,139
        .byte 42,137,106,248,131,232,2,116,17,137,209,131,193,8,72,139
        .byte 41,72,137,105,248,131,232,1,117,241,139,68,36,4,233,145
```

```
        .byte 6,0,0

        .globl lj_ff_type
        .hidden lj_ff_type
        .type lj_ff_type, @function
        .size lj_ff_type, 66
lj_ff_type:
        .byte 131,248,2,15,130,246,17,0,0,139,106,4,137,233,193,249
        .byte 15,131,249,254,116,37,184,13,0,0,0,247,213,57,232,15
        .byte 71,197,139,106,248,139,68,197,32,139,90,252,199,66,252,251
        .byte 255,255,255,137,66,248,233,77,6,0,0,184,3,0,0,0
        .byte 235,224

        .globl lj_ff_getmetatable
        .hidden lj_ff_getmetatable
        .type lj_ff_getmetatable, @function
        .size lj_ff_getmetatable, 162
lj_ff_getmetatable:
        .byte 131,248,2,15,130,180,17,0,0,139,106,4,139,90,252,131
        .byte 253,244,117,97,139,42,139,109,16,133,237,199,66,252,255,255
        .byte 255,255,15,132,30,6,0,0,65,139,134,108,245,255,255,199
        .byte 66,252,244,255,255,255,137,106,248,139,77,28,35,72,8,107
        .byte 201,24,3,77,20,131,121,12,251,117,5,57,65,8,116,12
        .byte 139,73,16,133,201,117,238,233,234,5,0,0,139,105,4,131
        .byte 253,255,15,132,222,5,0,0,139,1,137,106,252,137,66,248
        .byte 233,209,5,0,0,131,253,243,116,154,131,253,242,119,20,129
        .byte 253,255,255,254,255,118,7,189,252,255,255,255,235,5,189,242
        .byte 255,255,255,247,213,65,139,172,174,128,245,255,255,233,119,255
        .byte 255,255

        .globl lj_ff_setmetatable
        .hidden lj_ff_setmetatable
        .type lj_ff_setmetatable, @function
        .size lj_ff_setmetatable, 92
lj_ff_setmetatable:
        .byte 131,248,3,15,130,18,17,0,0,131,122,4,244,15,133,8
        .byte 17,0,0,139,42,131,125,16,0,15,133,252,16,0,0,131
        .byte 122,12,244,15,133,242,16,0,0,139,66,8,137,69,16,139
        .byte 90,252,199,66,252,244,255,255,255,137,106,248,246,69,4,4
        .byte 116,21,128,101,4,251,65,139,134,92,244,255,255,65,137,174
        .byte 92,244,255,255,137,69,12,233,72,5,0,0

        .globl lj_ff_rawget
        .hidden lj_ff_rawget
        .type lj_ff_rawget, @function
        .size lj_ff_rawget, 52
lj_ff_rawget:
        .byte 131,248,3,15,130,182,16,0,0,131,122,4,244,15,133,172
        .byte 16,0,0,137,213,139,50,141,82,8,139,124,36,24
        call lj_tab_get
        .byte 137,234,72,139,40,139,90,252,72,137,106,248,233,20,5,0
        .byte 0

        .globl lj_ff_tonumber
        .hidden lj_ff_tonumber
        .type lj_ff_tonumber, @function
        .size lj_ff_tonumber, 40
lj_ff_tonumber:
        .byte 131,248,2,15,133,130,16,0,0,129,122,4,255,255,254,255
        .byte 117,7,139,42,233,181,4,0,0,15,135,108,16,0,0,242
        .byte 15,16,2,233,228,4,0,0

        .globl lj_ff_tostring
        .hidden lj_ff_tostring
        .type lj_ff_tostring, @function
        .size lj_ff_tostring, 108
lj_ff_tostring:
        .byte 131,248,2,15,130,90,16,0,0,139,90,252,131,122,4,251
        .byte 117,17,139,2,199,66,252,251,255,255,255,137,66,248,233,201
        .byte 4,0,0,129,122,4,255,255,254,255,15,135,51,16,0,0
        .byte 65,131,190,180,245,255,255,0,15,133,37,16,0,0,65,139
        .byte 174,64,244,255,255,65,59,174,68,244,255,255,114,5,232,159
        .byte 16,0,0,139,108,36,24,137,85,16,137,92,36,28,137,214
        .byte 137,239
        call lj_strfmt_number
        .byte 139,85,16,235,168
```

```
1668
1669         .globl lj_ff_next
1670         .hidden lj_ff_next
1671         .type lj_ff_next, @function
1672         .size lj_ff_next, 72
1673 lj_ff_next:
1674         .byte 131,248,2,15,130,238,15,0,0,116,71,131,122,4,244,15
1675         .byte 133,226,15,0,0,139,108,36,24,137,85,16,137,85,24,139
1676         .byte 90,252,139,50,141,82,8,137,239,137,92,36,28
1677         call lj_tab_next
1678         .byte 139,85,16,133,192,116,34,72,139,106,8,72,139,66,16,72
1679         .byte 137,106,248,72,137,2
1680
1681         .globl lj_fff_res2
1682         .hidden lj_fff_res2
1683         .type lj_fff_res2, @function
1684         .size lj_fff_res2, 31
1685 lj_fff_res2:
1686         .byte 184,3,0,0,0,233,51,4,0,0,199,66,12,255,255,255
1687         .byte 255,235,176,199,66,252,255,255,255,255,233,25,4,0,0
1688
1689         .globl lj_ff_pairs
1690         .hidden lj_ff_pairs
1691         .type lj_ff_pairs, @function
1692         .size lj_ff_pairs, 57
1693 lj_ff_pairs:
1694         .byte 131,248,2,15,130,135,15,0,0,139,42,131,122,4,244,15
1695         .byte 133,123,15,0,0,139,106,248,139,69,32,139,90,252,199,66
1696         .byte 252,247,255,255,255,137,66,248,199,66,12,255,255,255,255,184
1697         .byte 4,0,0,0,233,229,3,0,0
1698
1699         .globl lj_ff_ipairs_aux
1700         .hidden lj_ff_ipairs_aux
1701         .type lj_ff_ipairs_aux, @function
1702         .size lj_ff_ipairs_aux, 104
1703 lj_ff_ipairs_aux:
1704         .byte 131,248,3,15,130,78,15,0,0,131,122,4,244,15,133,68
1705         .byte 15,0,0,129,122,12,255,255,254,255,15,133,55,15,0,0
1706         .byte 139,90,252,139,66,8,131,192,1,199,66,252,255,255,254,255
1707         .byte 137,66,248,139,42,59,69,24,115,23,193,224,3,3,69,8
1708         .byte 131,120,4,255,116,34,72,139,40,72,137,42,233,87,255,255
1709         .byte 255,131,125,28,0,116,17,137,239,137,213,137,198
1710         call lj_tab_getinth
1711         .byte 137,234,133,192,117,216
1712
1713         .globl lj_fff_res0
1714         .hidden lj_fff_res0
1715         .type lj_fff_res0, @function
1716         .size lj_fff_res0, 10
1717 lj_fff_res0:
1718         .byte 184,1,0,0,0,233,115,3,0,0
1719
1720         .globl lj_ff_ipairs
1721         .hidden lj_ff_ipairs
1722         .type lj_ff_ipairs, @function
1723         .size lj_ff_ipairs, 64
1724 lj_ff_ipairs:
1725         .byte 131,248,2,15,130,220,14,0,0,139,42,131,122,4,244,15
1726         .byte 133,208,14,0,0,139,106,248,139,69,32,139,90,252,199,66
1727         .byte 252,247,255,255,255,137,66,248,199,66,12,255,255,254,255,199
1728         .byte 66,8,0,0,0,0,184,4,0,0,0,233,51,3,0,0
1729
1730         .globl lj_ff_pcall
1731         .hidden lj_ff_pcall
1732         .type lj_ff_pcall, @function
1733         .size lj_ff_pcall, 41
1734 lj_ff_pcall:
1735         .byte 131,248,2,15,130,156,14,0,0,141,74,8,131,232,1,187
1736         .byte 14,0,0,0,65,15,182,174,157,244,255,255,193,237,4,131
1737         .byte 229,1,1,235,233,254,247,255,255
1738
1739         .globl lj_ff_xpcall
1740         .hidden lj_ff_xpcall
1741         .type lj_ff_xpcall, @function
1742         .size lj_ff_xpcall, 55
1743 lj_ff_xpcall:
```

```
        .byte 131,248,3,15,130,115,14,0,0,131,122,12,247,15,133,105
        .byte 14,0,0,139,106,4,137,106,12,199,66,4,247,255,255,255
        .byte 139,42,139,90,8,137,106,8,137,26,141,74,16,131,232,2
        .byte 187,22,0,0,0,235,180

        .globl lj_ff_coroutine_resume
        .hidden lj_ff_coroutine_resume
        .type lj_ff_coroutine_resume, @function
        .size lj_ff_coroutine_resume, 310
lj_ff_coroutine_resume:
        .byte 131,248,2,15,130,60,14,0,0,139,42,139,90,252,137,92
        .byte 36,28,137,44,36,131,122,4,249,15,133,38,14,0,0,72
        .byte 131,125,48,0,15,133,27,14,0,0,128,125,7,1,15,135
        .byte 17,14,0,0,139,77,24,116,9,59,77,16,15,132,3,14
        .byte 0,0,141,92,193,240,59,93,32,15,135,246,13,0,0,137
        .byte 93,24,139,108,36,24,137,85,16,131,194,8,137,85,24,141
        .byte 108,194,232,72,41,221,57,203,116,15,72,139,4,43,72,137
        .byte 67,248,131,235,8,57,203,117,241,137,206,139,60,36,232,80
        .byte 246,255,255,139,108,36,24,139,28,36,139,85,16,65,137,174
        .byte 24,245,255,255,65,199,134,120,244,255,255,255,255,255,255,131
        .byte 248,1,119,90,139,75,16,68,139,123,24,137,75,24,68,137
        .byte 251,41,203,116,31,141,4,26,193,235,3,59,69,32,119,91
        .byte 137,213,72,41,205,72,139,1,72,137,4,41,131,193,8,68
        .byte 57,249,117,241,141,67,2,199,66,252,253,255,255,255,139,92
        .byte 36,28,137,68,36,4,72,199,193,248,255,255,255,247,195,3
        .byte 0,0,0,15,132,133,237,255,255,233,132,244,255,255,199,66
        .byte 252,254,255,255,255,139,75,24,131,233,8,137,75,24,72,139
        .byte 1,72,137,2,184,3,0,0,0,235,195,139,12,36,68,137
        .byte 121,24,137,222,137,239
        call lj_state_growstack
        .byte 139,28,36,139,85,16,233,110,255,255,255

        .globl lj_ff_coroutine_wrap_aux
        .hidden lj_ff_coroutine_wrap_aux
        .type lj_ff_coroutine_wrap_aux, @function
        .size lj_ff_coroutine_wrap_aux, 257
lj_ff_coroutine_wrap_aux:
        .byte 139,106,248,139,109,32,139,90,252,137,92,36,28,137,44,36
        .byte 72,131,125,48,0,15,133,244,12,0,0,128,125,7,1,15
        .byte 135,234,12,0,0,139,77,24,116,9,59,77,16,15,132,220
        .byte 12,0,0,141,92,193,248,59,93,32,15,135,207,12,0,0
        .byte 137,93,24,139,108,36,24,137,85,16,137,85,24,141,108,194
        .byte 240,72,41,221,57,203,116,15,72,139,4,43,72,137,67,248
        .byte 131,235,8,57,203,117,241,137,206,139,60,36,232,44,245,255
        .byte 255,139,108,36,24,139,28,36,139,85,16,65,137,174,24,245
        .byte 255,255,65,199,134,120,244,255,255,255,255,255,255,131,248,1
        .byte 119,78,139,75,16,68,139,123,24,137,75,24,68,137,251,41
        .byte 203,116,31,141,4,26,193,235,3,59,69,32,119,59,137,213
        .byte 72,41,205,72,139,1,72,137,4,41,131,193,8,68,57,249
        .byte 117,241,141,67,1,139,92,36,28,137,68,36,4,49,201,247
        .byte 195,3,0,0,0,15,132,109,236,255,255,233,108,243,255,255
        .byte 137,222,137,239
        call lj_ffh_coroutine_wrap_err
        .byte 139,12,36,68,137,121,24,137,222,137,239
        call lj_state_growstack
        .byte 139,28,36,139,85,16,235,145

        .globl lj_ff_coroutine_yield
        .hidden lj_ff_coroutine_yield
        .type lj_ff_coroutine_yield, @function
        .size lj_ff_coroutine_yield, 44
lj_ff_coroutine_yield:
        .byte 139,108,36,24,72,247,69,48,1,0,0,0,15,132,252,11
        .byte 0,0,137,85,16,141,68,194,248,137,69,24,49,192,72,137
        .byte 69,48,176,1,136,69,7,233,117,243,255,255

        .globl lj_fff_resn
        .hidden lj_fff_resn
        .type lj_fff_resn, @function
        .size lj_fff_resn, 8
lj_fff_resn:
        .byte 139,90,252,221,90,248,235,99

        .globl lj_ff_math_abs
        .hidden lj_ff_math_abs
        .type lj_ff_math_abs, @function
```

```
        .size lj_ff_math_abs, 29
lj_ff_math_abs:
        .byte 131,248,2,15,130,209,11,0,0,129,122,4,255,255,254,255
        .byte 117,45,139,42,131,253,0,121,4,247,221,120,15

        .globl lj_fff_resi
        .hidden lj_fff_resi
        .type lj_fff_resi, @function
        .size lj_fff_resi, 0
lj_fff_resi:

        .globl lj_fff_resbit
        .hidden lj_fff_resbit
        .type lj_fff_resbit, @function
        .size lj_fff_resbit, 62
lj_fff_resbit:
        .byte 139,90,252,199,66,252,255,255,254,255,137,106,248,235,55,139
        .byte 90,252,199,66,252,0,0,224,65,199,66,248,0,0,0,0
        .byte 235,36,15,135,149,11,0,0,242,15,16,2,72,184,255,255
        .byte 255,255,255,255,255,127,102,72,15,110,200,15,84,193

        .globl lj_fff_resxmm0
        .hidden lj_fff_resxmm0
        .type lj_fff_resxmm0, @function
        .size lj_fff_resxmm0, 8
lj_fff_resxmm0:
        .byte 139,90,252,242,15,17,66,248

        .globl lj_fff_res1
        .hidden lj_fff_res1
        .type lj_fff_res1, @function
        .size lj_fff_res1, 5
lj_fff_res1:
        .byte 184,2,0,0,0

        .globl lj_fff_res
        .hidden lj_fff_res
        .type lj_fff_res, @function
        .size lj_fff_res, 4
lj_fff_res:
        .byte 137,68,36,4

        .globl lj_fff_res_
        .hidden lj_fff_res_
        .type lj_fff_res_, @function
        .size lj_fff_res_, 66
lj_fff_res_:
        .byte 247,195,3,0,0,0,117,46,56,67,255,119,28,15,182,75
        .byte 253,72,247,209,141,20,202,139,3,15,182,204,15,182,232,131
        .byte 195,4,193,232,16,65,255,36,238,199,68,194,244,255,255,255
        .byte 255,131,192,1,235,210,72,199,193,248,255,255,255,233,105,242
        .byte 255,255

        .globl lj_ff_math_floor
        .hidden lj_ff_math_floor
        .type lj_ff_math_floor, @function
        .size lj_ff_math_floor, 72
lj_ff_math_floor:
        .byte 129,122,4,255,255,254,255,117,7,139,42,233,95,255,255,255
        .byte 15,135,22,11,0,0,242,15,16,2,232,7,15,0,0,242
        .byte 15,44,232,129,253,0,0,0,128,15,133,64,255,255,255,242
        .byte 15,42,205,102,15,46,193,15,138,112,255,255,255,15,132,44
        .byte 255,255,255,233,101,255,255,255

        .globl lj_ff_math_ceil
        .hidden lj_ff_math_ceil
        .type lj_ff_math_ceil, @function
        .size lj_ff_math_ceil, 72
lj_ff_math_ceil:
        .byte 129,122,4,255,255,254,255,117,7,139,42,233,23,255,255,255
        .byte 15,135,206,10,0,0,242,15,16,2,232,26,15,0,0,242
        .byte 15,44,232,129,253,0,0,0,128,15,133,248,254,255,255,242
        .byte 15,42,205,102,15,46,193,15,138,40,255,255,255,15,132,228
        .byte 254,255,255,233,29,255,255,255

        .globl lj_ff_math_sqrt
```

```
        .hidden lj_ff_math_sqrt
        .type lj_ff_math_sqrt, @function
        .size lj_ff_math_sqrt, 31
lj_ff_math_sqrt:
        .byte 131,248,2,15,130,147,10,0,0,129,122,4,255,255,254,255
        .byte 15,131,134,10,0,0,242,15,81,2,233,254,254,255,255

        .globl lj_ff_math_log
        .hidden lj_ff_math_log
        .type lj_ff_math_log, @function
        .size lj_ff_math_log, 40
lj_ff_math_log:
        .byte 131,248,2,15,133,116,10,0,0,129,122,4,255,255,254,255
        .byte 15,131,103,10,0,0,242,15,16,2,137,213
        call log@PLT
        .byte 137,234,233,214,254,255,255

        .globl lj_ff_math_log10
        .hidden lj_ff_math_log10
        .type lj_ff_math_log10, @function
        .size lj_ff_math_log10, 40
lj_ff_math_log10:
        .byte 131,248,2,15,130,76,10,0,0,129,122,4,255,255,254,255
        .byte 15,131,63,10,0,0,242,15,16,2,137,213
        call log10@PLT
        .byte 137,234,233,174,254,255,255

        .globl lj_ff_math_exp
        .hidden lj_ff_math_exp
        .type lj_ff_math_exp, @function
        .size lj_ff_math_exp, 40
lj_ff_math_exp:
        .byte 131,248,2,15,130,36,10,0,0,129,122,4,255,255,254,255
        .byte 15,131,23,10,0,0,242,15,16,2,137,213
        call exp@PLT
        .byte 137,234,233,134,254,255,255

        .globl lj_ff_math_sin
        .hidden lj_ff_math_sin
        .type lj_ff_math_sin, @function
        .size lj_ff_math_sin, 40
lj_ff_math_sin:
        .byte 131,248,2,15,130,252,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,239,9,0,0,242,15,16,2,137,213
        call sin@PLT
        .byte 137,234,233,94,254,255,255

        .globl lj_ff_math_cos
        .hidden lj_ff_math_cos
        .type lj_ff_math_cos, @function
        .size lj_ff_math_cos, 40
lj_ff_math_cos:
        .byte 131,248,2,15,130,212,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,199,9,0,0,242,15,16,2,137,213
        call cos@PLT
        .byte 137,234,233,54,254,255,255

        .globl lj_ff_math_tan
        .hidden lj_ff_math_tan
        .type lj_ff_math_tan, @function
        .size lj_ff_math_tan, 40
lj_ff_math_tan:
        .byte 131,248,2,15,130,172,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,159,9,0,0,242,15,16,2,137,213
        call tan@PLT
        .byte 137,234,233,14,254,255,255

        .globl lj_ff_math_asin
        .hidden lj_ff_math_asin
        .type lj_ff_math_asin, @function
        .size lj_ff_math_asin, 40
lj_ff_math_asin:
        .byte 131,248,2,15,130,132,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,119,9,0,0,242,15,16,2,137,213
        call asin@PLT
        .byte 137,234,233,230,253,255,255
```

```
        .globl lj_ff_math_acos
        .hidden lj_ff_math_acos
        .type lj_ff_math_acos, @function
        .size lj_ff_math_acos, 40
lj_ff_math_acos:
        .byte 131,248,2,15,130,92,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,79,9,0,0,242,15,16,2,137,213
        call acos@PLT
        .byte 137,234,233,190,253,255,255

        .globl lj_ff_math_atan
        .hidden lj_ff_math_atan
        .type lj_ff_math_atan, @function
        .size lj_ff_math_atan, 40
lj_ff_math_atan:
        .byte 131,248,2,15,130,52,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,39,9,0,0,242,15,16,2,137,213
        call atan@PLT
        .byte 137,234,233,150,253,255,255

        .globl lj_ff_math_sinh
        .hidden lj_ff_math_sinh
        .type lj_ff_math_sinh, @function
        .size lj_ff_math_sinh, 40
lj_ff_math_sinh:
        .byte 131,248,2,15,130,12,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,255,8,0,0,242,15,16,2,137,213
        call sinh@PLT
        .byte 137,234,233,110,253,255,255

        .globl lj_ff_math_cosh
        .hidden lj_ff_math_cosh
        .type lj_ff_math_cosh, @function
        .size lj_ff_math_cosh, 40
lj_ff_math_cosh:
        .byte 131,248,2,15,130,228,8,0,0,129,122,4,255,255,254,255
        .byte 15,131,215,8,0,0,242,15,16,2,137,213
        call cosh@PLT
        .byte 137,234,233,70,253,255,255

        .globl lj_ff_math_tanh
        .hidden lj_ff_math_tanh
        .type lj_ff_math_tanh, @function
        .size lj_ff_math_tanh, 40
lj_ff_math_tanh:
        .byte 131,248,2,15,130,188,8,0,0,129,122,4,255,255,254,255
        .byte 15,131,175,8,0,0,242,15,16,2,137,213
        call tanh@PLT
        .byte 137,234,233,30,253,255,255

        .globl lj_ff_math_pow
        .hidden lj_ff_math_pow
        .type lj_ff_math_pow, @function
        .size lj_ff_math_pow, 58
lj_ff_math_pow:
        .byte 131,248,3,15,130,148,8,0,0,129,122,4,255,255,254,255
        .byte 15,131,135,8,0,0,129,122,12,255,255,254,255,15,131,122
        .byte 8,0,0,242,15,16,2,242,15,16,74,8,137,213
        call pow@PLT
        .byte 137,234,233,228,252,255,255

        .globl lj_ff_math_atan2
        .hidden lj_ff_math_atan2
        .type lj_ff_math_atan2, @function
        .size lj_ff_math_atan2, 58
lj_ff_math_atan2:
        .byte 131,248,3,15,130,90,8,0,0,129,122,4,255,255,254,255
        .byte 15,131,77,8,0,0,129,122,12,255,255,254,255,15,131,64
        .byte 8,0,0,242,15,16,2,242,15,16,74,8,137,213
        call atan2@PLT
        .byte 137,234,233,170,252,255,255

        .globl lj_ff_math_fmod
        .hidden lj_ff_math_fmod
        .type lj_ff_math_fmod, @function
```

```
        .size lj_ff_math_fmod, 58
lj_ff_math_fmod:
        .byte 131,248,3,15,130,32,8,0,0,129,122,4,255,255,254,255
        .byte 15,131,19,8,0,0,129,122,12,255,255,254,255,15,131,6
        .byte 8,0,0,242,15,16,2,242,15,16,74,8,137,213
        call fmod@PLT
        .byte 137,234,233,112,252,255,255

        .globl lj_ff_math_ldexp
        .hidden lj_ff_math_ldexp
        .type lj_ff_math_ldexp, @function
        .size lj_ff_math_ldexp, 49
lj_ff_math_ldexp:
        .byte 131,248,3,15,130,230,7,0,0,129,122,4,255,255,254,255
        .byte 15,131,217,7,0,0,129,122,12,255,255,254,255,15,131,204
        .byte 7,0,0,221,66,8,221,2,217,253,221,217,233,220,251,255
        .byte 255

        .globl lj_ff_math_frexp
        .hidden lj_ff_math_frexp
        .type lj_ff_math_frexp, @function
        .size lj_ff_math_frexp, 148
lj_ff_math_frexp:
        .byte 131,248,2,15,130,181,7,0,0,139,106,4,129,253,255,255
        .byte 254,255,15,131,166,7,0,0,139,90,252,139,2,137,106,252
        .byte 137,66,248,209,229,129,253,0,0,224,255,115,58,9,232,116
        .byte 54,184,254,3,0,0,129,253,0,0,32,0,114,46,193,237
        .byte 21,41,197,242,15,42,197,139,106,252,129,229,255,255,15,128
        .byte 129,205,0,0,224,63,137,106,252,242,15,17,2,184,3,0
        .byte 0,0,233,229,251,255,255,15,87,192,235,237,242,15,16,2
        .byte 72,189,0,0,0,0,0,0,80,67,102,72,15,110,205,242
        .byte 15,89,193,242,15,17,66,248,139,106,252,184,52,4,0,0
        .byte 209,229,235,170

        .globl lj_ff_math_modf
        .hidden lj_ff_math_modf
        .type lj_ff_math_modf, @function
        .size lj_ff_math_modf, 99
lj_ff_math_modf:
        .byte 131,248,2,15,130,33,7,0,0,129,122,4,255,255,254,255
        .byte 15,131,20,7,0,0,242,15,16,2,139,106,4,139,90,252
        .byte 209,229,129,253,0,0,224,255,116,52,15,40,224,232,168,11
        .byte 0,0,242,15,92,224,242,15,17,66,248,242,15,17,34,139
        .byte 66,252,139,106,4,49,232,120,10,184,3,0,0,0,233,101
        .byte 251,255,255,129,245,0,0,0,128,137,106,4,235,235,15,87
        .byte 228,235,211

        .globl lj_ff_math_min
        .hidden lj_ff_math_min
        .type lj_ff_math_min, @function
        .size lj_ff_math_min, 117
lj_ff_math_min:
        .byte 185,2,0,0,0,129,122,4,255,255,254,255,117,46,139,42
        .byte 57,193,15,131,242,250,255,255,129,124,202,252,255,255,254,255
        .byte 117,14,59,108,202,248,15,79,108,202,248,131,193,1,235,224
        .byte 15,135,145,6,0,0,242,15,42,197,235,42,15,135,133,6
        .byte 0,0,242,15,16,2,57,193,15,131,250,250,255,255,129,124
        .byte 202,252,255,255,254,255,114,14,15,135,105,6,0,0,242,15
        .byte 42,76,202,248,235,6,242,15,16,76,202,248,242,15,93,193
        .byte 131,193,1,235,209

        .globl lj_ff_math_max
        .hidden lj_ff_math_max
        .type lj_ff_math_max, @function
        .size lj_ff_math_max, 117
lj_ff_math_max:
        .byte 185,2,0,0,0,129,122,4,255,255,254,255,117,46,139,42
        .byte 57,193,15,131,125,250,255,255,129,124,202,252,255,255,254,255
        .byte 117,14,59,108,202,248,15,76,108,202,248,131,193,1,235,224
        .byte 15,135,28,6,0,0,242,15,42,197,235,42,15,135,16,6
        .byte 0,0,242,15,16,2,57,193,15,131,133,250,255,255,129,124
        .byte 202,252,255,255,254,255,114,14,15,135,244,5,0,0,242,15
        .byte 42,76,202,248,235,6,242,15,16,76,202,248,242,15,95,193
        .byte 131,193,1,235,209

