QSim Version 2.6 User Guide

Pranith Kumar
Georgia Institute of Technology

September 30, 2017

Contents

Introduction

QSim is a thread safe multicore emulation library based on the QEMU emulator (version 2.6.0). QSim provides instruction-level control of the emulated environment and detailed information about the executing instruction stream. X86 and ARM64 Linux guests are supported with multiple emulated hardware contexts. Applications of QSim include trace-gathering, multi-threaded software development, and its primary target: micro-architecture simulation.

In the domain of micro-archtiecture simulators, the advantage of using a library like QSim is that the instruction set is emulated with high fidelity by QEMU, so the simulator only needs to implement those aspects of the instruction set relevant to its results. This has both advantages and disadvantages that have been discussed at length elsewhere.

0.1 Requirements

The following are needed to run QSim on your machine. Before you attempt to build and run QSim, make sure these are available:

- x86 CPU and a Linux OS (tested on Ubuntu 14.04)
- 2GB of RAM, 2GB of swap and 10GB of disk space
- Optional: Cross compiler tool chain to build ARM64 binaries
- The `git` version control tool (not installed by default on recent Ubuntu)

0.2 Installation and Setup

To ease the setup of the QSim project we have automated it in a single setup file hosted in the repository. The setup file in this repository installs all the required dependencies, downloads qsim, qemu and optionally the userland images. It also prompts to set up the required environment variables and builds qsim and qemu as libraries.

This repository is hosted on github at `https://github.com/gtcasl/qsim`. Clone this repository using git as follows:

```
git clone https://github.com/gtcasl/qsim
```

Ensure that you have atleast 12 GB free disk space before you download the userland imagegs. Change directory to the cloned qsim directory and run *setup.sh*. This will setup the qsim project. Once you are done setting up, explore the *examples* directory for sample applications.

Chapter 1

QSim Software Architecture

QSim is an effort in combining several large and complex open source software packages into a single entity. The QSim code is very small compared to the QEMU and Linux code bases it leverages. These other projects are not included with QSim, but are downloaded and patched by the provided script `setup.sh` in the qsim-setup repository.

1.1 QSim Design

1.1.1 The QEMU Emulator

A block diagram of QEMU is shown in Figure 1.1. QEMU is an open source emulator that uses dynamic binary translation to translate guest instructions to a host instructions in the translation cache. Multiple guest architectures are supported, both as guests and hosts, including x86, ARM, and PowerPC. The environment provided by QSim allows an operating system and applications to be run in an emulated environment, instruction by instruction, on multiple emulated cores. Currently only 32-bit ARM guests are supported, but other architectures are expected to become available.

Figure 1.2 illustrates the QEMU translation process. This is a two-step process in which guest instructions are translated into Tiny Code Generator (TCG) operations and then host instructions. QSim allows the creation of *helper functions*, which are functions that can be called from the translation cache. QEMU uses helper functions to implement uncommon but complex guest instructions, so that they do not have to be implemented entirely as large and complex blocks of code that are compiled at run-time.

QSim uses the helper function mechanism to instrument guest instructions. Calls to helper functions are added to the translations of instrumented instructions. From these helper functions, callbacks that have been registered by the user's program are called.

1.1.2 The QSim Library

Figure 1.3 is an illustration of the fully-integrated QSim software architecture. The QSim library makes calls into the QEMU CPUs on behalf of the client program, and the QEMU CPUs make the callbacks into the client programs based on events in the instruction stream. A variety of different callbacks are provided, including:

- Every instruction
- Atomic memory operations
- Memory reads/writes
- Register reads/writes
- Interrupts

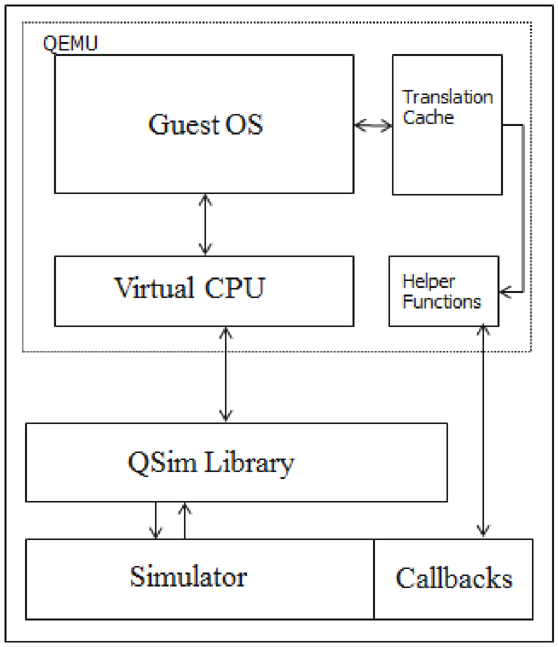
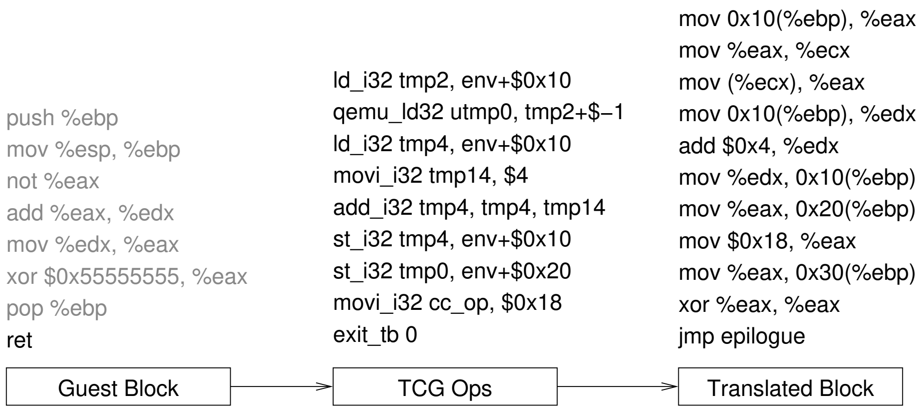


Figure 1.1: Block diagram of QEMU.



```
push %ebp
mov %esp, %ebp
not %eax
add %eax, %edx
mov %edx, %eax
xor $0x55555555, %eax
pop %ebp
ret
```

```
ld_i32 tmp2, env+$0x10
qemu_ld32 utmp0, tmp2+$−1
ld_i32 tmp4, env+$0x10
movi_i32 tmp14, $4
add_i32 tmp4, tmp4, tmp14
st_i32 tmp4, env+$0x10
st_i32 tmp0, env+$0x20
movi_i32 cc_op, $0x18
exit_tb 0
```

```
mov 0x10(%ebp), %eax
mov %eax, %ecx
mov (%ecx), %eax
mov 0x10(%ebp), %edx
add $0x4, %edx
mov %edx, 0x10(%ebp)
mov %eax, 0x20(%ebp)
mov $0x18, %eax
mov %eax, 0x30(%ebp)
xor %eax, %eax
jmp epilogue
```

Figure 1.2: QEMU translation process with expansion of return instruction from a guest basic block.

