CYENG 225 – SP 23: Microcontroller Essentials for Cyber Applications
Instructor: Dr. Shayan (Sean) Taheri
Gannon University (GU)

Assignment 2: Chapter 2 - Basic Computer Security Concepts

Total Points: 100**; and Deadline:** February/09/2023, 11:59 PM.

Note – Cheating and Plagiarism: Cheating and plagiarism are not permitted in any form and they cause certain penalties. The instructor reserves the right to fail culprits.

Deliverable: All of your responses to the questions of assignment should be included in a single compressed file to be uploaded to the Gannon University (GU) – Blackboard Learn environment.

Question. Provide short answers (i.e., no more than five lines on average with the font size of 12) for the following items. The grade for each item is **10 points**.

1. Explain what PUF is and mention its applications.
2. Discuss how PKI is used in secure processor design.
3. Specify how hashes are used in digital signatures.
4. Mention the similarities and the differences between symmetric-key and asymmetric-key cryptographic algorithms.
5. Discuss the threats to the computing systems, including their hardware and software, after the design phase.

Question 2. Complete the laboratory part, titled "**Introduction to the gem5 Simulator System**". The grade for this question is **50 points**. Provide a report that includes: (1) your overall understanding of the experiments; (B) the interesting points and the challenges that you faced in this laboratory; and (C) the screenshots for all of the major steps/processes in your experiments.

CYENG 225 – SP 23: Microcontroller Essentials for Cyber Applications
Instructor: Dr. Shayan (Sean) Taheri
Gannon University (GU)

Introduction to the gem5 Simulator System
(Reference: https://www.gem5.org)

Introduction
The goal of this document is to give you a thorough introduction on how to use gem5 and the gem5 codebase. The purpose of this document is not to provide a detailed description of every feature in gem5. After reading this document, you should feel comfortable using gem5 in the classroom and for computer architecture research. Additionally, you should be able to modify and extend gem5 and then contribute your improvements to the main gem5 repository. One important lesson I have learned (the hard way) is when using complex tools like gem5, it is important to actually understand how it works before using it.

What is gem5?
gem5 is a modular discrete event driven computer system simulator platform. That means that:

1. gem5's components can be rearranged, parameterized, extended or replaced easily to suit your needs.
2. It simulates the passing of time as a series of discrete events.
3. Its intended use is to simulate one or more computer systems in various ways.
4. It's more than just a simulator; it's a simulator platform that lets you use as many of its premade components as you want to build up your own simulation system.

gem5 is written primarily in C++ and python and most components are provided under a BSD style license. It can simulate a complete system with devices and an operating system in full system mode (FS mode), or user space only programs where system services are provided directly by the simulator in syscall emulation mode (SE mode). There are varying levels of support for executing Alpha, ARM, MIPS, Power, SPARC, RISC-V, and 64-bit x86 binaries on CPU models including two simple single CPI models, an out of order model, and an in order pipelined model. A memory system can be flexibly built out of caches and crossbars or the Ruby simulator which provides even more flexible memory system modeling.

Capabilities out of the box
gem5 is designed for use in computer architecture research, but if you're trying to research something new and novel it probably won't be able to evaluate your idea out of the box. If it could, that probably means someone has already evaluated a similar idea and published about it.

To get the most out of gem5, you'll most likely need to add new capabilities specific to your project's goals. gem5's modular design should help you make modifications without having to understand every part of the simulator.

As you add the new features you need, please consider contributing your changes back to gem5. That way others can take advantage of your hard work, and gem5 can become an even better simulator.

Asking for help
gem5 has two main mailing lists where you can ask for help or advice. gem5-dev is for developers who are working on the main version of gem5. This is the version that's distributed from the website and most likely what you'll base your own work off of. gem5-users is a larger mailing list and is for people

working on their own projects which are not, at least initially, going to be distributed as part of the official version of gem5.

Most of the time, gem5-users is the right mailing list to use. Most of the people on gem5-dev are also on gem5-users including all the main developers, and in addition many other members of the gem5 community will see your post. That helps you because they might be able to answer your question, and it also helps them because they'll be able to see the answers people send you.

Building gem5

Requirements for gem5: See gem5 requirements for more details.

On Ubuntu, you can install all of the required dependencies with the following command. The requirements are detailed below.

```
sudo apt install build-essential git m4 scons zlib1g zlib1g-dev libprotobuf-dev protobuf-compiler
libprotoc-dev libgoogle-perftools-dev python-dev python
```

1. **git (Git):**
   The gem5 project uses Git for version control. Git is a distributed version control system. More information about Git can be found by following the link. Git should be installed by default on most platforms. However, to install Git in Ubuntu use:

   ```
   sudo apt install git
   ```

2. **gcc 7+**
   You may need to use environment variables to point to a non-default version of gcc.
   On Ubuntu, you can install a development environment with

   ```
   sudo apt install build-essential
   ```

   We support GCC Versions >=7, up to GCC 11.

3. **SCons 3.0+**
   gem5 uses SCons as its build environment. SCons is like make on steroids and uses Python scripts for all aspects of the build process. This allows for a very flexible (if slow) build system. To get SCons on Ubuntu use

   ```
   sudo apt install scons
   ```

4. **Python 3.6+**
   gem5 relies on the Python development libraries. To install these on Ubuntu use:

   ```
   sudo apt install python3-dev
   ```

5. **protobuf 2.1+ (Optional)**
   "Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data." In gem5, the protobuf library is used for trace generation and playback. protobuf is not a required package, unless you plan on using it for trace generation and playback.

```
sudo apt install libprotobuf-dev protobuf-compiler libgoogle-perftools-dev
```

6. **Boost (Optional)**
   The Boost library is a set of general purpose C++ libraries. It is a necessary dependency if you wish to use the SystemC implementation.

   ```
   sudo apt install libboost-all-dev
   ```

Getting the code
Change directories to where you want to download the gem5 source. Then, to clone the repository, use the git clone command.

```
git clone https://gem5.googlesource.com/public/gem5
```

You can now change directories to gem5 which contains all of the gem5 code.

Your first gem5 build
Let's start by building a basic x86 system. Currently, you must compile gem5 separately for every ISA that you want to simulate. Additionally, if using ruby-intro-chapter, you have to have separate compilations for every cache coherence protocol.

To build gem5, we will use SCons. SCons uses the SConstruct file (gem5/SConstruct) to set up a number of variables and then uses the SConscript file in every subdirectory to find and compile all of the gem5 source.

SCons automatically creates a gem5/build directory when first executed. In this directory you'll find the files generated by SCons, the compiler, etc. There will be a separate directory for each set of options (ISA and cache coherence protocol) that you use to compile gem5.

There are a number of default compilations options in the build_opts directory. These files specify the parameters passed to SCons when initially building gem5. We'll use the X86 defaults and specify that we want to compile all of the CPU models. You can look at the file build_opts/X86 to see the default values for the SCons options. You can also specify these options on the command line to override any default.

```
python3 `which scons` build/X86/gem5.opt -j9
```

gem5 binary types

The SCons scripts in gem5 currently have 5 different binaries you can build for gem5: debug, opt, and fast. These names are mostly self-explanatory, but detailed below.

debug
Built with no optimizations and debug symbols. This binary is useful when using a debugger to debug if the variables you need to view are optimized out in the opt version of gem5. Running with debug is slow compared to the other binaries.

opt

This binary is build with most optimizations on (e.g., -O3), but with debug symbols included. This binary is much faster than debug, but still contains enough debug information to be able to debug most problems.

fast

Built with all optimizations on (including link-time optimizations on supported platforms) and with no debug symbols. Additionally, any asserts are removed, but panics and fatals are still included. fast is the highest performing binary, and is much smaller than opt. However, fast is only appropriate when you feel that it is unlikely your code has major bugs.

The main argument passed to SCons is what you want to build, build/X86/gem5.opt. In this case, we are building gem5.opt (an optimized binary with debug symbols). We want to build gem5 in the directory build/X86. Since this directory currently doesn't exist, SCons will look in build_opts to find the default parameters for X86. (Note: I'm using -j9 here to execute the build on 9 of my 8 cores on my machine. You should choose an appropriate number for your machine, usually cores+1.)

The output should look something like below:

```
Checking for C header file Python.h... yes
Checking for C library pthread... yes
Checking for C library dl... yes
Checking for C library util... yes
Checking for C library m... yes
Checking for C library python2.7... yes
Checking for accept(0,0,0) in C++ library None... yes
Checking for zlibVersion() in C++ library z... yes
Checking for GOOGLE_PROTOBUF_VERIFY_VERSION in C++ library protobuf... yes
Checking for clock_nanosleep(0,0,NULL,NULL) in C library None... yes
Checking for timer_create(CLOCK_MONOTONIC, NULL, NULL) in C library None... no
Checking for timer_create(CLOCK_MONOTONIC, NULL, NULL) in C library rt... yes
Checking for C library tcmalloc... yes
Checking for backtrace_symbols_fd((void*)0, 0, 0) in C library None... yes
Checking for C header file fenv.h... yes
Checking for C header file linux/kvm.h... yes
Checking size of struct kvm_xsave ... yes
Checking for member exclude_host in struct perf_event_attr...yes
Building in /local.chinook/gem5/gem5-tutorial/gem5/build/X86
Variables file /local.chinook/gem5/gem5-tutorial/gem5/build/variables/X86 not found,
  using defaults in /local.chinook/gem5/gem5-tutorial/gem5/build_opts/X86
scons: done reading SConscript files.
scons: Building targets ...
 [ISA DESC] X86/arch/x86/isa/main.isa -> generated/inc.d
 [NEW DEPS] X86/arch/x86/generated/inc.d -> x86-deps
 [ENVIRONS] x86-deps -> x86-environs
 [     CXX] X86/sim/main.cc -> .o
 ....
 .... <lots of output>
 ....
```

```
[   SHCXX] nomali/lib/mali_midgard.cc -> .os
[   SHCXX] nomali/lib/mali_t6xx.cc -> .os
[   SHCXX] nomali/lib/mali_t7xx.cc -> .os
[      AR]  -> drampower/libdrampower.a
[   SHCXX] nomali/lib/addrspace.cc -> .os
[   SHCXX] nomali/lib/mmu.cc -> .os
[  RANLIB]  -> drampower/libdrampower.a
[   SHCXX] nomali/lib/nomali_api.cc -> .os
[      AR]  -> nomali/libnomali.a
[  RANLIB]  -> nomali/libnomali.a
[     CXX] X86/base/date.cc -> .o
[    LINK]  -> X86/gem5.opt
scons: done building targets.
```

When compilation is finished you should have a working gem5 executable at build/X86/gem5.opt. The compilation can take a very long time, often 15 minutes or more, especially if you are compiling on a remote file system like AFS or NFS.

Common errors

Wrong gcc version

```
Error: gcc version 5 or newer required.
       Installed version: 4.4.7
```

Update your environment variables to point to the right gcc version, or install a more up to date version of gcc. See building-requirements-section.

Python in a non-default location

If you use a non-default version of Python, (e.g., version 3.6 when 2.5 is your default), there may be problems when using SCons to build gem5. RHEL6 version of SCons uses a hardcoded location for Python, which causes the issue. gem5 often builds successfully in this case, but may not be able to run. Below is one possible error you may see when you run gem5.

```
Traceback (most recent call last):
  File "........../gem5-stable/src/python/importer.py", line 93, in <module>
    sys.meta_path.append(importer)
TypeError: 'dict' object is not callable
```

To fix this, you can force SCons to use your environment’s Python version by running python3 `which scons` build/X86/gem5.opt instead of scons build/X86/gem5.opt.

M4 macro processor not installed

If the M4 macro processor isn’t installed you’ll see an error similar to this:

```
...
Checking for member exclude_host in struct perf_event_attr...yes
Error: Can't find version of M4 macro processor.  Please install M4 and try again.
```

Just installing the M4 macro package may not solve this issue. You may nee to also install all of the autoconf tools. On Ubuntu, you can use the following command.

```
sudo apt-get install automake
```

Protobuf 3.12.3 problem

Compiling gem5 using protobuf might result in the following error,

```
In file included from build/X86/cpu/trace/trace_cpu.hh:53,
                 from build/X86/cpu/trace/trace_cpu.cc:38:
build/X86/proto/inst_dep_record.pb.h:49:51: error: 'AuxiliaryParseTableField' in namespace
'google::protobuf::internal' does not name a type; did you mean 'AuxillaryParseTableField'?
   49 |   static const ::PROTOBUF_NAMESPACE_ID::internal::AuxiliaryParseTableField aux[]
```

The root cause of the problem is discussed here: [https://gem5.atlassian.net/browse/GEM5-1032].