        .globl lj_ff_string_byte
```

```
        .hidden lj_ff_string_byte
        .type lj_ff_string_byte, @function
        .size lj_ff_string_byte, 43
lj_ff_string_byte:
        .byte 131,248,2,15,133,212,5,0,0,131,122,4,251,15,133,202
        .byte 5,0,0,139,42,139,90,252,131,125,12,1,15,130,204,246
        .byte 255,255,15,182,109,16,233,245,249,255,255

        .globl lj_ff_string_char
        .hidden lj_ff_string_char
        .type lj_ff_string_char, @function
        .size lj_ff_string_char, 74
lj_ff_string_char:
        .byte 65,139,174,64,244,255,255,65,59,174,68,244,255,255,114,5
        .byte 232,44,6,0,0,131,248,2,15,133,148,5,0,0,129,122
        .byte 4,255,255,254,255,15,133,135,5,0,0,139,42,129,253,255
        .byte 0,0,0,15,135,121,5,0,0,137,108,36,4,199,68,36
        .byte 8,1,0,0,0,72,141,68,36,4

        .globl lj_fff_newstr
        .hidden lj_fff_newstr
        .type lj_fff_newstr, @function
        .size lj_fff_newstr, 25
lj_fff_newstr:
        .byte 139,108,36,24,137,85,16,139,84,36,8,72,137,198,137,239
        .byte 137,92,36,28
        call lj_str_new

        .globl lj_fff_resstr
        .hidden lj_fff_resstr
        .type lj_fff_resstr, @function
        .size lj_fff_resstr, 21
lj_fff_resstr:
        .byte 139,85,16,139,90,252,199,66,252,251,255,255,255,137,66,248
        .byte 233,195,249,255,255

        .globl lj_ff_string_sub
        .hidden lj_ff_string_sub
        .type lj_ff_string_sub, @function
        .size lj_ff_string_sub, 161
lj_ff_string_sub:
        .byte 65,139,174,64,244,255,255,65,59,174,68,244,255,255,114,5
        .byte 232,180,5,0,0,199,68,36,4,255,255,255,255,131,248,3
        .byte 15,130,20,5,0,0,118,20,129,122,20,255,255,254,255,15
        .byte 133,5,5,0,0,139,106,16,137,108,36,4,131,122,4,251
        .byte 15,133,244,4,0,0,129,122,12,255,255,254,255,15,133,231
        .byte 4,0,0,139,42,137,108,36,8,139,109,12,139,74,8,139
        .byte 68,36,4,57,197,114,30,133,201,126,38,139,108,36,8,41
        .byte 200,124,46,141,108,13,15,131,192,1,137,68,36,8,137,232
        .byte 233,77,255,255,255,124,6,141,68,40,1,235,218,137,232,235
        .byte 214,116,7,1,233,131,193,1,127,209,185,1,0,0,0,235
        .byte 202

        .globl lj_fff_emptystr
        .hidden lj_fff_emptystr
        .type lj_fff_emptystr, @function
        .size lj_fff_emptystr, 4
lj_fff_emptystr:
        .byte 49,192,235,213

        .globl lj_ff_string_reverse
        .hidden lj_ff_string_reverse
        .type lj_ff_string_reverse, @function
        .size lj_ff_string_reverse, 85
lj_ff_string_reverse:
        .byte 131,248,2,15,130,140,4,0,0,65,139,174,64,244,255,255
        .byte 65,59,174,68,244,255,255,114,5,232,6,5,0,0,131,122
        .byte 4,251,15,133,109,4,0,0,139,108,36,24,65,141,190,124
        .byte 244,255,255,137,85,16,139,50,139,71,8,137,111,12,137,7
        .byte 137,92,36,28
        call lj_buf_putstr_reverse
        .byte 137,199
        call lj_buf_tostr
        .byte 233,241,254,255,255

        .globl lj_ff_string_lower
```

```
        .hidden lj_ff_string_lower
        .type lj_ff_string_lower, @function
        .size lj_ff_string_lower, 85
lj_ff_string_lower:
        .byte 131,248,2,15,130,55,4,0,0,65,139,174,64,244,255,255
        .byte 65,59,174,68,244,255,255,114,5,232,177,4,0,0,131,122
        .byte 4,251,15,133,24,4,0,0,139,108,36,24,65,141,190,124
        .byte 244,255,255,137,85,16,139,50,139,71,8,137,111,12,137,7
        .byte 137,92,36,28
        call lj_buf_putstr_lower
        .byte 137,199
        call lj_buf_tostr
        .byte 233,156,254,255,255

        .globl lj_ff_string_upper
        .hidden lj_ff_string_upper
        .type lj_ff_string_upper, @function
        .size lj_ff_string_upper, 85
lj_ff_string_upper:
        .byte 131,248,2,15,130,226,3,0,0,65,139,174,64,244,255,255
        .byte 65,59,174,68,244,255,255,114,5,232,92,4,0,0,131,122
        .byte 4,251,15,133,195,3,0,0,139,108,36,24,65,141,190,124
        .byte 244,255,255,137,85,16,139,50,139,71,8,137,111,12,137,7
        .byte 137,92,36,28
        call lj_buf_putstr_upper
        .byte 137,199
        call lj_buf_tostr
        .byte 233,71,254,255,255

        .globl lj_ff_bit_tobit
        .hidden lj_ff_bit_tobit
        .type lj_ff_bit_tobit, @function
        .size lj_ff_bit_tobit, 63
lj_ff_bit_tobit:
        .byte 131,248,2,15,130,141,3,0,0,129,122,4,255,255,254,255
        .byte 117,7,139,42,233,192,247,255,255,15,135,119,3,0,0,242
        .byte 15,16,2,72,189,0,0,0,0,0,0,56,67,102,72,15
        .byte 110,205,242,15,88,193,102,15,126,197,233,154,247,255,255

        .globl lj_ff_bit_band
        .hidden lj_ff_bit_band
        .type lj_ff_bit_band, @function
        .size lj_ff_bit_band, 112
lj_ff_bit_band:
        .byte 131,248,2,15,130,78,3,0,0,72,189,0,0,0,0,0
        .byte 0,56,67,102,72,15,110,205,129,122,4,255,255,254,255,117
        .byte 4,139,42,235,18,15,135,44,3,0,0,242,15,16,2,242
        .byte 15,88,193,102,15,126,197,137,68,36,4,141,68,194,240,57
        .byte 208,15,134,83,247,255,255,129,120,4,255,255,254,255,117,7
        .byte 35,40,131,232,8,235,232,15,135,111,1,0,0,242,15,16
        .byte 0,242,15,88,193,102,15,126,193,33,205,131,232,8,235,207

        .globl lj_ff_bit_bor
        .hidden lj_ff_bit_bor
        .type lj_ff_bit_bor, @function
        .size lj_ff_bit_bor, 112
lj_ff_bit_bor:
        .byte 131,248,2,15,130,222,2,0,0,72,189,0,0,0,0,0
        .byte 0,56,67,102,72,15,110,205,129,122,4,255,255,254,255,117
        .byte 4,139,42,235,18,15,135,188,2,0,0,242,15,16,2,242
        .byte 15,88,193,102,15,126,197,137,68,36,4,141,68,194,240,57
        .byte 208,15,134,227,246,255,255,129,120,4,255,255,254,255,117,7
        .byte 11,40,131,232,8,235,232,15,135,255,0,0,0,242,15,16
        .byte 0,242,15,88,193,102,15,126,193,9,205,131,232,8,235,207

        .globl lj_ff_bit_bxor
        .hidden lj_ff_bit_bxor
        .type lj_ff_bit_bxor, @function
        .size lj_ff_bit_bxor, 112
lj_ff_bit_bxor:
        .byte 131,248,2,15,130,110,2,0,0,72,189,0,0,0,0,0
        .byte 0,56,67,102,72,15,110,205,129,122,4,255,255,254,255,117
        .byte 4,139,42,235,18,15,135,76,2,0,0,242,15,16,2,242
        .byte 15,88,193,102,15,126,197,137,68,36,4,141,68,194,240,57
        .byte 208,15,134,115,246,255,255,129,120,4,255,255,254,255,117,7
        .byte 51,40,131,232,8,235,232,15,135,143,0,0,0,242,15,16
```

```
        .byte 0,242,15,88,193,102,15,126,193,49,205,131,232,8,235,207

        .globl lj_ff_bit_bswap
        .hidden lj_ff_bit_bswap
        .type lj_ff_bit_bswap, @function
        .size lj_ff_bit_bswap, 62
lj_ff_bit_bswap:
        .byte 131,248,2,15,130,254,1,0,0,129,122,4,255,255,254,255
        .byte 117,4,139,42,235,33,15,135,235,1,0,0,242,15,16,2
        .byte 72,189,0,0,0,0,0,0,56,67,102,72,15,110,205,242
        .byte 15,88,193,102,15,126,197,15,205,233,12,246,255,255

        .globl lj_ff_bit_bnot
        .hidden lj_ff_bit_bnot
        .type lj_ff_bit_bnot, @function
        .size lj_ff_bit_bnot, 62
lj_ff_bit_bnot:
        .byte 131,248,2,15,130,192,1,0,0,129,122,4,255,255,254,255
        .byte 117,4,139,42,235,33,15,135,173,1,0,0,242,15,16,2
        .byte 72,189,0,0,0,0,0,0,56,67,102,72,15,110,205,242
        .byte 15,88,193,102,15,126,197,247,213,233,206,245,255,255

        .globl lj_fff_fallback_bit_op
        .hidden lj_fff_fallback_bit_op
        .type lj_fff_fallback_bit_op, @function
        .size lj_fff_fallback_bit_op, 9
lj_fff_fallback_bit_op:
        .byte 139,68,36,4,233,130,1,0,0

        .globl lj_ff_bit_lshift
        .hidden lj_ff_bit_lshift
        .type lj_ff_bit_lshift, @function
        .size lj_ff_bit_lshift, 78
lj_ff_bit_lshift:
        .byte 131,248,2,15,130,121,1,0,0,129,122,4,255,255,254,255
        .byte 117,4,139,42,235,33,15,135,102,1,0,0,242,15,16,2
        .byte 72,189,0,0,0,0,0,0,56,67,102,72,15,110,205,242
        .byte 15,88,193,102,15,126,197,129,122,12,255,255,254,255,15,133
        .byte 62,1,0,0,139,74,8,211,229,233,119,245,255,255

        .globl lj_ff_bit_rshift
        .hidden lj_ff_bit_rshift
        .type lj_ff_bit_rshift, @function
        .size lj_ff_bit_rshift, 78
lj_ff_bit_rshift:
        .byte 131,248,2,15,130,43,1,0,0,129,122,4,255,255,254,255
        .byte 117,4,139,42,235,33,15,135,24,1,0,0,242,15,16,2
        .byte 72,189,0,0,0,0,0,0,56,67,102,72,15,110,205,242
        .byte 15,88,193,102,15,126,197,129,122,12,255,255,254,255,15,133
        .byte 240,0,0,0,139,74,8,211,237,233,41,245,255,255

        .globl lj_ff_bit_arshift
        .hidden lj_ff_bit_arshift
        .type lj_ff_bit_arshift, @function
        .size lj_ff_bit_arshift, 78
lj_ff_bit_arshift:
        .byte 131,248,2,15,130,221,0,0,0,129,122,4,255,255,254,255
        .byte 117,4,139,42,235,33,15,135,202,0,0,0,242,15,16,2
        .byte 72,189,0,0,0,0,0,0,56,67,102,72,15,110,205,242
        .byte 15,88,193,102,15,126,197,129,122,12,255,255,254,255,15,133
        .byte 162,0,0,0,139,74,8,211,253,233,219,244,255,255

        .globl lj_ff_bit_rol
        .hidden lj_ff_bit_rol
        .type lj_ff_bit_rol, @function
        .size lj_ff_bit_rol, 74
lj_ff_bit_rol:
        .byte 131,248,2,15,130,143,0,0,0,129,122,4,255,255,254,255
        .byte 117,4,139,42,235,33,15,135,124,0,0,0,242,15,16,2
        .byte 72,189,0,0,0,0,0,0,56,67,102,72,15,110,205,242
        .byte 15,88,193,102,15,126,197,129,122,12,255,255,254,255,117,88
        .byte 139,74,8,211,197,233,145,244,255,255

        .globl lj_ff_bit_ror
        .hidden lj_ff_bit_ror
        .type lj_ff_bit_ror, @function
```

```
        .size lj_ff_bit_ror, 66
lj_ff_bit_ror:
        .byte 131,248,2,114,73,129,122,4,255,255,254,255,117,4,139,42
        .byte 235,29,119,58,242,15,16,2,72,189,0,0,0,0,0,0
        .byte 56,67,102,72,15,110,205,242,15,88,193,102,15,126,197,129
        .byte 122,12,255,255,254,255,117,22,139,74,8,211,205,233,79,244
        .byte 255,255

        .globl lj_fff_fallback_2
        .hidden lj_fff_fallback_2
        .type lj_fff_fallback_2, @function
        .size lj_fff_fallback_2, 7
lj_fff_fallback_2:
        .byte 184,3,0,0,0,235,5

        .globl lj_fff_fallback_1
        .hidden lj_fff_fallback_1
        .type lj_fff_fallback_1, @function
        .size lj_fff_fallback_1, 5
lj_fff_fallback_1:
        .byte 184,2,0,0,0

        .globl lj_fff_fallback
        .hidden lj_fff_fallback
        .type lj_fff_fallback, @function
        .size lj_fff_fallback, 87
lj_fff_fallback:
        .byte 139,108,36,24,139,90,252,137,92,36,28,137,85,16,141,68
        .byte 194,248,141,136,160,0,0,0,137,69,24,139,66,248,59,77
        .byte 32,119,89,137,239,255,80,24,139,85,16,133,192,15,143,91
        .byte 244,255,255,139,77,24,41,209,193,233,3,133,192,141,65,1
        .byte 139,106,248,117,18,139,93,16,139,11,15,182,233,15,182,205
        .byte 131,195,4,65,255,36,238

        .globl lj_vm_call_tail
        .hidden lj_vm_call_tail
        .type lj_vm_call_tail, @function
        .size lj_vm_call_tail, 56
lj_vm_call_tail:
        .byte 137,209,247,195,3,0,0,0,117,15,15,182,107,253,72,247
        .byte 213,141,20,234,233,18,233,255,255,137,221,131,229,248,41,234
        .byte 233,6,233,255,255,190,20,0,0,0,137,239
        call lj_state_growstack
        .byte 139,85,16,49,192,235,164

        .globl lj_fff_gcstep
        .hidden lj_fff_gcstep
        .type lj_fff_gcstep, @function
        .size lj_fff_gcstep, 52
lj_fff_gcstep:
        .byte 93,72,137,108,36,8,139,108,36,24,137,92,36,28,137,85
        .byte 16,141,68,194,248,137,239,137,69,24
        call lj_gc_step
        .byte 139,85,16,139,69,24,41,208,193,232,3,131,192,1,72,139
        .byte 108,36,8,85,195

        .globl lj_vm_record
        .hidden lj_vm_record
        .type lj_vm_record, @function
        .size lj_vm_record, 29
lj_vm_record:
        .byte 65,15,182,134,157,244,255,255,168,32,117,83,168,16,117,56
        .byte 168,12,116,52,65,255,142,240,244,255,255,235,43

        .globl lj_vm_rethook
        .hidden lj_vm_rethook
        .type lj_vm_rethook, @function
        .size lj_vm_rethook, 14
lj_vm_rethook:
        .byte 65,15,182,134,157,244,255,255,168,16,117,54,235,29

        .globl lj_vm_inshook
        .hidden lj_vm_inshook
        .type lj_vm_inshook, @function
        .size lj_vm_inshook, 68
lj_vm_inshook:
```

```
        .byte 65,15,182,134,157,244,255,255,168,16,117,40,168,12,116,36
        .byte 65,255,142,240,244,255,255,116,4,168,4,116,23,139,108,36
        .byte 24,137,85,16,137,222,137,239
        call lj_dispatch_ins
        .byte 139,85,16,15,182,75,253,15,182,107,252,15,183,67,254,65
        .byte 255,164,238,208,4,0,0

        .globl lj_cont_hook
        .hidden lj_cont_hook
        .type lj_cont_hook, @function
        .size lj_cont_hook, 12
lj_cont_hook:
        .byte 131,195,4,139,77,232,137,76,36,4,235,224

        .globl lj_vm_hotloop
        .hidden lj_vm_hotloop
        .type lj_vm_hotloop, @function
        .size lj_vm_hotloop, 50
lj_vm_hotloop:
        .byte 139,106,248,139,109,16,15,182,69,199,141,4,194,139,108,36
        .byte 24,137,85,16,137,69,24,137,222,65,141,190,192,245,255,255
        .byte 73,137,174,32,246,255,255,137,92,36,28
        call lj_trace_hot
        .byte 235,171

        .globl lj_vm_callhook
        .hidden lj_vm_callhook
        .type lj_vm_callhook, @function
        .size lj_vm_callhook, 6
lj_vm_callhook:
        .byte 137,92,36,28,235,7

        .globl lj_vm_hotcall
        .hidden lj_vm_hotcall
        .type lj_vm_hotcall, @function
        .size lj_vm_hotcall, 67
lj_vm_hotcall:
        .byte 137,92,36,28,131,203,1,141,68,194,248,139,108,36,24,137
        .byte 85,16,137,69,24,137,222,137,239
        call lj_dispatch_call
        .byte 199,68,36,28,0,0,0,0,131,227,254,139,85,16,72,137
        .byte 193,139,69,24,41,208,72,137,205,15,182,75,253,193,232,3
        .byte 131,192,1,255,229

        .globl lj_cont_stitch
        .hidden lj_cont_stitch
        .type lj_cont_stitch, @function
        .size lj_cont_stitch, 166
lj_cont_stitch:
        .byte 139,77,232,137,12,36,68,137,116,36,8,68,139,116,36,4
        .byte 15,182,75,253,141,12,202,65,131,238,1,116,18,72,139,40
        .byte 72,137,41,131,192,8,131,193,8,65,131,238,1,117,238,15
        .byte 182,67,253,15,182,107,255,1,232,141,68,194,248,57,200,119
        .byte 89,68,139,116,36,8,139,44,36,65,139,142,240,246,255,255
        .byte 139,4,169,133,192,15,132,162,233,255,255,15,183,64,82,57
        .byte 232,15,132,150,233,255,255,133,192,15,133,214,226,255,255,65
        .byte 137,174,44,255,255,255,139,108,36,24,137,85,16,137,222,65
        .byte 141,190,192,245,255,255,73,137,174,32,246,255,255
        call lj_dispatch_stitch
        .byte 139,85,16,233,99,233,255,255,199,65,4,255,255,255,255,131
        .byte 193,8,235,151

        .globl lj_vm_profhook
        .hidden lj_vm_profhook
        .type lj_vm_profhook, @function
        .size lj_vm_profhook, 27
lj_vm_profhook:
        .byte 139,108,36,24,137,85,16,137,222,137,239
        call lj_dispatch_profile
        .byte 139,85,16,131,235,4,233,60,233,255,255

        .globl lj_vm_exit_handler
        .hidden lj_vm_exit_handler
        .type lj_vm_exit_handler, @function
        .size lj_vm_exit_handler, 247
lj_vm_exit_handler:
```

```
        .byte 65,85,65,84,65,83,65,82,65,81,65,80,87,86,85,72
        .byte 141,108,36,88,85,83,82,81,80,15,182,69,248,138,101,240
        .byte 76,137,125,248,76,137,117,240,68,139,117,0,65,139,142,120
        .byte 244,255,255,65,199,134,120,244,255,255,252,255,255,255,65,137
        .byte 134,44,255,255,255,65,137,142,40,255,255,255,72,129,236,128
        .byte 0,0,0,72,131,197,128,242,68,15,17,125,248,242,68,15
        .byte 17,117,240,242,68,15,17,109,232,242,68,15,17,101,224,242
        .byte 68,15,17,93,216,242,68,15,17,85,208,242,68,15,17,77
        .byte 200,242,68,15,17,69,192,242,15,17,125,184,242,15,17,117
        .byte 176,242,15,17,109,168,242,15,17,101,160,242,15,17,93,152
        .byte 242,15,17,85,144,242,15,17,77,136,242,15,17,69,128,65
        .byte 139,174,24,245,255,255,65,139,150,28,245,255,255,73,137,174
        .byte 32,246,255,255,137,85,16,72,137,230,65,141,190,192,245,255
        .byte 255,65,199,134,28,245,255,255,0,0,0,0
        call lj_trace_exit
        .byte 72,139,77,48,72,131,225,252,72,137,204,137,105,24,139,85
        .byte 16,139,89,28,235,4

        .globl lj_vm_exit_interp
        .hidden lj_vm_exit_interp
        .type lj_vm_exit_interp, @function
        .size lj_vm_exit_interp, 137
lj_vm_exit_interp:
        .byte 72,131,196,16,76,139,108,36,8,76,139,36,36,133,192,120
        .byte 109,139,108,36,24,137,68,36,4,68,139,122,248,69,139,127
        .byte 16,69,139,127,208,137,85,16,65,199,134,28,245,255,255,0
        .byte 0,0,0,65,199,134,120,244,255,255,255,255,255,255,139,3
        .byte 15,182,204,15,182,232,131,195,4,193,232,16,131,253,89,114
        .byte 9,131,253,97,115,8,139,68,36,4,65,255,36,238,139,66
        .byte 252,169,3,0,0,0,117,238,15,182,64,253,72,247,208,68
        .byte 139,124,194,248,69,139,127,16,69,139,127,208,235,216,247,216
        .byte 137,239,137,198
        call lj_err_throw

        .globl lj_vm_floor_sse
        .hidden lj_vm_floor_sse
        .type lj_vm_floor_sse, @function
        .size lj_vm_floor_sse, 0
lj_vm_floor_sse:

        .globl lj_vm_floor
        .hidden lj_vm_floor
        .type lj_vm_floor, @function
        .size lj_vm_floor, 91
lj_vm_floor:
        .byte 72,184,255,255,255,255,255,255,255,127,102,72,15,110,208,72
        .byte 184,0,0,0,0,0,0,48,67,102,72,15,110,216,15,40
        .byte 200,102,15,84,202,102,15,46,217,118,47,102,15,85,208,242
        .byte 15,88,203,242,15,92,203,102,15,86,202,72,184,0,0,0
        .byte 0,0,0,240,63,102,72,15,110,208,242,15,194,193,1,102
        .byte 15,84,194,242,15,92,200,15,40,193,195

        .globl lj_vm_ceil_sse
        .hidden lj_vm_ceil_sse
        .type lj_vm_ceil_sse, @function
        .size lj_vm_ceil_sse, 0
lj_vm_ceil_sse:

        .globl lj_vm_ceil
        .hidden lj_vm_ceil
        .type lj_vm_ceil, @function
        .size lj_vm_ceil, 91
lj_vm_ceil:
        .byte 72,184,255,255,255,255,255,255,255,127,102,72,15,110,208,72
        .byte 184,0,0,0,0,0,0,48,67,102,72,15,110,216,15,40
        .byte 200,102,15,84,202,102,15,46,217,118,47,102,15,85,208,242
        .byte 15,88,203,242,15,92,203,102,15,86,202,72,184,0,0,0
        .byte 0,0,0,240,191,102,72,15,110,208,242,15,194,193,6,102
        .byte 15,84,194,242,15,92,200,15,40,193,195

        .globl lj_vm_trunc_sse
        .hidden lj_vm_trunc_sse
        .type lj_vm_trunc_sse, @function
        .size lj_vm_trunc_sse, 0
lj_vm_trunc_sse:

```

```
        .globl lj_vm_trunc
        .hidden lj_vm_trunc
        .type lj_vm_trunc, @function
        .size lj_vm_trunc, 94
lj_vm_trunc:
        .byte 72,184,255,255,255,255,255,255,255,127,102,72,15,110,208,72
        .byte 184,0,0,0,0,0,0,48,67,102,72,15,110,216,15,40
        .byte 200,102,15,84,202,102,15,46,217,118,50,102,15,85,208,15
        .byte 40,193,242,15,88,203,242,15,92,203,72,184,0,0,0,0
        .byte 0,0,240,63,102,72,15,110,216,242,15,194,193,1,102,15
        .byte 84,195,242,15,92,200,102,15,86,202,15,40,193,195

        .globl lj_vm_mod
        .hidden lj_vm_mod
        .type lj_vm_mod, @function
        .size lj_vm_mod, 118
lj_vm_mod:
        .byte 15,40,232,242,15,94,193,72,184,255,255,255,255,255,255,255
        .byte 127,102,72,15,110,208,72,184,0,0,0,0,0,0,48,67
        .byte 102,72,15,110,216,15,40,224,102,15,84,226,102,15,46,220
        .byte 118,56,102,15,85,208,242,15,88,227,242,15,92,227,102,15
        .byte 86,226,72,184,0,0,0,0,0,0,240,63,102,72,15,110
        .byte 208,242,15,194,196,1,102,15,84,194,242,15,92,224,15,40
        .byte 197,242,15,89,204,242,15,92,193,195,242,15,89,200,15,40
        .byte 197,242,15,92,193,195

        .globl lj_vm_powi_sse
        .hidden lj_vm_powi_sse
        .type lj_vm_powi_sse, @function
        .size lj_vm_powi_sse, 98
lj_vm_powi_sse:
        .byte 131,248,1,126,43,169,1,0,0,0,117,8,242,15,89,192
        .byte 209,232,235,241,209,232,116,23,15,40,200,242,15,89,192,209
        .byte 232,116,8,115,246,242,15,89,200,235,240,242,15,89,193,195
        .byte 116,253,114,30,247,216,232,202,255,255,255,72,184,0,0,0
        .byte 0,0,0,240,63,102,72,15,110,200,242,15,94,200,15,40
        .byte 193,195,72,184,0,0,0,0,0,0,240,63,102,72,15,110
        .byte 192,195

        .globl lj_vm_cpuid
        .hidden lj_vm_cpuid
        .type lj_vm_cpuid, @function
        .size lj_vm_cpuid, 18
lj_vm_cpuid:
        .byte 137,248,83,15,162,137,6,137,94,4,137,78,8,137,86,12
        .byte 91,195

        .globl lj_assert_bad_for_arg_type
        .hidden lj_assert_bad_for_arg_type
        .type lj_assert_bad_for_arg_type, @function
        .size lj_assert_bad_for_arg_type, 2
lj_assert_bad_for_arg_type:
        .byte 204,204