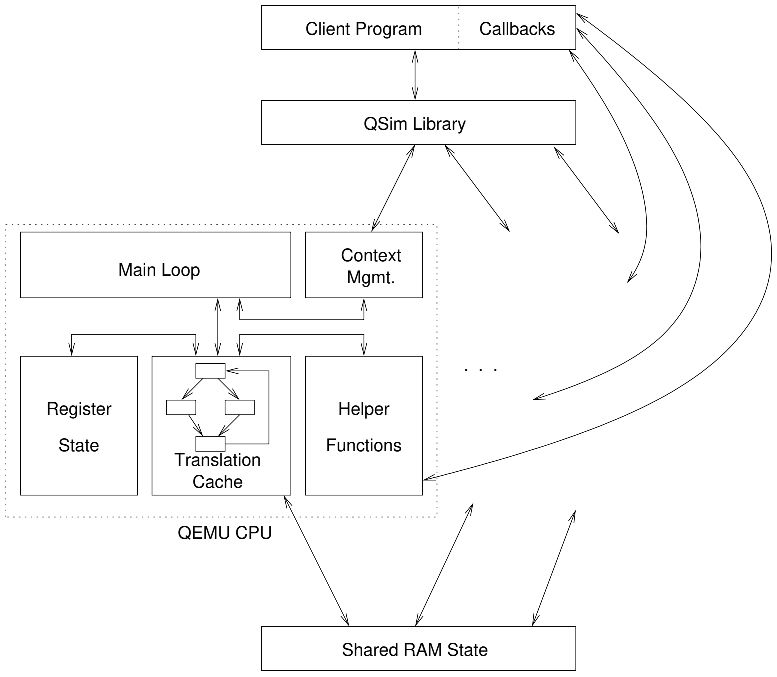


Figure 1.3: Block diagram of QSim library design with arrows representing data flow between components.

- “Magic” instructions
- Instruction read for translation

The atomic memory operation and read/write callbacks are given in addition to the instruction callbacks for their corresponding instructions. Magic instructions are used to provide out-of-band communication between the guest and QSim, and are explained in detail in Section 1.3.

1.1.3 Guest Environment

The QSim guest environment as is the norm includes a kernel component and a userland component. We use Linux for our kernel, and either a busybox based userspace or a Debian based userspace. When using the busybox userland, an init script and an application binary need to be provided. QSim utilizes the emulation of disk and network devices in qemu. So you can read from mounted disks like sd cards and access the web from the guest system.

The debian based userland is provided in the form of disk images. The login credentials for the provided disk image are as follows:

```
Username: qsim or root
Password: qsim12
```

1.1.4 QDB, The QSim Debugger

QDB provides a commandline interface to QSim that can be used to debug guest applications, guest operating systems, and the QSim library itself. Compared to the rest of QSim, QDB is in an earlier state of development. It is for this reason that QDB maintains its own list of bugs and unimplemented features.

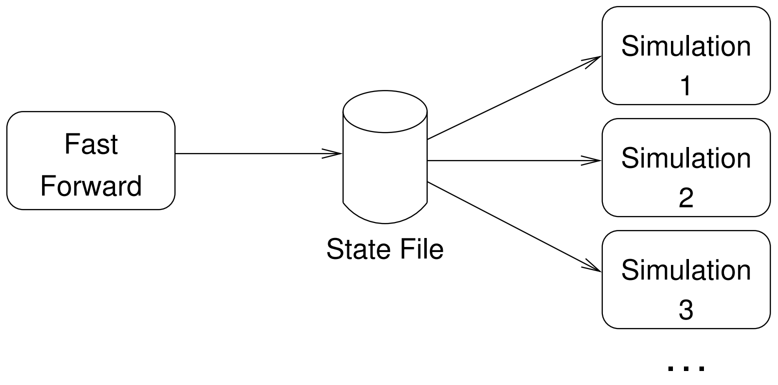


Figure 1.4: Operation of `qsim-fastforward` generates a reuseable state file.

1.1.5 The Fastforwarder

The QSim fastforwarder, `qsim-fastforward` is used to generate state files that represent the state of RAM after the machine has booted, but before the application has began running. This is so that the potentially very long time spent waiting for Linux to boot can be amortized over all simulations using a given `bzImage` and number of QEMU CPUs.

1.2 QSim Directory Hierarchy

What follows is a tour of the directory structure of the QSim codebase. This is provided as both a reference and an introduction.

```
LICENSE
README
setup.sh
```

1.2.1 QSim Core - qsim

```
qsim.h
qsim-vm.h
qsim-context.h
mgzd.h
vm-func.h
qsim.cpp
Makefile
```

These files form the core of QSim. `qsim-regs.h` defines `enum regs`, an important enumeration of guest register names shared between the modifications to QEMU and QSim. `qsim-vm.h` defines the types of the basic callback functions that are called from within QEMU. `mgzd.h` and `qsim-context.h` provide the services necessary to maintain multiple separate instances of QEMU as though they were components of the same program. `qsim.h` and `qsim.cpp` are the actual nuts and bolts of QSim. `qsim.h` must be included by any client program of QSim and `qsim.cpp` is the only source in what becomes `qsim.so`.

`Makefile` leaves a lot to be desired, but it is responsible for building QSim. For more information on building QSim, see `INSTALL` in the QSim root directory.

1.2.2 Fast Forwarder

```
fastforwarder.cpp
```

```
statesaver.cpp
statesaver.h
```

These files compile to a program called `qsim-fastforwarder`, which is used as described in Section 1.1.5 to eliminate the start-up portion of long simulations, especially ones with high core counts.

1.2.3 QEMU

```
qemu/
```

This directory is the clone of the qemu repository which contains the callbacks branch containing the modified QEMU. The result of a successful compile of QEMU for QSim is a shared object file which will be used by the QSim library.

1.2.4 Initial Ram Filesystem

```
initrd/
initrd/init
initrd/mark_app.c
initrd/qsim_io.c
initrd/test_threads.c
initrd/busybox-1.24.1
initrd/etc
```

The `initrd` directory contains the userspace portion of the guest environment, which is packaged into a Linux initial RAM filesystem and compiled into the Linux kernel. Information on how to configure the guest system is in Chapter 3.

1.2.5 Linux Kernel Sources

```
linux/
linux/getkernel.sh
linux/linux-4.1.15.qsim.patch
linux/bzImage
linux/linux/
```

A modified Linux kernel is used by QSim. The `getkernel.sh` script downloads the unomdified kernel from upstream and patches it with modifications that allow it to boot in QSim guest environment, without using BIOS or usual PC hardware services that QSim does not provide.

