Machine Problem 1: GEM5

ECE 511: Computer Architecture (Spring 2025)

In this machine problem (MP), you will explore the differences between CISC and RISC ISAs. One purpose of this MP is to introduce the gem5 simulator to you, so that you can choose to develop your final project based on gem5. This document guides you to set up the gem5 simulator and run a subset of the Splash-3 benchmark suite on the simulator. You will compare the x86 ISA and the RISC-V ISA, as well as analyze whether and how the choice of ISA will impact the system performance. This document begins with an introduction to the RISC vs. CISC debate, helps you set up the gem5 simulator, details the experiments you need to conduct and the results you need to include in the report, and the last section lists all the submission requirements.

1. Background

You probably have heard of the myth that CISC ISAs like x86 are faster and consume more power than RISC ISAs like ARM and RISC-V. In fact, the RISC vs. CISC wars have a long history back to the 1980s, and we have seen more or less the x86 ISA dominate desktops and servers, while the ARM ISA is widely deployed in tablets and smartphones. In this MP, we will compare x86 and RISC-V in the gem5 simulator and answer the question: will the choice of a specific ISA impact the performance of a processor?

2. Building Gem5 Simulator

Before you begin, it is recommended that you are working with a Ubuntu 24.04 workstation or virtual machine with at least 50GB free disk space and 8GB memory for ECE 511. The following steps are for setup on Ubuntu 24.04, please follow the instructions on this guide if you would like to use a different distribution.

1. Install the dependencies

```
sudo apt install build-essential scons python3-dev git pre-commit zlib1g zlib1g-dev \
    libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-dev \
    libboost-all-dev  libhdf5-serial-dev python3-pydot python3-venv python3-tk mypy \
    m4 libcapstone-dev libpng-dev libelf-dev pkg-config wget cmake doxygen
```

2. Clone the Gem5 repo

```
git clone https://github.com/gem5/gem5
```

3. Install graphviz

```
sudo apt install graphviz
```

4. **Building Gem5**
   Now, we are ready to compile the gem5 simulator. As we need to experiment with both x86 and RISC-V, we need to compile gem5 for both ISAs. At the root of the gem5 repo, run **scons build/{ISA}/gem5.{variant} -j {cpus}** to build gem5. For example for our purposes, we can compile with

   ```
   scons build/X86/gem5.fast -j `nproc`
   ```

   and

   ```
   scons build/RISCV/gem5.fast -j `nproc`
   ```

5. **Verifying the build**
   As the last step, you may want to verify that gem5 works properly with, these execute a statically linked hello world program.

   ```
   build/X86/gem5.fast configs/learning_gem5/part1/simple.py
   ```

   and

   ```
   build/RISCV/gem5.fast configs/learning_gem5/part1/simple-riscv.py
   ```

3. Comparing the x86 ISA and the RISC-V ISA

3.1. Objectives

In this section, you will configure gem5 to simulate different processors and ISAs, collect the statistics, and analyze the results. The section is divided into two parts: simulation with weak processors and with powerful processors. In each part, you will compare the performance of the two ISAs and analyze the results. This section also provides you with some basic guidance for using gem5. However, you should refer to the gem5 website or search the Internet to figure out the concrete steps that are not documented here. If you are still stuck, feel free to ask on Campuswire or come to office hours.

3.2. The Benchmark Programs

To ease your effort, we provide you with all the necessary binaries you need to run in gem5. There should be a total of 6 binaries, including three benchmarks *FFT*, *CHOLESKY*, and *FMM*, each compiled into both x86 and RISC-V. FFT computes the Fast Fourier Transform of a one-dimensional complex sequence. CHOLESKY performs blocked Cholesky factorization on a sparse matrix. FMM stands for Greengard–Rokhlin Fast Multipole Method, and it provides a way to calculate the two-dimensional stream

function and velocity field at numerous target points presented in a large system of vortices. These binaries have the
following command line interface:

```
./FFT -p# -m16
./CHOLESKY -p# matrixDescriptionFile.O
./FMM -p#
```

where # is the placeholder for the number of threads. Use -h to get a complete list of options for each benchmark.

3.3. Simulation with Weak Processors

At this stage, you are ready to run your first experiment with gem5! To ease your effort, this MP only requires the System Emulation (SE) mode, which emulates all the syscalls and avoids simulating the whole OS. As a comparison, the Full System (FS) mode supports booting an entire OS in gem5 at the expense of slow simulation speed and complex setup.