        .globl lj_vm_ffi_callback
        .hidden lj_vm_ffi_callback
        .type lj_vm_ffi_callback, @function
        .size lj_vm_ffi_callback, 179
lj_vm_ffi_callback:
        .byte 83,65,87,65,86,72,131,236,40,68,141,181,224,11,0,0
        .byte 139,157,4,1,0,0,15,183,192,137,131,208,0,0,0,72
        .byte 137,123,112,72,137,115,120,72,137,147,128,0,0,0,72,137
        .byte 139,136,0,0,0,242,15,17,67,48,242,15,17,75,56,242
        .byte 15,17,83,64,242,15,17,91,72,72,141,68,36,80,76,137
        .byte 131,144,0,0,0,76,137,139,152,0,0,0,242,15,17,99
        .byte 80,242,15,17,107,88,242,15,17,115,96,242,15,17,123,104
        .byte 72,137,131,176,0,0,0,72,137,230,137,92,36,28,137,223
        call lj_ccallback_enter
        .byte 65,199,134,120,244,255,255,255,255,255,255,139,80,16,139,64
        .byte 24,41,208,139,106,248,193,232,3,131,192,1,139,93,16,139
        .byte 11,15,182,233,15,182,205,131,195,4,65,255,36,238

        .globl lj_cont_ffi_callback
        .hidden lj_cont_ffi_callback
        .type lj_cont_ffi_callback, @function
        .size lj_cont_ffi_callback, 44
```

```
lj_cont_ffi_callback:
        .byte 139,76,36,24,65,139,158,36,245,255,255,72,137,75,16,137
        .byte 81,16,137,105,24,137,223,137,198
        call lj_ccallback_leave
        .byte 72,139,67,112,242,15,16,67,48,233,186,224,255,255

        .globl lj_vm_ffi_call
        .hidden lj_vm_ffi_call
        .type lj_vm_ffi_call, @function
        .size lj_vm_ffi_call, 160
lj_vm_ffi_call:
        .byte 85,72,137,229,83,72,137,251,139,67,8,72,41,196,15,182
        .byte 75,12,131,233,1,120,17,72,139,132,203,192,0,0,0,72
        .byte 137,4,204,131,233,1,121,239,15,182,67,15,72,139,187,144
        .byte 0,0,0,72,139,179,152,0,0,0,72,139,147,160,0,0
        .byte 0,72,139,139,168,0,0,0,76,139,131,176,0,0,0,76
        .byte 139,139,184,0,0,0,133,192,116,40,15,40,67,16,15,40
        .byte 75,32,15,40,83,48,15,40,91,64,131,248,4,118,19,15
        .byte 40,99,80,15,40,107,96,15,40,115,112,15,40,187,128,0
        .byte 0,0,255,19,72,137,131,144,0,0,0,15,41,67,16,72
        .byte 137,147,152,0,0,0,15,41,75,32,72,139,93,248,201,195

        .section .note.GNU-stack,"",@progbits
        .ident "DynASM 1.3.0"

        .section .debug_frame,"",@progbits
.Lframe0:
        .long .LECIE0-.LSCIE0
.LSCIE0:
        .long 0xffffffff
        .byte 0x1
        .string ""
        .uleb128 0x1
        .sleb128 -8
        .byte 0x10
        .byte 0xc
        .uleb128 0x7
        .uleb128 8
        .byte 0x80+0x10
        .uleb128 0x1
        .align 8
.LECIE0:

.LSFDE0:
        .long .LEFDE0-.LASFDE0
.LASFDE0:
        .long .Lframe0
        .quad .Lbegin
        .quad 15760
        .byte 0xe
        .uleb128 80
        .byte 0x86
        .uleb128 0x2
        .byte 0x83
        .uleb128 0x3
        .byte 0x8f
        .uleb128 0x4
        .byte 0x8e
        .uleb128 0x5
        .align 8
.LEFDE0:

.LSFDE1:
        .long .LEFDE1-.LASFDE1
.LASFDE1:
        .long .Lframe0
        .quad lj_vm_ffi_call
        .quad 160
        .byte 0xe
        .uleb128 16
        .byte 0x86
        .uleb128 0x2
        .byte 0xd
        .uleb128 0x6
        .byte 0x83
        .uleb128 0x3
```

```
        .align 8
.LEFDE1:

        .section .eh_frame,"a",@progbits
.Lframe1:
        .long .LECIE1-.LSCIE1
.LSCIE1:
        .long 0
        .byte 0x1
        .string "zPR"
        .uleb128 0x1
        .sleb128 -8
        .byte 0x10
        .uleb128 6
        .byte 0x1b
        .long lj_err_unwind_dwarf-.
        .byte 0x1b
        .byte 0xc
        .uleb128 0x7
        .uleb128 8
        .byte 0x80+0x10
        .uleb128 0x1
        .align 8
.LECIE1:

.LSFDE2:
        .long .LEFDE2-.LASFDE2
.LASFDE2:
        .long .LASFDE2-.Lframe1
        .long .Lbegin-.
        .long 15760
        .uleb128 0
        .byte 0xe
        .uleb128 80
        .byte 0x86
        .uleb128 0x2
        .byte 0x83
        .uleb128 0x3
        .byte 0x8f
        .uleb128 0x4
        .byte 0x8e
        .uleb128 0x5
        .align 8
.LEFDE2:

.Lframe2:
        .long .LECIE2-.LSCIE2
.LSCIE2:
        .long 0
        .byte 0x1
        .string "zR"
        .uleb128 0x1
        .sleb128 -8
        .byte 0x10
        .uleb128 1
        .byte 0x1b
        .byte 0xc
        .uleb128 0x7
        .uleb128 8
        .byte 0x80+0x10
        .uleb128 0x1
        .align 8
.LECIE2:

.LSFDE3:
        .long .LEFDE3-.LASFDE3
.LASFDE3:
        .long .LASFDE3-.Lframe2
        .long lj_vm_ffi_call-.
        .long 160
        .uleb128 0
        .byte 0xe
        .uleb128 16
        .byte 0x86
        .uleb128 0x2
        .byte 0xd
```

```
2808          .uleb128 0x6
2809          .byte 0x83
2810          .uleb128 0x3
2811          .align 8
2812  .LEFDE3:
2813
```

One Level Up Top Level

src/lj_vm.s - luajit-2.0-src

```
1          .file "buildvm_x86.dasc"
2          .text
3          .p2align 4
4
5          .globl lj_vm_asm_begin
6          .hidden lj_vm_asm_begin
7          .type lj_vm_asm_begin, @object
8          .size lj_vm_asm_begin, 0
9  lj_vm_asm_begin:
10  .Lbegin:
11
12           .globl lj_BC_ISLT
13           .hidden lj_BC_ISLT
14           .type lj_BC_ISLT, @function
15           .size lj_BC_ISLT, 72
16  lj_BC_ISLT:
17           .byte 129,124,202,4,255,255,254,255,15,131,63,29,0,0,129,124
18           .byte 194,4,255,255,254,255,15,131,49,29,0,0,242,15,16,4
19           .byte 194,131,195,4,102,15,46,4,202,118,11,15,183,67,254,141
20           .byte 156,131,0,0,254,255,139,3,15,182,204,15,182,232,131,195
21           .byte 4,193,232,16,65,255,36,238
22
23           .globl lj_BC_ISGE
24           .hidden lj_BC_ISGE
25           .type lj_BC_ISGE, @function
26           .size lj_BC_ISGE, 72
27  lj_BC_ISGE:
28           .byte 129,124,202,4,255,255,254,255,15,131,247,28,0,0,129,124
29           .byte 194,4,255,255,254,255,15,131,233,28,0,0,242,15,16,4
30           .byte 194,131,195,4,102,15,46,4,202,119,11,15,183,67,254,141
31           .byte 156,131,0,0,254,255,139,3,15,182,204,15,182,232,131,195
32           .byte 4,193,232,16,65,255,36,238
33
34           .globl lj_BC_ISLE
35           .hidden lj_BC_ISLE
36           .type lj_BC_ISLE, @function
37           .size lj_BC_ISLE, 72
38  lj_BC_ISLE:
39           .byte 129,124,202,4,255,255,254,255,15,131,175,28,0,0,129,124
40           .byte 194,4,255,255,254,255,15,131,161,28,0,0,242,15,16,4
41           .byte 194,131,195,4,102,15,46,4,202,114,11,15,183,67,254,141
42           .byte 156,131,0,0,254,255,139,3,15,182,204,15,182,232,131,195
43           .byte 4,193,232,16,65,255,36,238
44
45           .globl lj_BC_ISGT
46           .hidden lj_BC_ISGT
47           .type lj_BC_ISGT, @function
48           .size lj_BC_ISGT, 72
49  lj_BC_ISGT:
50           .byte 129,124,202,4,255,255,254,255,15,131,103,28,0,0,129,124
51           .byte 194,4,255,255,254,255,15,131,89,28,0,0,242,15,16,4
52           .byte 194,131,195,4,102,15,46,4,202,115,11,15,183,67,254,141
53           .byte 156,131,0,0,254,255,139,3,15,182,204,15,182,232,131,195
54           .byte 4,193,232,16,65,255,36,238
55
56           .globl lj_BC_ISEQV
57           .hidden lj_BC_ISEQV
58           .type lj_BC_ISEQV, @function
59           .size lj_BC_ISEQV, 139
60  lj_BC_ISEQV:
61           .byte 139,108,194,4,131,195,4,129,253,255,255,254,255,115,53,129
62           .byte 124,202,4,255,255,254,255,115,43,242,15,16,4,202,102,15
63           .byte 46,4,194,122,13,117,11,15,183,67,254,141,156,131,0,0
64           .byte 254,255,139,3,15,182,204,15,182,232,131,195,4,193,232,16
65           .byte 65,255,36,238,131,253,245,15,132,88,28,0,0,131,124,202
66           .byte 4,245,15,132,77,28,0,0,57,108,202,4,117,212,131,253
67           .byte 253,115,196,139,12,202,139,4,194,57,193,116,186,131,253,244
68           .byte 119,192,131,253,243,114,187,139,105,16,133,237,116,180,246,69
69           .byte 6,16,117,174,49,237,233,253,27,0,0
70
71           .globl lj_BC_ISNEV
```

```
        .hidden lj_BC_ISNEV
        .type lj_BC_ISNEV, @function
        .size lj_BC_ISNEV, 142
lj_BC_ISNEV:
        .byte 139,108,194,4,131,195,4,129,253,255,255,254,255,115,53,129
        .byte 124,202,4,255,255,254,255,115,43,242,15,16,4,202,102,15
        .byte 46,4,194,122,2,116,11,15,183,67,254,141,156,131,0,0
        .byte 254,255,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238,131,253,245,15,132,205,27,0,0,131,124,202
        .byte 4,245,15,132,194,27,0,0,57,108,202,4,117,201,131,253
        .byte 253,115,207,139,12,202,139,4,194,57,193,116,197,131,253,244
        .byte 119,181,131,253,243,114,176,139,105,16,133,237,116,169,246,69
        .byte 6,16,117,163,189,1,0,0,0,233,111,27,0,0

        .globl lj_BC_ISEQS
        .hidden lj_BC_ISEQS
        .type lj_BC_ISEQS, @function
        .size lj_BC_ISEQS, 63
lj_BC_ISEQS:
        .byte 72,247,208,139,108,202,4,131,195,4,131,253,251,117,38,139
        .byte 12,202,65,59,12,135,117,11,15,183,67,254,141,156,131,0
        .byte 0,254,255,139,3,15,182,204,15,182,232,131,195,4,193,232
        .byte 16,65,255,36,238,131,253,245,117,233,233,77,27,0,0

        .globl lj_BC_ISNES
        .hidden lj_BC_ISNES
        .type lj_BC_ISNES, @function
        .size lj_BC_ISNES, 63
 lj_BC_ISNES:
         .byte 72,247,208,139,108,202,4,131,195,4,131,253,251,117,38,139
         .byte 12,202,65,59,12,135,116,11,15,183,67,254,141,156,131,0
         .byte 0,254,255,139,3,15,182,204,15,182,232,131,195,4,193,232
         .byte 16,65,255,36,238,131,253,245,117,222,233,14,27,0,0

         .globl lj_BC_ISEQN
         .hidden lj_BC_ISEQN
         .type lj_BC_ISEQN, @function
         .size lj_BC_ISEQN, 69
 lj_BC_ISEQN:
         .byte 139,108,202,4,131,195,4,129,253,255,255,254,255,115,44,242
         .byte 65,15,16,4,199,102,15,46,4,202,122,13,117,11,15,183
         .byte 67,254,141,156,131,0,0,254,255,139,3,15,182,204,15,182
         .byte 232,131,195,4,193,232,16,65,255,36,238,131,253,245,117,233
         .byte 233,201,26,0,0

         .globl lj_BC_ISNEN
         .hidden lj_BC_ISNEN
         .type lj_BC_ISNEN, @function
         .size lj_BC_ISNEN, 69
 lj_BC_ISNEN:
         .byte 139,108,202,4,131,195,4,129,253,255,255,254,255,115,44,242
         .byte 65,15,16,4,199,102,15,46,4,202,122,2,116,11,15,183
         .byte 67,254,141,156,131,0,0,254,255,139,3,15,182,204,15,182
         .byte 232,131,195,4,193,232,16,65,255,36,238,131,253,245,117,222
         .byte 233,132,26,0,0

         .globl lj_BC_ISEQP
         .hidden lj_BC_ISEQP
         .type lj_BC_ISEQP, @function
         .size lj_BC_ISEQP, 53
 lj_BC_ISEQP:
         .byte 72,247,208,139,108,202,4,131,195,4,57,197,117,29,15,183
         .byte 67,254,141,156,131,0,0,254,255,139,3,15,182,204,15,182
         .byte 232,131,195,4,193,232,16,65,255,36,238,131,253,245,117,233
         .byte 233,79,26,0,0

         .globl lj_BC_ISNEP
         .hidden lj_BC_ISNEP
         .type lj_BC_ISNEP, @function
         .size lj_BC_ISNEP, 52
 lj_BC_ISNEP:
         .byte 72,247,208,139,108,202,4,131,195,4,57,197,116,20,131,253
         .byte 245,15,132,56,26,0,0,15,183,67,254,141,156,131,0,0
         .byte 254,255,139,3,15,182,204,15,182,232,131,195,4,193,232,16
         .byte 65,255,36,238

```

```
        .globl lj_BC_ISTC
        .hidden lj_BC_ISTC
        .type lj_BC_ISTC, @function
        .size lj_BC_ISTC, 51
lj_BC_ISTC:
        .byte 139,108,194,4,131,195,4,131,253,254,115,21,137,108,202,4
        .byte 139,44,194,137,44,202,15,183,67,254,141,156,131,0,0,254
        .byte 255,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65
        .byte 255,36,238

        .globl lj_BC_ISFC
        .hidden lj_BC_ISFC
        .type lj_BC_ISFC, @function
        .size lj_BC_ISFC, 51
lj_BC_ISFC:
        .byte 139,108,194,4,131,195,4,131,253,254,114,21,137,108,202,4
        .byte 139,44,194,137,44,202,15,183,67,254,141,156,131,0,0,254
        .byte 255,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65
        .byte 255,36,238

        .globl lj_BC_IST
        .hidden lj_BC_IST
        .type lj_BC_IST, @function
        .size lj_BC_IST, 41
lj_BC_IST:
        .byte 139,108,194,4,131,195,4,131,253,254,115,11,15,183,67,254
        .byte 141,156,131,0,0,254,255,139,3,15,182,204,15,182,232,131
        .byte 195,4,193,232,16,65,255,36,238

        .globl lj_BC_ISF
        .hidden lj_BC_ISF
        .type lj_BC_ISF, @function
        .size lj_BC_ISF, 41
lj_BC_ISF:
        .byte 139,108,194,4,131,195,4,131,253,254,114,11,15,183,67,254
        .byte 141,156,131,0,0,254,255,139,3,15,182,204,15,182,232,131
        .byte 195,4,193,232,16,65,255,36,238

        .globl lj_BC_MOV
        .hidden lj_BC_MOV
        .type lj_BC_MOV, @function
        .size lj_BC_MOV, 26
lj_BC_MOV:
        .byte 72,139,44,194,72,137,44,202,139,3,15,182,204,15,182,232
        .byte 131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_NOT
        .hidden lj_BC_NOT
        .type lj_BC_NOT, @function
        .size lj_BC_NOT, 32
lj_BC_NOT:
        .byte 49,237,131,124,194,4,254,131,213,253,137,108,202,4,139,3
        .byte 15,182,204,15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_UNM
        .hidden lj_BC_UNM
        .type lj_BC_UNM, @function
        .size lj_BC_UNM, 60
lj_BC_UNM:
        .byte 129,124,194,4,255,255,254,255,15,131,69,25,0,0,242,15
        .byte 16,4,194,72,184,0,0,0,0,0,0,0,128,102,72,15
        .byte 110,200,15,87,193,242,15,17,4,202,139,3,15,182,204,15
        .byte 182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_LEN
        .hidden lj_BC_LEN
        .type lj_BC_LEN, @function
        .size lj_BC_LEN, 74
lj_BC_LEN:
        .byte 131,124,194,4,251,117,34,139,4,194,15,87,192,242,15,42
        .byte 64,12,242,15,17,4,202,139,3,15,182,204,15,182,232,131
        .byte 195,4,193,232,16,65,255,36,238,131,124,194,4,244,15,133
        .byte 47,25,0,0,139,60,194,137,213,232
        .long lj_tab_len-.-4
        .byte 242,15,42,192,137,234,15,182,75,253,235,200

```

```
        .globl lj_BC_ADDVN
        .hidden lj_BC_ADDVN
        .type lj_BC_ADDVN, @function
        .size lj_BC_ADDVN, 54
lj_BC_ADDVN:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 169,24,0,0,242,15,16,4,234,242,65,15,88,4,199,242
        .byte 15,17,4,202,139,3,15,182,204,15,182,232,131,195,4,193
        .byte 232,16,65,255,36,238

        .globl lj_BC_SUBVN
        .hidden lj_BC_SUBVN
        .type lj_BC_SUBVN, @function
        .size lj_BC_SUBVN, 54
lj_BC_SUBVN:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 115,24,0,0,242,15,16,4,234,242,65,15,92,4,199,242
        .byte 15,17,4,202,139,3,15,182,204,15,182,232,131,195,4,193
        .byte 232,16,65,255,36,238

        .globl lj_BC_MULVN
        .hidden lj_BC_MULVN
        .type lj_BC_MULVN, @function
        .size lj_BC_MULVN, 54
lj_BC_MULVN:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 61,24,0,0,242,15,16,4,234,242,65,15,89,4,199,242
        .byte 15,17,4,202,139,3,15,182,204,15,182,232,131,195,4,193
        .byte 232,16,65,255,36,238

        .globl lj_BC_DIVVN
        .hidden lj_BC_DIVVN
        .type lj_BC_DIVVN, @function
        .size lj_BC_DIVVN, 54
lj_BC_DIVVN:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 7,24,0,0,242,15,16,4,234,242,65,15,94,4,199,242
        .byte 15,17,4,202,139,3,15,182,204,15,182,232,131,195,4,193
        .byte 232,16,65,255,36,238

        .globl lj_BC_MODVN
        .hidden lj_BC_MODVN
        .type lj_BC_MODVN, @function
        .size lj_BC_MODVN, 59
lj_BC_MODVN:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 209,23,0,0,242,15,16,4,234,242,65,15,16,12,199,232
        .byte 237,46,0,0,242,15,17,4,202,139,3,15,182,204,15,182
        .byte 232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_ADDNV
        .hidden lj_BC_ADDNV
        .type lj_BC_ADDNV, @function
        .size lj_BC_ADDNV, 54
lj_BC_ADDNV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 156,23,0,0,242,65,15,16,4,199,242,15,88,4,234,242
        .byte 15,17,4,202,139,3,15,182,204,15,182,232,131,195,4,193
        .byte 232,16,65,255,36,238

        .globl lj_BC_SUBNV
        .hidden lj_BC_SUBNV
        .type lj_BC_SUBNV, @function
        .size lj_BC_SUBNV, 54
lj_BC_SUBNV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 102,23,0,0,242,65,15,16,4,199,242,15,92,4,234,242
        .byte 15,17,4,202,139,3,15,182,204,15,182,232,131,195,4,193
        .byte 232,16,65,255,36,238

        .globl lj_BC_MULNV
        .hidden lj_BC_MULNV
        .type lj_BC_MULNV, @function
        .size lj_BC_MULNV, 54
lj_BC_MULNV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
```

```
        .byte 48,23,0,0,242,65,15,16,4,199,242,15,89,4,234,242
        .byte 15,17,4,202,139,3,15,182,204,15,182,232,131,195,4,193
        .byte 232,16,65,255,36,238

        .globl lj_BC_DIVNV
        .hidden lj_BC_DIVNV
        .type lj_BC_DIVNV, @function
        .size lj_BC_DIVNV, 54
lj_BC_DIVNV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 250,22,0,0,242,65,15,16,4,199,242,15,94,4,234,242
        .byte 15,17,4,202,139,3,15,182,204,15,182,232,131,195,4,193
        .byte 232,16,65,255,36,238

        .globl lj_BC_MODNV
        .hidden lj_BC_MODNV
        .type lj_BC_MODNV, @function
        .size lj_BC_MODNV, 36
lj_BC_MODNV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 196,22,0,0,242,65,15,16,4,199,242,15,16,12,234,233
        .byte 232,254,255,255

        .globl lj_BC_ADDVV
        .hidden lj_BC_ADDVV
        .type lj_BC_ADDVV, @function
        .size lj_BC_ADDVV, 67
lj_BC_ADDVV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 177,22,0,0,129,124,194,4,255,255,254,255,15,131,163,22
        .byte 0,0,242,15,16,4,234,242,15,88,4,194,242,15,17,4
        .byte 202,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65
        .byte 255,36,238

        .globl lj_BC_SUBVV
        .hidden lj_BC_SUBVV
        .type lj_BC_SUBVV, @function
        .size lj_BC_SUBVV, 67
lj_BC_SUBVV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 110,22,0,0,129,124,194,4,255,255,254,255,15,131,96,22
        .byte 0,0,242,15,16,4,234,242,15,92,4,194,242,15,17,4
        .byte 202,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65
        .byte 255,36,238

        .globl lj_BC_MULVV
        .hidden lj_BC_MULVV
        .type lj_BC_MULVV, @function
        .size lj_BC_MULVV, 67
lj_BC_MULVV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 43,22,0,0,129,124,194,4,255,255,254,255,15,131,29,22
        .byte 0,0,242,15,16,4,234,242,15,89,4,194,242,15,17,4
        .byte 202,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65
        .byte 255,36,238

        .globl lj_BC_DIVVV
        .hidden lj_BC_DIVVV
        .type lj_BC_DIVVV, @function
        .size lj_BC_DIVVV, 67
lj_BC_DIVVV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 232,21,0,0,129,124,194,4,255,255,254,255,15,131,218,21
        .byte 0,0,242,15,16,4,234,242,15,94,4,194,242,15,17,4
        .byte 202,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65
        .byte 255,36,238

        .globl lj_BC_MODVV
        .hidden lj_BC_MODVV
        .type lj_BC_MODVV, @function
        .size lj_BC_MODVV, 49
lj_BC_MODVV:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 165,21,0,0,129,124,194,4,255,255,254,255,15,131,151,21
        .byte 0,0,242,15,16,4,234,242,15,16,12,194,233,171,253,255
        .byte 255
```

```
        .globl lj_BC_POW
        .hidden lj_BC_POW
        .type lj_BC_POW, @function
        .size lj_BC_POW, 72
lj_BC_POW:
        .byte 15,182,236,15,182,192,129,124,234,4,255,255,254,255,15,131
        .byte 116,21,0,0,129,124,194,4,255,255,254,255,15,131,102,21
        .byte 0,0,242,15,16,4,234,242,15,16,12,194,232,47,45,0
        .byte 0,242,15,17,4,202,139,3,15,182,204,15,182,232,131,195
        .byte 4,193,232,16,65,255,36,238

        .globl lj_BC_CAT
        .hidden lj_BC_CAT
        .type lj_BC_CAT, @function
        .size lj_BC_CAT, 76
lj_BC_CAT:
        .byte 15,182,236,15,182,192,139,124,36,24,137,87,16,141,52,194
        .byte 137,194,41,234,137,253,137,92,36,28,232
        .long lj_meta_cat-.-4
        .byte 139,85,16,133,192,15,133,71,21,0,0,15,182,107,255,15
        .byte 182,75,253,72,139,4,234,72,137,4,202,139,3,15,182,204
        .byte 15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_KSTR
        .hidden lj_BC_KSTR
        .type lj_BC_KSTR, @function
        .size lj_BC_KSTR, 36
lj_BC_KSTR:
        .byte 72,247,208,65,139,4,135,199,68,202,4,251,255,255,255,137
        .byte 4,202,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238

        .globl lj_BC_KCDATA
        .hidden lj_BC_KCDATA
        .type lj_BC_KCDATA, @function
        .size lj_BC_KCDATA, 36
lj_BC_KCDATA:
        .byte 72,247,208,65,139,4,135,199,68,202,4,245,255,255,255,137
        .byte 4,202,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238

        .globl lj_BC_KSHORT
        .hidden lj_BC_KSHORT
        .type lj_BC_KSHORT, @function
        .size lj_BC_KSHORT, 30
lj_BC_KSHORT:
        .byte 15,191,192,242,15,42,192,242,15,17,4,202,139,3,15,182
        .byte 204,15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_KNUM
        .hidden lj_BC_KNUM
        .type lj_BC_KNUM, @function
        .size lj_BC_KNUM, 29
lj_BC_KNUM:
        .byte 242,65,15,16,4,199,242,15,17,4,202,139,3,15,182,204
        .byte 15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_KPRI
        .hidden lj_BC_KPRI
        .type lj_BC_KPRI, @function
        .size lj_BC_KPRI, 25
lj_BC_KPRI:
        .byte 72,247,208,137,68,202,4,139,3,15,182,204,15,182,232,131
        .byte 195,4,193,232,16,65,255,36,238

        .globl lj_BC_KNIL
        .hidden lj_BC_KNIL
        .type lj_BC_KNIL, @function
        .size lj_BC_KNIL, 43
lj_BC_KNIL:
        .byte 141,76,202,12,141,68,194,4,189,255,255,255,255,137,105,248
        .byte 137,41,131,193,8,57,193,118,247,139,3,15,182,204,15,182
        .byte 232,131,195,4,193,232,16,65,255,36,238

        .globl lj_BC_UGET
```

```
        .hidden lj_BC_UGET
        .type lj_BC_UGET, @function
        .size lj_BC_UGET, 36
lj_BC_UGET:
        .byte 139,106,248,139,108,133,20,139,109,16,72,139,69,0,72,137
        .byte 4,202,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238

        .globl lj_BC_USETV
        .hidden lj_BC_USETV
        .type lj_BC_USETV, @function
        .size lj_BC_USETV, 87
lj_BC_USETV:
        .byte 139,106,248,139,108,141,20,128,125,6,0,139,109,16,139,12
        .byte 194,139,68,194,4,137,77,0,137,69,4,116,6,246,69,252
        .byte 4,117,18,139,3,15,182,204,15,182,232,131,195,4,193,232
        .byte 16,65,255,36,238,131,232,252,131,248,246,118,230,246,65,4
        .byte 3,116,224,137,238,137,213,65,141,190,72,244,255,255,232
        .long lj_gc_barrieruv-.-4
        .byte 137,234,235,204