1.2.6 Disassembler

```
distorm/
capstone/
```

Several of the examples and the QSim Debugger require a disassembler. Distorm was chosen as a simple, portable, open source disassembly library for x86 and capstone was chosen for ARM64 programs. Note that these libraries are not part of QSim proper; they are just utility libraries that the bundled applications happen to require that was considered uncommon enough to be included with the source distribution.

1.2.7 Simplesim

```
simplesim/
```

SimpleSim is an example multicore out-of-order CPU simulator.

Value in `%rax` or `pmcr_el0`	**Function**
`0xc501e0xx`	Console output, character `x`.
`0x1d1e1d1e`	Calling CPU now executing idle loop.
`0xc75cxxxx`	Context switch to TID `x`.
`0xb007xxxx`	Bootstrap CPU `x`, `%ip` provided by subsequent magic instruction.
`0x1dxxxxyy`	Inter-processor interrupt. Interrupt CPU `x` with vector `y`.
`0xc7c7c7c7`	Request CPU count, returned in `%rax`.
`0x512e512e`	Request RAM size in MB, returned in `%rax`.
`0xaaaaaaaa`	Application start signal. Causes application start callback to be called.
`0xfa11dead`	Application end signal. Causes application end callback to be called.

Table 1.1: Magic Instructions provided by `OSDomain`.

1.2.8 Examples

```
examples/
examples/x86/simple.cpp
examples/x86/cachesim.cpp
```

The programs in "examples/x86" are example client program using the QSim library. One of these, `simple.cpp`, is examined in detail in Chapter 2 as an example client program using QSim.

1.2.9 QDB: The QSim Debugger

```
qdb/
qdb/BUGS
```

QDB provides a commandline interface to a QSim. QDB allows users to load in symbol tables for both the kernel and individual user address spaces. It also provides dynamic instruction count based flat profiling with uniform statistical sampling. The best documentation for QDB is its own built-in help system, which can be accessed at any time by typing `help`.

1.3 Magic Instructions

QSim provides an interface for out-of-band communication between the guest environment, QSim library, and its client programs, through use of magic instructions. Magic instructions are, on x86, QSim-defined regions of the CPUID address space that perform special functions. On ARM64, we use the performance monitor counter register, **pmcr_el0** accessed using the **MSR** instruction. CPUID is ordinarily used by software to get information about the CPU on which it is running. A value is loaded into **RAX** or the **pmcr_el0** register and once the CPUID instruction has run, the registers are set according to the results. This is used ordinarily to query CPUs for features.

In QSim, the magic instructions are used to provide simplified, paravirtualized, emulated hardware. Magic instruction callbacks can also be set by client programs to extend this interface. Table 1.1 lists the magic instructions defined by QSim itself and used by the Linux kernel and some of the guest software.

Chapter 2

A Sample Client Program

The use of QSim fundamentally boils down to a few simple steps that a client program must take:

- Instantiate one (and only one) `OSDomain`.
- Attach instrumentation to that `OSDomain`.
- Call `run()` on that `OSDomain`, ocasionally calling `timer_interrupt()`.
- Do something with the information collected by the instrumentations.

The last step is purely optional, but there would not be much reason to use QSim without the instrumentation.

For the purposes of this Chapter, we will examine `examples/x86/simple.cpp`, the simple example; a multicore trace gathering utility.

2.1 Operation

As illustrated in Figure 2.1, `simple.cpp` is basically a trace gatherer. It installs a set of callbacks that print messages to a textual trace file whenever they are called.

2.2 Listing

```
/*****************************************************************************\
* Qemu Simulation Framework (qsim)                                            *
* Qsim is a modified version of the Qemu emulator (www.qemu.org), couled     *
* a C++ API, for the use of computer architecture researchers.                *
*                                                                             *
* This work is licensed under the terms of the GNU GPL, version 2. See the    *
* COPYING file in the top-level directory.                                    *
\*****************************************************************************/
#include <iostream>
#include <fstream>
#include <iomanip>

#include "distorm.h"

#include <qsim.h>
#include <qsim-load.h>

using Qsim::OSDomain;
```