To resolve this problem, you may need to update the version of ProtocolBuffer,

```
sudo apt update
sudo apt install libprotobuf-dev protobuf-compiler libgoogle-perftools-dev
```

After that, you may need to clean the gem5 build folder **before** recompiling gem5,

```
python3 `which scons` --clean --no-cache        # cleaning the build folder
python3 `which scons` build/X86/gem5.opt -j 9   # re-compiling gem5
```

If the problem persists, you may need to completely remove the gem5 build folder **before** compiling gem5 again,

```
rm -rf build/                                   # completely removing the gem5 build folder
python3 `which scons` build/X86/gem5.opt -j 9   # re-compiling gem5
```

Creating a simple configuration script

This chapter of the tutorial will walk you through how to set up a simple simulation script for gem5 and to run gem5 for the first time. It’s assumed that you’ve completed the first chapter of the tutorial and have successfully built gem5 with an executable build/X86/gem5.opt.

Our configuration script is going to model a very simple system. We’ll have just one simple CPU core. This CPU core will be connected to a system-wide memory bus. And we’ll have a single DDR3 memory channel, also connected to the memory bus.

gem5 configuration scripts

The gem5 binary takes, as a parameter, a python script which sets up and executes the simulation. In this script, you create a system to simulate, create all of the components of the system, and specify all of the parameters for the system components. Then, from the script, you can begin the simulation.

This script is completely user-defined. You can choose to use any valid Python code in the configuration scripts. This book provides on example of a style that relies heavily classes and inheritance in Python. As a gem5 user, it's up to you how simple or complicated to make your configuration scripts.

There are a number of example configuration scripts that ship with gem5 in configs/examples. Most of these scripts are all-encompassing and allow users to specify almost all options on the command line. Instead of starting with these complex script, in this book we are going to start with the most simple script that can run gem5 and build from there. Hopefully, by the end of this section you'll have a good idea of how simulation scripts work.

An aside on SimObjects

gem5's modular design is built around the **SimObject** type. Most of the components in the simulated system are SimObjects: CPUs, caches, memory controllers, buses, etc. gem5 exports all of these objects from their C++ implementation to python. Thus, from the python configuration script you can create any SimObject, set its parameters, and specify the interactions between SimObjects.

See SimObject details for more information.

Creating a config file

Let's start by creating a new config file and opening it:

```
mkdir configs/tutorial/part1/
touch configs/tutorial/part1/simple.py
```

This is just a normal python file that will be executed by the embedded python in the gem5 executable. Therefore, you can use any features and libraries available in python.

The first thing we'll do in this file is import the m5 library and all SimObjects that we've compiled.

```
import m5
from m5.objects import *
```

Next, we'll create the first SimObject: the system that we are going to simulate. The System object will be the parent of all the other objects in our simulated system. The System object contains a lot of functional (not timing-level) information, like the physical memory ranges, the root clock domain, the root voltage domain, the kernel (in full-system simulation), etc. To create the system SimObject, we simply instantiate it like a normal python class:

```
system = System()
```

Now that we have a reference to the system we are going to simulate, let's set the clock on the system. We first have to create a clock domain. Then we can set the clock frequency on that domain. Setting parameters on a SimObject is exactly the same as setting members of an object in python, so we can simply set the clock to 1 GHz, for instance. Finally, we have to specify a voltage domain for this clock domain. Since we don't care about system power right now, we'll just use the default options for the voltage domain.

```
system.clk_domain = SrcClockDomain()
system.clk_domain.clock = '1GHz'
system.clk_domain.voltage_domain = VoltageDomain()
```

Once we have a system, let's set up how the memory will be simulated. We are going to use *timing* mode for the memory simulation. You will almost always use timing mode for the memory simulation, except in special cases like fast-forwarding and restoring from a checkpoint. We will also set up a single memory range of size 512 MB, a very small system. Note that in the python configuration scripts, whenever a size is required you can specify that size in common vernacular and units like '512MB'. Similarly, with time you can use time units (e.g., '5ns'). These will automatically be converted to a common representation, respectively.

```
system.mem_mode = 'timing'
system.mem_ranges = [AddrRange('512MB')]
```

Now, we can create a CPU. We'll start with the most simple timing-based CPU in gem5 for the X86 ISA, *X86TimingSimpleCPU*. This CPU model executes each instruction in a single clock cycle to execute, except memory requests, which flow through the memory system. To create the CPU you can simply just instantiate the object:

```
system.cpu = X86TimingSimpleCPU()
```

If we wanted to use the RISCV ISA we could use RiscvTimingSimpleCPU or if we wanted to use the ARM ISA we could use ArmTimingSimpleCPU. However, we will continue to use the X86 ISA for this exercise.

Next, we're going to create the system-wide memory bus:

```
system.membus = SystemXBar()
```

Now that we have a memory bus, let's connect the cache ports on the CPU to it. In this case, since the system we want to simulate doesn't have any caches, we will connect the I-cache and D-cache ports directly to the membus. In this example system, we have no caches.

```
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports
```

An aside on gem5 ports

To connect memory system components together, gem5 uses a port abstraction. Each memory object can have two kinds of ports, *request ports* and *response ports*. Requests are sent from a request port to a response port, and responses are sent from a response port to a request port. When connecting ports, you must connect a request port to a response port.

Connecting ports together is easy to do from the python configuration files. You can simply set the request port = to the response port and they will be connected. For instance:

```
system.cpu.icache_port = system.l1_cache.cpu_side
```

In this example, the cpu's icache_port is a request port, and the cache's cpu_side is a response port. The request port and the response port can be on either side of the = and the same connection will be made. After making the connection, the requestor can send requests to the responder. There is a lot of magic going on behind the scenes to set up the connection, the details of which are unimportant to most users.

Another notable kind of magic of the = of two ports in a gem5 Python configuration is that, it is allowed to have one port on one side, and an array of ports on the other side. For example:

```
system.cpu.icache_port = system.membus.cpu_side_ports
```

In this example, the cpu's icache_port is a request port, and the membus's cpu_side_ports is an array of response ports. In this case, a new response port is spawned on the cpu_side_ports, and this newly created port will be connected to the request port.

We will discuss ports and MemObject in more detail in the MemObject chapter.

Next, we need to connect up a few other ports to make sure that our system will function correctly. We need to create an I/O controller on the CPU and connect it to the memory bus. Also, we need to connect a special port in the system up to the membus. This port is a functional-only port to allow the system to read and write memory.

Connecting the PIO and interrupt ports to the memory bus is an x86-specific requirement. Other ISAs (e.g., ARM) do not require these 3 extra lines.

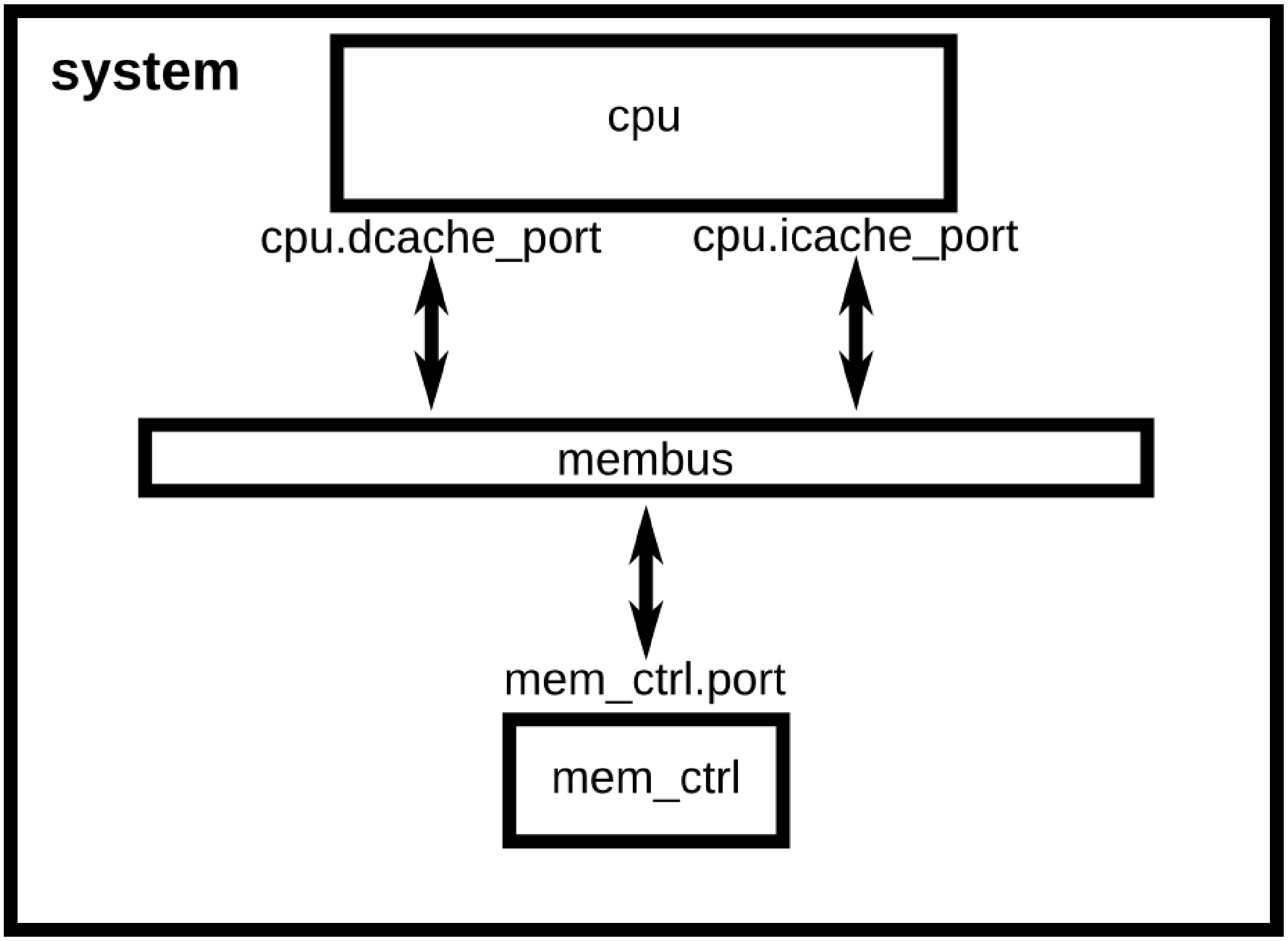
```
system.cpu.createInterruptController()
system.cpu.interrupts[0].pio = system.membus.mem_side_ports
system.cpu.interrupts[0].int_requestor = system.membus.cpu_side_ports
system.cpu.interrupts[0].int_responder = system.membus.mem_side_ports

system.system_port = system.membus.cpu_side_ports
```

Next, we need to create a memory controller and connect it to the membus. For this system, we'll use a simple DDR3 controller and it will be responsible for the entire memory range of our system.

```
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.mem_side_ports
```

After those final connections, we've finished instantiating our simulated system! Our system should look like the figure below.



Next, we need to set up the process we want the CPU to execute. Since we are executing in syscall emulation mode (SE mode), we will just point the CPU at the compiled executable. We'll execute a simple "Hello world" program. There's already one that is compiled that ships with gem5, so we'll use that. You can specify any application built for x86 and that's been statically compiled.

Full system versus syscall emulation

gem5 can run in two different modes called "syscall emulation" and "full system" or SE and FS modes. In full system mode (covered later full-system-part), gem5 emulates the entire hardware system and runs an unmodified kernel. Full system mode is similar to running a virtual machine.

Syscall emulation mode, on the other hand, does not emulate all of the devices in a system and focuses on simulating the CPU and memory system. Syscall emulation is much easier to configure since you are not required to instantiate all of the hardware devices required in a real system. However, syscall emulation only emulates Linux system calls, and thus only models user-mode code.

If you do not need to model the operating system for your research questions, and you want extra performance, you should use SE mode. However, if you need high fidelity modeling of the system, or OS interaction like page table walks are important, then you should use FS mode.

First, we have to create the process (another SimObject). Then we set the processes command to the command we want to run. This is a list similar to argv, with the executable in the first position and the arguments to the executable in the rest of the list. Then we set the CPU to use the process as it's workload, and finally create the functional execution contexts in the CPU.

```
binary = 'tests/test-progs/hello/bin/x86/linux/hello'

# for gem5 V21 and beyond
system.workload = SEWorkload.init_compatible(binary)

process = Process()
process.cmd = [binary]
system.cpu.workload = process
system.cpu.createThreads()
```

The final thing we need to do is instantiate the system and begin execution. First, we create the Root object. Then we instantiate the simulation. The instantiation process goes through all of the SimObjects we've created in python and creates the C++ equivalents.