        .globl lj_BC_USETS
        .hidden lj_BC_USETS
        .type lj_BC_USETS, @function
        .size lj_BC_USETS, 82
lj_BC_USETS:
        .byte 72,247,208,139,106,248,139,108,141,20,65,139,12,135,139,69
        .byte 16,137,8,199,64,4,251,255,255,255,246,69,4,4,117,18
        .byte 139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,255
        .byte 36,238,246,65,4,3,116,232,128,125,6,0,116,226,137,213
        .byte 137,198,65,141,190,72,244,255,255,232
        .long lj_gc_barrieruv-.-4
        .byte 137,234,235,206

        .globl lj_BC_USETN
        .hidden lj_BC_USETN
        .type lj_BC_USETN, @function
        .size lj_BC_USETN, 38
lj_BC_USETN:
        .byte 139,106,248,242,65,15,16,4,199,139,108,141,20,139,77,16
        .byte 242,15,17,1,139,3,15,182,204,15,182,232,131,195,4,193
        .byte 232,16,65,255,36,238

        .globl lj_BC_USETP
        .hidden lj_BC_USETP
        .type lj_BC_USETP, @function
        .size lj_BC_USETP, 34
lj_BC_USETP:
        .byte 72,247,208,139,106,248,139,108,141,20,139,77,16,137,65,4
        .byte 139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,255
        .byte 36,238

        .globl lj_BC_UCLO
        .hidden lj_BC_UCLO
        .type lj_BC_UCLO, @function
        .size lj_BC_UCLO, 51
lj_BC_UCLO:
        .byte 141,156,131,0,0,254,255,139,108,36,24,131,125,40,0,116
        .byte 16,137,85,16,141,52,202,137,239,232
        .long lj_func_closeuv-.-4
        .byte 139,85,16,139,3,15,182,204,15,182,232,131,195,4,193,232
        .byte 16,65,255,36,238

        .globl lj_BC_FNEW
        .hidden lj_BC_FNEW
        .type lj_BC_FNEW, @function
        .size lj_BC_FNEW, 64
lj_BC_FNEW:
        .byte 72,247,208,139,108,36,24,137,85,16,139,82,248,65,139,52
        .byte 135,137,239,137,92,36,28,232
        .long lj_func_newL_gc-.-4
        .byte 139,85,16,15,182,75,253,137,4,202,199,68,202,4,247,255
        .byte 255,255,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238

        .globl lj_BC_TNEW
```

```
        .hidden lj_BC_TNEW
        .type lj_BC_TNEW, @function
        .size lj_BC_TNEW, 109
lj_BC_TNEW:
        .byte 139,108,36,24,137,85,16,65,139,142,104,244,255,255,65,59
        .byte 142,108,244,255,255,137,92,36,28,115,69,137,194,37,255,7
        .byte 0,0,193,234,11,61,255,7,0,0,116,45,137,239,137,198
        .byte 232
        .long lj_tab_new-.-4
        .byte 139,85,16,15,182,75,253,137,4,202,199,68,202,4,244,255
        .byte 255,255,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238,184,1,8,0,0,235,204,137,239,232
        .long lj_gc_step_fixtop-.-4
        .byte 15,183,67,254,235,174

        .globl lj_BC_TDUP
        .hidden lj_BC_TDUP
        .type lj_BC_TDUP, @function
        .size lj_BC_TDUP, 93
lj_BC_TDUP:
        .byte 72,247,208,139,108,36,24,65,139,142,104,244,255,255,137,92
        .byte 36,28,65,59,142,108,244,255,255,137,85,16,115,47,65,139
        .byte 52,135,137,239,232
        .long lj_tab_dup-.-4
        .byte 139,85,16,15,182,75,253,137,4,202,199,68,202,4,244,255
        .byte 255,255,139,3,15,182,204,15,182,232,131,195,4,193,232,16
        .byte 65,255,36,238,137,239,232
        .long lj_gc_step_fixtop-.-4
        .byte 15,183,67,254,72,247,208,235,193

        .globl lj_BC_GGET
        .hidden lj_BC_GGET
        .type lj_BC_GGET, @function
        .size lj_BC_GGET, 18
lj_BC_GGET:
        .byte 72,247,208,139,106,248,139,109,8,65,139,4,135,233,193,0
        .byte 0,0

        .globl lj_BC_GSET
        .hidden lj_BC_GSET
        .type lj_BC_GSET, @function
        .size lj_BC_GSET, 18
lj_BC_GSET:
        .byte 72,247,208,139,106,248,139,109,8,65,139,4,135,233,59,2
        .byte 0,0

        .globl lj_BC_TGETV
        .hidden lj_BC_TGETV
        .type lj_BC_TGETV, @function
        .size lj_BC_TGETV, 148
lj_BC_TGETV:
        .byte 15,182,236,15,182,192,131,124,234,4,244,15,133,243,15,0
        .byte 0,139,44,234,129,124,194,4,255,255,254,255,115,102,242,15
        .byte 16,4,194,242,15,45,192,242,15,42,200,102,15,46,193,15
        .byte 133,207,15,0,0,59,69,24,15,131,198,15,0,0,193,224
        .byte 3,3,69,8,131,120,4,255,116,25,72,139,40,72,137,44
        .byte 202,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65
        .byte 255,36,238,131,125,16,0,116,17,139,77,16,246,65,6,1
        .byte 15,132,142,15,0,0,15,182,75,253,199,68,202,4,255,255
        .byte 255,255,235,205,131,124,194,4,251,15,133,117,15,0,0,139
        .byte 4,194,235,27

        .globl lj_BC_TGETS
        .hidden lj_BC_TGETS
        .type lj_BC_TGETS, @function
        .size lj_BC_TGETS, 124
lj_BC_TGETS:
        .byte 15,182,236,15,182,192,72,247,208,65,139,4,135,131,124,234
        .byte 4,244,15,133,28,15,0,0,139,44,234,139,77,28,35,72
        .byte 8,107,201,24,3,77,20,131,121,12,251,117,54,57,65,8
        .byte 117,49,131,121,4,255,116,50,15,182,67,253,72,139,41,72
        .byte 137,44,194,139,3,15,182,204,15,182,232,131,195,4,193,232
        .byte 16,65,255,36,238,15,182,67,253,199,68,194,4,255,255,255
        .byte 255,235,224,139,73,16,133,201,117,189,139,77,16,133,201,116
        .byte 228,246,65,6,1,117,222,233,184,14,0,0

```

```
        .globl lj_BC_TGETB
        .hidden lj_BC_TGETB
        .type lj_BC_TGETB, @function
        .size lj_BC_TGETB, 99
lj_BC_TGETB:
        .byte 15,182,236,15,182,192,131,124,234,4,244,15,133,208,14,0
        .byte 0,139,44,234,59,69,24,15,131,196,14,0,0,193,224,3
        .byte 3,69,8,131,120,4,255,116,25,72,139,40,72,137,44,202
        .byte 139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,255
        .byte 36,238,131,125,16,0,116,17,139,77,16,246,65,6,1,15
        .byte 132,140,14,0,0,15,182,75,253,199,68,202,4,255,255,255
        .byte 255,235,205

        .globl lj_BC_TSETV
        .hidden lj_BC_TSETV
        .type lj_BC_TSETV, @function
        .size lj_BC_TSETV, 173
lj_BC_TSETV:
        .byte 15,182,236,15,182,192,131,124,234,4,244,15,133,29,15,0
        .byte 0,139,44,234,129,124,194,4,255,255,254,255,115,100,242,15
        .byte 16,4,194,242,15,45,192,242,15,42,200,102,15,46,193,15
        .byte 133,249,14,0,0,59,69,24,15,131,240,14,0,0,193,224
        .byte 3,3,69,8,131,120,4,255,116,31,246,69,4,4,117,66
        .byte 72,139,44,202,72,137,40,139,3,15,182,204,15,182,232,131
        .byte 195,4,193,232,16,65,255,36,238,131,125,16,0,116,219,139
        .byte 77,16,246,65,6,2,15,132,178,14,0,0,15,182,75,253
        .byte 235,200,131,124,194,4,251,15,133,161,14,0,0,139,4,194
        .byte 235,54,128,101,4,251,65,139,142,132,244,255,255,65,137,174
        .byte 132,244,255,255,137,77,12,15,182,75,253,235,163

        .globl lj_BC_TSETS
        .hidden lj_BC_TSETS
        .type lj_BC_TSETS, @function
        .size lj_BC_TSETS, 229
lj_BC_TSETS:
        .byte 15,182,236,15,182,192,72,247,208,65,139,4,135,131,124,234
        .byte 4,244,15,133,45,14,0,0,139,44,234,139,77,28,35,72
        .byte 8,107,201,24,198,69,6,0,3,77,20,131,121,12,251,117
        .byte 77,57,65,8,117,72,131,121,4,255,116,39,246,69,4,4
        .byte 15,133,133,0,0,0,15,182,67,253,72,139,44,194,72,137
        .byte 41,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65
        .byte 255,36,238,131,125,16,0,116,211,137,12,36,139,77,16,246
        .byte 65,6,2,15,132,204,13,0,0,139,12,36,235,190,139,73
        .byte 16,133,201,117,166,139,77,16,133,201,116,10,246,65,6,2
        .byte 15,132,175,13,0,0,137,4,36,199,68,36,4,251,255,255
        .byte 255,137,108,36,8,139,124,36,24,137,87,16,72,141,20,36
        .byte 137,238,137,253,137,92,36,28,232
        .long lj_tab_newkey-.-4
        .byte 139,85,16,139,108,36,8,137,193,233,113,255,255,255,128,101
        .byte 4,251,65,139,134,132,244,255,255,65,137,174,132,244,255,255
        .byte 137,69,12,233,97,255,255,255

        .globl lj_BC_TSETB
        .hidden lj_BC_TSETB
        .type lj_BC_TSETB, @function
        .size lj_BC_TSETB, 124
lj_BC_TSETB:
        .byte 15,182,236,15,182,192,131,124,234,4,244,15,133,120,13,0
        .byte 0,139,44,234,59,69,24,15,131,108,13,0,0,193,224,3
        .byte 3,69,8,131,120,4,255,116,31,246,69,4,4,117,50,72
        .byte 139,12,202,72,137,8,139,3,15,182,204,15,182,232,131,195
        .byte 4,193,232,16,65,255,36,238,131,125,16,0,116,219,139,77
        .byte 16,246,65,6,2,15,132,46,13,0,0,15,182,75,253,235
        .byte 200,128,101,4,251,65,139,142,132,244,255,255,65,137,174,132
        .byte 244,255,255,137,77,12,15,182,75,253,235,179

        .globl lj_BC_TSETM
        .hidden lj_BC_TSETM
        .type lj_BC_TSETM, @function
        .size lj_BC_TSETM, 142
lj_BC_TSETM:
        .byte 68,137,60,36,69,139,60,199,141,12,202,139,105,248,246,69
        .byte 4,4,117,99,139,68,36,4,131,232,1,116,37,68,1,248
        .byte 59,69,24,119,51,68,41,248,65,193,231,3,68,3,125,8
        .byte 72,139,41,131,193,8,73,137,47,65,131,199,8,131,232,1
        .byte 117,238,68,139,60,36,139,3,15,182,204,15,182,232,131,195
```

```
        .byte 4,193,232,16,65,255,36,238,139,124,36,24,137,87,16,137
        .byte 238,137,194,137,253,137,92,36,28,232
        .long lj_tab_reasize-.-4
        .byte 139,85,16,15,182,75,253,235,145,128,101,4,251,65,139,134
        .byte 132,244,255,255,65,137,174,132,244,255,255,137,69,12,235,134

        .globl lj_BC_CALLM
        .hidden lj_BC_CALLM
        .type lj_BC_CALLM, @function
        .size lj_BC_CALLM, 46
lj_BC_CALLM:
        .byte 15,182,192,3,68,36,4,131,124,202,4,247,139,44,202,15
        .byte 133,242,13,0,0,141,84,202,8,137,90,252,139,93,16,139
        .byte 11,15,182,233,15,182,205,131,195,4,65,255,36,238

        .globl lj_BC_CALL
        .hidden lj_BC_CALL
        .type lj_BC_CALL, @function
        .size lj_BC_CALL, 42
lj_BC_CALL:
        .byte 15,182,192,131,124,202,4,247,139,44,202,15,133,200,13,0
        .byte 0,141,84,202,8,137,90,252,139,93,16,139,11,15,182,233
        .byte 15,182,205,131,195,4,65,255,36,238

        .globl lj_BC_CALLMT
        .hidden lj_BC_CALLMT
        .type lj_BC_CALLMT, @function
        .size lj_BC_CALLMT, 4
lj_BC_CALLMT:
        .byte 3,68,36,4

        .globl lj_BC_CALLT
        .hidden lj_BC_CALLT
        .type lj_BC_CALLT, @function
        .size lj_BC_CALLT, 148
lj_BC_CALLT:
        .byte 141,76,202,8,65,137,215,139,105,248,131,121,252,247,15,133
        .byte 155,13,0,0,139,90,252,247,195,3,0,0,0,117,91,137
        .byte 106,248,137,68,36,4,131,232,1,116,21,72,139,41,131,193
        .byte 8,73,137,47,65,131,199,8,131,232,1,117,238,139,106,248
        .byte 139,68,36,4,128,125,6,1,119,18,139,93,16,139,11,15
        .byte 182,233,15,182,205,131,195,4,65,255,36,238,247,195,3,0
        .byte 0,0,117,230,15,182,75,253,72,247,209,68,139,124,202,248
        .byte 69,139,127,16,69,139,127,208,235,208,131,235,3,247,195,7
        .byte 0,0,0,117,10,41,218,65,137,215,139,90,252,235,144,131
        .byte 195,3,235,139

        .globl lj_BC_ITERC
        .hidden lj_BC_ITERC
        .type lj_BC_ITERC, @function
        .size lj_BC_ITERC, 68
lj_BC_ITERC:
        .byte 141,76,202,8,72,139,105,232,72,139,65,240,72,137,41,72
        .byte 137,65,8,139,105,224,139,65,228,137,105,248,137,65,252,131
        .byte 248,247,184,3,0,0,0,15,133,238,12,0,0,137,202,137
        .byte 90,252,139,93,16,139,11,15,182,233,15,182,205,131,195,4
        .byte 65,255,36,238

        .globl lj_BC_ITERN
        .hidden lj_BC_ITERN
        .type lj_BC_ITERN, @function
        .size lj_BC_ITERN, 165
lj_BC_ITERN:
        .byte 68,137,60,36,68,137,116,36,4,139,108,202,240,139,68,202
        .byte 248,68,139,117,24,131,195,4,68,139,125,8,68,57,240,115
        .byte 76,65,131,124,199,4,255,116,63,242,15,42,192,73,139,44
        .byte 199,72,137,108,202,8,131,192,1,242,15,17,4,202,137,68
        .byte 202,248,15,183,67,254,141,156,131,0,0,254,255,68,139,116
        .byte 36,4,68,139,60,36,139,3,15,182,204,15,182,232,131,195
        .byte 4,193,232,16,65,255,36,238,131,192,1,235,175,68,41,240
        .byte 59,69,28,119,216,68,107,248,24,68,3,125,20,65,131,127
        .byte 4,255,116,28,70,141,116,48,1,73,139,111,8,73,139,7
        .byte 72,137,44,202,72,137,68,202,8,68,137,116,202,248,235,162
        .byte 131,192,1,235,203

        .globl lj_BC_VARG
```

```
        .hidden lj_BC_VARG
        .type lj_BC_VARG, @function
        .size lj_BC_VARG, 191
lj_BC_VARG:
        .byte 15,182,236,15,182,192,68,137,60,36,68,141,124,194,11,141
        .byte 12,202,68,43,122,252,133,237,116,68,141,108,233,248,65,57
        .byte 215,115,23,73,139,71,248,65,131,199,8,72,137,1,131,193
        .byte 8,57,233,115,19,65,57,215,114,233,199,65,4,255,255,255
        .byte 255,131,193,8,57,233,114,242,68,139,60,36,139,3,15,182
        .byte 204,15,182,232,131,195,4,193,232,16,65,255,36,238,199,68
        .byte 36,4,1,0,0,0,137,208,68,41,248,118,219,137,197,193
        .byte 237,3,131,197,1,137,108,36,4,139,108,36,24,1,200,59
        .byte 69,32,119,21,73,139,71,248,65,131,199,8,72,137,1,131
        .byte 193,8,65,57,215,114,237,235,175,137,85,16,137,77,24,137
        .byte 92,36,28,65,41,215,139,116,36,4,131,238,1,137,239,232
        .long lj_state_growstack-.-4
        .byte 139,85,16,139,77,24,65,1,215,235,197

        .globl lj_BC_ISNEXT
        .hidden lj_BC_ISNEXT
        .type lj_BC_ISNEXT, @function
        .size lj_BC_ISNEXT, 88
lj_BC_ISNEXT:
        .byte 131,124,202,236,247,117,65,139,108,202,232,131,124,202,244,244
        .byte 117,54,131,124,202,252,255,117,47,128,125,6,4,117,41,141
        .byte 156,131,0,0,254,255,199,68,202,248,0,0,0,0,199,68
        .byte 202,252,255,127,254,255,139,3,15,182,204,15,182,232,131,195
        .byte 4,193,232,16,65,255,36,238,198,67,252,84,141,156,131,0
        .byte 0,254,255,198,3,65,235,222

        .globl lj_BC_RETM
        .hidden lj_BC_RETM
        .type lj_BC_RETM, @function
        .size lj_BC_RETM, 4
lj_BC_RETM:
        .byte 3,68,36,4

        .globl lj_BC_RET
        .hidden lj_BC_RET
        .type lj_BC_RET, @function
        .size lj_BC_RET, 136
lj_BC_RET:
        .byte 193,225,3,139,90,252,137,68,36,4,247,195,3,0,0,0
        .byte 117,94,65,137,215,131,232,1,116,17,73,139,44,15,73,137
        .byte 111,248,65,131,199,8,131,232,1,117,239,139,68,36,4,15
        .byte 182,107,255,57,197,119,40,15,182,75,253,72,247,209,141,20
        .byte 202,68,139,122,248,69,139,127,16,69,139,127,208,139,3,15
        .byte 182,204,15,182,232,131,195,4,193,232,16,65,255,36,238,65
        .byte 199,71,252,255,255,255,255,65,131,199,8,131,192,1,235,195
        .byte 141,107,253,247,197,7,0,0,0,15,133,36,5,0,0,41
        .byte 234,1,233,233,123,255,255,255

        .globl lj_BC_RET0
        .hidden lj_BC_RET0
        .type lj_BC_RET0, @function
        .size lj_BC_RET0, 92
lj_BC_RET0:
        .byte 139,90,252,137,68,36,4,247,195,3,0,0,0,117,58,56
        .byte 67,255,119,40,15,182,75,253,72,247,209,141,20,202,68,139
        .byte 122,248,69,139,127,16,69,139,127,208,139,3,15,182,204,15
        .byte 182,232,131,195,4,193,232,16,65,255,36,238,199,68,194,244
        .byte 255,255,255,255,131,192,1,235,198,141,107,253,247,197,7,0
        .byte 0,0,15,133,195,4,0,0,41,234,235,164

        .globl lj_BC_RET1
        .hidden lj_BC_RET1
        .type lj_BC_RET1, @function
        .size lj_BC_RET1, 105
lj_BC_RET1:
        .byte 193,225,3,139,90,252,137,68,36,4,247,195,3,0,0,0
        .byte 117,66,72,139,44,10,72,137,106,248,56,67,255,119,40,15
        .byte 182,75,253,72,247,209,141,20,202,68,139,122,248,69,139,127
        .byte 16,69,139,127,208,139,3,15,182,204,15,182,232,131,195,4
        .byte 193,232,16,65,255,36,238,199,68,194,244,255,255,255,255,131
        .byte 192,1,235,198,141,107,253,247,197,7,0,0,0,15,133,92
        .byte 4,0,0,41,234,1,233,235,154
```

```
        .globl lj_BC_FORI
        .hidden lj_BC_FORI
        .type lj_BC_FORI, @function
        .size lj_BC_FORI, 97
lj_BC_FORI:
        .byte 141,12,202,129,121,4,255,255,254,255,15,131,11,10,0,0
        .byte 129,121,12,255,255,254,255,15,131,254,9,0,0,139,105,20
        .byte 129,253,255,255,254,255,15,131,239,9,0,0,242,15,16,1
        .byte 242,15,16,73,8,124,36,102,15,46,200,242,15,17,65,24
        .byte 115,7,141,156,131,0,0,254,255,139,3,15,182,204,15,182
        .byte 232,131,195,4,193,232,16,65,255,36,238,102,15,46,193,235
        .byte 218

        .globl lj_BC_JFORI
        .hidden lj_BC_JFORI
        .type lj_BC_JFORI, @function
        .size lj_BC_JFORI, 105
lj_BC_JFORI:
        .byte 141,12,202,129,121,4,255,255,254,255,15,131,170,9,0,0
        .byte 129,121,12,255,255,254,255,15,131,157,9,0,0,139,105,20
        .byte 129,253,255,255,254,255,15,131,142,9,0,0,242,15,16,1
        .byte 242,15,16,73,8,124,44,102,15,46,200,242,15,17,65,24
        .byte 141,156,131,0,0,254,255,15,183,67,254,15,131,118,1,0
        .byte 0,139,3,15,182,204,15,182,232,131,195,4,193,232,16,65
        .byte 255,36,238,102,15,46,193,235,210

        .globl lj_BC_FORL
        .hidden lj_BC_FORL
        .type lj_BC_FORL, @function
        .size lj_BC_FORL, 20
lj_BC_FORL:
        .byte 137,221,209,237,131,229,126,102,65,131,108,46,128,2,15,130
        .byte 187,28,0,0

        .globl lj_BC_IFORL
        .hidden lj_BC_IFORL
        .type lj_BC_IFORL, @function
        .size lj_BC_IFORL, 96
lj_BC_IFORL:
        .byte 141,12,202,129,121,12,255,255,254,255,15,131,222,34,0,0
        .byte 129,121,20,255,255,254,255,15,131,209,34,0,0,139,105,20
        .byte 242,15,16,1,242,15,16,73,8,242,15,88,65,16,242,15
        .byte 17,1,133,237,120,36,102,15,46,200,242,15,17,65,24,114
        .byte 7,141,156,131,0,0,254,255,139,3,15,182,204,15,182,232
        .byte 131,195,4,193,232,16,65,255,36,238,102,15,46,193,235,218

        .globl lj_BC_JFORL
        .hidden lj_BC_JFORL
        .type lj_BC_JFORL, @function
        .size lj_BC_JFORL, 93
lj_BC_JFORL:
        .byte 141,12,202,129,121,12,255,255,254,255,15,131,126,34,0,0
        .byte 129,121,20,255,255,254,255,15,131,113,34,0,0,139,105,20
        .byte 242,15,16,1,242,15,16,73,8,242,15,88,65,16,242,15
        .byte 17,1,133,237,120,33,102,15,46,200,242,15,17,65,24,15
        .byte 131,165,0,0,0,139,3,15,182,204,15,182,232,131,195,4
        .byte 193,232,16,65,255,36,238,102,15,46,193,235,221

        .globl lj_BC_ITERL
        .hidden lj_BC_ITERL
        .type lj_BC_ITERL, @function
        .size lj_BC_ITERL, 20
lj_BC_ITERL:
        .byte 137,221,209,237,131,229,126,102,65,131,108,46,128,2,15,130
        .byte 234,27,0,0

        .globl lj_BC_IITERL
        .hidden lj_BC_IITERL
        .type lj_BC_IITERL, @function
        .size lj_BC_IITERL, 44
lj_BC_IITERL:
        .byte 141,12,202,139,105,4,131,253,255,116,15,141,156,131,0,0
        .byte 254,255,139,1,137,105,252,137,65,248,139,3,15,182,204,15
        .byte 182,232,131,195,4,193,232,16,65,255,36,238
```

```
        .globl lj_BC_JITERL
        .hidden lj_BC_JITERL
        .type lj_BC_JITERL, @function
        .size lj_BC_JITERL, 39
lj_BC_JITERL:
        .byte 141,12,202,139,105,4,131,253,255,116,10,137,105,252,139,41
        .byte 137,105,248,235,56,139,3,15,182,204,15,182,232,131,195,4
        .byte 193,232,16,65,255,36,238

        .globl lj_BC_LOOP
        .hidden lj_BC_LOOP
        .type lj_BC_LOOP, @function
        .size lj_BC_LOOP, 20
lj_BC_LOOP:
        .byte 137,221,209,237,131,229,126,102,65,131,108,46,128,2,15,130
        .byte 131,27,0,0

        .globl lj_BC_ILOOP
        .hidden lj_BC_ILOOP
        .type lj_BC_ILOOP, @function
        .size lj_BC_ILOOP, 18
lj_BC_ILOOP:
        .byte 139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,255
        .byte 36,238

        .globl lj_BC_JLOOP
        .hidden lj_BC_JLOOP
        .type lj_BC_JLOOP, @function
        .size lj_BC_JLOOP, 47
lj_BC_JLOOP:
        .byte 65,139,142,16,247,255,255,139,4,129,72,139,64,64,139,108
        .byte 36,24,65,137,150,64,245,255,255,65,137,174,60,245,255,255
        .byte 76,137,36,36,76,137,108,36,8,72,131,236,16,255,224

        .globl lj_BC_JMP
        .hidden lj_BC_JMP
        .type lj_BC_JMP, @function
        .size lj_BC_JMP, 25
lj_BC_JMP:
        .byte 141,156,131,0,0,254,255,139,3,15,182,204,15,182,232,131
        .byte 195,4,193,232,16,65,255,36,238

        .globl lj_BC_FUNCF
        .hidden lj_BC_FUNCF
        .type lj_BC_FUNCF, @function
        .size lj_BC_FUNCF, 20
lj_BC_FUNCF:
        .byte 137,221,209,237,131,229,126,102,65,131,108,46,128,1,15,130
        .byte 77,27,0,0

        .globl lj_BC_IFUNCF
        .hidden lj_BC_IFUNCF
        .type lj_BC_IFUNCF, @function
        .size lj_BC_IFUNCF, 63
lj_BC_IFUNCF:
        .byte 68,139,123,204,139,108,36,24,141,12,202,59,77,32,15,135
        .byte 209,2,0,0,15,182,75,194,57,200,118,18,139,3,15,182
        .byte 204,15,182,232,131,195,4,193,232,16,65,255,36,238,199,68
        .byte 194,252,255,255,255,255,131,192,1,57,200,118,241,235,221

        .globl lj_BC_JFUNCF
        .hidden lj_BC_JFUNCF
        .type lj_BC_JFUNCF, @function
        .size lj_BC_JFUNCF, 54
lj_BC_JFUNCF:
        .byte 68,139,123,204,139,108,36,24,141,12,202,59,77,32,15,135
        .byte 146,2,0,0,15,182,75,194,57,200,118,9,15,183,67,254
        .byte 233,64,255,255,255,199,68,194,252,255,255,255,255,131,192,1
        .byte 57,200,118,241,235,230