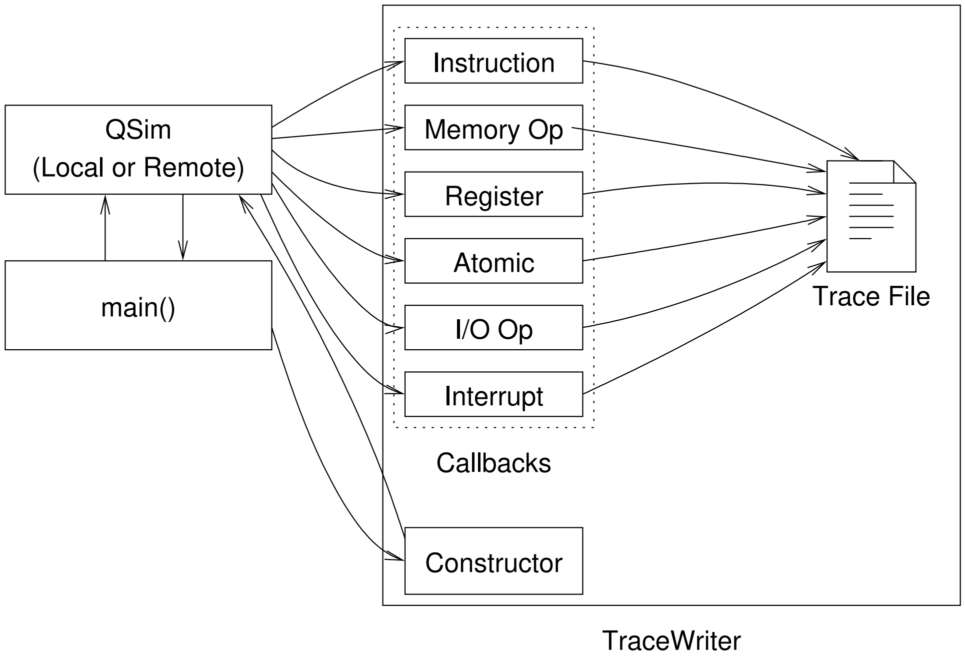


Figure 2.1: Design of the `simple.cpp` example program.

```
using std::ostream;

class TraceWriter {
public:
  TraceWriter(OSDomain &osd, ostream &tracefile) :
    osd(osd), tracefile(tracefile), finished(false)
  {
    osd.set_app_start_cb(this, &TraceWriter::app_start_cb);
  }

  bool hasFinished() { return finished; }

  int app_start_cb(int c) {
    static bool ran = false;
    if (!ran) {
      ran = true;
      osd.set_inst_cb(this, &TraceWriter::inst_cb);
      osd.set_atomic_cb(this, &TraceWriter::atomic_cb);
      osd.set_mem_cb(this, &TraceWriter::mem_cb);
      osd.set_int_cb(this, &TraceWriter::int_cb);
      osd.set_io_cb(this, &TraceWriter::io_cb);
      osd.set_reg_cb(this, &TraceWriter::reg_cb);
      osd.set_app_end_cb(this, &TraceWriter::app_end_cb);

      return 0;
    }

    return 0;
```

```
}

int app_end_cb(int c)   { finished = true; return 0; }

int atomic_cb(int c) {
  tracefile << std::dec << c << ": Atomic\n";
  return 0;
}

void inst_cb(int c, uint64_t v, uint64_t p, uint8_t l, const uint8_t *b,
             enum inst_type t)
{
  _DecodedInst inst[15];
  unsigned int shouldBeOne;
  distorm_decode(0, b, l, Decode64Bits, inst, 15, &shouldBeOne);

  tracefile << std::dec << c << ": Inst@(0x" << std::hex << v << "/0x" << p
            << ", tid=" << std::dec << osd.get_tid(c) << ", "
            << ((osd.get_prot(c) == Qsim::OSDomain::PROT_USER)?"USR":"KRN")
            << (osd.idle(c)?"[IDLE]":"[ACTIVE]")
            << "): " << std::hex;

  //while (l--) tracefile << ' ' << std::setw(2) << std::setfill('0')
  //                      << (unsigned)*(b++);

  if (shouldBeOne != 1) tracefile << "[Decoding Error]";
  else tracefile << inst[0].mnemonic.p << ' ' << inst[0].operands.p;

  tracefile << " (" << itype_str[t] << ")\n";
}

void mem_cb(int c, uint64_t v, uint64_t p, uint8_t s, int w) {
  tracefile << std::dec << c << ":  " << (w?"WR":"RD") << "(0x" << std::hex
            << v << "/0x" << p << "): " << std::dec << (unsigned)(s*8)
            << " bits.\n";
  return;
}

int int_cb(int c, uint8_t v) {
  tracefile << std::dec << c << ": Interrupt 0x" << std::hex << std::setw(2)
            << std::setfill('0') << (unsigned)v << '\n';
  return 0;
}

uint32_t *io_cb(int c, uint64_t p, uint8_t s, int w, uint32_t v) {
  tracefile << std::dec << c << ": I/O " << (w?"WR":"RD") << ": (0x"
            << std::hex << p << "): " << std::dec << (unsigned)(s*8)
            << " bits.\n";
  return NULL;
}

void reg_cb(int c, int r, uint8_t s, int type) {
  tracefile << std::dec << c << (s == 0?": Flag ":": Reg ")
            << (type?"WR":"RD") << std::dec;
```

```
    if (s != 0) tracefile << ' ' << r << ": " << (unsigned)(s*8) << " bits.\n";
    else tracefile << ": mask=0x" << std::hex << r << '\n';
  }

private:
  OSDomain &osd;
  ostream &tracefile;
  bool finished;

  static const char * itype_str[];
};

const char *TraceWriter::itype_str[] = {
  "QSIM_INST_NULL",
  "QSIM_INST_INTBASIC",
  "QSIM_INST_INTMUL",
  "QSIM_INST_INTDIV",
  "QSIM_INST_STACK",
  "QSIM_INST_BR",
  "QSIM_INST_CALL",
  "QSIM_INST_RET",
  "QSIM_INST_TRAP",
  "QSIM_INST_FPBASIC",
  "QSIM_INST_FPMUL",
  "QSIM_INST_FPDIV"
};

int main(int argc, char** argv) {
  using std::istringstream;
  using std::ofstream;

  ofstream *outfile(NULL);

  unsigned n_cpus = 1;

  std::string qsim_prefix(getenv("QSIM_PREFIX"));

  // Read number of CPUs as a parameter.
  if (argc >= 2) {
    istringstream s(argv[1]);
    s >> n_cpus;
  } else {
    fprintf(stderr, "Usage:\n INTERACTIVE: %s <num_cpus>\n"
            " HEADLESS: %s <num_cpus> -state <state_file> -bench <benchmark.tar> -out <tra
            argv[0], argv[0]);
    exit(0);
  }

  // Read trace file as a parameter.
  if (argc >= 7) {
    outfile = new ofstream(argv[7]);
  } else {
    outfile = new ofstream("trace.log");
```

```
126   }

127   OSDomain *osd_p(NULL);

128   if (argc >= 4) {
129     // Create new OSDomain from saved state.
130     osd_p = new OSDomain(n_cpus, argv[3]);
131   } else {
132     osd_p = new OSDomain(n_cpus, qsim_prefix + "/images/x86_64_images/vmlinuz", "x86", QSIM_INTE
133   }
134   OSDomain &osd(*osd_p);

135   // Attach a TraceWriter if a trace file is given.
136   TraceWriter tw(osd, outfile?*outfile:std::cout);

137   // If this OSDomain was created from a saved state, the app start callback was
138   // received prior to the state being saved.
139   if (argc >= 6) {
140     Qsim::load_file(osd, argv[5]);
141     tw.app_start_cb(0);
142   }

143   osd.connect_console(std::cout);

144   // The main loop: run until 'finished' is true.
145   unsigned long inst_per_iter = 1000, inst_run;
146   inst_run = inst_per_iter;
147   while (!(inst_per_iter - inst_run)) {
148     inst_run = 0;
149     inst_run = osd.run(inst_per_iter);
150     osd.timer_interrupt();
151   }
152
153   if (outfile) { outfile->close(); }
154   delete outfile;

155   delete osd_p;

156   return 0;
157 }
```

2.3 Instantiating OSDomain

As is often found in simple QSim client programs, the single `OSDomain` (there can only be one) is allocated on the heap in the `main()` function at lines 123 through 132. This allows the same `osd` reference to be used for either an `OSDomain` constructed with a number of CPUs and a kernel image or an `OSDomain` constructed from a state file.

2.4 The Callback System

`simple.cpp` uses instruction, atomic, memory, interrupt, and I/O callbacks and logs the occurrence of each to a trace file. Application start and end callbacks are used. `TraceWriter::app_start_cb()`

sets the callbacks that perform tracing, and `TraceWriter::app_end_cb()` sets a flag that causes the main loop to exit.

Chapter 3

Porting Guest Software

You can analyze or collect characteristics of applications from within the qemu environment. If source for the application is available, all you need to do is annotate it and recompile. If no source is available, you will need to analyze the entire application by using the app start and end markers.

