Gem5 Lab 1: Gem5 and Superscalar OOO Experiments

Due: This lab is to be completed in Lab Section during your time slot. After you are done, submit everything to github.

Gem5 is a system simulator that models CPUs at the micro-architecture level and all associated structures such as caches, memory and interconnect buses. It implements many architectural features that will be studied this term. The purpose of this assignment is to help you get acquainted with gem5 and with the methodology for running simulations to estimate the performance of machines.

About gem5

Gem5 allows the simulation of a wide range of CPUs, both functionally (correctness only) and for timing (correctness and efficiency). It supports multiple ISAs, including x86-64 and ARM. For this course, gem5 has been pre-compiled as Linux virtual machine disk image. The pre-compiled gem5 works under Ubuntu 16.04. You can use Oracle Virtual Box for running the Linux disk image. Download Virtual Box from the Oracle website. It supports Windows, MacOS, and Linux.

Cloud Access

We created a remote lab environment using Google cloud. You can connect to the cloud Linux server (**2440.cs.uh.edu**) with gem5 pre-installed. You can use any ssh client connecting to the server.

Windows ssh client
https://mobaxterm.mobatek.net/download.html
https://www.putty.org/
Mac OS
Mac OS has a built-in SSH client.
Android
https://android.md/2019/05/10/5-best-android-ssh-clients/

TA will provide instruction related to your accounts and login information. Each student has a home folder with 2GB space.

In the VirtualBox disk image that we provide, gem5 is installed in /home/gem5/. The default use account is gem5. In the cloud server, gem5 is installed under /opt/gem5.

You need to download the benchmark applications (compressed as tar.z file) and extract the files into your home folder (/opt/benchmark-apps.tar.z). To use tar command,

```
tar zxvf benchmark-apps.tar.z
```

The Assignment

Gem5 can simulate CPUs with different configurations (e.g. number of cores, pipeline complexity, cache size...) based on configuration scripts. We provide you with pre-installed gem5, and binary benchmarks.

In this Lab Section, you can experiment different settings of processor configuration, analyze the results of your simulation and answer questions.

Benchmarks

There are a total of five benchmarks which we will simulate with:

401.bzip2
429.mcf
456.hmmer
458.sjeng
470.lbm

In the VirtualBox disk image, the benchmarks are pre-installed under "/home/gem5/src/".

Within each benchmark directory, you can find a "**src**" directory that contains the source code and an executable binary, and a "**data**" directory that contains any necessary input files/arguments. Within the "**src**" directory, the executable file named "**benchmark**" is the program that you need to simulate with. You can also open the script "**run.sh**" to understand how to run the benchmark.

Within each folder, you can also find the script "**runGem5.sh**". It is a sample script that runs gem5 on the benchmark**.** You can use this as an example how to specify the command line to run a benchmark program on gem5.

As some of the benchmark programs can take considerably long time (e.g. bzip2 can take 20 hours, and sjeng can take 10 hours), you can define a max number of instructions to be executed, e.g. a maximum of 5*10^8 (500000000) instructions. The way to specify this constraint is discussed in the following section.

If you want to build gem5 from scratch, you can follow the instruction on how to get, compile and run gem5. For your own understanding, and run the provided benchmarks. You should also read through the script source code because you will need to run the same benchmark from your custom script.

Use the provided script (*2wayo3.py*) under /home/gem5/bin (or /opt/gem5/bin/) as the CPU model:

- An Out-of-Order CPU (O3CPU) capable of **fetching, decoding, issuing, and dispatching 2 instructions per cycle**

For your convenience, you can use "runGem5_O3CPU.sh" (can be found under each benchmark application folder). Inside the shell script, the command line below is used to launch gem5.

```
/bin/gem5.opt -d ./m5out $GEM5_CONFIG/2wayo3.py -c $BENCHMARK -o "$ARGUMENT" -I 100000000 --cpu-type=DerivO3CPU --caches --l2cache --l1d_size=128kB --l1i_size=128kB --l2_size=8MB --l1d_assoc=2 --l1i_assoc=2 --l2_assoc=1 --cacheline_size=64
```