As a note, you don't have to instantiate the python class then specify the parameters explicitly as member variables. You can also pass the parameters as named arguments, like the Root object below.

```
root = Root(full_system = False, system = system)
m5.instantiate()
```

Finally, we can kick off the actual simulation! As a side now, gem5 is now using Python 3-style print functions, so print is no longer a statement and must be called as a function.

```
print("Beginning simulation!")
exit_event = m5.simulate()
```

And once simulation finishes, we can inspect the state of the system.

```
print('Exiting @ tick {} because {}'
      .format(m5.curTick(), exit_event.getCause()))
```

Running gem5

Now that we've created a simple simulation script (the full version of which can be found in the gem5 code base at configs/learning_gem5/part1/simple.py) we're ready to run gem5. gem5 can take many parameters, but requires just one positional argument, the simulation script. So, we can simply run gem5 from the root gem5 directory as:

```
build/X86/gem5.opt configs/tutorial/part1/simple.py
```

The output should be:

```
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.0.0.0
gem5 compiled May 17 2021 18:05:59
gem5 started May 17 2021 22:05:20
gem5 executing on amarillo, pid 75197
command line: build/X86/gem5.opt configs/tutorial/part1/simple.py

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7005
Beginning simulation!
info: Entering event queue @ 0.  Starting simulation...
Hello world!
Exiting @ tick 490394000 because exiting with last active thread context
```

Parameters in the configuration file can be changed and the results should be different. For instance, if you double the system clock, the simulation should finish faster. Or, if you change the DDR controller to DDR4, the performance should be better.

Additionally, you can change the CPU model to X86MinorCPU to model an in-order CPU, or X86O3CPU to model an out-of-order CPU. However, note that X86O3CPU currently does not work with simple.py, because X86O3CPU requires a system with separate instruction and data caches (X86O3CPU does work with the configuration in the next section).

All gem5 BaseCPU's take the naming format {ISA}{Type}CPU. Ergo, if we wanted a RISCV Minor CPU we'd use RiscvMinorCPU.

The Valid ISAs are:

- Riscv
- Arm
- X86
- Sparc
- Power
- Mips

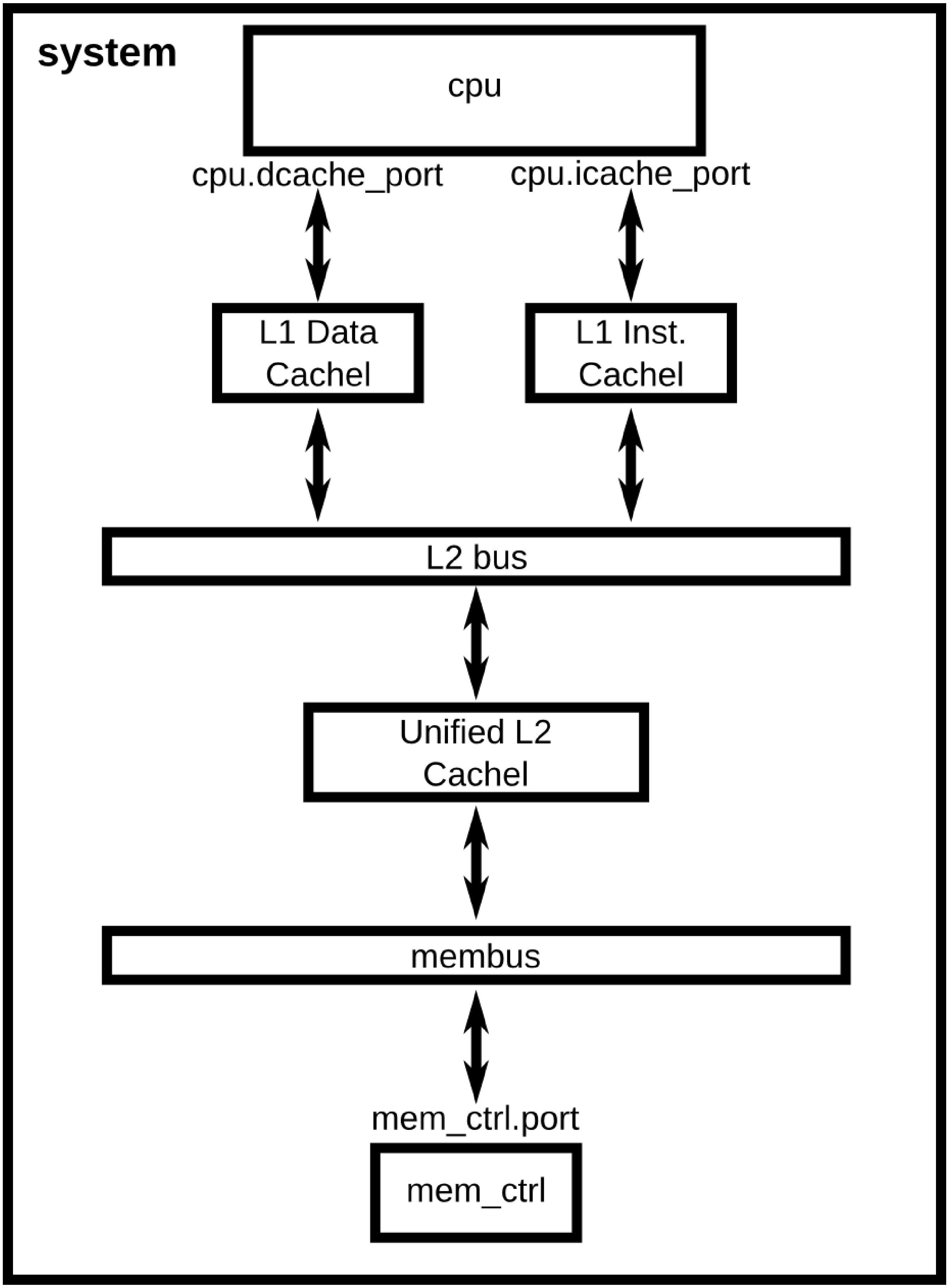
The CPU types are:

- AtomicSimpleCPU
- O3CPU
- TimingSimpleCPu
- KvmCPU
- MinorCPU

Next, we will add caches to our configuration file to model a more complex system.

Adding cache to the configuration script

Using the previous configuration script as a starting point, this chapter will walk through a more complex configuration. We will add a cache hierarchy to the system as shown in the figure below. Additionally, this chapter will cover understanding the gem5 statistics output and adding command line parameters to your scripts.


system
cpu
cpu.dcache_port
cpu.icache_port
L1 Data Cachel
L1 Inst. Cachel
L2 bus
Unified L2 Cachel
membus
mem_ctrl.port
mem_ctrl

Creating cache objects
We are going to use the classic caches, instead of ruby-intro-chapter, since we are modeling a single CPU system and we don't care about modeling cache coherence. We will extend the Cache SimObject and configure it for our system. First, we must understand the parameters that are used to configure Cache objects.

Classic caches and Ruby

gem5 currently has two completely distinct subsystems to model the on-chip caches in a system, the "Classic caches" and "Ruby". The historical reason for this is that gem5 is a combination of m5 from Michigan and GEMS from Wisconsin. GEMS used Ruby as its cache model, whereas the classic caches came from the m5 codebase (hence "classic"). The difference between these two models is that Ruby is designed to model cache coherence in detail. Part of Ruby is SLICC, a language for defining cache coherence protocols. On the other hand, the classic caches implement a simplified and inflexible MOESI coherence protocol.

To choose which model to use, you should ask yourself what you are trying to model. If you are modeling changes to the cache coherence protocol or the coherence protocol could have a first-order impact on your results, use Ruby. Otherwise, if the coherence protocol isn't important to you, use the classic caches.

A long-term goal of gem5 is to unify these two cache models into a single holistic model.

Cache

The Cache SimObject declaration can be found in src/mem/cache/Cache.py. This Python file defines the parameters which you can set of the SimObject. Under the hood, when the SimObject is instantiated these parameters are passed to the C++ implementation of the object. The Cache SimObject inherits from the BaseCache object shown below.

Within the BaseCache class, there are a number of *parameters*. For instance, assoc is an integer parameter. Some parameters, like write_buffers have a default value, 8 in this case. The default parameter is the first argument to Param.*, unless the first argument is a string. The string argument of each of the parameters is a description of what the parameter is (e.g., tag_latency = Param.Cycles("Tag lookup latency") means that the tag_latency controls "The hit latency for this cache").

Many of these parameters do not have defaults, so we are required to set these parameters before calling m5.instantiate().

Now, to create caches with specific parameters, we are first going to create a new file, caches.py, in the same directory as simple.py, configs/tutorial. The first step is to import the SimObject(s) we are going to extend in this file.

```
from m5.objects import Cache
```

Next, we can treat the BaseCache object just like any other Python class and extend it. We can name the new cache anything we want. Let's start by making an L1 cache.

```
class L1Cache(Cache):
    assoc = 2
    tag_latency = 2
    data_latency = 2
    response_latency = 2
    mshrs = 4
    tgts_per_mshr = 20
```

Here, we are setting some of the parameters of the BaseCache that do not have default values. To see all of the possible configuration options, and to find which are required and which are optional, you have to look at the source code of the SimObject. In this case, we are using BaseCache.

We have extended BaseCache and set most of the parameters that do not have default values in the BaseCache SimObject. Next, let's two more sub-classes of L1Cache, an L1DCache and L1ICache

```
class L1ICache(L1Cache):
    size = '16kB'

class L1DCache(L1Cache):
    size = '64kB'
```

Let's also create an L2 cache with some reasonable parameters.

```
class L2Cache(Cache):
    size = '256kB'
    assoc = 8
    tag_latency = 20
    data_latency = 20
    response_latency = 20
    mshrs = 20
    tgts_per_mshr = 12
```

Now that we have specified all of the necessary parameters required for BaseCache, all we have to do is instantiate our sub-classes and connect the caches to the interconnect. However, connecting lots of objects up to complex interconnects can make configuration files quickly grow and become unreadable. Therefore, let's first add some helper functions to our sub-classes of Cache. Remember, these are just Python classes, so we can do anything with them that you can do with a Python class.

To the L1 cache let's add two functions, connectCPU to connect a CPU to the cache and connectBus to connect the cache to a bus. We need to add the following code to the L1Cache class.

```
def connectCPU(self, cpu):
    # need to define this in a base class!
    raise NotImplementedError

def connectBus(self, bus):
    self.mem_side = bus.cpu_side_ports
```

Next, we have to define a separate connectCPU function for the instruction and data caches, since the I-cache and D-cache ports have a different names. Our L1ICache and L1DCache classes now become:

```
class L1ICache(L1Cache):
    size = '16kB'

    def connectCPU(self, cpu):
        self.cpu_side = cpu.icache_port

class L1DCache(L1Cache):
    size = '64kB'

    def connectCPU(self, cpu):
        self.cpu_side = cpu.dcache_port
```

Finally, let's add functions to the L2Cache to connect to the memory-side and CPU-side bus, respectively.

```
def connectCPUSideBus(self, bus):
    self.cpu_side = bus.mem_side_ports

def connectMemSideBus(self, bus):
    self.mem_side = bus.cpu_side_ports
```

The full file can be found in the gem5 source at configs/learning_gem5/part1/caches.py.

Adding caches to the simple config file

Now, let's add the caches we just created to the configuration script we created in the last chapter.

First, let's copy the script to a new name.

```
cp ./configs/tutorial/simple.py ./configs/tutorial/two_level.py
```

First, we need to import the names from the caches.py file into the namespace. We can add the following to the top of the file (after the m5.objects import), as you would with any Python source.

```
from caches import *
```

Now, after creating the CPU, let's create the L1 caches:

```
system.cpu.icache = L1ICache()
system.cpu.dcache = L1DCache()
```

And connect the caches to the CPU ports with the helper function we created.

```
system.cpu.icache.connectCPU(system.cpu)
system.cpu.dcache.connectCPU(system.cpu)
```

You need to *remove* the following two lines which connected the cache ports directly to the memory bus.

```
system.cpu.icache_port = system.membus.cpu_side_ports
system.cpu.dcache_port = system.membus.cpu_side_ports
```

We can't directly connect the L1 caches to the L2 cache since the L2 cache only expects a single port to connect to it. Therefore, we need to create an L2 bus to connect our L1 caches to the L2 cache. The, we can use our helper function to connect the L1 caches to the L2 bus.

```
system.l2bus = L2XBar()

system.cpu.icache.connectBus(system.l2bus)
system.cpu.dcache.connectBus(system.l2bus)
```

Next, we can create our L2 cache and connect it to the L2 bus and the memory bus.

```
system.l2cache = L2Cache()
system.l2cache.connectCPUSideBus(system.l2bus)
system.membus = SystemXBar()
system.l2cache.connectMemSideBus(system.membus)
```

Note that system.membus = SystemXBar() has been defined before system.l2cache.connectMemSideBus so we can pass it to system.l2cache.connectMemSideBus. Everything else in the file stays the same! Now we have a complete configuration with a two-level cache hierarchy. If you run the current file, hello should now finish in 57467000 ticks. The full script can be found in the gem5 source at `configs/learning_gem5/part1/two_level.py.