        .globl lj_BC_FUNCV
        .hidden lj_BC_FUNCV
        .type lj_BC_FUNCV, @function
        .size lj_BC_FUNCV, 0
lj_BC_FUNCV:

```

```
        .globl lj_BC_IFUNCV
        .hidden lj_BC_IFUNCV
        .type lj_BC_IFUNCV, @function
        .size lj_BC_IFUNCV, 125
lj_BC_IFUNCV:
        .byte 141,44,197,3,0,0,0,141,4,194,68,139,122,248,137,104
        .byte 252,68,137,120,248,139,108,36,24,141,12,200,59,77,32,15
        .byte 135,70,2,0,0,137,209,137,194,15,182,107,194,133,237,116
        .byte 37,131,193,8,57,209,115,52,68,139,121,248,68,137,56,68
        .byte 139,121,252,68,137,120,4,131,192,8,199,65,252,255,255,255
        .byte 255,131,237,1,117,219,68,139,123,204,139,3,15,182,204,15
        .byte 182,232,131,195,4,193,232,16,65,255,36,238,199,64,4,255
        .byte 255,255,255,131,192,8,131,237,1,117,241,235,217

        .globl lj_BC_JFUNCV
        .hidden lj_BC_JFUNCV
         .type lj_BC_JFUNCV, @function
         .size lj_BC_JFUNCV, 1
lj_BC_JFUNCV:
         .byte 204

         .globl lj_BC_FUNCC
         .hidden lj_BC_FUNCC
         .type lj_BC_FUNCC, @function
         .size lj_BC_FUNCC, 79
lj_BC_FUNCC:
         .byte 139,106,248,76,139,125,24,139,108,36,24,141,68,194,248,137
         .byte 85,16,141,136,160,0,0,0,59,77,32,137,69,24,137,239
         .byte 15,135,192,1,0,0,65,199,134,48,245,255,255,254,255,255
         .byte 255,65,255,215,65,199,134,48,245,255,255,255,255,255,255,139
         .byte 85,16,141,12,194,247,217,3,77,24,139,90,252,235,119

         .globl lj_BC_FUNCCW
         .hidden lj_BC_FUNCCW
         .type lj_BC_FUNCCW, @function
         .size lj_BC_FUNCCW, 86
lj_BC_FUNCCW:
         .byte 139,106,248,76,139,125,24,139,108,36,24,141,68,194,248,137
         .byte 85,16,141,136,160,0,0,0,59,77,32,137,69,24,76,137
         .byte 254,137,239,15,135,110,1,0,0,65,199,134,48,245,255,255
         .byte 254,255,255,255,65,255,150,32,245,255,255,65,199,134,48,245
         .byte 255,255,255,255,255,255,139,85,16,141,12,194,247,217,3,77
         .byte 24,139,90,252,235,33

         .globl lj_vm_returnp
         .hidden lj_vm_returnp
         .type lj_vm_returnp, @function
         .size lj_vm_returnp, 33
lj_vm_returnp:
         .byte 247,195,4,0,0,0,15,132,216,2,0,0,131,227,248,41
         .byte 218,72,141,76,25,248,139,90,252,199,68,10,4,253,255,255
         .byte 255

         .globl lj_vm_returnc
         .hidden lj_vm_returnc
         .type lj_vm_returnc, @function
         .size lj_vm_returnc, 25
lj_vm_returnc:
         .byte 131,192,1,15,132,167,0,0,0,137,68,36,4,247,195,3
         .byte 0,0,0,15,132,111,250,255,255

         .globl lj_vm_return
         .hidden lj_vm_return
         .type lj_vm_return, @function
         .size lj_vm_return, 75
lj_vm_return:
         .byte 131,243,1,247,195,3,0,0,0,117,187,65,199,134,48,245
         .byte 255,255,254,255,255,255,131,227,248,41,211,247,219,131,232,1
         .byte 116,16,72,139,44,10,72,137,106,248,131,194,8,131,232,1
         .byte 117,240,139,108,36,24,137,93,16,139,68,36,4,139,76,36
         .byte 16,57,193,117,28,131,234,8,137,85,24

         .globl lj_vm_leave_cp
         .hidden lj_vm_leave_cp
         .type lj_vm_leave_cp, @function
         .size lj_vm_leave_cp, 11
```

```
lj_vm_leave_cp:
        .byte 72,139,76,36,32,72,137,77,48,49,192

        .globl lj_vm_leave_unw
        .hidden lj_vm_leave_unw
        .type lj_vm_leave_unw, @function
        .size lj_vm_leave_unw, 65
lj_vm_leave_unw:
        .byte 72,131,196,40,65,94,65,95,91,93,195,114,20,59,85,32
        .byte 119,26,199,66,252,255,255,255,255,131,194,8,131,192,1,235
        .byte 202,133,201,116,202,41,193,141,20,202,235,195,137,85,24,137
        .byte 68,36,4,137,206,137,239,232
        .long lj_state_growstack-.-4
        .byte 139,85,24,235,162

        .globl lj_vm_unwind_yield
        .hidden lj_vm_unwind_yield
        .type lj_vm_unwind_yield, @function
        .size lj_vm_unwind_yield, 4
lj_vm_unwind_yield:
        .byte 176,1,235,5

        .globl lj_vm_unwind_c
        .hidden lj_vm_unwind_c
        .type lj_vm_unwind_c, @function
        .size lj_vm_unwind_c, 5
lj_vm_unwind_c:
        .byte 137,240,72,137,252

        .globl lj_vm_unwind_c_eh
        .hidden lj_vm_unwind_c_eh
        .type lj_vm_unwind_c_eh, @function
        .size lj_vm_unwind_c_eh, 19
lj_vm_unwind_c_eh:
        .byte 139,108,36,24,139,109,8,199,133,232,0,0,0,254,255,255
        .byte 255,235,163

        .globl lj_vm_unwind_rethrow
        .hidden lj_vm_unwind_rethrow
        .type lj_vm_unwind_rethrow, @function
        .size lj_vm_unwind_rethrow, 21
lj_vm_unwind_rethrow:
        .byte 139,124,36,24,137,198,72,131,196,40,65,94,65,95,91,93
        .byte 233
        .long lj_err_throw-.-4

        .globl lj_vm_unwind_ff
        .hidden lj_vm_unwind_ff
        .type lj_vm_unwind_ff, @function
        .size lj_vm_unwind_ff, 7
lj_vm_unwind_ff:
        .byte 72,131,231,252,72,137,252

        .globl lj_vm_unwind_ff_eh
        .hidden lj_vm_unwind_ff_eh
        .type lj_vm_unwind_ff_eh, @function
        .size lj_vm_unwind_ff_eh, 56
lj_vm_unwind_ff_eh:
        .byte 139,108,36,24,72,199,193,248,255,255,255,184,2,0,0,0
        .byte 139,85,16,68,139,117,8,65,129,198,184,11,0,0,139,90
        .byte 252,199,66,252,254,255,255,255,65,199,134,48,245,255,255,255
        .byte 255,255,255,233,224,254,255,255

        .globl lj_vm_growstack_c
        .hidden lj_vm_growstack_c
        .type lj_vm_growstack_c, @function
        .size lj_vm_growstack_c, 7
lj_vm_growstack_c:
        .byte 190,20,0,0,0,235,28

        .globl lj_vm_growstack_v
        .hidden lj_vm_growstack_v
        .type lj_vm_growstack_v, @function
        .size lj_vm_growstack_v, 5
lj_vm_growstack_v:
        .byte 131,232,8,235,4
```

```
1136
1137         .globl lj_vm_growstack_f
1138         .hidden lj_vm_growstack_f
1139         .type lj_vm_growstack_f, @function
1140         .size lj_vm_growstack_f, 65
1141 lj_vm_growstack_f:
1142         .byte 141,68,194,248,15,182,75,195,131,195,4,137,85,16,137,69
1143         .byte 24,137,92,36,28,137,206,137,239,232
1144         .long lj_state_growstack-.-4
1145         .byte 139,85,16,139,69,24,139,106,248,41,208,193,232,3,131,192
1146         .byte 1,139,93,16,139,11,15,182,233,15,182,205,131,195,4,65
1147         .byte 255,36,238
1148
1149         .globl lj_vm_resume
1150         .hidden lj_vm_resume
1151         .type lj_vm_resume, @function
1152         .size lj_vm_resume, 125
1153 lj_vm_resume:
1154         .byte 85,83,65,87,65,86,72,131,236,40,137,253,137,124,36,24
1155         .byte 137,241,187,5,0,0,0,49,192,76,141,124,36,1,68,139
1156         .byte 117,8,65,129,198,184,11,0,0,76,137,125,48,137,68,36
1157         .byte 28,72,137,68,36,32,137,68,36,16,137,68,36,20,56,69
1158         .byte 7,15,132,130,0,0,0,65,199,134,48,245,255,255,255,255
1159         .byte 255,255,136,69,7,139,85,16,139,69,24,41,200,193,232,3
1160         .byte 131,192,1,41,209,139,90,252,137,68,36,4,247,195,3,0
1161         .byte 0,0,15,132,163,248,255,255,233,47,254,255,255
1162
1163         .globl lj_vm_pcall
1164         .hidden lj_vm_pcall
1165         .type lj_vm_pcall, @function
1166         .size lj_vm_pcall, 21
1167 lj_vm_pcall:
1168         .byte 85,83,65,87,65,86,72,131,236,40,187,5,0,0,0,137
1169         .byte 76,36,20,235,15
1170
1171         .globl lj_vm_call
1172         .hidden lj_vm_call
1173         .type lj_vm_call, @function
1174         .size lj_vm_call, 84
1175 lj_vm_call:
1176         .byte 85,83,65,87,65,86,72,131,236,40,187,1,0,0,0,137
1177         .byte 84,36,16,137,253,137,124,36,24,137,241,76,139,125,48,76
1178         .byte 137,124,36,32,137,108,36,28,72,137,101,48,68,139,117,8
1179         .byte 65,129,198,184,11,0,0,65,199,134,48,245,255,255,255,255
1180         .byte 255,255,139,85,16,1,203,41,211,139,69,24,41,200,193,232
1181         .byte 3,131,192,1
1182
1183         .globl lj_vm_call_dispatch
1184         .hidden lj_vm_call_dispatch
1185         .type lj_vm_call_dispatch, @function
1186         .size lj_vm_call_dispatch, 13
1187 lj_vm_call_dispatch:
1188         .byte 139,105,248,131,121,252,247,15,133,45,3,0,0
1189
1190         .globl lj_vm_call_dispatch_f
1191         .hidden lj_vm_call_dispatch_f
1192         .type lj_vm_call_dispatch_f, @function
1193         .size lj_vm_call_dispatch_f, 23
1194 lj_vm_call_dispatch_f:
1195         .byte 137,202,137,90,252,139,93,16,139,11,15,182,233,15,182,205
1196         .byte 131,195,4,65,255,36,238
1197
1198         .globl lj_vm_cpcall
1199         .hidden lj_vm_cpcall
1200         .type lj_vm_cpcall, @function
1201         .size lj_vm_cpcall, 76
1202 lj_vm_cpcall:
1203         .byte 85,83,65,87,65,86,72,131,236,40,137,253,137,124,36,24
1204         .byte 137,108,36,28,68,139,125,36,68,43,125,24,199,68,36,20
1205         .byte 0,0,0,0,68,137,124,36,16,76,139,125,48,76,137,124
1206         .byte 36,32,72,137,101,48,255,209,133,192,15,132,173,253,255,255
1207         .byte 137,193,187,5,0,0,0,233,104,255,255,255
1208
1209         .globl lj_cont_dispatch
1210         .hidden lj_cont_dispatch
1211         .type lj_cont_dispatch, @function
```

```
        .size lj_cont_dispatch, 74
lj_cont_dispatch:
        .byte 1,209,131,227,248,137,213,41,218,199,68,193,252,255,255,255
        .byte 255,137,200,139,93,244,72,99,77,240,131,249,1,118,24,76
        .byte 141,61,75,228,255,255,76,1,249,68,139,122,248,69,139,127
        .byte 16,69,139,127,208,255,225,15,132,68,29,0,0,41,213,193
        .byte 237,3,141,69,255,233,104,21,0,0

        .globl lj_cont_cat
        .hidden lj_cont_cat
        .type lj_cont_cat, @function
        .size lj_cont_cat, 46
lj_cont_cat:
        .byte 15,182,75,255,131,237,16,141,12,202,41,233,15,132,132,0
        .byte 0,0,247,217,193,233,3,139,124,36,24,137,87,16,137,202
        .byte 72,139,8,72,137,77,0,137,238,233,195,236,255,255

        .globl lj_vmeta_tgets
        .hidden lj_vmeta_tgets
        .type lj_vmeta_tgets, @function
        .size lj_vmeta_tgets, 41
lj_vmeta_tgets:
        .byte 137,4,36,199,68,36,4,251,255,255,255,72,141,4,36,128
        .byte 123,252,52,117,46,65,141,142,232,244,255,255,137,41,199,65
        .byte 4,244,255,255,255,137,205,235,33

        .globl lj_vmeta_tgetb
        .hidden lj_vmeta_tgetb
        .type lj_vmeta_tgetb, @function
        .size lj_vmeta_tgetb, 19
lj_vmeta_tgetb:
        .byte 15,182,67,254,242,15,42,192,242,15,17,4,36,72,141,4
        .byte 36,235,7

        .globl lj_vmeta_tgetv
        .hidden lj_vmeta_tgetv
        .type lj_vmeta_tgetv, @function
        .size lj_vmeta_tgetv, 44
lj_vmeta_tgetv:
        .byte 15,182,67,254,141,4,194,15,182,107,255,141,44,234,139,124
        .byte 36,24,137,87,16,137,238,72,137,194,137,253,137,92,36,28
        .byte 232
        .long lj_meta_tget-.-4
        .byte 139,85,16,133,192,116,29

        .globl lj_cont_ra
        .hidden lj_cont_ra
        .type lj_cont_ra, @function
        .size lj_cont_ra, 53
lj_cont_ra:
        .byte 15,182,75,253,72,139,40,72,137,44,202,139,3,15,182,204
        .byte 15,182,232,131,195,4,193,232,16,65,255,36,238,139,77,24
        .byte 137,89,244,141,89,2,41,211,139,105,248,184,3,0,0,0
        .byte 233,136,254,255,255

        .globl lj_vmeta_tsets
        .hidden lj_vmeta_tsets
        .type lj_vmeta_tsets, @function
        .size lj_vmeta_tsets, 41
lj_vmeta_tsets:
        .byte 137,4,36,199,68,36,4,251,255,255,255,72,141,4,36,128
        .byte 123,252,53,117,46,65,141,142,232,244,255,255,137,41,199,65
        .byte 4,244,255,255,255,137,205,235,33

        .globl lj_vmeta_tsetb
        .hidden lj_vmeta_tsetb
        .type lj_vmeta_tsetb, @function
        .size lj_vmeta_tsetb, 19
lj_vmeta_tsetb:
        .byte 15,182,67,254,242,15,42,192,242,15,17,4,36,72,141,4
        .byte 36,235,7

        .globl lj_vmeta_tsetv
        .hidden lj_vmeta_tsetv
        .type lj_vmeta_tsetv, @function
        .size lj_vmeta_tsetv, 55
```

```
lj_vmeta_tsetv:
        .byte 15,182,67,254,141,4,194,15,182,107,255,141,44,234,139,124
        .byte 36,24,137,87,16,137,238,72,137,194,137,253,137,92,36,28
        .byte 232
        .long lj_meta_tset-.-4
        .byte 139,85,16,133,192,116,29,15,182,75,253,72,139,44,202,72
        .byte 137,40

        .globl lj_cont_nop
        .hidden lj_cont_nop
        .type lj_cont_nop, @function
        .size lj_cont_nop, 54
lj_cont_nop:
        .byte 139,3,15,182,204,15,182,232,131,195,4,193,232,16,65,255
        .byte 36,238,139,77,24,137,89,244,15,182,67,253,72,139,44,194
        .byte 72,137,105,16,141,89,2,41,211,139,105,248,184,4,0,0
        .byte 0,233,223,253,255,255

        .globl lj_vmeta_comp
        .hidden lj_vmeta_comp
        .type lj_vmeta_comp, @function
        .size lj_vmeta_comp, 74
lj_vmeta_comp:
        .byte 139,108,36,24,137,85,16,141,52,202,141,20,194,137,239,15
        .byte 182,75,252,137,92,36,28,232
        .long lj_meta_comp-.-4
        .byte 139,85,16,131,248,1,15,135,178,0,0,0,141,91,4,114
        .byte 11,15,183,67,254,141,156,131,0,0,254,255,139,3,15,182
        .byte 204,15,182,232,131,195,4,193,232,16,65,255,36,238

        .globl lj_cont_condt
        .hidden lj_cont_condt
        .type lj_cont_condt, @function
        .size lj_cont_condt, 11
lj_cont_condt:
        .byte 131,195,4,131,120,4,254,114,218,235,227

        .globl lj_cont_condf
        .hidden lj_cont_condf
        .type lj_cont_condf, @function
        .size lj_cont_condf, 6
lj_cont_condf:
        .byte 131,120,4,254,235,205

        .globl lj_vmeta_equal
        .hidden lj_vmeta_equal
        .type lj_vmeta_equal, @function
        .size lj_vmeta_equal, 29
lj_vmeta_equal:
        .byte 131,235,4,137,206,137,233,139,108,36,24,137,85,16,137,194
        .byte 137,239,137,92,36,28,232
        .long lj_meta_equal-.-4
        .byte 235,164

        .globl lj_vmeta_equal_cd
        .hidden lj_vmeta_equal_cd
        .type lj_vmeta_equal_cd, @function
        .size lj_vmeta_equal_cd, 26
lj_vmeta_equal_cd:
        .byte 131,235,4,139,108,36,24,137,85,16,137,239,139,115,252,137
        .byte 92,36,28,232
        .long lj_meta_equal_cd-.-4
        .byte 235,138

        .globl lj_vmeta_arith_vno
        .hidden lj_vmeta_arith_vno
        .type lj_vmeta_arith_vno, @function
        .size lj_vmeta_arith_vno, 0
lj_vmeta_arith_vno:

        .globl lj_vmeta_arith_vn
        .hidden lj_vmeta_arith_vn
        .type lj_vmeta_arith_vn, @function
        .size lj_vmeta_arith_vn, 6
lj_vmeta_arith_vn:
        .byte 65,141,4,199,235,20
```

```
1364
1365         .globl lj_vmeta_arith_nvo
1366         .hidden lj_vmeta_arith_nvo
1367         .type lj_vmeta_arith_nvo, @function
1368         .size lj_vmeta_arith_nvo, 0
1369 lj_vmeta_arith_nvo:
1370
1371         .globl lj_vmeta_arith_nv
1372         .hidden lj_vmeta_arith_nv
1373         .type lj_vmeta_arith_nv, @function
1374         .size lj_vmeta_arith_nv, 10
1375 lj_vmeta_arith_nv:
1376         .byte 65,141,4,199,141,44,234,149,235,13
1377
1378         .globl lj_vmeta_unm
1379         .hidden lj_vmeta_unm
1380         .type lj_vmeta_unm, @function
1381         .size lj_vmeta_unm, 7
1382 lj_vmeta_unm:
1383         .byte 141,4,194,137,197,235,6
1384
1385         .globl lj_vmeta_arith_vvo
1386         .hidden lj_vmeta_arith_vvo
1387         .type lj_vmeta_arith_vvo, @function
1388         .size lj_vmeta_arith_vvo, 0
1389 lj_vmeta_arith_vvo:
1390
1391         .globl lj_vmeta_arith_vv
1392         .hidden lj_vmeta_arith_vv
1393         .type lj_vmeta_arith_vv, @function
1394         .size lj_vmeta_arith_vv, 49
1395 lj_vmeta_arith_vv:
1396         .byte 141,4,194,141,44,234,141,12,202,68,15,182,67,252,137,206
1397         .byte 137,193,139,124,36,24,137,87,16,137,234,137,253,137,92,36
1398         .byte 28,232
1399         .long lj_meta_arith-.-4
1400         .byte 139,85,16,133,192,15,132,240,254,255,255
1401
1402         .globl lj_vmeta_binop
1403         .hidden lj_vmeta_binop
1404         .type lj_vmeta_binop, @function
1405         .size lj_vmeta_binop, 20
1406 lj_vmeta_binop:
1407         .byte 137,193,41,208,137,89,244,141,88,2,184,3,0,0,0,233
1408         .byte 228,252,255,255
1409
1410         .globl lj_vmeta_len
1411         .hidden lj_vmeta_len
1412         .type lj_vmeta_len, @function
1413         .size lj_vmeta_len, 26
1414 lj_vmeta_len:
1415         .byte 139,108,36,24,137,85,16,141,52,194,137,239,137,92,36,28
1416         .byte 232
1417         .long lj_meta_len-.-4
1418         .byte 139,85,16,235,210
1419
1420         .globl lj_vmeta_call_ra
1421         .hidden lj_vmeta_call_ra
1422         .type lj_vmeta_call_ra, @function
1423         .size lj_vmeta_call_ra, 4
1424 lj_vmeta_call_ra:
1425         .byte 141,76,202,8
1426
1427         .globl lj_vmeta_call
1428         .hidden lj_vmeta_call
1429         .type lj_vmeta_call, @function
1430         .size lj_vmeta_call, 81
1431 lj_vmeta_call:
1432         .byte 137,76,36,4,137,4,36,131,233,8,139,108,36,24,137,85
1433         .byte 16,137,206,141,20,193,137,239,137,92,36,28,232
1434         .long lj_meta_call-.-4
1435         .byte 139,85,16,139,76,36,4,139,4,36,139,105,248,131,192,1
1436         .byte 65,57,215,15,132,43,242,255,255,137,202,137,90,252,139,93
1437         .byte 16,139,11,15,182,233,15,182,205,131,195,4,65,255,36,238
1438
1439         .globl lj_vmeta_for
```

```
        .hidden lj_vmeta_for
        .type lj_vmeta_for, @function
        .size lj_vmeta_for, 43
lj_vmeta_for:
        .byte 139,108,36,24,137,85,16,137,206,137,239,137,92,36,28,232
        .long lj_meta_for-.-4
        .byte 139,85,16,139,67,252,15,182,204,15,182,232,193,232,16,65
        .byte 255,164,238,216,4,0,0

        .globl lj_ff_assert
        .hidden lj_ff_assert
        .type lj_ff_assert, @function
        .size lj_ff_assert, 67
lj_ff_assert:
        .byte 131,248,2,15,130,12,18,0,0,139,106,4,131,253,254,15
        .byte 131,0,18,0,0,139,90,252,137,68,36,4,137,106,252,139
        .byte 42,137,106,248,131,232,2,116,17,137,209,131,193,8,72,139
        .byte 41,72,137,105,248,131,232,1,117,241,139,68,36,4,233,86
        .byte 6,0,0

        .globl lj_ff_type
        .hidden lj_ff_type
        .type lj_ff_type, @function
        .size lj_ff_type, 66
lj_ff_type:
        .byte 131,248,2,15,130,201,17,0,0,139,106,4,137,233,193,249
        .byte 15,131,249,254,116,37,184,13,0,0,0,247,213,57,232,15
        .byte 71,197,139,106,248,139,68,197,32,139,90,252,199,66,252,251
        .byte 255,255,255,137,66,248,233,18,6,0,0,184,3,0,0,0
        .byte 235,224

        .globl lj_ff_getmetatable
        .hidden lj_ff_getmetatable
        .type lj_ff_getmetatable, @function
        .size lj_ff_getmetatable, 162
lj_ff_getmetatable:
        .byte 131,248,2,15,130,135,17,0,0,139,106,4,139,90,252,131
        .byte 253,244,117,97,139,42,139,109,16,133,237,199,66,252,255,255
        .byte 255,255,15,132,227,5,0,0,65,139,134,140,245,255,255,199
        .byte 66,252,244,255,255,255,137,106,248,139,77,28,35,72,8,107
        .byte 201,24,3,77,20,131,121,12,251,117,5,57,65,8,116,12
        .byte 139,73,16,133,201,117,238,233,175,5,0,0,139,105,4,131
        .byte 253,255,15,132,163,5,0,0,139,1,137,106,252,137,66,248
        .byte 233,150,5,0,0,131,253,243,116,154,131,253,242,119,20,129
        .byte 253,255,255,254,255,118,7,189,252,255,255,255,235,5,189,242
        .byte 255,255,255,247,213,65,139,172,174,160,245,255,255,233,119,255
        .byte 255,255

        .globl lj_ff_setmetatable
        .hidden lj_ff_setmetatable
        .type lj_ff_setmetatable, @function
        .size lj_ff_setmetatable, 92
lj_ff_setmetatable:
        .byte 131,248,3,15,130,229,16,0,0,131,122,4,244,15,133,219
        .byte 16,0,0,139,42,131,125,16,0,15,133,207,16,0,0,131
        .byte 122,12,244,15,133,197,16,0,0,139,66,8,137,69,16,139
        .byte 90,252,199,66,252,244,255,255,255,137,106,248,246,69,4,4
        .byte 116,21,128,101,4,251,65,139,134,132,244,255,255,65,137,174
        .byte 132,244,255,255,137,69,12,233,13,5,0,0

        .globl lj_ff_rawget
        .hidden lj_ff_rawget
        .type lj_ff_rawget, @function
        .size lj_ff_rawget, 52
lj_ff_rawget:
        .byte 131,248,3,15,130,137,16,0,0,131,122,4,244,15,133,127
        .byte 16,0,0,137,213,139,50,141,82,8,139,124,36,24,232
        .long lj_tab_get-.-4
        .byte 137,234,72,139,40,139,90,252,72,137,106,248,233,217,4,0
        .byte 0

        .globl lj_ff_tonumber
        .hidden lj_ff_tonumber
        .type lj_ff_tonumber, @function
        .size lj_ff_tonumber, 31
lj_ff_tonumber:
```

```
        .byte 131,248,2,15,133,85,16,0,0,129,122,4,255,255,254,255
        .byte 15,131,72,16,0,0,242,15,16,2,233,178,4,0,0

        .globl lj_ff_tostring
        .hidden lj_ff_tostring
        .type lj_ff_tostring, @function
        .size lj_ff_tostring, 108
lj_ff_tostring:
        .byte 131,248,2,15,130,54,16,0,0,139,90,252,131,122,4,251
        .byte 117,17,139,2,199,66,252,251,255,255,255,137,66,248,233,151
        .byte 4,0,0,129,122,4,255,255,254,255,15,135,15,16,0,0
        .byte 65,131,190,212,245,255,255,0,15,133,1,16,0,0,65,139
        .byte 174,104,244,255,255,65,59,174,108,244,255,255,114,5,232,123
        .byte 16,0,0,139,108,36,24,137,85,16,137,92,36,28,137,214
        .byte 137,239,232
        .long lj_str_fromnum-.-4
        .byte 139,85,16,235,168