Take a look at the script *configs/deprecated/example/se.py* and the corresponding tutorial.This script provides almost everything you need for this part of the MP. Assuming you have already placed the benchmark program FFT at *path/to/FFT*, for example, you can use *se.py* in the following way:

```
### This is just an example. Please change the parameters for this MP.
build/X86/gem5.fast --outdir=m5out/ configs/deprecated/example/se.py \
        --cpu-type=X86O3CPU \
        --cpu-clock=4GHz \
        --cacheline_size=64 \
        --caches \
        --l1i_size=8kB \
        --l1d_size=8kB \
        --l2cache \
        --l2_size=4MB \
        --l2_assoc=16 \
        --num-cpus=4 \
        --cmd=path/to/FFT --options="-p4 -m16"
```

The *--outdir=[xxx]* option specifies where the statistics and simulation results will be stored. The *--cmd=[xxx]* option specifies the binary to execute, and *--options=[xxx]* gives the command line arguments for the binary. A complete list of usage can be retrieved with the *--help* option.

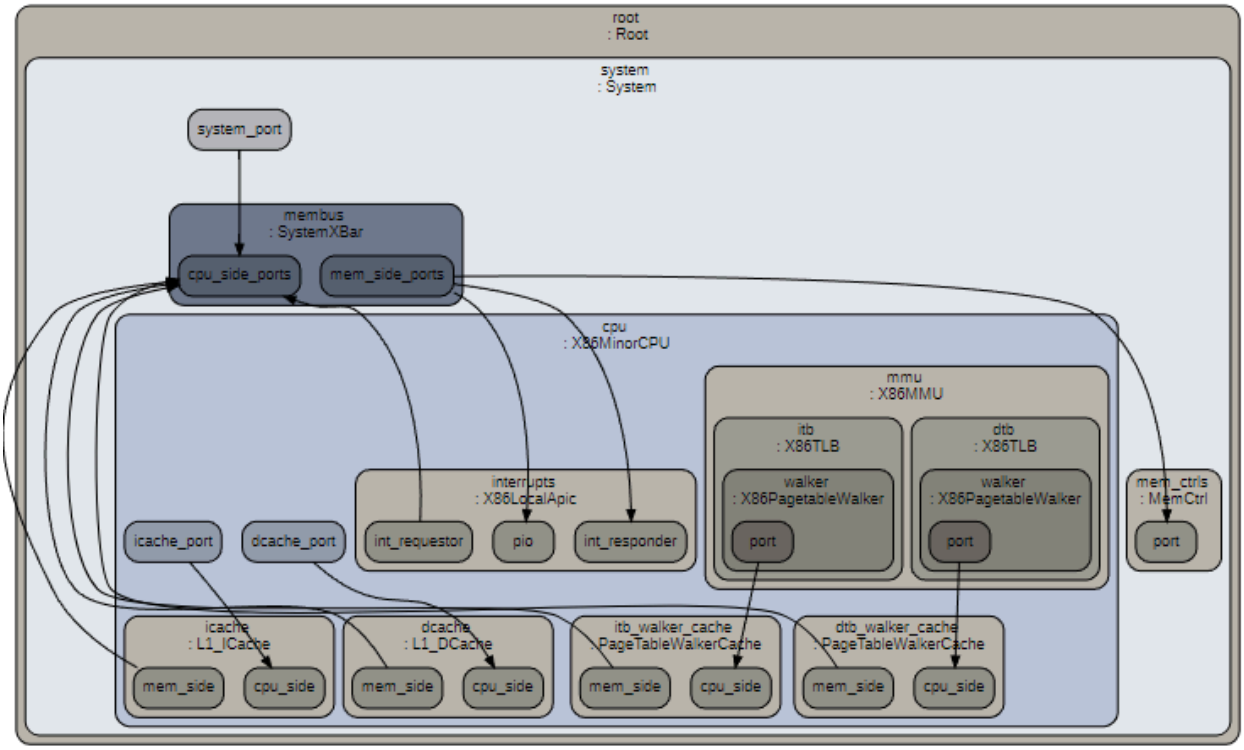

Figure 1: Expected configuration for the weak x86 processor. For RISC-V, the interrupts object will not be present.