--cpu-type=DerivO3CPU
--l1d_size=128kB (L1 data cache size)
--l1i_size=128kB (L1 instruction cache size)
--l2_size=8MB (L2 cache size)
--l1d_assoc=2 (L1 data cache associativity, 2)
--l1i_assoc=2 (L1 instruction cache associativity, 2)
--l2_assoc=1 (L2 cache associativity, direct-mapped)
--cacheline_size=64 (cache line size)

The CPU above simulated has the following functional units:

- *Separate 128KB, two-way set-associative L1 caches for instructions and data with 64-byte cache lines*
- *an 8MB, direct-mapped L2 unified cache with 64-byte cache lines*

You can run a simulation of the **bzip2** benchmark using the default setting. The results will be saved under folder **m5out**,

Note that you cannot modify file under /opt/gem5/bin. To change **2wayo3.py**, please make a copy of 2wayo3.to your own home folder (use cp Linux command). You can edit **2wayo3.py** using vim or nano – command line editor.

Please study runGem5_O3CPU.sh. If you use 2wayo3.py copied to your home folder, you need to modify *$GEM5_CONFIG/2wayo3.py so it points to where the copied 2wayo3.py locates.*

Modify the cpu model configuration to simulate a machine with the following setting:

cpu.decodeWidth = 2
cpu.fetchWidth = 2
cpu.issueWidth = 2
cpu.dispatchWidth = 2
cpu.commitWidth=2
cpu.numROBEntries=64
cpu.numIQEntries=32
cpu.fetchQueueSize=32

Run a simulation of the bzip2 benchmark using the provided 2way Superscalar CPU model scripts (runGem5_O3CPU.sh and 2wayo3.py). The simulation should run for 100 million instructions.

Answer the following questions as lab report:

For these questions, extract and label the values for the following metrics on your ***m5out/stats.txt*** (copy each line from **stats.txt** entirely):

Q1: The total number of instructions committed. This total includes all the instructions that have completed all the phases of instruction execution, including writing their results to the register file.

Why is this number not equal to the number of instructions executed as reported by the stats? Is this latter number the same as the one set-up in options.maxinsts (10^8)?

Numberofinstructionscommitted=100000000

Q2: What is the average CPI (cycles per instruction)?

system.cpu.cpi1.213822#cpi:CyclesperInstruction

Q3: The total number of committed branches and their frequency with respect to the number of committed instructions.

system.cpu.commit.branches5990529#Numberofbranchescommitted.Frequency=totalno.ofcommittedbranches/totalno.ofcommittedinstructions=5990529/100000000=0.05990529

Q4: Average number of instructions fetched per cycle.

system.cpu.fetch.rateDist:mean1.291364#Numberofinstructionsfetchedeachcycle

Q5: Average number of instructions issued per cycle.

system.cpu.iq.issued_per_cycle:mean1.203765#Numberofinstsissuedeachcycle

Q6: Average number of instructions committed per cycle.

system.cpu.commit.committed_per_cycle:mean1.110137#Numberofinstsissuedeachcycle

Use the provided script (*2wayo3.py*) under /home/gem5/bin as reference, and modify the CPU model to the parameters below:

- An Out-of-Order CPU (O3CPU) capable of **fetching, decoding, issuing, and dispatching 4 instructions per cycle**

- *Separate **64KB, four-way** set-associative L1 caches for instructions and data with 64-byte cache lines*
- ***an 8MB, direct-mapped L2 unified cache with 64-byte cache lines***

Note that you cannot modify file under **/opt/gem5/bin**. Please make a copy of 2wayo3.to your own home folder (use cp Linux command). Change its name to **6wayo3.py**.

After a copy is created, you can edit **6wayo3.py** under your home folder using vim or nano. Modify the Python script (*6wayo3.py*) so it will simulate a CPU model with the following parameters

cpu.decodeWidth = 6
cpu.fetchWidth = 6
cpu.issueWidth = 4
cpu.dispatchWidth = 8
cpu.commitWidth= 4
cpu.numROBEntries= 192
cpu.numIQEntries= 64
cpu.fetchQueueSize= 56

Please study runGem5_O3CPU.sh how to modify simulation settings of caches.