Adding parameters to your script

When performing experiments with gem5, you don't want to edit your configuration script every time you want to test the system with different parameters. To get around this, you can add command-line parameters to your gem5 configuration script. Again, because the configuration script is just Python, you can use the Python libraries that support argument parsing. Although pyoptparse is officially deprecated, many of the configuration scripts that ship with gem5 use it instead of pyargparse since gem5's minimum Python version used to be 2.5. The minimum Python version is now 3.6, so Python's argparse is a better option when writing new scripts that don't need to interact with the current gem5 scripts. To get started using :pyoptparse, you can consult the online Python documentation.

To add options to our two-level cache configuration, after importing our caches, let's add some options.

```
import argparse

parser = argparse.ArgumentParser(description='A simple system with 2-level cache.')
parser.add_argument("binary", default="", nargs="?", type=str,
                    help="Path to the binary to execute.")
parser.add_argument("--l1i_size",
                    help=f"L1 instruction cache size. Default: 16kB.")
parser.add_argument("--l1d_size",
                    help="L1 data cache size. Default: Default: 64kB.")
parser.add_argument("--l2_size",
                    help="L2 cache size. Default: 256kB.")

options = parser.parse_args()
```

Note that if you wanted to pass the binary file's path the way shown above and use it through options, you should specify it as options.binary. For example:

```
system.workload = SEWorkload.init_compatible(options.binary)
```

Now, you can run build/X86/gem5.opt configs/tutorial/two_level.py --help which will display the options you just added.

Next, we need to pass these options onto the caches that we create in the configuration script. To do this, we'll simply change two_level_opts.py to pass the options into the caches as a parameter to their constructor and add an appropriate constructor, next.

```
system.cpu.icache = L1ICache(options)
system.cpu.dcache = L1DCache(options)
...
system.l2cache = L2Cache(options)
```

In caches.py, we need to add constructors (__init__ functions in Python) to each of our classes. Starting with our base L1 cache, we'll just add an empty constructor since we don't have any parameters which apply to the base L1 cache. However, we can't forget to call the super class's constructor in this case. If the call to the super class constructor is skipped, gem5's SimObject attribute finding function will fail and the result will be "RuntimeError: maximum recursion depth exceeded" when you try to instantiate the cache object. So, in L1Cache we need to add the following after the static class members.

```
def __init__(self, options=None):
    super(L1Cache, self).__init__()
    pass
```

Next, in the L1ICache, we need to use the option that we created (l1i_size) to set the size. In the following code, there is guards for if options is not passed to the L1ICache constructor and if no option was specified on the command line. In these cases, we'll just use the default we've already specified for the size.

```
def __init__(self, options=None):
    super(L1ICache, self).__init__(options)
    if not options or not options.l1i_size:
        return
    self.size = options.l1i_size
```

We can use the same code for the L1DCache:

```
def __init__(self, options=None):
    super(L1DCache, self).__init__(options)
    if not options or not options.l1d_size:
        return
    self.size = options.l1d_size
```

And the unified L2Cache:

```
def __init__(self, options=None):
    super(L2Cache, self).__init__()
    if not options or not options.l2_size:
        return
    self.size = options.l2_size
```

With these changes, you can now pass the cache sizes into your script from the command line like below.

```
build/X86/gem5.opt configs/tutorial/two_level.py --l2_size='1MB' --l1d_size='128kB'
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.0.0.0
gem5 compiled May 17 2021 18:05:59
gem5 started May 18 2021 00:00:33
gem5 executing on amarillo, pid 83118
command line: build/X86/gem5.opt configs/tutorial/two_level.py --l2_size=1MB --l1d_size=128kB

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7005
Beginning simulation!
info: Entering event queue @ 0.  Starting simulation...
Hello world!
Exiting @ tick 57467000 because exiting with last active thread context
```

The full scripts can be found in the gem5 source at configs/learning_gem5/part1/caches.py and configs/learning_gem5/part1/two_level.py.

Understanding gem5 statistics and output
In addition to any information which your simulation script prints out, after running gem5, there are three files generated in a directory called m5out:

config.ini
Contains a list of every SimObject created for the simulation and the values for its parameters.
config.json
The same as config.ini, but in json format.
stats.txt
A text representation of all of the gem5 statistics registered for the simulation.

config.ini
This file is the definitive version of what was simulated. All of the parameters for each SimObject that is simulated, whether they were set in the configuration scripts or the defaults were used, are shown in this file.

Below is pulled from the config.ini generated when the simple.py configuration file from simple-config-chapter is run.

```
[root]
type=Root
children=system
eventq_index=0
full_system=false
sim_quantum=0
time_sync_enable=false
time_sync_period=100000000000
time_sync_spin_threshold=100000000

[system]
type=System
children=clk_domain cpu dvfs_handler mem_ctrl membus
boot_osflags=a
cache_line_size=64
clk_domain=system.clk_domain
default_p_state=UNDEFINED
eventq_index=0
exit_on_work_items=false
init_param=0
kernel=
kernel_addr_check=true
kernel_extras=
kvm_vm=Null
load_addr_mask=18446744073709551615
```

```
load_offset=0
mem_mode=timing

...

[system.membus]
type=CoherentXBar
children=snoop_filter
clk_domain=system.clk_domain
default_p_state=UNDEFINED
eventq_index=0
forward_latency=4
frontend_latency=3
p_state_clk_gate_bins=20
p_state_clk_gate_max=1000000000000
p_state_clk_gate_min=1000
point_of_coherency=true
point_of_unification=true
power_model=
response_latency=2
snoop_filter=system.membus.snoop_filter
snoop_response_latency=4
system=system
use_default_range=false
width=16
master=system.cpu.interrupts.pio system.cpu.interrupts.int_slave system.mem_ctrl.port
slave=system.cpu.icache_port system.cpu.dcache_port system.cpu.interrupts.int_master
system.system_port

[system.membus.snoop_filter]
type=SnoopFilter
eventq_index=0
lookup_latency=1
max_capacity=8388608
system=system
```

Here we see that at the beginning of the description of each SimObject is first its name as created in the configuration file surrounded by square brackets (e.g., [system.membus]).

Next, every parameter of the SimObject is shown with its value, including parameters not explicitly set in the configuration file. For instance, the configuration file sets the clock domain to be 1 GHz (1000 ticks in this case). However, it did not set the cache line size (which is 64 in the system) object.

The config.ini file is a valuable tool for ensuring that you are simulating what you think you're simulating. There are many possible ways to set default values, and to override default values, in gem5. It is a "best-practice" to always check the config.ini as a sanity check that values set in the configuration file are propagated to the actual SimObject instantiation.

stats.txt
gem5 has a flexible statistics generating system. gem5 statistics is covered in some detail on the gem5 wiki site. Each instantiation of a SimObject has it's own statistics. At the end of simulation, or when special statistic-dumping commands are issued, the current state of the statistics for all SimObjects is dumped to a file.

First, the statistics file contains general statistics about the execution:

```
---------- Begin Simulation Statistics ----------
simSeconds                    0.000057                 # Number of seconds simulated (Second)
simTicks                     57467000                 # Number of ticks simulated (Tick)
finalTick                    57467000                 # Number of ticks from beginning of simulation
(restored from checkpoints and never reset) (Tick)
simFreq                  1000000000000                 # The number of ticks per simulated second
((Tick/Second))
hostSeconds                       0.03                 # Real time elapsed on the host (Second)
hostTickRate                2295882330                 # The number of ticks simulated per host
second (ticks/s) ((Tick/Second))
hostMemory                      665792                 # Number of bytes of host memory used (Byte)
simInsts                          6225                 # Number of instructions simulated (Count)
simOps                           11204                 # Number of ops (including micro ops) simulated
(Count)
hostInstRate                    247382                 # Simulator instruction rate (inst/s)
((Count/Second))
hostOpRate                      445086                 # Simulator op (including micro ops) rate (op/s)
((Count/Second))

---------- Begin Simulation Statistics ----------
simSeconds                    0.000490                 # Number of seconds simulated (Second)
simTicks                     490394000                 # Number of ticks simulated (Tick)
finalTick                    490394000                 # Number of ticks from beginning of simulation
(restored from checkpoints and never reset) (Tick)
simFreq                  1000000000000                 # The number of ticks per simulated second
((Tick/Second))
hostSeconds                       0.03                 # Real time elapsed on the host (Second)
hostTickRate               15979964060                 # The number of ticks simulated per host
second (ticks/s) ((Tick/Second))
hostMemory                      657488                 # Number of bytes of host memory used (Byte)
simInsts                          6225                 # Number of instructions simulated (Count)
simOps                           11204                 # Number of ops (including micro ops) simulated
(Count)
hostInstRate                    202054                 # Simulator instruction rate (inst/s)
((Count/Second))
hostOpRate                      363571                 # Simulator op (including micro ops) rate (op/s)
((Count/Second))
```

The statistic dump begins with ---------- Begin Simulation Statistics ----------. There may be multiple of these in a single file if there are multiple statistic dumps during the gem5 execution. This is common for long running applications, or when restoring from checkpoints.

Each statistic has a name (first column), a value (second column), and a description (last column preceded by #) followed by the unit of the statistic.

Most of the statistics are self explanatory from their descriptions. A couple of important statistics are sim_seconds which is the total simulated time for the simulation, sim_insts which is the number of instructions committed by the CPU, and host_inst_rate which tells you the performance of gem5.

Next, the SimObjects' statistics are printed. For instance, the CPU statistics, which contains information on the number of syscalls, statistics for cache system and translation buffers, etc.

```
system.clk_domain.clock                       1000                    # Clock period in ticks (Tick)
system.clk_domain.voltage_domain.voltage           1                   # Voltage in Volts (Volt)
system.cpu.numCycles                         57467                   # Number of cpu cycles simulated (Cycle)
system.cpu.numWorkItemsStarted                     0                   # Number of work items this cpu started
(Count)
system.cpu.numWorkItemsCompleted                   0                   # Number of work items this cpu
completed (Count)
system.cpu.dcache.demandHits::cpu.data          1941                   # number of demand (read+write)
hits (Count)
system.cpu.dcache.demandHits::total             1941                   # number of demand (read+write) hits
(Count)
system.cpu.dcache.overallHits::cpu.data         1941                   # number of overall hits (Count)
system.cpu.dcache.overallHits::total            1941                   # number of overall hits (Count)
system.cpu.dcache.demandMisses::cpu.data         133                   # number of demand (read+write)
misses (Count)
system.cpu.dcache.demandMisses::total            133                   # number of demand (read+write)
misses (Count)
system.cpu.dcache.overallMisses::cpu.data        133                   # number of overall misses (Count)
system.cpu.dcache.overallMisses::total           133                   # number of overall misses (Count)
system.cpu.dcache.demandMissLatency::cpu.data     14301000                   # number of demand
(read+write) miss ticks (Tick)
system.cpu.dcache.demandMissLatency::total     14301000                   # number of demand
(read+write) miss ticks (Tick)
system.cpu.dcache.overallMissLatency::cpu.data     14301000                   # number of overall miss
ticks (Tick)
system.cpu.dcache.overallMissLatency::total     14301000                   # number of overall miss ticks
(Tick)
system.cpu.dcache.demandAccesses::cpu.data        2074                   # number of demand (read+write)
accesses (Count)
system.cpu.dcache.demandAccesses::total          2074                   # number of demand (read+write)
accesses (Count)
system.cpu.dcache.overallAccesses::cpu.data        2074                   # number of overall (read+write)
accesses (Count)
system.cpu.dcache.overallAccesses::total         2074                   # number of overall (read+write)
accesses (Count)
```

```
system.cpu.dcache.demandMissRate::cpu.data     0.064127                       # miss rate for demand
accesses (Ratio)
system.cpu.dcache.demandMissRate::total      0.064127                       # miss rate for demand accesses
(Ratio)
system.cpu.dcache.overallMissRate::cpu.data     0.064127                       # miss rate for overall accesses
(Ratio)
system.cpu.dcache.overallMissRate::total     0.064127                       # miss rate for overall accesses
(Ratio)
system.cpu.dcache.demandAvgMissLatency::cpu.data 107526.315789                       # average overall
miss latency ((Cycle/Count))
system.cpu.dcache.demandAvgMissLatency::total 107526.315789                       # average overall miss
latency ((Cycle/Count))
system.cpu.dcache.overallAvgMissLatency::cpu.data 107526.315789                       # average overall
miss latency ((Cycle/Count))
system.cpu.dcache.overallAvgMissLatency::total 107526.315789                       # average overall miss
latency ((Cycle/Count))
...
system.cpu.mmu.dtb.rdAccesses                    1123                       # TLB accesses on read requests
(Count)
system.cpu.mmu.dtb.wrAccesses                     953                       # TLB accesses on write requests
(Count)
system.cpu.mmu.dtb.rdMisses                        11                       # TLB misses on read requests (Count)
system.cpu.mmu.dtb.wrMisses                         9                       # TLB misses on write requests (Count)
system.cpu.mmu.dtb.walker.power_state.pwrStateResidencyTicks::UNDEFINED     57467000
# Cumulative time (in ticks) in various power states (Tick)
system.cpu.mmu.itb.rdAccesses                       0                       # TLB accesses on read requests (Count)
system.cpu.mmu.itb.wrAccesses                    7940                       # TLB accesses on write requests
(Count)
system.cpu.mmu.itb.rdMisses                         0                       # TLB misses on read requests (Count)
system.cpu.mmu.itb.wrMisses                        37                       # TLB misses on write requests (Count)
system.cpu.mmu.itb.walker.power_state.pwrStateResidencyTicks::UNDEFINED     57467000
# Cumulative time (in ticks) in various power states (Tick)
system.cpu.power_state.pwrStateResidencyTicks::ON     57467000                       # Cumulative time (in
ticks) in various power states (Tick)
system.cpu.thread_0.numInsts                        0                       # Number of Instructions committed
(Count)
system.cpu.thread_0.numOps                          0                       # Number of Ops committed (Count)
system.cpu.thread_0.numMemRefs                      0                       # Number of Memory References
(Count)
system.cpu.workload.numSyscalls                    11                       # Number of system calls (Count)
```