3.3.1. Required Experiments

You are expected to setup the weak processors with the following parameters:

single-core, 1GHz, 64-byte cache line, 16KB 4-way L1 instruction cache, 32KB 8-way L1 data cache, no L2 cache

Use X86MinorCPU and RiscvMinorCPU for the --cpu-type option. MinorCPU models a 4-stage in-order pipeline, which we will use as a representative of weak cores.

Use the following arguments for the benchmark programs:

```
./FFT -p1 -m16
./CHOLESKY -p1 /path/to/CHOLESKY_INPUT/tk15.O
./FMM -p1
```

With the above settings, you are required to configure a weak x86 processor and a weak RISC-V processor, and run all three benchmarks on both processors. You should also specify the commands you use to run each experiment in your MP report.

3.3.2. Collecting Simulation Outputs

The simulation outputs are available at the directory specified by --outdir (by default this will be m5out/). You may want to verify that your config.dot.pdf represents the same configuration as that in Figure 1.

The files config.ini and config.json list the parameters for all simulation objects. The simulation statistics are in stats.txt. You are expected to find the following numbers and include them in the MP report:

1. Benchmark execution time in seconds
2. Instructions per cycle (IPC)
3. Number of instructions (use simInsts)
4. Number of μOps (use simOps)
5. Cache hit ratio for L1I and L1D
6. Total DRAM bandwidth

As an example, the simSeconds line in stats.txt is the number of simulated seconds, which in this case represents the benchmark execution time. When you report the actual numbers, please also indicate the name of the statistic where this number comes from (for example, the statistic name for benchmark execution time is simSeconds).

3.4. Simulation with Powerful Processors

For our powerful processors, we will be adding a shared L3 Cache. There are at least two approaches to set up the new configuration. The first approach is to write your own Python script and connect all the components by yourself. You may want to start with the gem5 tutorial to get a better idea on how to write your own simulation script in gem5. You may also refer to other scripts in the *configs/example/* and *configs/common/* directories at your convenience. Note that this approach may be cumbersome and require more coding than the second approach.

The second approach is to leverage the *se.py* script to configure the processor. However, this script by default does not properly generate the unified L3 cache. Therefore, you will need to modify some codes under the *configs/* folder.

You can change the script in any way you like, but it is suggested to start with the *config_cache()* function in *configs/common/CacheConfig.py*. This function is invoked by the *se.py* script, and is responsible for instantiating the cache objects and connecting them into the system.

As a quick guide, inside the config cache() function, you can initialize the L3 cache instance by writing

```
system.l3 = L3Cache(...)
```

Then, you can initialize an *L2XBar* instance as the bus between the L3 cache and the upper-level memory hierarchy

```
system.tollcbus = L2XBar(...)
```

To connect the L3 cache with the rest of the system, take a look at how this is done for the L2 cache:

```
system.l2.cpu_side = system.tol2bus.mem_side_ports
system.l2.mem_side = system.membus.cpu_side_ports
```

It should be very similar for connecting the L3 cache. Another useful function to look at is *addPrivateSplitL1Caches()*, which is invoked as:

```
system.cpu[i].addPrivateSplitL1Caches(icache, dcache, iwalkcache,
dwalkcache)
```

This function is defined in the *BaseCPU* class in *src/cpu/BaseCPU.py*, and it adds the split L1I/D caches for the CPU instance. If you take a look at the *BaseCPU* class, you will find another helpful function *addTwoLevelCacheHierarchy()*, which adds the L1 caches and the private L2 cache for this CPU core. The *connectAllPorts()* function is also invoked when setting up the L2 cache, and you may want to adapt it for the L3 cache as well.

You may also want to extend the *configs/common/Options.py* script so that you can change the parameters more conveniently from the command line. For example, in the function *addNoISAOptions()*, the statement:

```
parser.add_argument("--l2cache", action="store_true")
```

enables you to use "*--l2cache*" when you call the se.py script. The "*store_true*" option means that if *--l2cache* is not specified when you call *se.py*, the value of *l2cache* is by default False. In se.py, the statement:

```
args = parser.parse_args()
```

gets all the command line arguments, and *args.l2cache* indicates whether --l2cache is specified

Note that the two approaches listed above are only recommendations. You are also free to choose other methods to accomplish the requirements in this part. For a coherent result with the previous part, please use the caches in *configs/common/Caches.py*. For the L3 cache, please add the following class to *Caches.py*:

```
class L3Cache(Cache):
        assoc = 16
        tag_latency = 20
        data_latency = 20
        response_latency = 20
        mshrs = 512
        tgts_per_mshr = 20
        write_buffers = 256
```

Note that the above code only specifies the default parameters for *L3Cache*. You will need to use *L3Cache* in your code and override the associativity and capacity by yourself.