Note: before porting an application or benchmark suite to QSim, make sure it is not already present in the QSim benchmarks collection. This collection, available through the QSim website, contains a growing set of common benchmarks, including all of Splash 2 and Parsec.

You can login to the qemu VM using the following ssh command:

```
ssh -p 2222 qsim@localhost

username: qsim
password: qsim12
```

Porting an application to QSim consists of the following steps:

- Add an application start callback to the application where the "region of interest" begins.
- Add an application end callback to the application where the "region of interest" ends.
- Compile the application for ARM 64-bit Linux, preferably statically linked in the qemu VM.
- Run the application.

3.1 Application Start and End Markers

QSim provides callbacks for the start and end of applications. These are used by QSim client programs to fast-forward through irrelevant code to the simulation *region of interest* (ROI). These can be inserted before and after a program by adding,

```
~/qsim_tests/qsim_enable

~/qsim_tests/qsim_disable
```

If more fine-grained demarcation of the ROI is needed, the following lines of C code can be used, at the start and end of the ROI, respectively:

```
#include <qsim_magic.h>

qsim_magic_enable();
<ROI>
qsim_magic_disable();
```

3.2 Compiling for QSim

On a 64-bit host, creating statically-linked binaries is a matter of the `-static` option to the linker. In a typical GNU Autotools based build environment, this can be accomplished with:

```
$ LDFLAGS=-static ./configure
$ make
```

The `file` command can be used to ensure that the resulting binary is actually a statically-linked ELF.

3.3 Setting Up the Linux Initial Ramdisk

The `initrd` makefile is set up so that any files placed in `initrd/sbin` will be added to the generated `initrd.cpio`. User applications are not, however, included as dependencies, so when modifying them it is prudent to perform a `make clean` before running `make` to produce the ramdisk file. Typically the only actions required to set up the linux filesystem is to copy the application binary into `initrd/sbin`, modify the `initrd/init` script to launch the application, and run `make clean` followed by `make`.

The `init` script contains, out of the box, a set of sample application invocations commented out with octothorpes. If using one of the prepackaged benchmarks, it is only necessary to uncomment the corresponding line. Otherwise, the appropriate command can be added anywhere between the invocation of `mark_app` and `mark_app END`.

3.3.1 Busybox Utility Programs

The command interpreter used to execute the `init` script, as well as most of the utility programs available within it (`cat`, `grep`, `ls`) are provided by the Busybox all-in-one userland. It provides smaller-footprint versions of common Unix utilities, accessed by a set of symbolic links to the same file.

3.3.2 QSim Guest Utilities

The initial ramdisk also contains a set of QSim-specific utilities and test programs:

```
qsim-out
```

Uses the magic instruction mechanism to copy program input to the QSim console.

```
mark_app [END]
```

Uses the magic instruction mechanism to mark the start and end of the application. This causes the application start and end callbacks described on pages 29-31 to be called.

```
test-threads
```

Simply spawns 100 threads and waits for them to exit. Each thread grabs a mutex, prints a message, and exits. Used to check that atomic memory operations are being treated as such by the client program; not a simple task if this application is multithreaded.

3.4 Compiling the Kernel Image

3.4.1 ARM64 Kernel

Initially, the kernel image must be configured with the absolute path to the `initrd.cpio` file. The configuration system can be launched by changing to the kernel root and running:

```
$ make menuconfig
```

The option to configure the path to the RAM disk CPIO file is under "General Options". After this initial configuration has been performed, subsuquent builds only require the second step of the process, running `make`.

```
$ make -j4
```

3.5 Using the Kernel with QSim

A symlink to the kernel this produces is conveniently placed in the `linux/` directory. This is the default kernel image as used by all of the programs in `examples/`. QDB takes the path of the kernel image as a commandline option. Within client programs, the path to the kernel is an argument to the `OSDomain` constructor (see `OSDomain`, page 26 for details).

Glossary

atomic memory operation A memory operation that is required by the architecture to complete as an indivisible operation. Atomic read-modify-write operations like `XCHG` are used in the implementation of synchronization primitives.

callback A function passed as an argument to another layer of the software stack, so that it can be called by software in that layer as necessary. In QSim, callbacks are code within the *client program* that is called from *helper functions*. Callbacks are the primary way QSim allows its users to instrument the code executing in the *guest*. The word "callback" can also be used to refer to the instance of a specific call to a callback function. For example, "A memory operation callback will be received after any exception conditions on the address being accessed are cleared up." In this sentence, the "memory operation callback" is not the function itself, but a specific instance of a call to the function.

client program An application that makes use of a library; for the purposes of this document the "client program" is always an application that uses QSim.

guest The simulated system, or virtual machine, within QSim. Often used as an adjective to describe a data structure: "guest CPU state;" the registers of the virtual CPU stored in a data structure within QEMU, "guest RAM;" the memory image of the guest machine. Used as an antonym of *host*, the machine on which QSim and the client program runs.

helper function Part of QEMU, a C function intended to be called from the *translation cache*. Helper functions are used to implement *guest* functionality that is complex and not time critical. Their use results in more compact translated code and a simpler binary translator.

host The machine on which QSim and its client program runs, antonym of *guest*. Used to describe abstractions that may otherwise be confused for aspects of the guest system: "host virtual address", "host instruction set."

magic instruction An instruction that provides a service not found in current hardware, decoded using an unorthodox mechanism. In QSim, magic instructions are `CPUID`(x86) or `msr pmcr_el0`(ARM64) instructions initiated with invalid values in `%rax`(x86) or `%pmcr_el0`(ARM64) registers. These can be tied to callbacks and used to implement new features or create debugging output.

OS domain A collection of CPUs among which memory is sequentially consistent and cache coherent, usually associated with a single operating system image. The `OSDomain` class gets its moniker from this concept.

QEMU CPU A hardware context within QEMU. Not to be confused with a CPU in the sense of an entire chip, or in the sense of a core within that chip that may support multiple contexts through and SMT arrangement. A four-core chip with 2-way SMT, for example, would be emulated with 8 QEMU CPUs.

translation cache The data structure into which a dynamic binary translator like QEMU places the result of translation and from which the translated code is run. The only code emulating *guest* instructions is executed by the *host* through the translation cache, either directly or by calling *helper functions*.

Appendix A

API Documentation

What follow are descriptions of the user-facing classes of which QSim is comprised and their interfaces, member types, and member variables. By far the most important class of these is `OSDomain`, as described in Chapter 2 and documented on page 26. Entire simulators can be built interacting only with `OSDomain`. The other available classes provide various utility functions and the ability to interact directly with the QEMU CPU objects that normally comprise an `OSDomain`.