Later in the file is memory controller statistics. This has information like the bytes read by each component and the average bandwidth used by those components.

```
system.mem_ctrl.bytesReadWrQ                        0                       # Total number of bytes read from write
queue (Byte)
system.mem_ctrl.bytesReadSys                    23168                       # Total read bytes from the system
interface side (Byte)
```

```
system.mem_ctrl.bytesWrittenSys                    0                        # Total written bytes from the system
interface side (Byte)
system.mem_ctrl.avgRdBWSys              403153113.96105593                       # Average system read
bandwidth in Byte/s ((Byte/Second))
system.mem_ctrl.avgWrBWSys                0.00000000                       # Average system write bandwidth
in Byte/s ((Byte/Second))
system.mem_ctrl.totGap                       57336000                        # Total gap between requests (Tick)
system.mem_ctrl.avgGap                       158386.74                        # Average gap between requests
((Tick/Count))
system.mem_ctrl.requestorReadBytes::cpu.inst        14656                       # Per-requestor bytes read from
memory (Byte)
system.mem_ctrl.requestorReadBytes::cpu.data         8512                       # Per-requestor bytes read from
memory (Byte)
system.mem_ctrl.requestorReadRate::cpu.inst 255033323.472601681948                       # Per-requestor
bytes read from memory rate ((Byte/Second))
system.mem_ctrl.requestorReadRate::cpu.data 148119790.488454252481                       # Per-requestor
bytes read from memory rate ((Byte/Second))
system.mem_ctrl.requestorReadAccesses::cpu.inst          229                       # Per-requestor read serviced
memory accesses (Count)
system.mem_ctrl.requestorReadAccesses::cpu.data          133                       # Per-requestor read serviced
memory accesses (Count)
system.mem_ctrl.requestorReadTotalLat::cpu.inst      6234000                       # Per-requestor read total
memory access latency (Tick)
system.mem_ctrl.requestorReadTotalLat::cpu.data      4141000                       # Per-requestor read total
memory access latency (Tick)
system.mem_ctrl.requestorReadAvgLat::cpu.inst     27222.71                       # Per-requestor read average
memory access latency ((Tick/Count))
system.mem_ctrl.requestorReadAvgLat::cpu.data     31135.34                       # Per-requestor read
average memory access latency ((Tick/Count))
system.mem_ctrl.dram.bytesRead::cpu.inst        14656                       # Number of bytes read from this
memory (Byte)
system.mem_ctrl.dram.bytesRead::cpu.data         8512                       # Number of bytes read from this
memory (Byte)
system.mem_ctrl.dram.bytesRead::total           23168                       # Number of bytes read from this
memory (Byte)
system.mem_ctrl.dram.bytesInstRead::cpu.inst        14656                       # Number of instructions bytes
read from this memory (Byte)
system.mem_ctrl.dram.bytesInstRead::total        14656                       # Number of instructions bytes read
from this memory (Byte)
system.mem_ctrl.dram.numReads::cpu.inst            229                       # Number of read requests
responded to by this memory (Count)
system.mem_ctrl.dram.numReads::cpu.data            133                       # Number of read requests
responded to by this memory (Count)
system.mem_ctrl.dram.numReads::total              362                       # Number of read requests responded
to by this memory (Count)
system.mem_ctrl.dram.bwRead::cpu.inst       255033323                       # Total read bandwidth from this
memory ((Byte/Second))
system.mem_ctrl.dram.bwRead::cpu.data       148119790                       # Total read bandwidth from this
memory ((Byte/Second))
```

```
system.mem_ctrl.dram.bwRead::total         403153114                      # Total read bandwidth from this
memory ((Byte/Second))
system.mem_ctrl.dram.bwInstRead::cpu.inst    255033323                      # Instruction read bandwidth
from this memory ((Byte/Second))
system.mem_ctrl.dram.bwInstRead::total      255033323                      # Instruction read bandwidth from
this memory ((Byte/Second))
system.mem_ctrl.dram.bwTotal::cpu.inst      255033323                      # Total bandwidth to/from this
memory ((Byte/Second))
system.mem_ctrl.dram.bwTotal::cpu.data      148119790                      # Total bandwidth to/from this
memory ((Byte/Second))
system.mem_ctrl.dram.bwTotal::total        403153114                      # Total bandwidth to/from this
memory ((Byte/Second))
system.mem_ctrl.dram.readBursts                362                   # Number of DRAM read bursts (Count)
system.mem_ctrl.dram.writeBursts                 0                   # Number of DRAM write bursts (Count)
```

Using the default configuration scripts

In this chapter, we'll explore using the default configuration scripts that come with gem5. gem5 ships with many configuration scripts that allow you to use gem5 very quickly. However, a common pitfall is to use these scripts without fully understanding what is being simulated. It is important when doing computer architecture research with gem5 to fully understand the system you are simulating. This chapter will walk you through some important options and parts of the default configuration scripts.

In the last few chapters you have created your own configuration scripts from scratch. This is very powerful, as it allows you to specify every single system parameter. However, some systems are very complex to set up (e.g., a full-system ARM or x86 machine). Luckily, the gem5 developers have provided many scripts to bootstrap the process of building systems.

A tour of the directory structure

All of gem5's configuration files can be found in configs/. The directory structure is shown below:

```
configs/boot:
bbench-gb.rcS  bbench-ics.rcS  hack_back_ckpt.rcS  halt.sh

configs/common:
Benchmarks.py   Caches.py  cpu2000.py   FileSystemConfig.py  GPUTLBConfig.py   HMC.py
MemConfig.py   Options.py    Simulation.py
CacheConfig.py  cores     CpuConfig.py  FSConfig.py        GPUTLBOptions.py  __init__.py
ObjectList.py  SimpleOpts.py  SysPaths.py

configs/dist:
sw.py

configs/dram:
lat_mem_rd.py  low_power_sweep.py  sweep.py

configs/example:
apu_se.py  etrace_replay.py  garnet_synth_traffic.py  hmctest.py   hsaTopology.py  memtest.py
read_config.py  ruby_direct_test.py     ruby_mem_test.py     sc_main.py
```

```
arm       fs.py          hmc_hello.py          hmc_tgen.cfg  memcheck.py    noc_config  riscv
ruby_gpu_random_test.py  ruby_random_test.py  se.py

configs/learning_gem5:
part1  part2  part3  README

configs/network:
__init__.py  Network.py

configs/nvm:
sweep_hybrid.py  sweep.py

configs/ruby:
AMD_Base_Constructor.py  CHI.py       Garnet_standalone.py  __init__.py
MESI_Three_Level.py  MI_example.py     MOESI_CMP_directory.py  MOESI_hammer.py
CHI_config.py           CntrlBase.py  GPU_VIPER.py         MESI_Three_Level_HTM.py
MESI_Two_Level.py    MOESI_AMD_Base.py  MOESI_CMP_token.py      Ruby.py

configs/splash2:
cluster.py  run.py

configs/topologies:
BaseTopology.py  Cluster.py  CrossbarGarnet.py  Crossbar.py  CustomMesh.py  __init__.py
MeshDirCorners_XY.py  Mesh_westfirst.py  Mesh_XY.py  Pt2Pt.py
```

Each directory is briefly described below:

boot/

These are rcS files which are used in full-system mode. These files are loaded by the simulator after Linux boots and are executed by the shell. Most of these are used to control benchmarks when running in full-system mode. Some are utility functions, like hack_back_ckpt.rcS. These files are covered in more depth in the chapter on full-system simulation.

common/

This directory contains a number of helper scripts and functions to create simulated systems. For instance, Caches.py is similar to the caches.py and caches_opts.py files created in previous chapters.

Options.py contains a variety of options that can be set on the command line. Like the number of CPUs, system clock, and many, many more. This is a good place to look to see if the option you want to change already has a command line parameter.

CacheConfig.py contains the options and functions for setting cache parameters for the classic memory system.

MemConfig.py provides some helper functions for setting the memory system.

FSConfig.py contains the necessary functions to set up full-system simulation for many different kinds of systems. Full-system simulation is discussed further in it's own chapter.

Simulation.py contains many helper functions to set up and run gem5. A lot of the code contained in this file manages saving and restoring checkpoints. The example configuration files below in examples/ use the functions in this file to execute the gem5 simulation. This file is quite complicated, but it also allows a lot of flexibility in how the simulation is run.

dram/
Contains scripts to test DRAM.

example/
This directory contains some example gem5 configuration scripts that can be used out-of-the-box to run gem5. Specifically, se.py and fs.py are quite useful. More on these files can be found in the next section. There are also some other utility configuration scripts in this directory.

learning_gem5/
This directory contains all gem5 configuration scripts found in the learning_gem5 book.

network/
This directory contains the configurations scripts for a HeteroGarnet network.

nvm/
This directory contains example scripts using the NVM interface.

ruby/
This directory contains the configurations scripts for Ruby and its included cache coherence protocols. More details can be found in the chapter on Ruby.

splash2/
This directory contains scripts to run the splash2 benchmark suite with a few options to configure the simulated system.

topologies/
This directory contains the implementation of the topologies that can be used when creating the Ruby cache hierarchy. More details can be found in the chapter on Ruby.

Using se.py and fs.py

In this section, I'll discuss some of the common options that can be passed on the command line to se.py and fs.py. More details on how to run full-system simulation can be found in the full-system simulation chapter. Here I'll discuss the options that are common to the two files.

Most of the options discussed in this section are found in Options.py and are registered in the function addCommonOptions. This section does not detail all of the options. To see all of the options, run the configuration script with --help, or read the script's source code.