        .globl lj_ff_next
        .hidden lj_ff_next
        .type lj_ff_next, @function
        .size lj_ff_next, 72
lj_ff_next:
        .byte 131,248,2,15,130,202,15,0,0,116,71,131,122,4,244,15
        .byte 133,190,15,0,0,139,108,36,24,137,85,16,137,85,24,139
        .byte 90,252,139,50,141,82,8,137,239,137,92,36,28,232
        .long lj_tab_next-.-4
        .byte 139,85,16,133,192,116,34,72,139,106,8,72,139,66,16,72
        .byte 137,106,248,72,137,2

        .globl lj_fff_res2
        .hidden lj_fff_res2
        .type lj_fff_res2, @function
        .size lj_fff_res2, 31
lj_fff_res2:
        .byte 184,3,0,0,0,233,1,4,0,0,199,66,12,255,255,255
        .byte 255,235,176,199,66,252,255,255,255,255,233,231,3,0,0

        .globl lj_ff_pairs
        .hidden lj_ff_pairs
        .type lj_ff_pairs, @function
        .size lj_ff_pairs, 57
lj_ff_pairs:
        .byte 131,248,2,15,130,99,15,0,0,139,42,131,122,4,244,15
        .byte 133,87,15,0,0,139,106,248,139,69,32,139,90,252,199,66
        .byte 252,247,255,255,255,137,66,248,199,66,12,255,255,255,255,184
        .byte 4,0,0,0,233,179,3,0,0

        .globl lj_ff_ipairs_aux
        .hidden lj_ff_ipairs_aux
        .type lj_ff_ipairs_aux, @function
        .size lj_ff_ipairs_aux, 121
lj_ff_ipairs_aux:
        .byte 131,248,3,15,130,42,15,0,0,131,122,4,244,15,133,32
        .byte 15,0,0,129,122,12,255,255,254,255,15,131,19,15,0,0
        .byte 139,90,252,242,15,16,66,8,72,189,0,0,0,0,0,0
        .byte 240,63,102,72,15,110,205,242,15,88,193,242,15,45,192,242
        .byte 15,17,66,248,139,42,59,69,24,115,23,193,224,3,3,69
        .byte 8,131,120,4,255,116,34,72,139,40,72,137,42,233,70,255
        .byte 255,255,131,125,28,0,116,17,137,239,137,213,137,198,232
        .long lj_tab_getinth-.-4
        .byte 137,234,133,192,117,216

        .globl lj_fff_res0
        .hidden lj_fff_res0
        .type lj_fff_res0, @function
        .size lj_fff_res0, 10
lj_fff_res0:
        .byte 184,1,0,0,0,233,48,3,0,0

        .globl lj_ff_ipairs
        .hidden lj_ff_ipairs
        .type lj_ff_ipairs, @function
        .size lj_ff_ipairs, 58
lj_ff_ipairs:
        .byte 131,248,2,15,130,167,14,0,0,139,42,131,122,4,244,15
```

```
        .byte 133,155,14,0,0,139,106,248,139,69,32,139,90,252,199,66
        .byte 252,247,255,255,255,137,66,248,15,87,192,242,15,17,66,8
        .byte 184,4,0,0,0,233,246,2,0,0

        .globl lj_ff_pcall
        .hidden lj_ff_pcall
        .type lj_ff_pcall, @function
        .size lj_ff_pcall, 41
lj_ff_pcall:
        .byte 131,248,2,15,130,109,14,0,0,141,74,8,131,232,1,187
        .byte 14,0,0,0,65,15,182,174,217,244,255,255,193,237,4,131
        .byte 229,1,1,235,233,130,248,255,255

        .globl lj_ff_xpcall
        .hidden lj_ff_xpcall
        .type lj_ff_xpcall, @function
        .size lj_ff_xpcall, 55
lj_ff_xpcall:
        .byte 131,248,3,15,130,68,14,0,0,131,122,12,247,15,133,58
        .byte 14,0,0,139,106,4,137,106,12,199,66,4,247,255,255,255
        .byte 139,42,139,90,8,137,106,8,137,26,141,74,16,131,232,2
        .byte 187,22,0,0,0,235,180

        .globl lj_ff_coroutine_resume
        .hidden lj_ff_coroutine_resume
        .type lj_ff_coroutine_resume, @function
        .size lj_ff_coroutine_resume, 303
lj_ff_coroutine_resume:
        .byte 131,248,2,15,130,13,14,0,0,139,42,139,90,252,137,92
        .byte 36,28,137,44,36,131,122,4,249,15,133,247,13,0,0,72
        .byte 131,125,48,0,15,133,236,13,0,0,128,125,7,1,15,135
        .byte 226,13,0,0,139,77,24,116,9,59,77,16,15,132,212,13
        .byte 0,0,141,92,193,240,59,93,32,15,135,199,13,0,0,137
        .byte 93,24,139,108,36,24,137,85,16,131,194,8,137,85,24,141
        .byte 108,194,232,72,41,221,57,203,116,15,72,139,4,43,72,137
        .byte 67,248,131,235,8,57,203,117,241,137,206,139,60,36,232,226
        .byte 246,255,255,65,199,134,48,245,255,255,255,255,255,255,139,108
        .byte 36,24,139,28,36,139,85,16,131,248,1,119,90,139,75,16
        .byte 68,139,123,24,137,75,24,68,137,251,41,203,116,31,141,4
        .byte 26,193,235,3,59,69,32,119,91,137,213,72,41,205,72,139
        .byte 1,72,137,4,41,131,193,8,68,57,249,117,241,141,67,2
        .byte 199,66,252,253,255,255,255,139,92,36,28,137,68,36,4,72
        .byte 199,193,248,255,255,255,247,195,3,0,0,0,15,132,142,239
        .byte 255,255,233,26,245,255,255,199,66,252,254,255,255,255,139,75
        .byte 24,131,233,8,137,75,24,72,139,1,72,137,2,184,3,0
        .byte 0,0,235,195,139,12,36,68,137,121,24,137,222,137,239,232
        .long lj_state_growstack-.-4
        .byte 139,28,36,139,85,16,233,110,255,255,255

        .globl lj_ff_coroutine_wrap_aux
        .hidden lj_ff_coroutine_wrap_aux
        .type lj_ff_coroutine_wrap_aux, @function
        .size lj_ff_coroutine_wrap_aux, 250
lj_ff_coroutine_wrap_aux:
        .byte 139,106,248,139,109,32,139,90,252,137,92,36,28,137,44,36
        .byte 72,131,125,48,0,15,133,204,12,0,0,128,125,7,1,15
        .byte 135,194,12,0,0,139,77,24,116,9,59,77,16,15,132,180
        .byte 12,0,0,141,92,193,248,59,93,32,15,135,167,12,0,0
        .byte 137,93,24,139,108,36,24,137,85,16,137,85,24,141,108,194
        .byte 240,72,41,221,57,203,116,15,72,139,4,43,72,137,67,248
        .byte 131,235,8,57,203,117,241,137,206,139,60,36,232,197,245,255
        .byte 255,65,199,134,48,245,255,255,255,255,255,255,139,108,36,24
        .byte 139,28,36,139,85,16,131,248,1,119,78,139,75,16,68,139
        .byte 123,24,137,75,24,68,137,251,41,203,116,31,141,4,26,193
        .byte 235,3,59,69,32,119,59,137,213,72,41,205,72,139,1,72
        .byte 137,4,41,131,193,8,68,57,249,117,241,141,67,1,139,92
        .byte 36,28,137,68,36,4,49,201,247,195,3,0,0,0,15,132
        .byte 125,238,255,255,233,9,244,255,255,137,222,137,239,232
        .long lj_ffh_coroutine_wrap_err-.-4
        .byte 139,12,36,68,137,121,24,137,222,137,239,232
        .long lj_state_growstack-.-4
        .byte 139,28,36,139,85,16,235,145

        .globl lj_ff_coroutine_yield
        .hidden lj_ff_coroutine_yield
        .type lj_ff_coroutine_yield, @function
```

```
        .size lj_ff_coroutine_yield, 44
lj_ff_coroutine_yield:
        .byte 139,108,36,24,72,247,69,48,1,0,0,0,15,132,219,11
        .byte 0,0,137,85,16,141,68,194,248,137,69,24,49,192,72,137
        .byte 69,48,176,1,136,69,7,233,18,244,255,255

        .globl lj_fff_resi
        .hidden lj_fff_resi
        .type lj_fff_resi, @function
        .size lj_fff_resi, 0
lj_fff_resi:

        .globl lj_fff_resn
        .hidden lj_fff_resn
        .type lj_fff_resn, @function
        .size lj_fff_resn, 8
lj_fff_resn:
        .byte 139,90,252,221,90,248,235,52

        .globl lj_ff_math_abs
        .hidden lj_ff_math_abs
        .type lj_ff_math_abs, @function
        .size lj_ff_math_abs, 44
lj_ff_math_abs:
        .byte 131,248,2,15,130,176,11,0,0,129,122,4,255,255,254,255
        .byte 15,131,163,11,0,0,242,15,16,2,72,184,255,255,255,255
        .byte 255,255,255,127,102,72,15,110,200,15,84,193

        .globl lj_fff_resxmm0
        .hidden lj_fff_resxmm0
        .type lj_fff_resxmm0, @function
        .size lj_fff_resxmm0, 8
lj_fff_resxmm0:
        .byte 139,90,252,242,15,17,66,248

        .globl lj_fff_res1
        .hidden lj_fff_res1
        .type lj_fff_res1, @function
        .size lj_fff_res1, 5
lj_fff_res1:
        .byte 184,2,0,0,0

        .globl lj_fff_res
        .hidden lj_fff_res
        .type lj_fff_res, @function
        .size lj_fff_res, 4
lj_fff_res:
        .byte 137,68,36,4

        .globl lj_fff_res_
        .hidden lj_fff_res_
        .type lj_fff_res_, @function
        .size lj_fff_res_, 66
lj_fff_res_:
        .byte 247,195,3,0,0,0,117,46,56,67,255,119,28,15,182,75
        .byte 253,72,247,209,141,20,202,139,3,15,182,204,15,182,232,131
        .byte 195,4,193,232,16,65,255,36,238,199,68,194,244,255,255,255
        .byte 255,131,192,1,235,210,72,199,193,248,255,255,255,233,53,243
        .byte 255,255

        .globl lj_ff_math_floor
        .hidden lj_ff_math_floor
        .type lj_ff_math_floor, @function
        .size lj_ff_math_floor, 24
lj_ff_math_floor:
        .byte 129,122,4,255,255,254,255,15,131,45,11,0,0,242,15,16
        .byte 2,232,49,14,0,0,235,149

        .globl lj_ff_math_ceil
        .hidden lj_ff_math_ceil
        .type lj_ff_math_ceil, @function
        .size lj_ff_math_ceil, 27
lj_ff_math_ceil:
        .byte 129,122,4,255,255,254,255,15,131,21,11,0,0,242,15,16
        .byte 2,232,116,14,0,0,233,122,255,255,255

```

```
        .globl lj_ff_math_sqrt
        .hidden lj_ff_math_sqrt
        .type lj_ff_math_sqrt, @function
        .size lj_ff_math_sqrt, 31
lj_ff_math_sqrt:
        .byte 131,248,2,15,130,254,10,0,0,129,122,4,255,255,254,255
        .byte 15,131,241,10,0,0,242,15,81,2,233,91,255,255,255

        .globl lj_ff_math_log
        .hidden lj_ff_math_log
        .type lj_ff_math_log, @function
        .size lj_ff_math_log, 33
lj_ff_math_log:
        .byte 131,248,2,15,133,223,10,0,0,129,122,4,255,255,254,255
        .byte 15,131,210,10,0,0,217,237,221,2,217,241,233,6,255,255
        .byte 255

        .globl lj_ff_math_log10
        .hidden lj_ff_math_log10
        .type lj_ff_math_log10, @function
        .size lj_ff_math_log10, 33
lj_ff_math_log10:
        .byte 131,248,2,15,130,190,10,0,0,129,122,4,255,255,254,255
        .byte 15,131,177,10,0,0,217,236,221,2,217,241,233,229,254,255
        .byte 255

        .globl lj_ff_math_exp
        .hidden lj_ff_math_exp
        .type lj_ff_math_exp, @function
        .size lj_ff_math_exp, 34
lj_ff_math_exp:
        .byte 131,248,2,15,130,157,10,0,0,129,122,4,255,255,254,255
        .byte 15,131,144,10,0,0,221,2,232,57,15,0,0,233,195,254
        .byte 255,255

        .globl lj_ff_math_sin
        .hidden lj_ff_math_sin
        .type lj_ff_math_sin, @function
        .size lj_ff_math_sin, 31
lj_ff_math_sin:
        .byte 131,248,2,15,130,123,10,0,0,129,122,4,255,255,254,255
        .byte 15,131,110,10,0,0,221,2,217,254,233,164,254,255,255

        .globl lj_ff_math_cos
        .hidden lj_ff_math_cos
        .type lj_ff_math_cos, @function
        .size lj_ff_math_cos, 31
lj_ff_math_cos:
        .byte 131,248,2,15,130,92,10,0,0,129,122,4,255,255,254,255
        .byte 15,131,79,10,0,0,221,2,217,255,233,133,254,255,255

        .globl lj_ff_math_tan
        .hidden lj_ff_math_tan
        .type lj_ff_math_tan, @function
        .size lj_ff_math_tan, 33
lj_ff_math_tan:
        .byte 131,248,2,15,130,61,10,0,0,129,122,4,255,255,254,255
        .byte 15,131,48,10,0,0,221,2,217,242,221,216,233,100,254,255
        .byte 255

        .globl lj_ff_math_asin
        .hidden lj_ff_math_asin
        .type lj_ff_math_asin, @function
        .size lj_ff_math_asin, 41
lj_ff_math_asin:
        .byte 131,248,2,15,130,28,10,0,0,129,122,4,255,255,254,255
        .byte 15,131,15,10,0,0,221,2,217,192,216,200,217,232,222,225
        .byte 217,250,217,243,233,59,254,255,255

        .globl lj_ff_math_acos
        .hidden lj_ff_math_acos
        .type lj_ff_math_acos, @function
        .size lj_ff_math_acos, 43
lj_ff_math_acos:
        .byte 131,248,2,15,130,243,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,230,9,0,0,221,2,217,192,216,200,217,232,222,225
```

```
        .byte 217,250,217,201,217,243,233,16,254,255,255

        .globl lj_ff_math_atan
        .hidden lj_ff_math_atan
        .type lj_ff_math_atan, @function
        .size lj_ff_math_atan, 33
lj_ff_math_atan:
        .byte 131,248,2,15,130,200,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,187,9,0,0,221,2,217,232,217,243,233,239,253,255
        .byte 255

        .globl lj_ff_math_sinh
        .hidden lj_ff_math_sinh
        .type lj_ff_math_sinh, @function
        .size lj_ff_math_sinh, 40
lj_ff_math_sinh:
        .byte 131,248,2,15,130,167,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,154,9,0,0,242,15,16,2,137,213,232
        .long lj_vm_sinh-.-4
        .byte 137,234,233,251,253,255,255

        .globl lj_ff_math_cosh
        .hidden lj_ff_math_cosh
        .type lj_ff_math_cosh, @function
        .size lj_ff_math_cosh, 40
lj_ff_math_cosh:
        .byte 131,248,2,15,130,127,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,114,9,0,0,242,15,16,2,137,213,232
        .long lj_vm_cosh-.-4
        .byte 137,234,233,211,253,255,255

        .globl lj_ff_math_tanh
        .hidden lj_ff_math_tanh
        .type lj_ff_math_tanh, @function
        .size lj_ff_math_tanh, 40
lj_ff_math_tanh:
        .byte 131,248,2,15,130,87,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,74,9,0,0,242,15,16,2,137,213,232
        .long lj_vm_tanh-.-4
        .byte 137,234,233,171,253,255,255

        .globl lj_ff_math_deg
        .hidden lj_ff_math_deg
        .type lj_ff_math_deg, @function
        .size lj_ff_math_deg, 0
lj_ff_math_deg:

        .globl lj_ff_math_rad
        .hidden lj_ff_math_rad
        .type lj_ff_math_rad, @function
        .size lj_ff_math_rad, 39
lj_ff_math_rad:
        .byte 131,248,2,15,130,47,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,34,9,0,0,242,15,16,2,139,106,248,242,15,89
        .byte 69,32,233,132,253,255,255

        .globl lj_ff_math_atan2
        .hidden lj_ff_math_atan2
        .type lj_ff_math_atan2, @function
        .size lj_ff_math_atan2, 47
lj_ff_math_atan2:
        .byte 131,248,3,15,130,8,9,0,0,129,122,4,255,255,254,255
        .byte 15,131,251,8,0,0,129,122,12,255,255,254,255,15,131,238
        .byte 8,0,0,221,2,221,66,8,217,243,233,33,253,255,255

        .globl lj_ff_math_ldexp
        .hidden lj_ff_math_ldexp
        .type lj_ff_math_ldexp, @function
        .size lj_ff_math_ldexp, 49
lj_ff_math_ldexp:
        .byte 131,248,3,15,130,217,8,0,0,129,122,4,255,255,254,255
        .byte 15,131,204,8,0,0,129,122,12,255,255,254,255,15,131,191
        .byte 8,0,0,221,66,8,221,2,217,253,221,217,233,240,252,255
        .byte 255

        .globl lj_ff_math_frexp
```

```
        .hidden lj_ff_math_frexp
        .type lj_ff_math_frexp, @function
        .size lj_ff_math_frexp, 148
lj_ff_math_frexp:
        .byte 131,248,2,15,130,168,8,0,0,139,106,4,129,253,255,255
        .byte 254,255,15,131,153,8,0,0,139,90,252,139,2,137,106,252
        .byte 137,66,248,209,229,129,253,0,0,224,255,115,58,9,232,116
        .byte 54,184,254,3,0,0,129,253,0,0,32,0,114,46,193,237
        .byte 21,41,197,242,15,42,197,139,106,252,129,229,255,255,15,128
        .byte 129,205,0,0,224,63,137,106,252,242,15,17,2,184,3,0
        .byte 0,0,233,202,252,255,255,15,87,192,235,237,242,15,16,2
        .byte 72,189,0,0,0,0,0,0,80,67,102,72,15,110,205,242
        .byte 15,89,193,242,15,17,66,248,139,106,252,184,52,4,0,0
        .byte 209,229,235,170

        .globl lj_ff_math_modf
        .hidden lj_ff_math_modf
        .type lj_ff_math_modf, @function
        .size lj_ff_math_modf, 99
lj_ff_math_modf:
        .byte 131,248,2,15,130,20,8,0,0,129,122,4,255,255,254,255
        .byte 15,131,7,8,0,0,242,15,16,2,139,106,4,139,90,252
        .byte 209,229,129,253,0,0,224,255,116,52,15,40,224,232,174,11
        .byte 0,0,242,15,92,224,242,15,17,66,248,242,15,17,34,139
        .byte 66,252,139,106,4,49,232,120,10,184,3,0,0,0,233,74
        .byte 252,255,255,129,245,0,0,0,128,137,106,4,235,235,15,87
        .byte 228,235,211

        .globl lj_ff_math_fmod
        .hidden lj_ff_math_fmod
        .type lj_ff_math_fmod, @function
        .size lj_ff_math_fmod, 57
lj_ff_math_fmod:
        .byte 131,248,3,15,130,177,7,0,0,129,122,4,255,255,254,255
        .byte 15,131,164,7,0,0,129,122,12,255,255,254,255,15,131,151
        .byte 7,0,0,221,66,8,221,2,217,248,223,224,102,37,0,4
        .byte 117,246,221,217,233,192,251,255,255

        .globl lj_ff_math_pow
        .hidden lj_ff_math_pow
        .type lj_ff_math_pow, @function
        .size lj_ff_math_pow, 54
lj_ff_math_pow:
        .byte 131,248,3,15,130,120,7,0,0,129,122,4,255,255,254,255
        .byte 15,131,107,7,0,0,129,122,12,255,255,254,255,15,131,94
        .byte 7,0,0,242,15,16,2,242,15,16,74,8,232,52,12,0
        .byte 0,233,190,251,255,255

        .globl lj_ff_math_min
        .hidden lj_ff_math_min
        .type lj_ff_math_min, @function
        .size lj_ff_math_min, 59
lj_ff_math_min:
        .byte 185,2,0,0,0,129,122,4,255,255,254,255,15,131,57,7
        .byte 0,0,242,15,16,2,57,193,15,131,160,251,255,255,129,124
        .byte 202,252,255,255,254,255,15,131,31,7,0,0,242,15,16,76
        .byte 202,248,242,15,93,193,131,193,1,235,219

        .globl lj_ff_math_max
        .hidden lj_ff_math_max
        .type lj_ff_math_max, @function
        .size lj_ff_math_max, 59
lj_ff_math_max:
        .byte 185,2,0,0,0,129,122,4,255,255,254,255,15,131,254,6
        .byte 0,0,242,15,16,2,57,193,15,131,101,251,255,255,129,124
        .byte 202,252,255,255,254,255,15,131,228,6,0,0,242,15,16,76
        .byte 202,248,242,15,95,193,131,193,1,235,219

        .globl lj_ff_string_len
        .hidden lj_ff_string_len
        .type lj_ff_string_len, @function
        .size lj_ff_string_len, 31
lj_ff_string_len:
        .byte 131,248,2,15,130,204,6,0,0,131,122,4,251,15,133,194
        .byte 6,0,0,139,42,242,15,42,69,12,233,41,251,255,255
```

```
        .globl lj_ff_string_byte
        .hidden lj_ff_string_byte
        .type lj_ff_string_byte, @function
        .size lj_ff_string_byte, 47
lj_ff_string_byte:
        .byte 131,248,2,15,133,173,6,0,0,131,122,4,251,15,133,163
        .byte 6,0,0,139,42,139,90,252,131,125,12,1,15,130,218,247
        .byte 255,255,15,182,109,16,242,15,42,197,233,250,250,255,255

        .globl lj_ff_string_char
        .hidden lj_ff_string_char
        .type lj_ff_string_char, @function
        .size lj_ff_string_char, 76
lj_ff_string_char:
        .byte 65,139,174,104,244,255,255,65,59,174,108,244,255,255,114,5
        .byte 232,1,7,0,0,131,248,2,15,133,105,6,0,0,129,122
        .byte 4,255,255,254,255,15,131,92,6,0,0,242,15,44,42,129
        .byte 253,255,0,0,0,15,135,76,6,0,0,137,108,36,4,199
        .byte 68,36,8,1,0,0,0,72,141,68,36,4

        .globl lj_fff_newstr
        .hidden lj_fff_newstr
        .type lj_fff_newstr, @function
        .size lj_fff_newstr, 46
lj_fff_newstr:
        .byte 139,108,36,24,137,85,16,139,84,36,8,72,137,198,137,239
        .byte 137,92,36,28,232
        .long lj_str_new-.-4
        .byte 139,85,16,139,90,252,199,66,252,251,255,255,255,137,66,248
        .byte 233,136,250,255,255

        .globl lj_ff_string_sub
        .hidden lj_ff_string_sub
        .type lj_ff_string_sub, @function
        .size lj_ff_string_sub, 165
lj_ff_string_sub:
        .byte 65,139,174,104,244,255,255,65,59,174,108,244,255,255,114,5
        .byte 232,135,6,0,0,199,68,36,4,255,255,255,255,131,248,3
        .byte 15,130,231,5,0,0,118,22,129,122,20,255,255,254,255,15
        .byte 131,216,5,0,0,242,15,44,106,16,137,108,36,4,131,122
        .byte 4,251,15,133,197,5,0,0,129,122,12,255,255,254,255,15
        .byte 131,184,5,0,0,139,42,137,108,36,8,139,109,12,242,15
        .byte 44,74,8,139,68,36,4,57,197,114,30,133,201,126,38,139
        .byte 108,36,8,41,200,124,46,141,108,13,15,131,192,1,137,68
        .byte 36,8,137,232,233,73,255,255,255,124,6,141,68,40,1,235
        .byte 218,137,232,235,214,116,7,1,233,131,193,1,127,209,185,1
        .byte 0,0,0,235,202

        .globl lj_fff_emptystr
        .hidden lj_fff_emptystr
        .type lj_fff_emptystr, @function
        .size lj_fff_emptystr, 4
lj_fff_emptystr:
        .byte 49,192,235,213

        .globl lj_ff_string_rep
        .hidden lj_ff_string_rep
        .type lj_ff_string_rep, @function
        .size lj_ff_string_rep, 127
lj_ff_string_rep:
        .byte 65,139,174,104,244,255,255,65,59,174,108,244,255,255,114,5
        .byte 232,222,5,0,0,131,248,3,15,133,70,5,0,0,131,122
        .byte 4,251,15,133,60,5,0,0,129,122,12,255,255,254,255,139
        .byte 42,15,131,45,5,0,0,242,15,44,66,8,133,192,126,188
        .byte 131,125,12,1,114,182,15,133,12,5,0,0,65,57,134,172
        .byte 244,255,255,15,130,255,4,0,0,15,182,77,16,65,139,174
        .byte 160,244,255,255,137,68,36,8,136,77,0,131,197,1,131,232
        .byte 1,117,245,65,139,134,160,244,255,255,233,170,254,255,255

        .globl lj_ff_string_reverse
        .hidden lj_ff_string_reverse
        .type lj_ff_string_reverse, @function
        .size lj_ff_string_reverse, 110
lj_ff_string_reverse:
        .byte 131,248,2,15,130,220,4,0,0,65,139,174,104,244,255,255
        .byte 65,59,174,108,244,255,255,114,5,232,86,5,0,0,131,122
```

```
        .byte 4,251,15,133,189,4,0,0,139,42,139,69,12,133,192,15
        .byte 132,72,255,255,255,65,57,134,172,244,255,255,15,130,158,4
        .byte 0,0,131,197,16,137,92,36,4,137,68,36,8,65,139,158
        .byte 160,244,255,255,15,182,77,0,131,197,1,131,232,1,136,12
        .byte 3,117,241,137,216,139,92,36,4,233,60,254,255,255

        .globl lj_ff_string_lower
        .hidden lj_ff_string_lower
        .type lj_ff_string_lower, @function
        .size lj_ff_string_lower, 115
lj_ff_string_lower:
        .byte 131,248,2,15,130,110,4,0,0,65,139,174,104,244,255,255
        .byte 65,59,174,108,244,255,255,114,5,232,232,4,0,0,131,122
        .byte 4,251,15,133,79,4,0,0,139,42,139,69,12,65,57,134
        .byte 172,244,255,255,15,130,56,4,0,0,131,197,16,137,92,36
        .byte 4,137,68,36,8,65,139,158,160,244,255,255,235,21,15,182
        .byte 76,5,0,131,249,65,114,8,131,249,90,119,3,131,241,32
        .byte 136,12,3,131,232,1,121,230,137,216,139,92,36,4,233,201
        .byte 253,255,255