A.1 `Qsim::OSDomain`

`OSDomain` is a singleton class that creates and manages a multicore processor emulator. The simplest way to use QSim is to instantiate a QSim `OSDomain`, provide it with a set of callbacks, and start running instructions on it, periodically calling `OSDomain::timer_interrupt()`. This is exactly what the sample program in Chapter 2 does. `OSDomain` means "Operating System domain." It is so named because all of the CPUs in it share one cache coherent and sequentially consistent block of RAM that contains a single OS image.

A.1.1 Member Types

```
enum cpu_mode {
  MODE_REAL, MODE_PROT, MODE_LONG
};
```

Used to describe the current mode of the CPU. While 32-bit real mode and 16-bit protected mode do exist on the hardware, for modern operating systems it can be assumed that `MODE_REAL` means 16-bit default operand sizes and is only encountered during boot, and `MODE_PROT` means 32-bit default operand sizes. `MODE_LONG` only works in protected mode, and has 64-bit default operand sizes.

```
enum cpu_prot {
  PROT_KERN, PROT_USER
};
```

Describes the current protection level of the CPU. While x86 actually provides a MULTICS-like ring system complete with hardware-defined task and call gate structures and four levels of protection, these still exist only for backwards compatability. This data type is used to describe whether a given CPU within QSim is running in supervisor mode (ring 0) or user mode.

A.1.2 Member Functions

```
OSDomain(uint16_t    n,
         std::string kernel_path,
         unsigned    ram_mb = 2048);
```

This `OSDomain` constructor takes as parameters `n`, the number of CPUs, `kernel_path`, the path to an appropriate *bzImage* and optionally `ram_mb`, the size of system RAM in megabytes.

```
OSDomain(const char *state_file);
```

The other `OSDomain` constructor loads the state from a file. This allows for fast startup compared to emulating the entire boot process.

```
bool idle(unsigned i);
```

Returns true if CPU `i` is running its idle loop according to the OS, false otherwise.

```
int get_tid(uint16_t i);
```

If Linux has booted, returns the task ID of the thread currently running on CPU `i`. Otherwise returns -1.

```
enum cpu_mode get_mode(uint16_t i);
```

Returns the mode (real, protected, or long) of CPU `i` (see `enum cpu_mode`, page 26). Not capable of distinguishing 16-bit protected mode, but this is not used by any current OS.

```
    enum cpu_prot get_prot(uint16_t i);
```

Returns the current protection ring (kernel or user) of CPU `i` (see `enum cpu_prot`, page 26).

```
    int get_n() const;
```

Returns the number of CPUs being emulated.

```
    qemu_ramdesc_t get_ramdesc() const;
```

Returns the `qemu_ramdesc_t` structure used by the instances of QEMU within this `OSDomain` to share their memory region (see `qemu_ramdesc_t`, page **??**).

```
    unsigned run(uint16_t i, unsigned n);
```

Run the emulator on CPU `i` for `n` instructions or until an exception condition occurs. Returns the number of instructions for which the CPU actually ran. Exception conditions are caused by callbacks that return nonzero and by the CPU not currently being in a runnable state. Since the callbacks are managed by the client program and whether the CPU is running can be determined through other means (see `OSDomain::runnable()`, page 27), a direct method of determining which condition terminated the `run` is not provided.

```
    void connect_console(std::ostream &s);
```

The given stream is added to the list of streams to which character output from the guest software stack is delivered. This text is delivered one character at a time using a magic instruction.

```
    void timer_interrupt();
```

Send a timer interrupt to all of the CPUs in the `OSDomain`. Linux expects to see these at a rate of 100Hz, or once every 10ms. Spacing calls to `timer_interrupt()` appropriately is the fundamental way the progress of simulation time can be communicated to the guest.

```
    void interrupt(unsigned i, uint8_t vec);
```

Interrupt CPU `i`, causing it to run the interrupt service routine pointed to by vector `vec`.

```
    bool booted(unsigned i);
    bool runnable(unsigned i);
```

Returns `true` if CPU `i` has been brought up by the operating system and remains runnable, and `false` otherwise. CPUs are no longer runnable after the application's end has been reached, and the `booted()` function will return false in that case as well. These functions have the exact same function and the `booted()` alias is provided for compatibility reasons only. Its use causes a warning.

```
    void save_state(const char* state_file);
```

Save the state of the emulator to the given file. The OSDomain must be booted and running in protected mode, and must not be using floating point, SSE, or MMX register state when this is called. Also, unless this is called during an instruction callback, the previous instruction will be re-executed when the state is loaded. For an example demonstrating how to overcome this limitation, see `examples/fastforwarder.cpp` and its dependencies.

```
    typedef int (*atomic_cb_t)(int cpu);
    void set_atomic_cb(uint16_t i, atomic_cb_t cb);
    void set_atomic_cb(atomic_cb_t cb);
```

Set a callback to be called from within the translation cache every time an atomic memory operation is encountered. Can be set for a given CPU `i` or, omitting this parameter, for all CPUs. This callback is of the form shown in the definition of `atomic_cb_t` and occurs after the corresponding instruction callback is made but before the instruction is executed. If the callback returns nonzero, the atomic operation will not complete and the call to `run()` (see `OSDomain::run()`, page `func:run`) will return. New callback assignments replace the previous assignment. If a null pointer is provided, the callback will be disabled.

```
typedef void (*inst_cb_t)(int            cpu,
                          uint64_t       vaddr,
                          uint64_t       paddr,
                          uint8_t        len,
                          const uint8_t *bytes);
void set_inst_cb(uint16_t i, inst_cb_t cb);
void set_inst_cb(inst_cb_t cb);
```

Set a callback to be called from within the translation cache every time a guest instruction begins executing. This callback is made prior to the instruction's execution, and upon its return, the instruction will begin. In the event of a page fault while accessing the instruction itself, the appropriate interrupt callback will be generated with no prior instruction callback for the faulting instruction, since the fetch itself and not the execution of the instruction failed, followed by the execution of the interrupt handler and finally the execution of the faulting instruction including the appropriate instruction callback.

If, however, a memory operation that is part of the instruction execution fails, the instruction callback will be seen twice, once for each attempt at executing the instruction.

Instruction callbacks provide an identifier for the CPU on which the instruction will execute, the virtual and physical address of the instruction, its size, and a pointer to the instruction code. They can be set for specific CPUs `i` or all CPUs in the `OSDomain`. New callback assignments replace the previous assignment. If a null pointer is provided, the callback will be disabled.

```
typedef int (*mem_cb_t)(int      cpu,
                        uint64_t vaddr,
                        uint64_t paddr,
                        uint8_t  size,
                        int      write);
void set_mem_cb(uint16_t i, mem_cb_t cb);
void set_mem_cb(mem_cb_t cb);
```

Set up a function to be called from within the translation cache every time a memory operation is encountered. Read operation callbacks occur immediately before the read occurs, to allow the value being read to be modified by the callback, and write operation callbacks occur immediately after the write, to allow the value just written to be read. Reads are, however, tried before the callback is called. This means that any page fault that would occur always happens before the callback is called, so that once the callback is reached the addresses are guaranteed to be reachable.