First, let's simply run the hello world program without any parameters:

```
build/X86/gem5.opt configs/example/se.py --cmd=tests/test-progs/hello/bin/x86/linux/hello
```

And we get the following as output:

```
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.0.0.0
gem5 compiled May 17 2021 18:05:59
gem5 started May 18 2021 00:33:42
```

```
gem5 executing on amarillo, pid 85168
command line: build/X86/gem5.opt configs/example/se.py --cmd=tests/test-
progs/hello/bin/x86/linux/hello

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7005
**** REAL SIMULATION ****
info: Entering event queue @ 0.  Starting simulation...
Hello world!
Exiting @ tick 5943000 because exiting with last active thread context
```

However, this isn’t a very interesting simulation at all! By default, gem5 uses the atomic CPU and uses atomic memory accesses, so there’s no real timing data reported! To confirm this, you can look at m5out/config.ini. The CPU is shown on line 51:

```
[system.cpu]
type=X86AtomicSimpleCPU
children=interrupts isa mmu power_state tracer workload
branchPred=Null
checker=Null
clk_domain=system.cpu_clk_domain
cpu_id=0
do_checkpoint_insts=true
do_statistics_insts=true
```

To actually run gem5 in timing mode, let’s specify a CPU type. While we’re at it, we can also specify sizes for the L1 caches.

```
build/X86/gem5.opt configs/example/se.py --cmd=tests/test-progs/hello/bin/x86/linux/hello --cpu-
type=TimingSimpleCPU --l1d_size=64kB --l1i_size=16kB
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.0.0.0
gem5 compiled May 17 2021 18:05:59
gem5 started May 18 2021 00:36:10
gem5 executing on amarillo, pid 85269
command line: build/X86/gem5.opt configs/example/se.py --cmd=tests/test-
progs/hello/bin/x86/linux/hello --cpu-type=TimingSimpleCPU --l1d_size=64kB --l1i_size=16kB

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7005
**** REAL SIMULATION ****
```

```
info: Entering event queue @ 0.  Starting simulation...
Hello world!
Exiting @ tick 454646000 because exiting with last active thread context
```

Now, let's check the config.ini file and make sure that these options propagated correctly to the final system. If you search m5out/config.ini for "cache", you'll find that no caches were created! Even though we specified the size of the caches, we didn't specify that the system should use caches, so they weren't created. The correct command line should be:

```
build/X86/gem5.opt configs/example/se.py --cmd=tests/test-progs/hello/bin/x86/linux/hello --cpu-type=TimingSimpleCPU --l1d_size=64kB --l1i_size=16kB --caches
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.0.0.0
gem5 compiled May 17 2021 18:05:59
gem5 started May 18 2021 00:37:03
gem5 executing on amarillo, pid 85560
command line: build/X86/gem5.opt configs/example/se.py --cmd=tests/test-progs/hello/bin/x86/linux/hello --cpu-type=TimingSimpleCPU --l1d_size=64kB --l1i_size=16kB --caches

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7005
**** REAL SIMULATION ****
info: Entering event queue @ 0.  Starting simulation...
Hello world!
Exiting @ tick 31680000 because exiting with last active thread context
```

On the last line, we see that the total time went from 454646000 ticks to 31680000, much faster! Looks like caches are probably enabled now. But, it's always a good idea to double check the config.ini file.

```
[system.cpu.dcache]
type=Cache
children=power_state replacement_policy tags
addr_ranges=0:18446744073709551615
assoc=2
clk_domain=system.cpu_clk_domain
clusivity=mostly_incl
compressor=Null
data_latency=2
demand_mshr_reserve=1
eventq_index=0
is_read_only=false
max_miss_count=0
```

```
move_contractions=true
mshrs=4
power_model=
power_state=system.cpu.dcache.power_state
prefetch_on_access=false
prefetcher=Null
replace_expansions=true
replacement_policy=system.cpu.dcache.replacement_policy
response_latency=2
sequential_access=false
size=65536
system=system
tag_latency=2
tags=system.cpu.dcache.tags
tgts_per_mshr=20
warmup_percentage=0
write_allocator=Null
write_buffers=8
writeback_clean=false
cpu_side=system.cpu.dcache_port
mem_side=system.membus.cpu_side_ports[2]
```

Some common options se.py and fs.py
All of the possible options are printed when you run:

```
build/X86/gem5.opt configs/example/se.py --help
```

Below are a few important options from that list:

- --cpu-type=CPU_TYPE
  - The type of cpu to run with. This is an important parameter to always set. The default is atomic, which doesn't perform a timing simulation.
- --sys-clock=SYS_CLOCK
  - Top-level clock for blocks running at system speed.
- --cpu-clock=CPU_CLOCK
  - Clock for blocks running at CPU speed. This is separate from the system clock above.
- --mem-type=MEM_TYPE
  - Type of memory to use. Options include different DDR memories, and the ruby memory controller.
- --caches
  - Perform the simulation with classic caches.
- --l2cache
  - Perform the simulation with an L2 cache, if using classic caches.
- --ruby
  - Use Ruby instead of the classic caches as the cache system simulation.
- -m TICKS, --abs-max-tick=TICKS
  - Run to absolute simulated tick specified including ticks from a restored checkpoint. This is useful if you only want simulate for a certain amount of simulated time.

- -I MAXINSTS, --maxinsts=MAXINSTS
    - Total number of instructions to simulate (default: run forever). This is useful if you want to stop simulation after a certain number of instructions has been executed.
- -c CMD, --cmd=CMD
    - The binary to run in syscall emulation mode.
- -o OPTIONS, --options=OPTIONS
    - The options to pass to the binary, use " " around the entire string. This is useful when you are running a command which takes options. You can pass both arguments and options (e.g., –whatever) through this variable.
- --output=OUTPUT
    - Redirect stdout to a file. This is useful if you want to redirect the output of the simulated application to a file instead of printing to the screen. Note: to redirect gem5 output, you have to pass a parameter before the configuration script.
- --errout=ERROUT
    - Redirect stderr to a file. Similar to above.

Extending gem5 for ARM

This chapter assumes you've already built a basic x86 system with gem5 and created a simple configuration script.

Downloading ARM Binaries

Let's start by downloading some ARM benchmark binaries. Begin from the root of the gem5 folder:

```
mkdir -p cpu_tests/benchmarks/bin/arm
cd cpu_tests/benchmarks/bin/arm
wget dist.gem5.org/dist/v22-0/test-progs/cpu-tests/bin/arm/Bubblesort
wget dist.gem5.org/dist/v22-0/test-progs/cpu-tests/bin/arm/FloatMM
```

We'll use these to further test our ARM system.

Building gem5 to run ARM Binaries

Just as we did when we first built our basic x86 system, we run the same command, except this time we want it to compile with the default ARM configurations. To do so, we just replace x86 with ARM:

```
scons build/ARM/gem5.opt -j 20
```

When compilation is finished you should have a working gem5 executable at build/ARM/gem5.opt.

Modifying simple.py to run ARM Binaries

Before we can run any ARM binaries with our new system, we'll have to make a slight tweak to our simple.py.

If you recall when we created our simple configuration script, it was noted that we did not have to connect the PIO and interrupt ports to the memory bus for any ISA other than for an x86 system. So let's remove those 3 lines:

```
system.cpu.createInterruptController()
#system.cpu.interrupts[0].pio = system.membus.mem_side_ports
#system.cpu.interrupts[0].int_requestor = system.membus.cpu_side_ports
```

```
#system.cpu.interrupts[0].int_responder = system.membus.mem_side_ports

system.system_port = system.membus.cpu_side_ports
```

You can either delete or comment them out as above. Next let's set the processes command to one of our ARM benchmark binaries:

```
process.cmd = ['cpu_tests/benchmarks/bin/arm/Bubblesort']
```

If you'd like to test a simple hello program as before, just replace x86 with arm:

```
process.cmd = ['tests/test-progs/hello/bin/arm/linux/hello']
```

Running gem5
Simply run it as before, except replace X86 with ARM:

```
build/ARM/gem5.opt configs/tutorial/simple.py
```

If you set your process to be the Bubblesort benchmark, your output should look like this:

```
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled Oct  3 2019 16:02:35
gem5 started Oct  6 2019 13:22:25
gem5 executing on amarillo, pid 77129
command line: build/ARM/gem5.opt configs/tutorial/simple.py

Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7002
Beginning simulation!
info: Entering event queue @ 0.  Starting simulation...
info: Increasing stack size by one page.
warn: readlink() called on '/proc/self/exe' may yield unexpected results in various settings.
      Returning '/home/jtoya/gem5/cpu_tests/benchmarks/bin/arm/Bubblesort'
-50000
Exiting @ tick 258647411000 because exiting with last active thread context
```

ARM Full System Simulation
To run an ARM FS Simulation, there are some changes required to the setup.

If you haven't already, from the gem5 repository's root directory, cd into the directory util/term/ by running

```
$ cd util/term/
```

and then compile the m5term binary by running

```
$ make
```

The gem5 repository comes with example system setups and configurations. These can be found in the configs/example/arm/ directory.

A collection of full system Linux image files are available here. Save these in a directory and remember the path to it. For example, you could store them in

```
/path/to/user/gem5/fs_images/
```

The fs_images directory will be assumed to contain the extracted FS images for the rest of this example.

With the image(s) downloaded, execute the following command in your terminal:

```
$ export IMG_ROOT=/absolute/path/to/fs_images/<image-directory-name>
```

replacing "<image-directory-name>" with the name of the directory extracted from the downloaded image file, without the angle-brackets.

We are now ready to run a FS ARM simulation. From the root of the gem5 repository, run:

```
$ ./build/ARM/gem5.opt configs/example/arm/fs_bigLITTLE.py \
    --caches \
    --bootloader="$IMG_ROOT/binaries/<bootloader-name>" \
    --kernel="$IMG_ROOT/binaries/<kernel-name>" \
    --disk="$IMG_ROOT/disks/<disk-image-name>" \
    --bootscript=path/to/bootscript.rcS
```

replacing anything in angle-brackets with the name of the directory or file, without the angle-brackets.

You can then attach to the simulation by, in a different terminal window, running:

```
$ ./util/term/m5term 3456
```

The full details of what the fs_bigLITTLE.py script supports can be gotten by running:

```
$ ./build/ARM/gem5.opt configs/example/arm/fs_bigLITTLE.py –help
```

An aside on FS simulations:

Note that FS simulations take a long time; like "1 hour to load the kernel" long time! There are ways to "fast-forward" a simulation and then resume the detailed simulation at the interesting point, but these are beyond the scope of this chapter.

Setting up your development environment

This is going to talk about getting started developing gem5.

gem5-style guidelines

When modifying any open source project, it is important to follow the project's style guidelines. Details on gem5 style can be found on the gem5 Coding Style page.

To help you conform to the style guidelines, gem5 includes a script which runs whenever you commit a changeset in git. This script should be automatically added to your .git/config file by SCons the first time you build gem5. Please do not ignore these warnings/errors. However, in the rare case where you are trying to commit a file that doesn't conform to the gem5 style guidelines (e.g., something from outside the gem5 source tree) you can use the git option --no-verify to skip running the style checker.

The key takeaways from the style guide are:

- Use 4 spaces, not tabs
- Sort the includes
- Use capitalized camel case for class names, camel case for member variables and functions, and snake case for local variables.
- Document your code

git branches

Most people developing with gem5 use the branch feature of git to track their changes. This makes it quite simple to commit your changes back to gem5. Additionally, using branches can make it easier to update gem5 with new changes that other people make while keeping your own changes separate. The Git book has a great chapter describing the details of how to use branches.

Creating a *very* simple SimObject

Note: gem5 has SimObject named SimpleObject. Implementing another SimpleObject SimObject will result in confusing compiler issues.

Almost all objects in gem5 inherit from the base SimObject type. SimObjects export the main interfaces to all objects in gem5. SimObjects are wrapped C++ objects that are accessible from the Python configuration scripts.

SimObjects can have many parameters, which are set via the Python configuration files. In addition to simple parameters like integers and floating point numbers, they can also have other SimObjects as parameters. This allows you to create complex system hierarchies, like real machines.

In this chapter, we will walk through creating a simple "HelloWorld" SimObject. The goal is to introduce you to how SimObjects are created and the required boilerplate code for all SimObjects. We will also create a simple Python configuration script which instantiates our SimObject.

In the next few chapters, we will take this simple SimObject and expand on it to include debugging support, dynamic events, and parameters.

Using git branches

It is common to use a new git branch for each new feature you add to gem5.

The first step when adding a new feature or modifying something in gem5, is to create a new branch to store your changes. Details on git branches can be found in the Git book.

```
git checkout -b hello-simobject
```

Step 1: Create a Python class for your new SimObject

Each SimObject has a Python class which is associated with it. This Python class describes the parameters of your SimObject that can be controlled from the Python configuration files. For our simple SimObject, we are just going to start out with no parameters. Thus, we simply need to declare a new class for our SimObject and set it's name and the C++ header that will define the C++ class for the SimObject.

We can create a file, HelloObject.py, in src/learning_gem5/part2. If you have cloned the gem5 repository you'll have the files mentioned in this tutorial completed under src/learning_gem5/part2 and configs/learning_gem5/part2. You can delete these or move them elsewhere to follow this tutorial.

```
from m5.params import *
from m5.SimObject import SimObject

class HelloObject(SimObject):
    type = 'HelloObject'
    cxx_header = "learning_gem5/part2/hello_object.hh"
    cxx_class = "gem5::HelloObject"
```

It is not required that the type be the same as the name of the class, but it is convention. The type is the C++ class that you are wrapping with this Python SimObject. Only in special circumstances should the type and the class name be different.