After you are confident with your modifications, you should generate a patch for submission. Add and commit your changes using git. Then, run git diff > mp1.patch. You are required to submit the mp1.patch file with your report.

3.4.1. Required Experiments

You are expected to setup the powerful processors with the following parameters:

dual-core, 2.8GHz, 64-byte cache line, 32KB 8-way L1 instruction cache, 64KB 8-way L1 data cache, per-core 2MB 16-way L2 cache, unified 6MB 24-way L3 cache

Use *X86O3CPU* and *RiscvO3CPU* for the CPU core type. *O3CPU* models an out-of-order pipeline, which we use to represent powerful cores.

Run the benchmark programs with the *-p2* flag, and keep other flags the same as the weak processors. With the above settings, you are required to run all three benchmarks on the powerful x86 processor and the powerful RISC-V processor.

3.4.2. Collecting Simulation Outputs

1. Benchmark execution time in seconds
2. Instructions per cycle (IPC)
3. Cache hit ratio for L1I, L1D, L2, and L3
4. total DRAM bandwidth

For IPC and cache hit ratio, you may report the average of both cores. In addition, please also specify how you run the simulations in your MP report (e.g., which scripts or commands you use).

4. Submission Guideline

You are expected to turn in the following deliverables:

1. The MP report, report.pdf. Please include your name and Net ID in the report. Your report should include at least **two parts**, one for the weak processors (45 points) and one for the powerful processors (45 points).

   For **each part**, please include the following:
   a. (30 points) Required simulation output statistics for each experiment. There are a total of 6 experiments in each part. Please use one or more tables to report the numbers.

   b. (5 points) How you run each experiment. Specify the command you use (if you use the se.py script), or how you use your own script (if you choose to write your own). If you write your own script files, please include them in the patch file.

   c. Answer the following questions by analyzing the results you get from the experiments:
      i. (5 points) Which ISA has better performance for each benchmark program, in terms of execution time and IPC? Is the performance difference significant enough for us to conclude that one ISA outperforms the other?
      ii. (5 points) Which ISA has a better L1I cache hit ratio? Does the choice of ISA have a significant impact on L1D, L2, and L3 cache hit ratio, respectively?

Please also answer the following questions in your report based on your results from both parts:

1. (5 points) It may be expected that since CISC ISAs have higher code density (i.e., less assembly instructions for the same program) than RISC ISAs, CISC may have better performance by saving memory bandwidth. Is this really the case, based on your experiments? Why or why not?

2. 5 points) Compare and contrast your findings in Part I and Part II. Based on your experiments, can you conclude which one has more impact on

processor performance: the choice of ISA, or the microarchitectural design choices?

2. Gem5 simulation outputs for each experiment, including stats.txt, config.dot.pdf, config.ini, and config.json. **No points will be given if you put your simulation results in the report without submitting these files.**

3. The patch file, mp1.patch from the powerful processors part. This is used so that we know what changes you make to gem5. You may still get some partial credits in case your code doesn't work or you fail to report the simulation results. **No points will be given if this file is not submitted**.

You will be submitting your work on Canvas. Please submit a zip file organized in the following manner:

```
<your_netid>_mp1.zip
        |---> report.pdf
        |---> mp1.patch
        |---> m5out/
                |---> x86/
                        |---> weak/
                                |---> FFT/
                                        |---> stats.txt
                                        |---> config.dot.pdf
                                        |---> config.ini
                                        |---> config.json
                                |---> CHOLESKY/
                                        |---> ... (same structure as in FFT/)
                                |---> FMM/
                                        |---> ... (same structure as in FFT/)
                        |---> powerful/
                                |---> ... (same structure as in weak/)
                |---> riscv/
                        |---> ... (same structure as in x86/)
```

References:

- gem5 Documentation. https://www.gem5.org/documentation

- Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In 2016 *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111, 2016.