Memory operation callbacks take as arguments the CPU identifier for the CPU on which they execute, virtual and physical addresses, their size, and a flag that is 1 if the operation is a write and 0 if the operation is a read. These callbacks can be set for specific CPUs `i` or for all of the CPUs in the `OSDomain`. New callback assignments replace the previous assignment. If a null pointer is provided, the callback will be disabled. Some memory operations occur at the physical level with no corresponding virtual address. For these, the virtual address is set to zero.

If the return value of the memory operation callback is nonzero, `run()` will return from that point. If the operation is a read, this will allow other routines in the main loop to provide data.

```
typedef int (*int_cb_t)(int cpu, uint8_t vec);
void set_int_cb(uint16_t i, int_cb_t cb);
void set_int_cb(int_cb_t cb);
```

Set up an interrupt callback, either for a specific CPU `i` or all CPUs in the `OSDomain`. These callbacks are called at the moment any interrupt, hardware or software, occurs. Hardware interrupts always happen after an instruction has finished executing. Software exceptions such as page faults occur at the point in the execution of their instruction that the exception condition arises. In the case of page faults, this is prior to the receipt of a memory callback for the faulting address but after the receipt of an instruction callback for the instruction performing the access.

Interrupt callbacks take the CPU identifier and the 8-bit interrupt vector as parameters. Other related information, such as the fault address in the case of a page fault, can be retreived with `OSDomain::get_reg()` (see page 31). Like the atomic memory operation callback (see page 27), the interrupt callback can return nonzero to halt instruction execution at a given point and force `run()` to return prematurely. This can be used to provide resolution to an exception condition from within the main loop instead of a callback. New callbacks assigned with this function replace the old ones. A null assignment will disable the callback.

```
typedef void (*io_cb_t)(int      cpu,
                        uint64_t port,
                        uint8_t  size,
                        int      type,
                        uint32_t val);
void set_io_cb(uint16_t i, io_cb_t cb);
void set_io_cb(io_cb_t cb);
```

Set an I/O callback. Similar to the memory callback, but much simpler due to the simple flat I/O address space provided by the x86 architecture. Does not yet allow for modifying the value returned by an `IN` instruction, but does allow reading the value (`val`) provided by the `OUT` instruction. New callbacks assigned with this function replace the old ones. A null assignment will disable the callback.

```
typedef void (*trans_cb_t)(int cpu);
void set_trans_cb(uint16_t i, trans_cb_t cb);
void set_trans_cb(trans_cb_t cb);
```

Set a translation callback. When this callback is set, any set memory operation callbacks will be called when instructions are being read for translation. The addition of the translation callback enables memory state to be maintained externally to QSim.

```
typedef void (*reg_cb_t)(int     cpu,
                         int     reg,
                         uint8_t size,
                         int     type);
void set_reg_cb(uint16_t i, reg_cb_t cb);
void set_reg_cb(reg_cb_t cb);
```

Set a register callback. Same semantics as memory callback. The `reg` parameter can be interpreted as a member of `enum regs` (see page ??). In the special case that `size` is zero, the register callback can be interpreted as a flag bit read or write. In this case, `type` is the logical or of one or more elements of `enum flags` (see page ??).

```
void set_app_start_cb(int (*f)(int));
```

A simple callback that uses a magic instruction delivered by the guest application or a three-instruction utility program to provide an indication of when this application begins running. Includes a CPU identifier, but this cannot be taken to mean the CPU on which the benchmark begins execution, since the magic instruction may have been deliviered by a utility program and not the benchmark application itself. Common uses include fast-forwarding simulators past the boot sequence in a purely emulated mode and resetting performance counters within simulators so that at application end they reflect only the operations created by the application being evaluated. New callbacks assigned with this function replace the old ones. A null assignment will disable the callback.

This callback will return 0 to continue execution, or nonzero to cause `OSDomain::run()` to exit prematurely, just like the interrupt and atomic callbacks.

```
void set_app_end_cb(void (*f)(int));
```

A simple callback that uses a magic instruction delivered by a three-instruction utility program to indicate when the guest application has terminated. Typically used to end the simulation. New callbacks assigned with this function replace the old ones. A null assignment will disable the callback.

```
template <typename T>
  void set_atomic_cb(T* o,
                     int (T::*f)(int cpu));
```

Same as `set_atomic_cb()` on page 27, but instead of operating on a static or global function is capable of making calls to arbitrary member functions of any instance `o` of any class `T`, that fit the prototype. Also capable of making multiple calls, so that functionality can be combined. New calls to this version of `set_atomic_cb()` do not replace the previous callback but instead add a new one. In this way, functionality can be combined.

It is important to note that this callback system is incompatible with the other one. Calls to this version of `set_atomic_cb()` disable static callbacks, and new static callbacks disable these class member callbacks.

```
template <typename T>
  void set_magic_cb(T* o,
                    int (T::*f)(int      cpu,
                                uint64_t rax));
```

The only way to set magic instruction callbacks in `OSDomain`, since `OSDomain`'s static magic instruction callback is already of central importance and cannot be overridden. Provides an easy way to add new magic instructions by intercepting `CPUID`(x86) or `MSR`(ARM64) instructions.

```
template <typename T>
  void set_io_cb(T* o,
                 void (T::*f)(int      cpu,
                              uint64_t port,
                              uint8_t  size,
                              int      write,
                              uint32_t value));
```

The equivalent of `set_atomic_cb()` (see page 30) for the I/O callback (see `set_io_cb()`, page 29).

```
template <typename T>
  void set_reg_cb(T* o,
                  void (T::*f)(int      cpu,
                               uint64_t port,
                               uint8_t  size,
                               int      type));
```

The equivalent of `set_atomic_cb()` (see page 30) for the register callback (see `set_reg_cb()`, page 29).

```
template <typename T>
  void set_mem_cb(T* o,
                  int (T::*f)(int      cpu,
                                  uint64_t vaddr,
                                  uint64_t paddr,
                                  uint8_t  size,
                                  int      write));
```

The equivalent of set_atomic_cb() (see page 30) for the memory operation callback (see set_mem_cb(), page 28).

```
template <typename T>
  void set_int_cb(T* o,
                  int (T::*f)(int     cpu,
                              uint8_t vector));
```

The equivalent of set_atomic_cb() (see page 30) for the interrupt callback (see set_int_cb(), page 28).

```
template <typename T>
  void set_trans_cb(T* o, int (T::*f)(int cpu));
```