The cxx_header is the file that contains the declaration of the class used as the type parameter. Again, the convention is to use the name of the SimObject with all lowercase and underscores, but this is only convention. You can specify any header file here.

The cxx_class is an attribute specifying the newly created SimObject is declared within the gem5 namespace. Most SimObjects in the gem5 code base are declared within the gem5 namespace!

Step 2: Implement your SimObject in C++

Next, we need to create hello_object.hh and hello_object.cc in src/learning_gem5/part2/ directory which will implement the HelloObject.

We'll start with the header file for our C++ object. By convention, gem5 wraps all header files in #ifndef/#endif with the name of the file and the directory its in so there are no circular includes.

SimObjects should be declared within the gem5 namespace. Therefore, we declare our class within the namespace gem5 scope.

The only thing we need to do in the file is to declare our class. Since HelloObject is a SimObject, it must inherit from the C++ SimObject class. Most of the time, your SimObject's parent will be a subclass of SimObject, not SimObject itself.

The SimObject class specifies many virtual functions. However, none of these functions are pure virtual, so in the simplest case, there is no need to implement any functions except for the constructor.

The constructor for all SimObjects assumes it will take a parameter object. This parameter object is automatically created by the build system and is based on the Python class for the SimObject, like the one we created above. The name for this parameter type is generated automatically from the name of your object. For our "HelloObject" the parameter type's name is "HelloObjectParams".

The code required for our simple header file is listed below.

```
#ifndef __LEARNING_GEM5_HELLO_OBJECT_HH__
#define __LEARNING_GEM5_HELLO_OBJECT_HH__

#include "params/HelloObject.hh"
#include "sim/sim_object.hh"

namespace gem5
{

class HelloObject : public SimObject
{
  public:
    HelloObject(const HelloObjectParams &p);
};

} // namespace gem5

#endif // __LEARNING_GEM5_HELLO_OBJECT_HH__
```

Next, we need to implement *two* functions in the .cc file, not just one. The first function, is the constructor for the HelloObject. Here we simply pass the parameter object to the SimObject parent and print "Hello world!"

Normally, you would **never** use std::cout in gem5. Instead, you should use debug flags. In the next chapter, we will modify this to use debug flags instead. However, for now, we'll simply use std::cout because it is simple.

```
#include "learning_gem5/part2/hello_object.hh"

#include <iostream>

namespace gem5
```

```
{

HelloObject::HelloObject(const HelloObjectParams &params) :
    SimObject(params)
{
    std::cout << "Hello World! From a SimObject!" << std::endl;
}

} // namespace gem5
```

Note: If the constructor of your SimObject follows the following signature,

```
Foo(const FooParams &)
```

then a FooParams::create() method will be automatically defined. The purpose of the create() method is to call the SimObject constructor and return an instance of the SimObject. Most SimObject will follow this pattern; however, if your SimObject does not follow this pattern, the gem5 SimObject documetation provides more information about manually implementing the create() method.

Step 3: Register the SimObject and C++ file

In order for the C++ file to be compiled and the Python file to be parsed we need to tell the build system about these files. gem5 uses SCons as the build system, so you simply have to create a SConscript file in the directory with the code for the SimObject. If there is already a SConscript file for that directory, simply add the following declarations to that file.

This file is simply a normal Python file, so you can write any Python code you want in this file. Some of the scripting can become quite complicated. gem5 leverages this to automatically create code for SimObjects and to compile the domain-specific languages like SLICC and the ISA language.

In the SConscript file, there are a number of functions automatically defined after you import them. See the section on that…

To get your new SimObject to compile, you simply need to create a new file with the name "SConscript" in the src/learning_gem5/part2 directory. In this file, you have to declare the SimObject and the .cc file. Below is the required code.

```
Import('*')

SimObject('HelloObject.py', sim_objects=['HelloObject'])
Source('hello_object.cc')
```

Step 4: (Re)-build gem5

To compile and link your new files you simply need to recompile gem5. The below example assumes you are using the x86 ISA, but nothing in our object requires an ISA so, this will work with any of gem5's ISAs.

```
scons build/X86/gem5.opt
```

Step 5: Create the config scripts to use your new SimObject

Now that you have implemented a SimObject, and it has been compiled into gem5, you need to create or modify a Python config file run_hello.py in configs/learning_gem5/part2 to instantiate your object. Since your object is very simple a system object is not required! CPUs are not needed, or caches, or anything, except a Root object. All gem5 instances require a Root object.

Walking through creating a *very* simple configuration script, first, import m5 and all of the objects you have compiled.

```
import m5
from m5.objects import *
```

Next, you have to instantiate the Root object, as required by all gem5 instances.

```
root = Root(full_system = False)
```

Now, you can instantiate the HelloObject you created. All you need to do is call the Python "constructor". Later, we will look at how to specify parameters via the Python constructor. In addition to creating an instantiation of your object, you need to make sure that it is a child of the root object. Only SimObjects that are children of the Root object are instantiated in C++.

```
root.hello = HelloObject()
```

Finally, you need to call instantiate on the m5 module and actually run the simulation!

```
m5.instantiate()

print("Beginning simulation!")
exit_event = m5.simulate()
print('Exiting @ tick {} because {}'
    .format(m5.curTick(), exit_event.getCause()))
```

Remember to rebuild gem5 after modifying files in the src/ directory. The command line to run the config file is in the output below after 'command line:'. The output should look something like the following:

Note: If the code for the future section "Adding parameters to SimObjects and more events", (goodbye_object) is in your src/learning_gem5/part2 directory, run_hello.py will cause an error. If you delete those files or move them outside of the gem5 directory run_hello.py should give the output below.

```
    gem5 Simulator System.  http://gem5.org
    gem5 is copyrighted software; use the --copyright option for details.

    gem5 compiled May  4 2016 11:37:41
    gem5 started May  4 2016 11:44:28
    gem5 executing on mustardseed.cs.wisc.edu, pid 22480
    command line: build/X86/gem5.opt configs/learning_gem5/part2/run_hello.py

    Global frequency set at 1000000000000 ticks per second
```

```
   Hello World! From a SimObject!
   Beginning simulation!
   info: Entering event queue @ 0.  Starting simulation...
   Exiting @ tick 18446744073709551615 because simulate() limit reached
```

Congrats! You have written your first SimObject. In the next chapters, we will extend this SimObject and explore what you can do with SimObjects.

Debugging gem5

In the previous chapters we covered how to create a very simple SimObject. In this chapter, we will replace the simple print to stdout with gem5's debugging support.

gem5 provides support for printf-style tracing/debugging of your code via *debug flags*. These flags allow every component to have many debug-print statements, without all of them enabled at the same time. When running gem5, you can specify which debug flags to enable from the command line.

Using debug flags

For instance, when running the first simple.py script from simple-config-chapter, if you enable the DRAM debug flag, you get the following output. Note that this generates *a lot* of output to the console (about 7 MB).

```
build/X86/gem5.opt --debug-flags=DRAM configs/learning_gem5/part1/simple.py | head -n 50
gem5 Simulator System.  http://gem5.org
DRAM device capacity (gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled Jan  3 2017 16:03:38
gem5 started Jan  3 2017 16:09:53
gem5 executing on chinook, pid 19223
command line: build/X86/gem5.opt --debug-flags=DRAM configs/learning_gem5/part1/simple.py

Global frequency set at 1000000000000 ticks per second
      0: system.mem_ctrl: Memory capacity 536870912 (536870912) bytes
      0: system.mem_ctrl: Row buffer size 8192 bytes with 128 columns per row buffer
      0: system.remote_gdb.listener: listening for remote gdb #0 on port 7000
Beginning simulation!
info: Entering event queue @ 0.  Starting simulation...
      0: system.mem_ctrl: recvTimingReq: request ReadReq addr 400 size 8
      0: system.mem_ctrl: Read queue limit 32, current size 0, entries needed 1
      0: system.mem_ctrl: Address: 400 Rank 0 Bank 0 Row 0
      0: system.mem_ctrl: Read queue limit 32, current size 0, entries needed 1
      0: system.mem_ctrl: Adding to read queue
      0: system.mem_ctrl: Request scheduled immediately
      0: system.mem_ctrl: Single request, going to a free rank
      0: system.mem_ctrl: Timing access to addr 400, rank/bank/row 0 0 0
      0: system.mem_ctrl: Activate at tick 0
      0: system.mem_ctrl: Activate bank 0, rank 0 at tick 0, now got 1 active
      0: system.mem_ctrl: Access to 400, ready at 46250 bus busy until 46250.
  46250: system.mem_ctrl: processRespondEvent(): Some req has reached its readyTime
  46250: system.mem_ctrl: number of read entries for rank 0 is 0
  46250: system.mem_ctrl: Responding to Address 400..   46250: system.mem_ctrl: Done
  77000: system.mem_ctrl: recvTimingReq: request ReadReq addr 400 size 8
  77000: system.mem_ctrl: Read queue limit 32, current size 0, entries needed 1
  77000: system.mem_ctrl: Address: 400 Rank 0 Bank 0 Row 0
  77000: system.mem_ctrl: Read queue limit 32, current size 0, entries needed 1
  77000: system.mem_ctrl: Adding to read queue
  77000: system.mem_ctrl: Request scheduled immediately
  77000: system.mem_ctrl: Single request, going to a free rank
  77000: system.mem_ctrl: Timing access to addr 400, rank/bank/row 0 0 0
  77000: system.mem_ctrl: Access to 400, ready at 101750 bus busy until 101750.
 101750: system.mem_ctrl: processRespondEvent(): Some req has reached its readyTime
 101750: system.mem_ctrl: number of read entries for rank 0 is 0
 101750: system.mem_ctrl: Responding to Address 400..  101750: system.mem_ctrl: Done
 132000: system.mem_ctrl: recvTimingReq: request ReadReq addr 400 size 8
 132000: system.mem_ctrl: Read queue limit 32, current size 0, entries needed 1
 132000: system.mem_ctrl: Address: 400 Rank 0 Bank 0 Row 0
 132000: system.mem_ctrl: Read queue limit 32, current size 0, entries needed 1
 132000: system.mem_ctrl: Adding to read queue
 132000: system.mem_ctrl: Request scheduled immediately
 132000: system.mem_ctrl: Single request, going to a free rank
 132000: system.mem_ctrl: Timing access to addr 400, rank/bank/row 0 0 0
 132000: system.mem_ctrl: Access to 400, ready at 156750 bus busy until 156750.
 156750: system.mem_ctrl: processRespondEvent(): Some req has reached its readyTime
 156750: system.mem_ctrl: number of read entries for rank 0 is 0
```