        .globl lj_ff_string_upper
        .hidden lj_ff_string_upper
        .type lj_ff_string_upper, @function
        .size lj_ff_string_upper, 115
lj_ff_string_upper:
        .byte 131,248,2,15,130,251,3,0,0,65,139,174,104,244,255,255
        .byte 65,59,174,108,244,255,255,114,5,232,117,4,0,0,131,122
        .byte 4,251,15,133,220,3,0,0,139,42,139,69,12,65,57,134
        .byte 172,244,255,255,15,130,197,3,0,0,131,197,16,137,92,36
        .byte 4,137,68,36,8,65,139,158,160,244,255,255,235,21,15,182
        .byte 76,5,0,131,249,97,114,8,131,249,122,119,3,131,241,32
        .byte 136,12,3,131,232,1,121,230,137,216,139,92,36,4,233,86
        .byte 253,255,255

        .globl lj_ff_table_getn
        .hidden lj_ff_table_getn
        .type lj_ff_table_getn, @function
        .size lj_ff_table_getn, 39
lj_ff_table_getn:
        .byte 131,248,2,15,130,136,3,0,0,131,122,4,244,15,133,126
        .byte 3,0,0,137,213,139,58,232
        .long lj_tab_len-.-4
        .byte 137,234,242,15,42,192,233,221,247,255,255

        .globl lj_ff_bit_tobit
        .hidden lj_ff_bit_tobit
        .type lj_ff_bit_tobit, @function
        .size lj_ff_bit_tobit, 54
lj_ff_bit_tobit:
        .byte 131,248,2,15,130,97,3,0,0,129,122,4,255,255,254,255
        .byte 15,131,84,3,0,0,242,15,16,2,72,189,0,0,0,0
        .byte 0,0,56,67,102,72,15,110,205,242,15,88,193,102,15,126
        .byte 197,233,139,1,0,0

        .globl lj_ff_bit_band
        .hidden lj_ff_bit_band
        .type lj_ff_bit_band, @function
        .size lj_ff_bit_band, 97
lj_ff_bit_band:
        .byte 131,248,2,15,130,43,3,0,0,72,189,0,0,0,0,0
        .byte 0,56,67,102,72,15,110,205,129,122,4,255,255,254,255,15
        .byte 131,15,3,0,0,242,15,16,2,242,15,88,193,102,15,126
        .byte 197,137,68,36,4,141,68,194,240,57,208,15,134,74,1,0
        .byte 0,129,120,4,255,255,254,255,15,131,70,1,0,0,242,15
        .byte 16,0,242,15,88,193,102,15,126,193,33,205,131,232,8,235
        .byte 216

        .globl lj_ff_bit_bor
        .hidden lj_ff_bit_bor
        .type lj_ff_bit_bor, @function
        .size lj_ff_bit_bor, 97
lj_ff_bit_bor:
        .byte 131,248,2,15,130,202,2,0,0,72,189,0,0,0,0,0
        .byte 0,56,67,102,72,15,110,205,129,122,4,255,255,254,255,15
        .byte 131,174,2,0,0,242,15,16,2,242,15,88,193,102,15,126
        .byte 197,137,68,36,4,141,68,194,240,57,208,15,134,233,0,0
```

```
        .byte 0,129,120,4,255,255,254,255,15,131,229,0,0,0,242,15
        .byte 16,0,242,15,88,193,102,15,126,193,9,205,131,232,8,235
        .byte 216

        .globl lj_ff_bit_bxor
        .hidden lj_ff_bit_bxor
        .type lj_ff_bit_bxor, @function
        .size lj_ff_bit_bxor, 97
lj_ff_bit_bxor:
        .byte 131,248,2,15,130,105,2,0,0,72,189,0,0,0,0,0
        .byte 0,56,67,102,72,15,110,205,129,122,4,255,255,254,255,15
        .byte 131,77,2,0,0,242,15,16,2,242,15,88,193,102,15,126
        .byte 197,137,68,36,4,141,68,194,240,57,208,15,134,136,0,0
        .byte 0,129,120,4,255,255,254,255,15,131,132,0,0,0,242,15
        .byte 16,0,242,15,88,193,102,15,126,193,49,205,131,232,8,235
        .byte 216

        .globl lj_ff_bit_bswap
        .hidden lj_ff_bit_bswap
        .type lj_ff_bit_bswap, @function
        .size lj_ff_bit_bswap, 53
lj_ff_bit_bswap:
        .byte 131,248,2,15,130,8,2,0,0,129,122,4,255,255,254,255
        .byte 15,131,251,1,0,0,242,15,16,2,72,189,0,0,0,0
        .byte 0,0,56,67,102,72,15,110,205,242,15,88,193,102,15,126
        .byte 197,15,205,235,51

        .globl lj_ff_bit_bnot
        .hidden lj_ff_bit_bnot
        .type lj_ff_bit_bnot, @function
        .size lj_ff_bit_bnot, 51
lj_ff_bit_bnot:
        .byte 131,248,2,15,130,211,1,0,0,129,122,4,255,255,254,255
        .byte 15,131,198,1,0,0,242,15,16,2,72,189,0,0,0,0
        .byte 0,0,56,67,102,72,15,110,205,242,15,88,193,102,15,126
        .byte 197,247,213

        .globl lj_fff_resbit
        .hidden lj_fff_resbit
        .type lj_fff_resbit, @function
        .size lj_fff_resbit, 9
lj_fff_resbit:
        .byte 242,15,42,197,233,19,246,255,255

        .globl lj_fff_fallback_bit_op
        .hidden lj_fff_fallback_bit_op
        .type lj_fff_fallback_bit_op, @function
        .size lj_fff_fallback_bit_op, 9
lj_fff_fallback_bit_op:
        .byte 139,68,36,4,233,151,1,0,0

        .globl lj_ff_bit_lshift
        .hidden lj_ff_bit_lshift
        .type lj_ff_bit_lshift, @function
        .size lj_ff_bit_lshift, 79
lj_ff_bit_lshift:
        .byte 131,248,3,15,130,142,1,0,0,129,122,4,255,255,254,255
        .byte 15,131,129,1,0,0,129,122,12,255,255,254,255,15,131,116
        .byte 1,0,0,242,15,16,2,242,15,16,74,8,72,189,0,0
        .byte 0,0,0,0,56,67,102,72,15,110,213,242,15,88,194,242
        .byte 15,88,202,102,15,126,197,102,15,126,201,211,229,235,159

        .globl lj_ff_bit_rshift
        .hidden lj_ff_bit_rshift
        .type lj_ff_bit_rshift, @function
        .size lj_ff_bit_rshift, 82
lj_ff_bit_rshift:
        .byte 131,248,3,15,130,63,1,0,0,129,122,4,255,255,254,255
        .byte 15,131,50,1,0,0,129,122,12,255,255,254,255,15,131,37
        .byte 1,0,0,242,15,16,2,242,15,16,74,8,72,189,0,0
        .byte 0,0,0,0,56,67,102,72,15,110,213,242,15,88,194,242
        .byte 15,88,202,102,15,126,197,102,15,126,201,211,237,233,77,255
        .byte 255,255

        .globl lj_ff_bit_arshift
        .hidden lj_ff_bit_arshift
```

```
        .type lj_ff_bit_arshift, @function
        .size lj_ff_bit_arshift, 82
lj_ff_bit_arshift:
        .byte 131,248,3,15,130,237,0,0,0,129,122,4,255,255,254,255
        .byte 15,131,224,0,0,0,129,122,12,255,255,254,255,15,131,211
        .byte 0,0,0,242,15,16,2,242,15,16,74,8,72,189,0,0
        .byte 0,0,0,0,56,67,102,72,15,110,213,242,15,88,194,242
        .byte 15,88,202,102,15,126,197,102,15,126,201,211,253,233,251,254
        .byte 255,255

        .globl lj_ff_bit_rol
        .hidden lj_ff_bit_rol
        .type lj_ff_bit_rol, @function
        .size lj_ff_bit_rol, 82
lj_ff_bit_rol:
        .byte 131,248,3,15,130,155,0,0,0,129,122,4,255,255,254,255
        .byte 15,131,142,0,0,0,129,122,12,255,255,254,255,15,131,129
        .byte 0,0,0,242,15,16,2,242,15,16,74,8,72,189,0,0
        .byte 0,0,0,0,56,67,102,72,15,110,213,242,15,88,194,242
        .byte 15,88,202,102,15,126,197,102,15,126,201,211,197,233,169,254
        .byte 255,255

        .globl lj_ff_bit_ror
        .hidden lj_ff_bit_ror
        .type lj_ff_bit_ror, @function
        .size lj_ff_bit_ror, 70
lj_ff_bit_ror:
        .byte 131,248,3,114,77,129,122,4,255,255,254,255,115,68,129,122
        .byte 12,255,255,254,255,115,59,242,15,16,2,242,15,16,74,8
        .byte 72,189,0,0,0,0,0,0,56,67,102,72,15,110,213,242
        .byte 15,88,194,242,15,88,202,102,15,126,197,102,15,126,201,211
        .byte 205,233,99,254,255,255

        .globl lj_fff_fallback_2
        .hidden lj_fff_fallback_2
        .type lj_fff_fallback_2, @function
        .size lj_fff_fallback_2, 7
lj_fff_fallback_2:
        .byte 184,3,0,0,0,235,5

        .globl lj_fff_fallback_1
        .hidden lj_fff_fallback_1
        .type lj_fff_fallback_1, @function
        .size lj_fff_fallback_1, 5
lj_fff_fallback_1:
        .byte 184,2,0,0,0

        .globl lj_fff_fallback
        .hidden lj_fff_fallback
        .type lj_fff_fallback, @function
        .size lj_fff_fallback, 87
lj_fff_fallback:
        .byte 139,108,36,24,139,90,252,137,92,36,28,137,85,16,141,68
        .byte 194,248,141,136,160,0,0,0,137,69,24,139,66,248,59,77
        .byte 32,119,89,137,239,255,80,24,139,85,16,133,192,15,143,77
        .byte 244,255,255,139,77,24,41,209,193,233,3,133,192,141,65,1
        .byte 139,106,248,117,18,139,93,16,139,11,15,182,233,15,182,205
        .byte 131,195,4,65,255,36,238

        .globl lj_vm_call_tail
        .hidden lj_vm_call_tail
        .type lj_vm_call_tail, @function
        .size lj_vm_call_tail, 56
lj_vm_call_tail:
        .byte 137,209,247,195,3,0,0,0,117,15,15,182,107,253,72,247
        .byte 213,141,20,234,233,197,233,255,255,137,221,131,229,248,41,234
        .byte 233,185,233,255,255,190,20,0,0,0,137,239,232
        .long lj_state_growstack-.-4
        .byte 139,85,16,49,192,235,164

        .globl lj_fff_gcstep
        .hidden lj_fff_gcstep
        .type lj_fff_gcstep, @function
        .size lj_fff_gcstep, 52
lj_fff_gcstep:
        .byte 93,72,137,108,36,8,139,108,36,24,137,92,36,28,137,85
```

```
        .byte 16,141,68,194,248,137,239,137,69,24,232
        .long lj_gc_step-.-4
        .byte 139,85,16,139,69,24,41,208,193,232,3,131,192,1,72,139
        .byte 108,36,8,85,195

        .globl lj_vm_record
        .hidden lj_vm_record
        .type lj_vm_record, @function
        .size lj_vm_record, 29
lj_vm_record:
        .byte 65,15,182,134,217,244,255,255,168,32,117,83,168,16,117,56
        .byte 168,12,116,52,65,255,142,16,245,255,255,235,43

        .globl lj_vm_rethook
        .hidden lj_vm_rethook
        .type lj_vm_rethook, @function
        .size lj_vm_rethook, 14
lj_vm_rethook:
        .byte 65,15,182,134,217,244,255,255,168,16,117,54,235,29

        .globl lj_vm_inshook
        .hidden lj_vm_inshook
        .type lj_vm_inshook, @function
        .size lj_vm_inshook, 68
lj_vm_inshook:
        .byte 65,15,182,134,217,244,255,255,168,16,117,40,168,12,116,36
        .byte 65,255,142,16,245,255,255,116,4,168,4,116,23,139,108,36
        .byte 24,137,85,16,137,222,137,239,232
        .long lj_dispatch_ins-.-4
        .byte 139,85,16,15,182,75,253,15,182,107,252,15,183,67,254,65
        .byte 255,164,238,216,4,0,0

        .globl lj_cont_hook
        .hidden lj_cont_hook
        .type lj_cont_hook, @function
        .size lj_cont_hook, 12
lj_cont_hook:
        .byte 131,195,4,139,77,232,137,76,36,4,235,224

        .globl lj_vm_hotloop
        .hidden lj_vm_hotloop
        .type lj_vm_hotloop, @function
        .size lj_vm_hotloop, 50
lj_vm_hotloop:
        .byte 139,106,248,139,109,16,15,182,69,199,141,4,194,139,108,36
        .byte 24,137,85,16,137,69,24,137,222,65,141,190,224,245,255,255
        .byte 73,137,174,64,246,255,255,137,92,36,28,232
        .long lj_trace_hot-.-4
        .byte 235,171

        .globl lj_vm_callhook
        .hidden lj_vm_callhook
        .type lj_vm_callhook, @function
        .size lj_vm_callhook, 6
lj_vm_callhook:
        .byte 137,92,36,28,235,7

        .globl lj_vm_hotcall
        .hidden lj_vm_hotcall
        .type lj_vm_hotcall, @function
        .size lj_vm_hotcall, 67
lj_vm_hotcall:
        .byte 137,92,36,28,131,203,1,141,68,194,248,139,108,36,24,137
        .byte 85,16,137,69,24,137,222,137,239,232
        .long lj_dispatch_call-.-4
        .byte 199,68,36,28,0,0,0,0,131,227,254,139,85,16,72,137
        .byte 193,139,69,24,41,208,72,137,205,15,182,75,253,193,232,3
        .byte 131,192,1,255,229

        .globl lj_vm_exit_handler
        .hidden lj_vm_exit_handler
        .type lj_vm_exit_handler, @function
        .size lj_vm_exit_handler, 247
lj_vm_exit_handler:
        .byte 65,85,65,84,65,83,65,82,65,81,65,80,87,86,85,72
        .byte 141,108,36,88,85,83,82,81,80,15,182,69,248,138,101,240
```

```
        .byte 76,137,125,248,76,137,117,240,68,139,117,0,65,139,142,48
        .byte 245,255,255,65,199,134,48,245,255,255,252,255,255,255,65,137
        .byte 134,60,255,255,255,65,137,142,56,255,255,255,72,129,236,128
        .byte 0,0,0,72,131,197,128,242,68,15,17,125,248,242,68,15
        .byte 17,117,240,242,68,15,17,109,232,242,68,15,17,101,224,242
        .byte 68,15,17,93,216,242,68,15,17,85,208,242,68,15,17,77
        .byte 200,242,68,15,17,69,192,242,15,17,125,184,242,15,17,117
        .byte 176,242,15,17,109,168,242,15,17,101,160,242,15,17,93,152
        .byte 242,15,17,85,144,242,15,17,77,136,242,15,17,69,128,65
        .byte 139,174,60,245,255,255,65,139,150,64,245,255,255,73,137,174
        .byte 64,246,255,255,65,199,134,60,245,255,255,0,0,0,0,137
        .byte 85,16,72,137,230,65,141,190,224,245,255,255,232
        .long lj_trace_exit-.-4
        .byte 72,139,77,48,72,131,225,252,72,137,204,137,105,24,139,85
        .byte 16,139,89,28,235,4

        .globl lj_vm_exit_interp
        .hidden lj_vm_exit_interp
        .type lj_vm_exit_interp, @function
        .size lj_vm_exit_interp, 93
lj_vm_exit_interp:
        .byte 72,131,196,16,76,139,108,36,8,76,139,36,36,133,192,120
        .byte 65,137,68,36,4,68,139,122,248,69,139,127,16,69,139,127
        .byte 208,65,199,134,60,245,255,255,0,0,0,0,65,199,134,48
        .byte 245,255,255,255,255,255,255,139,3,15,182,204,15,182,232,131
        .byte 195,4,193,232,16,131,253,85,114,4,139,68,36,4,65,255
        .byte 36,238,247,216,137,239,137,198,232
        .long lj_err_throw-.-4

        .globl lj_vm_floor
        .hidden lj_vm_floor
        .type lj_vm_floor, @function
        .size lj_vm_floor, 0
lj_vm_floor:

        .globl lj_vm_floor_sse
        .hidden lj_vm_floor_sse
        .type lj_vm_floor_sse, @function
        .size lj_vm_floor_sse, 91
lj_vm_floor_sse:
        .byte 72,184,255,255,255,255,255,255,255,127,102,72,15,110,208,72
        .byte 184,0,0,0,0,0,0,48,67,102,72,15,110,216,15,40
        .byte 200,102,15,84,202,102,15,46,217,118,47,102,15,85,208,242
        .byte 15,88,203,242,15,92,203,102,15,86,202,72,184,0,0,0
        .byte 0,0,0,240,63,102,72,15,110,208,242,15,194,193,1,102
        .byte 15,84,194,242,15,92,200,15,40,193,195

        .globl lj_vm_ceil
        .hidden lj_vm_ceil
        .type lj_vm_ceil, @function
        .size lj_vm_ceil, 0
lj_vm_ceil:

        .globl lj_vm_ceil_sse
        .hidden lj_vm_ceil_sse
        .type lj_vm_ceil_sse, @function
        .size lj_vm_ceil_sse, 91
lj_vm_ceil_sse:
        .byte 72,184,255,255,255,255,255,255,255,127,102,72,15,110,208,72
        .byte 184,0,0,0,0,0,0,48,67,102,72,15,110,216,15,40
        .byte 200,102,15,84,202,102,15,46,217,118,47,102,15,85,208,242
        .byte 15,88,203,242,15,92,203,102,15,86,202,72,184,0,0,0
        .byte 0,0,0,240,191,102,72,15,110,208,242,15,194,193,6,102
        .byte 15,84,194,242,15,92,200,15,40,193,195

        .globl lj_vm_trunc
        .hidden lj_vm_trunc
        .type lj_vm_trunc, @function
        .size lj_vm_trunc, 0
lj_vm_trunc:

        .globl lj_vm_trunc_sse
        .hidden lj_vm_trunc_sse
        .type lj_vm_trunc_sse, @function
        .size lj_vm_trunc_sse, 94
lj_vm_trunc_sse:
```

```
        .byte 72,184,255,255,255,255,255,255,255,127,102,72,15,110,208,72
        .byte 184,0,0,0,0,0,0,48,67,102,72,15,110,216,15,40
        .byte 200,102,15,84,202,102,15,46,217,118,50,102,15,85,208,15
        .byte 40,193,242,15,88,203,242,15,92,203,72,184,0,0,0,0
        .byte 0,0,240,63,102,72,15,110,216,242,15,194,193,1,102,15
        .byte 84,195,242,15,92,200,102,15,86,202,15,40,193,195

        .globl lj_vm_mod
        .hidden lj_vm_mod
        .type lj_vm_mod, @function
        .size lj_vm_mod, 118
lj_vm_mod:
        .byte 15,40,232,242,15,94,193,72,184,255,255,255,255,255,255,255
        .byte 127,102,72,15,110,208,72,184,0,0,0,0,0,0,48,67
        .byte 102,72,15,110,216,15,40,224,102,15,84,226,102,15,46,220
        .byte 118,56,102,15,85,208,242,15,88,227,242,15,92,227,102,15
        .byte 86,226,72,184,0,0,0,0,0,0,240,63,102,72,15,110
        .byte 208,242,15,194,196,1,102,15,84,194,242,15,92,224,15,40
        .byte 197,242,15,89,204,242,15,92,193,195,242,15,89,200,15,40
        .byte 197,242,15,92,193,195

        .globl lj_vm_log2
        .hidden lj_vm_log2
        .type lj_vm_log2, @function
        .size lj_vm_log2, 25
lj_vm_log2:
        .byte 242,15,17,68,36,248,217,232,221,68,36,248,217,241,221,92
        .byte 36,248,242,15,16,68,36,248,195

        .globl lj_vm_exp_x87
        .hidden lj_vm_exp_x87
        .type lj_vm_exp_x87, @function
        .size lj_vm_exp_x87, 4
lj_vm_exp_x87:
        .byte 217,234,222,201

        .globl lj_vm_exp2_x87
        .hidden lj_vm_exp2_x87
        .type lj_vm_exp2_x87, @function
        .size lj_vm_exp2_x87, 24
lj_vm_exp2_x87:
        .byte 217,84,36,248,129,124,36,248,0,0,128,127,116,28,129,124
        .byte 36,248,0,0,128,255,116,19

        .globl lj_vm_exp2raw
        .hidden lj_vm_exp2raw
        .type lj_vm_exp2raw, @function
        .size lj_vm_exp2raw, 24
lj_vm_exp2raw:
        .byte 217,192,217,252,220,233,217,201,217,240,217,232,222,193,217,253
        .byte 221,217,195,221,216,217,238,195

        .globl lj_vm_pow
        .hidden lj_vm_pow
        .type lj_vm_pow, @function
        .size lj_vm_pow, 0
lj_vm_pow:

        .globl lj_vm_pow_sse
        .hidden lj_vm_pow_sse
        .type lj_vm_pow_sse, @function
        .size lj_vm_pow_sse, 24
lj_vm_pow_sse:
        .byte 242,15,45,193,242,15,42,208,102,15,46,202,15,133,104,0
        .byte 0,0,15,138,199,0,0,0

        .globl lj_vm_powi_sse
        .hidden lj_vm_powi_sse
        .type lj_vm_powi_sse, @function
        .size lj_vm_powi_sse, 320
lj_vm_powi_sse:
        .byte 131,248,1,126,43,169,1,0,0,0,117,8,242,15,89,192
        .byte 209,232,235,241,209,232,116,23,15,40,200,242,15,89,192,209
        .byte 232,116,8,115,246,242,15,89,200,235,240,242,15,89,193,195
        .byte 116,253,114,30,247,216,232,202,255,255,255,72,184,0,0,0
        .byte 0,0,0,240,63,102,72,15,110,200,242,15,94,200,15,40
```

```
        .byte 193,195,72,184,0,0,0,0,0,0,240,63,102,72,15,110
        .byte 192,195,102,72,15,126,200,72,209,224,72,193,192,12,72,61
        .byte 254,15,0,0,116,106,102,72,15,126,192,72,209,224,15,132
        .byte 164,0,0,0,72,193,192,12,72,61,254,15,0,0,15,132
        .byte 160,0,0,0,242,15,17,76,36,240,242,15,17,68,36,248
        .byte 221,68,36,240,221,68,36,248,217,241,217,192,217,252,220,233
        .byte 217,201,217,240,217,232,222,193,217,253,221,217,221,92,36,248
        .byte 242,15,16,68,36,248,195,72,184,0,0,0,0,0,0,240
        .byte 63,102,72,15,110,208,102,15,46,194,116,3,15,40,193,195
        .byte 72,184,255,255,255,255,255,255,255,127,102,72,15,110,208,102
        .byte 15,84,194,72,184,0,0,0,0,0,0,240,63,102,72,15
        .byte 110,208,102,15,46,194,116,215,102,15,80,193,15,87,192,136
        .byte 196,15,146,208,48,224,117,199,72,184,0,0,0,0,0,0
        .byte 240,127,102,72,15,110,192,195,102,15,80,193,133,192,117,232
        .byte 15,87,192,195,102,15,80,193,133,192,116,220,15,87,192,195

        .globl lj_vm_foldfpm
        .hidden lj_vm_foldfpm
        .type lj_vm_foldfpm, @function
        .size lj_vm_foldfpm, 131
lj_vm_foldfpm:
        .byte 131,255,1,15,130,200,252,255,255,15,132,29,253,255,255,131
        .byte 255,3,15,130,111,253,255,255,119,5,242,15,81,192,195,242
        .byte 15,17,68,36,248,221,68,36,248,131,255,5,119,16,116,7
        .byte 232,63,254,255,255,235,64,232,60,254,255,255,235,57,131,255
        .byte 7,116,10,119,16,217,237,217,201,217,241,235,42,217,232,217
        .byte 201,217,241,235,34,131,255,9,116,10,119,12,217,236,217,201
        .byte 217,241,235,19,217,254,235,15,131,255,11,116,6,119,19,217
        .byte 255,235,4,217,242,221,216,221,92,36,248,242,15,16,68,36
        .byte 248,195,204

        .globl lj_vm_foldarith
        .hidden lj_vm_foldarith
        .type lj_vm_foldarith, @function
        .size lj_vm_foldarith, 160
lj_vm_foldarith:
        .byte 131,255,1,116,7,119,10,242,15,88,193,195,242,15,92,193
        .byte 195,131,255,3,116,7,119,10,242,15,89,193,195,242,15,94
        .byte 193,195,131,255,5,15,130,55,253,255,255,15,132,244,253,255
        .byte 255,131,255,7,116,21,119,38,72,184,0,0,0,0,0,0
        .byte 0,128,102,72,15,110,200,15,87,193,195,72,184,255,255,255
        .byte 255,255,255,255,127,102,72,15,110,200,15,84,193,195,131,255
        .byte 9,119,43,242,15,17,68,36,248,242,15,17,76,36,240,221
        .byte 68,36,248,221,68,36,240,116,13,217,243,221,92,36,248,242
        .byte 15,16,68,36,248,195,217,201,217,253,221,217,235,237,131,255
        .byte 11,116,7,119,10,242,15,93,193,195,242,15,95,193,195,204

        .globl lj_vm_cpuid
        .hidden lj_vm_cpuid
        .type lj_vm_cpuid, @function
        .size lj_vm_cpuid, 18
lj_vm_cpuid:
        .byte 137,248,83,15,162,137,6,137,94,4,137,78,8,137,86,12
        .byte 91,195

        .globl lj_assert_bad_for_arg_type
        .hidden lj_assert_bad_for_arg_type
        .type lj_assert_bad_for_arg_type, @function
        .size lj_assert_bad_for_arg_type, 2
lj_assert_bad_for_arg_type:
        .byte 204,204

        .globl lj_vm_ffi_callback
        .hidden lj_vm_ffi_callback
        .type lj_vm_ffi_callback, @function
        .size lj_vm_ffi_callback, 179
lj_vm_ffi_callback:
        .byte 83,65,87,65,86,72,131,236,40,68,141,181,184,11,0,0
        .byte 139,157,252,0,0,0,15,183,192,137,131,208,0,0,0,72
        .byte 137,123,112,72,137,115,120,72,137,147,128,0,0,0,72,137
        .byte 139,136,0,0,0,242,15,17,67,48,242,15,17,75,56,242
        .byte 15,17,83,64,242,15,17,91,72,72,141,68,36,80,76,137
        .byte 131,144,0,0,0,76,137,139,152,0,0,0,242,15,17,99
        .byte 80,242,15,17,107,88,242,15,17,115,96,242,15,17,123,104
        .byte 72,137,131,176,0,0,0,72,137,230,137,92,36,28,137,223
        .byte 232
```