The equivalent of set_atomic_cb() (see page 30) for the translation (see set_trans_cb(), page 29).

```
template <typename T>
  void set_inst_cb(T* o,
                   void (T::f)(int            cpu,
                               uint64_t       vaddr,
                               uint64_t       paddr,
                               uint8_t        len,
                               const uint8_t *bytes));
```

The equivalent of set_atomic_cb() (see page 30) for the instruction callback (see set_inst_cb(), page 28).

```
template <typename T>
  void set_app_start_cb(T* o,
                        int (T::f)(int));
```

The equivalent of set_atomic_cb() (see page 30) for the application start callback (see set_app_start_cb(), page 29).

```
template <typename T>
  void set_app_end_cb(T* o,
                      int (T::f)(int));
```

The equivalent of set_atomic_cb() (see page 30) for the application end callback (see set_app_end_cb(), page 30).

```
uint64_t get_reg(unsigned cpu, enum regs r);
```

Retrieve the value of a register on CPU i, referenced using one of the names from enum regs (see page ??).

```
void set_reg(unsigned cpu, enum regs r, uint64_t value);
```

Alter the value of a register on CPU i, referenced using one of the names from enum regs (see page ??).

```
template <typename T> void mem_rd(T& d, uint64_t paddr);
```

Read a value of arbitrary size from guest RAM at physical address `paddr`.

```
template <typename T> void mem_wr(T& d, uint64_t paddr);
```

Write a value of arbitrary size to guest RAM at physical address `paddr`.

```
template <typename T>
  void mem_rd_virt(unsigned i, T& d, uint64_t vaddr);
```

Read a value of arbitrary size from guest RAM at virtual address `vaddr`, translating the address according to CPU `i`.

```
template <typename T>
  void mem_wr_virt(unsigned i, T& d, uint64_t vaddr);
```

Write a value of arbitrary size to guest RAM at virtual address `vaddr`, translating the address according to CPU `i`.

A.1.3 Utility Functions

The following functions are not part of `Qsim::OSDomain` proper, but act on OSDomains to provide important features:

```
void load_file(OSDomain &osd, const char *filename);
```

If the `OSDomain` is running the `qsim-io` guest program, this function will provide input through this system. This is used by the default initial ramdisk to load a `tar` archive containing the application to be run, along with any data and libraries it needs. A demonstration of the use of `load-file` can be seen in `examples/io-test.cpp`.

A.2 Qsim::QemuCpu

QemuCpu is the basic CPU element in QSim, instantiated once for each CPU in the OSDomain. QemuCpu can also be instantiated by the client program directly, which could be used in the simulation of more complex distributed or non-coherent architectures. The guest RAM state illustrated in Figure 1.3 is contained inside of a "master", QemuCpu allocated by QEMU. Multiple "slave" QemuCpu objects share the RAM state allocated by the master, including the kernel image loaded by it. The various callback setters and other functions behave similarly to the way they do in OSDomain, but with less emphasis on convenience of use for the end-user.

A.2.1 Member Functions

```
QemuCpu(int id, const char* kernel, unsigned ram_mb=1024);
```

Construct a master QemuCpu with ram_mb megabytes of guest RAM and load the kernel image at the path in kernel.

```
QemuCpu(int id, QemuCpu *master_cpu, unsigned ram_mb=1024);
```

Create a slave QemuCpu, with master_cpu as its master. ram_mb must match the value of ram_mb given for the master.

```
unsigned run(unsigned n);
```

Run for n instructions or until an exception condition occurrs. Explained more fully in the description OSDomain::run() (see page 27).

```
typedef int (*atomic_cb_t)(int cpu);
void set_atomic_cb(atomic_cb_t cb);
```

Set an atomic memory operation callback for this CPU as explained in the description of the corresponding member function of OSDomain on page 27.

```
typedef void (*inst_cb_t)(int             cpu,
                          uint64_t        vaddr,
                          uint64_t        paddr,
                          uint8_t         len,
                          const uint8_t *bytes);
 void set_inst_cb(inst_cb_t cb);
```

Set an instruction callback for this CPU as explained in the description of the corresponding member function of OSDomain on page 28.

```
typedef int (*mem_cb_t)(int      cpu,
                        uint64_t vaddr,
                        uint64_t paddr,
                        uint8_t  size,
                        int      write);
 void set_mem_cb(mem_cb_t cb);
```

Set a memory operation callback for this CPU as explained in the description of the corresponding member function of OSDomain on page 28.

```
typedef int (*int_cb_t)(int cpu, uint8_t vec);
 void set_int_cb(int_cb_t cb);
```

Set an interrupt callback for this CPU as explained in the description of the corresponding member function of OSDomain on page 28.

```
typedef int (*magic_cb_t)(int cpu, uint64_t rax);
void set_magic_cb(magic_cb_t cb);
```

Set a magic instruction callback for this CPU. While this works similar to the callback setters for `OSDomain`, there is no way to set a static magic instruction callback for `OSDomain` as it already provides one. It is, however, possible to set member function callbacks as described on page 30.

```
typedef void (*io_cb_t)(int      cpu,
                       uint64_t port,
                       uint8_t  size,
                       int      type,
                       uint32_t val);
void set_io_cb(io_cb_t cb);
```

Set an I/O operation callback for this CPU as explained in the description of the corresponding member function of `OSDomain` on page 29.

```
typedef void (*reg_cb_t)(int     cpu,
                        int      reg,
                        uint8_t  size,
                        int      type);
void set_io_cb(reg_cb_t cb);
```

Set a register access callback for this CPU as explained in the description of the corresponding member function of `OSDomain` on page 29.

```
uint8_t mem_rd(uint64_t pa);
```

Read byte at physical address `pa`.

```
void mem_wr(uint64_t pa, uint8_t val);
```

Write byte `val` to physical address `pa`.

```
uint8_t mem_rd_virt(uint64_t va);
```

Read byte at virtual address `va`.

```
void mem_wr_virt(uint64_t va, uint8_t val);
```

Write byte `val` to virtual address `va`.

```
int interrupt(uint8_t vec);
```

Trigger an interrupt on this CPU with vector `vec`.

```
uint64_t get_reg(enum regs r);
```

Retreive the contents of register `r`.

```
void set_reg(enum regs r, uint64_t v);
```

Set the contents of register `r` to value `v`.

A.3 `enum flags`

The flags enumeration contains at least the following entries, set up to be used as bits in a vector:

```
enum flags {
  QSIM_FLAG_OF = 0x01, QSIM_FLAG_SF = 0x02,
  QSIM_FLAG_ZF = 0x04, QSIM_FLAG_AF = 0x08,
  QSIM_FLAG_PF = 0x10, QSIM_FLAG_CF = 0x20
};
```

This is defined in `qsim-regs.h` and used to generate the value for the `reg` parameter to the register access callback in the case that `size` is zero, indicating a flag bit access.