Or, you may want to debug based on the exact instruction the CPU is executing. For this, the Exec debug flag may be useful. This debug flags shows details of how each instruction is executed by the simulated CPU.

```
build/X86/gem5.opt --debug-flags=Exec configs/learning_gem5/part1/simple.py | head -n 50
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled Jan  3 2017 16:03:38
gem5 started Jan  3 2017 16:11:47
gem5 executing on chinook, pid 19234
command line: build/X86/gem5.opt --debug-flags=Exec configs/learning_gem5/part1/simple.py

Global frequency set at 1000000000000 ticks per second
      0: system.remote_gdb.listener: listening for remote gdb #0 on port 7000
warn: ClockedObject: More than one power state change request encountered within the same
simulation tick
Beginning simulation!
info: Entering event queue @ 0.  Starting simulation...
  77000: system.cpu T0 : @_start    : xor   rbp, rbp
  77000: system.cpu T0 : @_start.0  :   XOR_R_R : xor   rbp, rbp, rbp : IntAlu :
D=0x0000000000000000
 132000: system.cpu T0 : @_start+3    : mov r9, rdx
 132000: system.cpu T0 : @_start+3.0  :   MOV_R_R : mov   r9, r9, rdx : IntAlu :
D=0x0000000000000000
 187000: system.cpu T0 : @_start+6    : pop rsi
 187000: system.cpu T0 : @_start+6.0  :   POP_R : ld   t1, SS:[rsp] : MemRead :
D=0x0000000000000001 A=0x7fffffffee30
 250000: system.cpu T0 : @_start+6.1  :   POP_R : addi   rsp, rsp, 0x8 : IntAlu :  D=0x00007fffffffee38
 250000: system.cpu T0 : @_start+6.2  :   POP_R : mov   rsi, rsi, t1 : IntAlu :  D=0x0000000000000001
 360000: system.cpu T0 : @_start+7    : mov rdx, rsp
 360000: system.cpu T0 : @_start+7.0  :   MOV_R_R : mov   rdx, rdx, rsp : IntAlu :
D=0x00007fffffffee38
 415000: system.cpu T0 : @_start+10    : and    rax, 0xfffffffffffffff0
 415000: system.cpu T0 : @_start+10.0  :   AND_R_I : limm   t1, 0xfffffffffffffff0 : IntAlu :
D=0xfffffffffffffff0
 415000: system.cpu T0 : @_start+10.1  :   AND_R_I : and   rsp, rsp, t1 : IntAlu :
D=0x0000000000000000
 470000: system.cpu T0 : @_start+14    : push   rax
 470000: system.cpu T0 : @_start+14.0  :   PUSH_R : st   rax, SS:[rsp + 0xfffffffffffffff8] : MemWrite :
D=0x0000000000000000 A=0x7fffffffee28
 491000: system.cpu T0 : @_start+14.1  :   PUSH_R : subi   rsp, rsp, 0x8 : IntAlu :
D=0x00007fffffffee28
 546000: system.cpu T0 : @_start+15    : push   rsp
 546000: system.cpu T0 : @_start+15.0  :   PUSH_R : st   rsp, SS:[rsp + 0xfffffffffffffff8] : MemWrite :
D=0x00007fffffffee28 A=0x7fffffffee20
 567000: system.cpu T0 : @_start+15.1  :   PUSH_R : subi   rsp, rsp, 0x8 : IntAlu :
D=0x00007fffffffee20
 622000: system.cpu T0 : @_start+16    : mov    r15, 0x40a060
 622000: system.cpu T0 : @_start+16.0  :   MOV_R_I : limm   r8, 0x40a060 : IntAlu :
D=0x000000000040a060
 732000: system.cpu T0 : @_start+23    : mov    rdi, 0x409ff0
 732000: system.cpu T0 : @_start+23.0  :   MOV_R_I : limm   rcx, 0x409ff0 : IntAlu :
D=0x0000000000409ff0
 842000: system.cpu T0 : @_start+30    : mov    rdi, 0x400274
 842000: system.cpu T0 : @_start+30.0  :   MOV_R_I : limm   rdi, 0x400274 : IntAlu :
D=0x0000000000400274
 952000: system.cpu T0 : @_start+37    : call   0x9846
 952000: system.cpu T0 : @_start+37.0  :   CALL_NEAR_I : limm   t1, 0x9846 : IntAlu :
D=0x0000000000009846
 952000: system.cpu T0 : @_start+37.1  :   CALL_NEAR_I : rdip   t7, %ctrl153,  : IntAlu :
D=0x00000000004001ba
 952000: system.cpu T0 : @_start+37.2  :   CALL_NEAR_I : st   t7, SS:[rsp + 0xfffffffffffffff8] :
MemWrite :  D=0x00000000004001ba A=0x7fffffffee18
 973000: system.cpu T0 : @_start+37.3  :   CALL_NEAR_I : subi   rsp, rsp, 0x8 : IntAlu :
D=0x00007fffffffee18
 973000: system.cpu T0 : @_start+37.4  :   CALL_NEAR_I : wrip   , t7, t1 : IntAlu :
1042000: system.cpu T0 : @__libc_start_main    : push   r15
1042000: system.cpu T0 : @__libc_start_main.0  :   PUSH_R : st   r15, SS:[rsp + 0xfffffffffffffff8] :
MemWrite :  D=0x0000000000000000 A=0x7fffffffee10
1063000: system.cpu T0 : @__libc_start_main.1  :   PUSH_R : subi   rsp, rsp, 0x8 : IntAlu :
D=0x00007fffffffee10
```

```
1118000: system.cpu T0 : @__libc_start_main+2    : movsxd   rax, rsi
1118000: system.cpu T0 : @__libc_start_main+2.0  :   MOVSXD_R_R : sexti   rax, rsi, 0x1f : IntAlu :
D=0x0000000000000001
1173000: system.cpu T0 : @__libc_start_main+5    : mov  r15, r9
1173000: system.cpu T0 : @__libc_start_main+5.0  :   MOV_R_R : mov   r15, r15, r9 : IntAlu :
D=0x0000000000000000
1228000: system.cpu T0 : @__libc_start_main+8    : push r14
```

In fact, the Exec flag is actually an agglomeration of multiple debug flags. You can see this, and all of the available debug flags, by running gem5 with the --debug-help parameter.

```
build/X86/gem5.opt --debug-help
Base Flags:
    Activity: None
    AddrRanges: None
    Annotate: State machine annotation debugging
    AnnotateQ: State machine annotation queue debugging
    AnnotateVerbose: Dump all state machine annotation details
    BaseXBar: None
    Branch: None
    Bridge: None
    CCRegs: None
    CMOS: Accesses to CMOS devices
    Cache: None
    CacheComp: None
    CachePort: None
    CacheRepl: None
    CacheTags: None
    CacheVerbose: None
    Checker: None
    Checkpoint: None
    ClockDomain: None
...
Compound Flags:
    All: Controls all debug flags. It should not be used within C++ code.
        All Base Flags
    AnnotateAll: All Annotation flags
        Annotate, AnnotateQ, AnnotateVerbose
    CacheAll: None
        Cache, CacheComp, CachePort, CacheRepl, CacheVerbose, HWPrefetch
    DiskImageAll: None
        DiskImageRead, DiskImageWrite
...
XBar: None
    BaseXBar, CoherentXBar, NoncoherentXBar, SnoopFilter
```

Adding a new debug flag

In the previous chapters, we used a simple std::cout to print from our SimObject. While it is possible to use the normal C/C++ I/O in gem5, it is highly discouraged. So, we are now going to replace this and use gem5's debugging facilities instead.

When creating a new debug flag, we first have to declare it in a SConscript file. Add the following to the SConscript file in the directory with your hello object code (src/learning_gem5/).

```
DebugFlag('HelloExample')
```

This declares a debug flag of "HelloExample". Now, we can use this in debug statements in our SimObject.

By declaring the flag in the SConscript file, a debug header is automatically generated that allows us to use the debug flag. The header file is in the debug directory and has the same name (and capitalization) as what we declare in the SConscript file. Therefore, we need to include the automatically generated header file in any files where we plan to use the debug flag.

In the hello_object.cc file, we need to include the header file.

```
#include "base/trace.hh"
#include "debug/HelloExample.hh"
```

Now that we have included the necessary header file, let's replace the std::cout call with a debug statement like so.

```
DPRINTF(HelloExample, "Created the hello object\n");
```

DPRINTF is a C++ macro. The first parameter is a *debug flag* that has been declared in a SConscript file. We can use the flag HelloExample since we declared it in the src/learning_gem5/SConscript file. The rest of the arguments are variable and can be anything you would pass to a printf statement.

Now, if you recompile gem5 and run it with the "HelloExample" debug flag, you get the following result.

```
build/X86/gem5.opt --debug-flags=HelloExample configs/learning_gem5/part2/run_hello.py
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled Jan  4 2017 09:40:10
gem5 started Jan  4 2017 09:41:01
gem5 executing on chinook, pid 29078
command line: build/X86/gem5.opt --debug-flags=HelloExample
configs/learning_gem5/part2/run_hello.py

Global frequency set at 1000000000000 ticks per second
      0: hello: Created the hello object
Beginning simulation!
info: Entering event queue @ 0.  Starting simulation...
```

```
Exiting @ tick 18446744073709551615 because simulate() limit reached
```

You can find the updated SConcript file here and the updated hello object code here.

Debug output
For each dynamic DPRINTF execution, three things are printed to stdout. First, the current tick when the DPRINTF is executed. Second, the *name of the SimObject* that called DPRINTF. This name is usually the Python variable name from the Python config file. However, the name is whatever the SimObject name() function returns. Finally, you see whatever format string you passed to the DPRINTF function.

You can control where the debug output goes with the --debug-file parameter. By default, all of the debugging output is printed to stdout. However, you can redirect the output to any file. The file is stored relative to the main gem5 output directory, not the current working directory.

Using functions other than DPRINTF
DPRINTF is the most commonly used debugging function in gem5. However, gem5 provides a number of other functions that are useful in specific circumstances.

These functions are like the previous functions :cppDDUMP, :cppDPRINTF, and :cppDPRINTFR except they do not take a flag as a parameter. Therefore, these statements will *always* print whenever debugging is enabled.

All of these functions are only enabled if you compile gem5 in "opt" or "debug" mode. All other modes use empty placeholder macros for the above functions. Therefore, if you want to use debug flags, you must use either "gem5.opt" or "gem5.debug".

Introduction to Ruby
Ruby comes from the multifacet GEMS project. Ruby provides a detailed cache memory and cache coherence models as well as a detailed network model (Garnet).

Ruby is flexible. It can model many different kinds of coherence implementations, including broadcast, directory, token, region-based coherence, and is simple to extend to new coherence models.

Ruby is a mostly drop-in replacement for the classic memory system. There are interfaces between the classic gem5 MemObjects and Ruby, but for the most part, the classic caches and Ruby are not compatible.

In this part of the book, we will first go through creating an example protocol from the protocol description to debugging and running the protocol.

Before diving into a protocol, we will first talk about some of the architecture of Ruby. The most important structure in Ruby is the controller, or state machine. Controllers are implemented by writing a SLICC state machine file.

SLICC is a domain-specific language (Specification Language including Cache Coherence) for specifying coherence protocols. SLICC files end in ".sm" because they are *state machine* files. Each file describes states, transitions from a begin to an end state on some event, and actions to take during the transition.

Each coherence protocol is made up of multiple SLICC state machine files. These files are compiled with the SLICC compiler which is written in Python and part of the gem5 source. The SLICC compiler takes the state machine files and output a set of C++ files that are compiled with all of gem5's other files. These files include the SimObject declaration file as well as implementation files for SimObjects and other C++ objects.

Currently, gem5 supports compiling only a single coherence protocol at a time. For instance, you can compile MI_example into gem5 (the default, poor performance, protocol), or you can use MESI_Two_Level. But, to use MESI_Two_Level, you have to recompile gem5 so the SLICC compiler can generate the correct files for the protocol. We discuss this further in the compilation section <MSI-building-section>

Now, let's dive into implementing our first coherence protocol!

MSI example cache protocol

Before we implement a cache coherence protocol, it is important to have a solid understanding of cache coherence. This section leans heavily on the great book *A Primer on Memory Consistency and Cache Coherence* by Daniel J. Sorin, Mark D. Hill, and David A. Wood which was published as part of the Synthesis Lectures on Computer Architecture in 2011 (DOI:10.2200/S00346ED1V01Y201104CAC016). If you are unfamiliar with cache coherence, I strongly advise reading that book before continuing.

In this chapter, we will be implementing an MSI protocol. (An MSI protocol has three stable states, modified with read-write permission, shared with read-only permission, and invalid with no permissions.) We will implement this as a three-hop directory protocol (i.e., caches can send data directly to other caches without going through the directory). Details for the protocol can be found in Section 8.2 of *A Primer on Memory Consistency and Cache Coherence* (pages 141-149). It will be helpful to print out Section 8.2 to reference as you are implementing the protocol.

You can download the Second Edition via this link.

First steps to writing a protocol

Let's start by creating a new directory for our protocol at src/learning_gem5/MSI_protocol. In this directory, like in all gem5 source directories, we need to create a file for SCons to know what to compile. However, this time, instead of creating a SConscript file, we are going to create a SConsopts file. (The SConsopts files are processed before the SConscript files and we need to run the SLICC compiler before SCons executes.)

We need to create a SConsopts file with the following:

```
Import('*')

main.Append(ALL_PROTOCOLS=['MSI'])

main.Append(PROTOCOL_DIRS=[Dir('.')])
```

We do two things in this file. First, we register the name of our protocol ('MSI'). Since we have named our protocol MSI, SCons will assume that there is a file named MSI.slicc which specifies all of the state machine files and auxiliary files. We will create that file after writing all of our state machine files.

Second, the SConsopts files tells the SCons to look in the current directory for files to pass to the SLICC compiler.

You can download the SConsopts file here.

Writing a state machine file

The next step, and most of the effort in writing a protocol, is to create the state machine files. State machine files generally follow the outline:

Parameters
: These are the parameters for the SimObject that will be generated from the SLICC code.

Declaring required structures and functions
: This section declares the states, events, and many other required structures for the state machine.

In port code blocks
: Contain code that looks at incoming messages from the (in_port) message buffers and determines what events to trigger.

Actions
: These are simple one-effect code blocks (e.g., send a message) that are executed when going through a transition.

Transitions
: Specify actions to execute given a starting state and an event and the final state. This is the meat of the state machine definition.

Over the next few sections we will go over how to write each of these components of the protocol.