```
        .long lj_ccallback_enter-.-4
        .byte 65,199,134,48,245,255,255,255,255,255,255,139,80,16,139,64
        .byte 24,41,208,139,106,248,193,232,3,131,192,1,139,93,16,139
        .byte 11,15,182,233,15,182,205,131,195,4,65,255,36,238

        .globl lj_cont_ffi_callback
        .hidden lj_cont_ffi_callback
        .type lj_cont_ffi_callback, @function
        .size lj_cont_ffi_callback, 44
lj_cont_ffi_callback:
        .byte 139,76,36,24,65,139,158,68,245,255,255,72,137,75,16,137
        .byte 81,16,137,105,24,137,223,137,198,232
        .long lj_ccallback_leave-.-4
        .byte 72,139,67,112,242,15,16,67,48,233,255,223,255,255

        .globl lj_vm_ffi_call
        .hidden lj_vm_ffi_call
        .type lj_vm_ffi_call, @function
        .size lj_vm_ffi_call, 160
lj_vm_ffi_call:
        .byte 85,72,137,229,83,72,137,251,139,67,8,72,41,196,15,182
        .byte 75,12,131,233,1,120,17,72,139,132,203,192,0,0,0,72
        .byte 137,4,204,131,233,1,121,239,15,182,67,15,72,139,187,144
        .byte 0,0,0,72,139,179,152,0,0,0,72,139,147,160,0,0
        .byte 0,72,139,139,168,0,0,0,76,139,131,176,0,0,0,76
        .byte 139,139,184,0,0,0,133,192,116,40,15,40,67,16,15,40
        .byte 75,32,15,40,83,48,15,40,91,64,131,248,4,118,19,15
        .byte 40,99,80,15,40,107,96,15,40,115,112,15,40,187,128,0
        .byte 0,0,255,19,72,137,131,144,0,0,0,15,41,67,16,72
        .byte 137,147,152,0,0,0,15,41,75,32,72,139,93,248,201,195

        .section .note.GNU-stack,"",@progbits
        .ident "DynASM 1.3.0"

        .section .debug_frame,"",@progbits
.Lframe0:
        .long .LECIE0-.LSCIE0
.LSCIE0:
        .long 0xffffffff
        .byte 0x1
        .string ""
        .uleb128 0x1
        .sleb128 -8
        .byte 0x10
        .byte 0xc
        .uleb128 0x7
        .uleb128 8
        .byte 0x80+0x10
        .uleb128 0x1
        .align 8
.LECIE0:

.LSFDE0:
        .long .LEFDE0-.LASFDE0
.LASFDE0:
        .long .Lframe0
        .quad .Lbegin
        .quad 14652
        .byte 0xe
        .uleb128 80
        .byte 0x86
        .uleb128 0x2
        .byte 0x83
        .uleb128 0x3
        .byte 0x8f
        .uleb128 0x4
        .byte 0x8e
        .uleb128 0x5
        .align 8
.LEFDE0:

.LSFDE1:
        .long .LEFDE1-.LASFDE1
.LASFDE1:
        .long .Lframe0
        .quad lj_vm_ffi_call
```

```
        .quad 160
        .byte 0xe
        .uleb128 16
        .byte 0x86
        .uleb128 0x2
        .byte 0xd
        .uleb128 0x6
        .byte 0x83
        .uleb128 0x3
        .align 8
.LEFDE1:

        .section .eh_frame,"a",@progbits
.Lframe1:
        .long .LECIE1-.LSCIE1
.LSCIE1:
        .long 0
        .byte 0x1
        .string "zPR"
        .uleb128 0x1
        .sleb128 -8
        .byte 0x10
        .uleb128 6
        .byte 0x1b
        .long lj_err_unwind_dwarf-.
        .byte 0x1b
        .byte 0xc
        .uleb128 0x7
        .uleb128 8
        .byte 0x80+0x10
        .uleb128 0x1
        .align 8
.LECIE1:

.LSFDE2:
        .long .LEFDE2-.LASFDE2
.LASFDE2:
        .long .LASFDE2-.Lframe1
        .long .Lbegin-.
        .long 14652
        .uleb128 0
        .byte 0xe
        .uleb128 80
        .byte 0x86
        .uleb128 0x2
        .byte 0x83
        .uleb128 0x3
        .byte 0x8f
        .uleb128 0x4
        .byte 0x8e
        .uleb128 0x5
        .align 8
.LEFDE2:

.Lframe2:
        .long .LECIE2-.LSCIE2
.LSCIE2:
        .long 0
        .byte 0x1
        .string "zR"
        .uleb128 0x1
        .sleb128 -8
        .byte 0x10
        .uleb128 1
        .byte 0x1b
        .byte 0xc
        .uleb128 0x7
        .uleb128 8
        .byte 0x80+0x10
        .uleb128 0x1
        .align 8
.LECIE2:

.LSFDE3:
        .long .LEFDE3-.LASFDE3
.LASFDE3:
```

```
2732          .long .LASFDE3-.Lframe2
2733          .long lj_vm_ffi_call-.
2734          .long 160
2735          .uleb128 0
2736          .byte 0xe
2737          .uleb128 16
2738          .byte 0x86
2739          .uleb128 0x2
2740          .byte 0xd
2741          .uleb128 0x6
2742          .byte 0x83
2743          .uleb128 0x3
2744          .align 8
2745  .LEFDE3:
2746
```

One Level Up Top Level

src/lua.hpp - luajit-2.0-src

```
1 // C++ wrapper for LuaJIT header files.
2
3 extern "C" {
4 #include "lua.h"
5 #include "lauxlib.h"
6 #include "lualib.h"
7 #include "luajit.h"
8 }
9
```

src/msvcbuild.bat - luajit-2.0-src

Global variables defined

- ALL_LIB
- DASM
- DASMDIR
- DASMFLAGS
- DASMFLAGS
- LJARCH
- LJARCH
- LJCOMPILE
- LJCOMPILE
- LJCOMPILE
- LJDLLNAME
- LJLIB
- LJLIBNAME
- LJLINK
- LJLINK
- LJMT

Source code

```
@rem Script to build LuaJIT with MSVC.
@rem Copyright (C) 2005-2015 Mike Pall. See Copyright Notice in luajit.h
@rem
@rem Either open a "Visual Studio .NET Command Prompt"
@rem (Note that the Express Edition does not contain an x64 compiler)
@rem -or-
@rem Open a "Windows SDK Command Shell" and set the compiler environment:
@rem     setenv /release /x86
@rem   -or-
 @rem     setenv /release /x64
 @rem
 @rem Then cd to this directory and run this script.

 @if not defined INCLUDE goto :FAIL

 @setlocal
 @set LJCOMPILE=cl /nologo /c /O2 /W3 /D_CRT_SECURE_NO_DEPRECATE
 @set LJLINK=link /nologo
 @set LJMT=mt /nologo
 @set LJLIB=lib /nologo /nodefaultlib
 @set DASMDIR=..\dynasm
 @set DASM=%DASMDIR%\dynasm.lua
 @set LJDLLNAME=lua51.dll
 @set LJLIBNAME=lua51.lib
 @set ALL_LIB=lib_base.c lib_math.c lib_bit.c lib_string.c lib_table.c lib_io.c lib_os.c lib_package.c
lib_debug.c lib_jit.c lib_ffi.c

 %LJCOMPILE% host\minilua.c
```

```
@if errorlevel 1 goto :BAD
%LJLINK% /out:minilua.exe minilua.obj
@if errorlevel 1 goto :BAD
if exist minilua.exe.manifest^
  %LJMT% -manifest minilua.exe.manifest -outputresource:minilua.exe

@set DASMFLAGS=-D WIN -D JIT -D FFI -D P64
@set LJARCH=x64
@minilua
@if errorlevel 8 goto :X64
@set DASMFLAGS=-D WIN -D JIT -D FFI
@set LJARCH=x86
@set LJCOMPILE=%LJCOMPILE% /arch:SSE2
:X64
minilua %DASM% -LN %DASMFLAGS% -o host\buildvm_arch.h vm_x86.dasc
@if errorlevel 1 goto :BAD

%LJCOMPILE% /I "." /I %DASMDIR% host\buildvm*.c
@if errorlevel 1 goto :BAD
%LJLINK% /out:buildvm.exe buildvm*.obj
@if errorlevel 1 goto :BAD
if exist buildvm.exe.manifest^
  %LJMT% -manifest buildvm.exe.manifest -outputresource:buildvm.exe

buildvm -m peobj -o lj_vm.obj
@if errorlevel 1 goto :BAD
buildvm -m bcdef -o lj_bcdef.h %ALL_LIB%
@if errorlevel 1 goto :BAD
buildvm -m ffdef -o lj_ffdef.h %ALL_LIB%
@if errorlevel 1 goto :BAD
buildvm -m libdef -o lj_libdef.h %ALL_LIB%
@if errorlevel 1 goto :BAD
buildvm -m recdef -o lj_recdef.h %ALL_LIB%
@if errorlevel 1 goto :BAD
buildvm -m vmdef -o jit\vmdef.lua %ALL_LIB%
@if errorlevel 1 goto :BAD
buildvm -m folddef -o lj_folddef.h lj_opt_fold.c
@if errorlevel 1 goto :BAD

@if "%1" neq "debug" goto :NODEBUG
@shift
@set LJCOMPILE=%LJCOMPILE% /Zi
@set LJLINK=%LJLINK% /debug
:NODEBUG
@if "%1"=="amalg" goto :AMALGDLL
@if "%1"=="static" goto :STATIC
%LJCOMPILE% /MD /DLUA_BUILD_AS_DLL lj_*.c lib_*.c
@if errorlevel 1 goto :BAD
%LJLINK% /DLL /out:%LJDLLNAME% lj_*.obj lib_*.obj
@if errorlevel 1 goto :BAD
@goto :MTDLL
:STATIC
%LJCOMPILE% lj_*.c lib_*.c
@if errorlevel 1 goto :BAD
%LJLIB% /OUT:%LJLIBNAME% lj_*.obj lib_*.obj
@if errorlevel 1 goto :BAD
@goto :MTDLL
:AMALGDLL
%LJCOMPILE% /MD /DLUA_BUILD_AS_DLL ljamalg.c
@if errorlevel 1 goto :BAD
%LJLINK% /DLL /out:%LJDLLNAME% ljamalg.obj lj_vm.obj
@if errorlevel 1 goto :BAD
:MTDLL
if exist %LJDLLNAME%.manifest^
  %LJMT% -manifest %LJDLLNAME%.manifest -outputresource:%LJDLLNAME%;2

%LJCOMPILE% luajit.c
@if errorlevel 1 goto :BAD
%LJLINK% /out:luajit.exe luajit.obj %LJLIBNAME%
@if errorlevel 1 goto :BAD
if exist luajit.exe.manifest^
  %LJMT% -manifest luajit.exe.manifest -outputresource:luajit.exe

 @del *.obj *.manifest minilua.exe buildvm.exe
 @echo.
 @echo === Successfully built LuaJIT for Windows/%LJARCH% ===
```

```
104
105 @goto :END
106 :BAD
107 @echo.
108 @echo ******************************************************
109 @echo *** Build FAILED -- Please check the error messages ***
110 @echo ******************************************************
111 @goto :END
112 :FAIL
113 @echo You must open a "Visual Studio .NET Command Prompt" to run this script
114 :END
```

One Level Up Top Level

src/ps4build.bat - luajit-2.0-src

Global variables defined

- ALL_LIB
- DASM
- DASMDIR
- DASMFLAGS
- INCLUDE
- LJCOMPILE
- LJCOMPILE
- LJCOMPILE
- LJCOMPILE
- LJLIB
- LJLINK
- LJMT
- TARGETLIB
- TARGETLIB

Source code

```
@rem Script to build LuaJIT with the PS4 SDK.
@rem Donated to the public domain.
@rem
@rem Open a "Visual Studio .NET Command Prompt" (64 bit host compiler)
@rem Then cd to this directory and run this script.

@if not defined INCLUDE goto :FAIL
@if not defined SCE_ORBIS_SDK_DIR goto :FAIL

 @setlocal
 @rem ---- Host compiler ----
 @set LJCOMPILE=cl /nologo /c /MD /O2 /W3 /D_CRT_SECURE_NO_DEPRECATE
 @set LJLINK=link /nologo
 @set LJMT=mt /nologo
 @set DASMDIR=..\dynasm
 @set DASM=%DASMDIR%\dynasm.lua
 @set ALL_LIB=lib_base.c lib_math.c lib_bit.c lib_string.c lib_table.c lib_io.c lib_os.c lib_package.c
lib_debug.c lib_jit.c lib_ffi.c

 %LJCOMPILE% host\minilua.c
 @if errorlevel 1 goto :BAD
 %LJLINK% /out:minilua.exe minilua.obj
 @if errorlevel 1 goto :BAD
 if exist minilua.exe.manifest^
   %LJMT% -manifest minilua.exe.manifest -outputresource:minilua.exe

 @rem Check for 64 bit host compiler.
 @minilua
 @if not errorlevel 8 goto :FAIL

 @set DASMFLAGS=-D P64
 minilua %DASM% -LN %DASMFLAGS% -o host\buildvm_arch.h vm_x86.dasc
 @if errorlevel 1 goto :BAD
```

```
 33
 34  %LJCOMPILE% /I "." /I %DASMDIR% -DLUAJIT_TARGET=LUAJIT_ARCH_X64 -DLUAJIT_OS=LUAJIT_OS_OTHER -
DLUAJIT_DISABLE_JIT -DLUAJIT_DISABLE_FFI host\buildvm*.c
 35  @if errorlevel 1 goto :BAD
 36  %LJLINK% /out:buildvm.exe buildvm*.obj
 37  @if errorlevel 1 goto :BAD
 38  if exist buildvm.exe.manifest^
 39    %LJMT% -manifest buildvm.exe.manifest -outputresource:buildvm.exe
 40
 41  buildvm -m elfasm -o lj_vm.s
 42  @if errorlevel 1 goto :BAD
 43  buildvm -m bcdef -o lj_bcdef.h %ALL_LIB%
 44  @if errorlevel 1 goto :BAD
 45  buildvm -m ffdef -o lj_ffdef.h %ALL_LIB%
 46  @if errorlevel 1 goto :BAD
 47  buildvm -m libdef -o lj_libdef.h %ALL_LIB%
 48  @if errorlevel 1 goto :BAD
 49  buildvm -m recdef -o lj_recdef.h %ALL_LIB%
 50  @if errorlevel 1 goto :BAD
 51  buildvm -m vmdef -o jit\vmdef.lua %ALL_LIB%
 52  @if errorlevel 1 goto :BAD
 53  buildvm -m folddef -o lj_folddef.h lj_opt_fold.c
 54  @if errorlevel 1 goto :BAD
 55
 56  @rem ---- Cross compiler ----
 57  @set LJCOMPILE="%SCE_ORBIS_SDK_DIR%\host_tools\bin\orbis-clang" -c -Wall -DLUAJIT_DISABLE_FFI
 58  @set LJLIB="%SCE_ORBIS_SDK_DIR%\host_tools\bin\orbis-ar" rcus
 59  @set INCLUDE=""
 60
 61  orbis-as -o lj_vm.o lj_vm.s
 62
 63  @if "%1" neq "debug" goto :NODEBUG
 64  @shift
 65  @set LJCOMPILE=%LJCOMPILE% -g -O0
 66  @set TARGETLIB=libluajitD.a
 67  goto :BUILD
 68  :NODEBUG
 69  @set LJCOMPILE=%LJCOMPILE% -O2
 70  @set TARGETLIB=libluajit.a
 71  :BUILD
 72  del %TARGETLIB%
 73  @if "%1"=="amalg" goto :AMALG
 74  for %%f in (lj_*.c lib_*.c) do (
 75    %LJCOMPILE% %%f
 76    @if errorlevel 1 goto :BAD
 77  )
 78
 79  %LJLIB% %TARGETLIB% lj_*.o lib_*.o
 80  @if errorlevel 1 goto :BAD
 81  @goto :NOAMALG
 82  :AMALG
 83  %LJCOMPILE% ljamalg.c
 84  @if errorlevel 1 goto :BAD
 85  %LJLIB% %TARGETLIB% ljamalg.o lj_vm.o
 86  @if errorlevel 1 goto :BAD
 87  :NOAMALG
 88
 89  @del *.o *.obj *.manifest minilua.exe buildvm.exe
 90  @echo.
 91  @echo === Successfully built LuaJIT for PS4 ===
 92
 93  @goto :END
 94  :BAD
 95  @echo.
 96  @echo *********************************************************
 97  @echo *** Build FAILED -- Please check the error messages ***
 98  @echo *********************************************************
 99  @goto :END
100   :FAIL
101   @echo To run this script you must open a "Visual Studio .NET Command Prompt"
102   @echo (64 bit host compiler). The PS4 Orbis SDK must be installed, too.
103   :END
```

One Level Up Top Level

src/psvitabuild.bat - luajit-2.0-src

Global variables defined

- ALL_LIB
- DASM
- DASMDIR
- DASMFLAGS
- INCLUDE
- LJCOMPILE
- LJCOMPILE
- LJCOMPILE
- LJCOMPILE
- LJLIB
- LJLINK
- LJMT
- TARGETLIB
- TARGETLIB

Source code

```
1  @rem Script to build LuaJIT with the PS Vita SDK.
2  @rem Donated to the public domain.
3  @rem
4  @rem Open a "Visual Studio .NET Command Prompt" (32 bit host compiler)
5  @rem Then cd to this directory and run this script.
6
7  @if not defined INCLUDE goto :FAIL
8  @if not defined SCE_PSP2_SDK_DIR goto :FAIL
9
10  @setlocal
11  @rem ---- Host compiler ----
12  @set LJCOMPILE=cl /nologo /c /MD /O2 /W3 /D_CRT_SECURE_NO_DEPRECATE
13  @set LJLINK=link /nologo
14  @set LJMT=mt /nologo
15  @set DASMDIR=..\dynasm
16  @set DASM=%DASMDIR%\dynasm.lua
17  @set ALL_LIB=lib_base.c lib_math.c lib_bit.c lib_string.c lib_table.c lib_io.c lib_os.c lib_package.c
lib_debug.c lib_jit.c lib_ffi.c
18
19  %LJCOMPILE% host\minilua.c
20  @if errorlevel 1 goto :BAD
21  %LJLINK% /out:minilua.exe minilua.obj
22  @if errorlevel 1 goto :BAD
23  if exist minilua.exe.manifest^
24    %LJMT% -manifest minilua.exe.manifest -outputresource:minilua.exe
25
26  @rem Check for 32 bit host compiler.
27  @minilua
28  @if errorlevel 8 goto :FAIL
29
30  @set DASMFLAGS=-D FPU -D HFABI
31  minilua %DASM% -LN %DASMFLAGS% -o host\buildvm_arch.h vm_arm.dasc
32  @if errorlevel 1 goto :BAD
```

```
33
34  %LJCOMPILE% /I "." /I %DASMDIR% -DLUAJIT_TARGET=LUAJIT_ARCH_ARM -DLUAJIT_OS=LUAJIT_OS_OTHER -
DLUAJIT_DISABLE_JIT -DLUAJIT_DISABLE_FFI -DLJ_TARGET_PSVITA=1 host\buildvm*.c
35  @if errorlevel 1 goto :BAD
36  %LJLINK% /out:buildvm.exe buildvm*.obj
37  @if errorlevel 1 goto :BAD
38  if exist buildvm.exe.manifest^
39    %LJMT% -manifest buildvm.exe.manifest -outputresource:buildvm.exe
40
41  buildvm -m elfasm -o lj_vm.s
42  @if errorlevel 1 goto :BAD
43  buildvm -m bcdef -o lj_bcdef.h %ALL_LIB%
44  @if errorlevel 1 goto :BAD
45  buildvm -m ffdef -o lj_ffdef.h %ALL_LIB%
46  @if errorlevel 1 goto :BAD
47  buildvm -m libdef -o lj_libdef.h %ALL_LIB%
48  @if errorlevel 1 goto :BAD
49  buildvm -m recdef -o lj_recdef.h %ALL_LIB%
50  @if errorlevel 1 goto :BAD
51  buildvm -m vmdef -o jit\vmdef.lua %ALL_LIB%
52  @if errorlevel 1 goto :BAD
53  buildvm -m folddef -o lj_folddef.h lj_opt_fold.c
54  @if errorlevel 1 goto :BAD
55
56  @rem ---- Cross compiler ----
57  @set LJCOMPILE="%SCE_PSP2_SDK_DIR%\host_tools\build\bin\psp2snc" -c -w -DLUAJIT_DISABLE_FFI -
DLUAJIT_USE_SYSMALLOC
58  @set LJLIB="%SCE_PSP2_SDK_DIR%\host_tools\build\bin\psp2ld32" -r --output=
59  @set INCLUDE=""
60
61  "%SCE_PSP2_SDK_DIR%\host_tools\build\bin\psp2as" -o lj_vm.o lj_vm.s
62
63  @if "%1" neq "debug" goto :NODEBUG
64  @shift
65  @set LJCOMPILE=%LJCOMPILE% -g -O0
66  @set TARGETLIB=libluajitD.a
67  goto :BUILD
68  :NODEBUG
69  @set LJCOMPILE=%LJCOMPILE% -O2
70  @set TARGETLIB=libluajit.a
71  :BUILD
72  del %TARGETLIB%
73
74  %LJCOMPILE% ljamalg.c
75  @if errorlevel 1 goto :BAD
76  %LJLIB%%TARGETLIB% ljamalg.o lj_vm.o
77  @if errorlevel 1 goto :BAD
78
79  @del *.o *.obj *.manifest minilua.exe buildvm.exe
80  @echo.
81  @echo === Successfully built LuaJIT for PS Vita ===
82
83  @goto :END
84  :BAD
85  @echo.
86  @echo *********************************************************
87  @echo *** Build FAILED -- Please check the error messages ***
88  @echo *********************************************************
89  @goto :END
90  :FAIL
91  @echo To run this script you must open a "Visual Studio .NET Command Prompt"
92  @echo (32 bit host compiler). The PS Vita SDK must be installed, too.
93  :END
```

One Level Up Top Level

src/xedkbuild.bat - luajit-2.0-src

Global variables defined

- ALL_LIB
- DASM
- DASMDIR
- DASMFLAGS
- LJCOMPILE
- LJCOMPILE
- LJLIB
- LJLINK
- LJMT

Source code

```
1  @rem Script to build LuaJIT with the Xbox 360 SDK.
2  @rem Donated to the public domain.
3  @rem
4  @rem Open a "Visual Studio .NET Command Prompt" (32 bit host compiler)
5  @rem Then cd to this directory and run this script.
6
7  @if not defined INCLUDE goto :FAIL
8  @if not defined XEDK goto :FAIL
9
10  @setlocal
11  @rem ---- Host compiler ----
12  @set LJCOMPILE=cl /nologo /c /MD /O2 /W3 /D_CRT_SECURE_NO_DEPRECATE
13  @set LJLINK=link /nologo
14  @set LJMT=mt /nologo
15  @set DASMDIR=..\dynasm
16  @set DASM=%DASMDIR%\dynasm.lua
17  @set ALL_LIB=lib_base.c lib_math.c lib_bit.c lib_string.c lib_table.c lib_io.c lib_os.c lib_package.c
lib_debug.c lib_jit.c lib_ffi.c
18
19  %LJCOMPILE% host\minilua.c
20  @if errorlevel 1 goto :BAD
21  %LJLINK% /out:minilua.exe minilua.obj
22  @if errorlevel 1 goto :BAD
23  if exist minilua.exe.manifest^
24    %LJMT% -manifest minilua.exe.manifest -outputresource:minilua.exe
25
26  @rem Error out for 64 bit host compiler
27  @minilua
28  @if errorlevel 8 goto :FAIL
29
30  @set DASMFLAGS=-D GPR64 -D FRAME32 -D PPE -D SQRT -D DUALNUM
31  minilua %DASM% -LN %DASMFLAGS% -o host\buildvm_arch.h vm_ppc.dasc
32  @if errorlevel 1 goto :BAD
33
34  %LJCOMPILE% /I "." /I %DASMDIR% /D_XBOX_VER=200 /DLUAJIT_TARGET=LUAJIT_ARCH_PPC  host\buildvm*.c
35  @if errorlevel 1 goto :BAD
36  %LJLINK% /out:buildvm.exe buildvm*.obj
37  @if errorlevel 1 goto :BAD
38  if exist buildvm.exe.manifest^
39    %LJMT% -manifest buildvm.exe.manifest -outputresource:buildvm.exe
40
41  buildvm -m peobj -o lj_vm.obj
42  @if errorlevel 1 goto :BAD
43  buildvm -m bcdef -o lj_bcdef.h %ALL_LIB%
```

```
@if errorlevel 1 goto :BAD
buildvm -m ffdef -o lj_ffdef.h %ALL_LIB%
@if errorlevel 1 goto :BAD
buildvm -m libdef -o lj_libdef.h %ALL_LIB%
@if errorlevel 1 goto :BAD
buildvm -m recdef -o lj_recdef.h %ALL_LIB%
@if errorlevel 1 goto :BAD
buildvm -m vmdef -o jit\vmdef.lua %ALL_LIB%
@if errorlevel 1 goto :BAD
buildvm -m folddef -o lj_folddef.h lj_opt_fold.c
@if errorlevel 1 goto :BAD

@rem ---- Cross compiler ----
@set LJCOMPILE="%XEDK%\bin\win32\cl" /nologo /c /MT /O2 /W3 /GF /Gm- /GR- /GS- /Gy /openmp-
/D_CRT_SECURE_NO_DEPRECATE /DNDEBUG /D_XBOX /D_LIB /DLUAJIT_USE_SYSMALLOC
@set LJLIB="%XEDK%\bin\win32\lib" /nologo
@set "INCLUDE=%XEDK%\include\xbox"

@if "%1" neq "debug" goto :NODEBUG
@shift
@set "LJCOMPILE=%LJCOMPILE% /Zi"
:NODEBUG
@if "%1"=="amalg" goto :AMALG
%LJCOMPILE% /DLUA_BUILD_AS_DLL lj_*.c lib_*.c
@if errorlevel 1 goto :BAD
%LJLIB% /OUT:luajit20.lib lj_*.obj lib_*.obj
@if errorlevel 1 goto :BAD
@goto :NOAMALG
:AMALG
%LJCOMPILE% /DLUA_BUILD_AS_DLL ljamalg.c
@if errorlevel 1 goto :BAD
%LJLIB% /OUT:luajit20.lib ljamalg.obj lj_vm.obj
@if errorlevel 1 goto :BAD
:NOAMALG

@del *.obj *.manifest minilua.exe buildvm.exe
@echo.
@echo === Successfully built LuaJIT for Xbox 360 ===

@goto :END
:BAD
@echo.
@echo ******************************************************
@echo *** Build FAILED -- Please check the error messages ***
@echo ******************************************************
@goto :END
:FAIL
@echo To run this script you must open a "Visual Studio .NET Command Prompt"
@echo (32 bit host compiler). The Xbox 360 SDK must be installed, too.
:END
```

One Level Up Top Level