--l1d_size=64kB (L1 data cache size)
--l1i_size=64kB (L1 instruction cache size)
--l2_size=8MB (L2 cache size)
--l1d_assoc=4 (L1 data cache associativity, 2)
--l1i_assoc=4 (L1 instruction cache associativity, 2)
--l2_assoc=1 (L2 cache associativity, direct-mapped)
--cacheline_size=64 (cache line size)

Run a simulation of the bzip2 benchmark using your modified CPU model scripts (runGem5_O3CPU.sh and 6wayo3.py). The simulation should run for 100 million instructions.

Q7: What is the average CPI (cycles per instruction)?

system.cpu.cpi1.054944#cpi:CyclesperInstruction

Q8: Average instructions fetched per cycle.

system.cpu.fetch.rateDist::mean1.963522#Numberofinstructionsfetchedeachcycle

Q9: Average instructions issued per cycle.

system.cpu.iq.issued_per_cycle:mean1.574188#Numberofinstsissuedeachcycle

Q10: Average instructions committed per cycle.

system.cpu.commit.committed_per_cycle:mean1.326443#Numberofinstsissuedeachcycle

For benchmark bzip2, write down results using the table below.

bzip2 experiment results	**2way CPU model**	**6way CPU model**
Average CPI	1.213822	1.054944
Average instructions fetched per cycle	1.291364	1.963522
Average instructions issued per cycle	1.203765	1.574188
Average instructions committed per cycle	1.110137	1.326443

Repeat the same set of experiments for mcf benchmark. The simulation should run for 100 million instructions.

Write down results using the table below.

mcf experiment results	**2way CPU model**	**6way CPU model**
Average CPI	8.242796	7.176579
Average instructions fetched per cycle	0.705640	1.199819
Average instructions issued per cycle	0.571508	0.788616
Average instructions committed per cycle	0.496613	0.603479

Submit

Your submission should consist of the following:

- The statistics output files generated by the simulator (**stats.txt)**

- c**onfig.ini** under m5out

- Answers for the questions: **gem5_lab1_report.pdf**

Note that you must submit stats.txt and config.ini files for all your experiments. To make TA's grading job easier, label the files that you have submitted using the table below:

	Name of stats.txt	Name of config.ini
bzip2 2way CPU model	stats_bzip22way.txt	config_bzip22way.ini
bzip2 6way CPU model	stats_bzip26way.txt	config_bzip26way.ini
mcf 2way CPU model	stats_mf_2way.txt	config_mcf_2way.ini
mcf 6way CPU model	stats_mf_6way.txt	config_mcf_2way.ini

Command line for submitting files to github

- *cd [your github repository location]*
- *git add "*"*
- *git commit -m "Gem5 lab1"*
- *git push origin master*

-

Stages of the Simulated Processor

The simulated processor executes instructions in several stages, which are important to understand to reason about the statistics reported:

- ***fetch*:** instructions are fetched from the instruction cache. By default, the processor fetches up to eight instructions at a time. The number fetched is controlled by the fetchWidth parameter. This stage does branch prediction and branch target prediction to determine what to fetch.
- ***decode*:** instructions from the fetch stage are preproecssed. This stage handles execution of unconditional branches (whose target address is not in a register, etc.). In a real processor, this would be where instruction register numbers, etc. would be identified (but the simulation does not work this way internally; it only simulates the timings). The maximum number of instructions processed per cycle is controlled by the decodeWidth parameter.
- ***rename*:** entries in the re-order buffer and the instruction queue (approximately a shared reservation buffer) are allocated for each instruction. Register operands of the instruction are renamed, updating a renaming map (blocking if not enough free registers are available). The maximum number of instructions processed per cycle is controlled by the renameWidth parameter.
- ***dispatch/issue*:** instructions whose renamed operands are available are dispatched to functional units. For loads, stores, they are dispatched to the Load/Store Queue (LSQ). The simulated processor has a single instruction queue from which all instructions issue. Ordinarily instructions are taken in-order from this queue The maximum number of instructions processed per cycle is controlled by the dispatchWidth parameter.
- ***execute*:** the functional unit actually processes their instruction. Each functional unit may have a different latency (number of cycles until it produces a result). Conditional branch mispredictions are identified here. The maximum number of instructions processed per cycle is controlled by the configuration of the types of operations, latencies, and counts of the functional units available.
- ***writeback*:** send the result of the instruction into the corresponding physical register (if any), marking the register as available and permitting the issue of dependent instructions. Update the reorder buffer entry for the instruction. The maximum number of instructions processed per cycle is controlled by the wbWidth parameter.
- ***commit*:** process the reorder buffer, freeing up reorder buffer entries. A second renaming map is updated. The maximum number of microps processed per cycle is controlled by the commitWidth parameter.

In the event of branch misprediction, trap, or other speculative execution event, *squashing* can occur at all stages of this pipeline. When a pending instruction is squashed, it is removed from the instruction queues, reorder buffers, requests to the instruction cache, etc.

The fetch, decode, rename and commit stages process instructions in program order. Other stages process instructions out-of-order based on availability of operands and results.

The simulated processor also lets one configure the latency between many of these stages — how many clock cycles it takes an instruction to pass from one phase to another in the best case.

To deal with the complicated instructions that are very common in X86, like a single instruction that performs a load and an add, the simulated processor splits many instructions into multiple microoperations. Confusingly, it is often not clear whether statistics are referring to microops (and calling them instructions) or real instructions. Generally, statistics about the issue, execute, and writeback phase will always concern microops (even if their descriptions use the word instructions), and statistics about the commit phase will make it clear which are referred to.

Some Terminology in the Program Statistics

- IQ: instruction queue, where instructions are placed when they are ready to be executed on functional units or the caches.
- LSQ, LQ, SQ: Load/Store Queue, Load Queue, Store Queue: queues that hold pending memory operations. These are also responsible for tracking accessed memory addresses to ensure that out-of-order load/store instructions that refer to the same memory address produce the correct results.
- squashing: undoing the effect of instructions due to a branch misprediction, fault, or other kind of incorrect speculative execution. Note that this occurs at all stages of the pipeline, rather than waiting for instructions to reach the reorder buffer.

Simulation performance

Beware that **gem5 is much slower than an actual processor**. A program taking 1 second to run on a real computer would take approximately 30 minutes to run in gem5.opt, because of all simulation overheads. In particular, notice that execution is unbounded if you do not specify the max number of instructions.

Introduction to gem5 configuration scripts

In the coming assignments, you will need to reconfigure the simulated environment. gem5 uses Python scripts to construct a virtual system and wire everything together. This means that in order to make system changes you must be relatively comfortable with editing Python code. It is recommended that you place all configuration into a script, rather than as command-line arguments. While basic changes work from the command line, you will outgrow them over the course, so it is better to learn to edit the scripts.

It is suggested that you read the explanation below to understand the ideas underlying the configuration system.

http://gem5.org/Configuration_/_Simulation_Scripts

When making changes to the configuration, you have two options.

(1) The scripts use the Python *Options* library to provide command-line arguments, so by setting values on the Options object you can alter the configuration. For example:

```
options.caches=1            # Enable L1 Caches
options.l2cache=1           # Enable L2 Cache
options.cpu_type="detailed"  # Use the Out of Order CPU
```

All options can be found in *configs/common/Options.py*, though you should note that some may not apply to your script. You should set these options early in the script, before they are used.

(2) The other option is to modify the already-configured Python objects. This lets you perform more in-depth changes, for example:

```
CPUClass.fetchWidth=1                  # Only Fetch 1 Instruction at a time
for fu in CPUClass.fuPool:
   if fu.__class__.__name__ == "IntALU"
      fu.count = 1                     # Only 1 Integer ALU
```