CSE 5600 Computer Systems Architecture I

Assignment 1, due Sept. 18, 2026

The goal of this lab assignment is to help familiarize you with simulating a system using `gem5` by simulating two programs with three different instruction set architectures and interpreting the accompanying statistics with meaningful analysis. Consequently, the systems that we will be simulating are relatively simple. In the coming weeks, we will simulate more complex systems to reinforce concepts covered in class. The system-level aspect of this assignment is based on introductory `gem5` materials from Jason Lowe-Power of UC Davis. The benchmark applications were obtained from UW-Madison and Charlie Reiss at UVA.

It is possible to obtain the source code for `gem5` from `github` and build it yourself, but we will describe how to work with the pre-built `gem5` simulator, version 25.1.0.1, on the WUSTL Linux cluster machines.

1. Log into `shell.cec.wustl` using your WUSTLKey credentials). If on a Mac, open a terminal window and issue the command `'ssh wustlkey@shell.cec.wustl.edu'`, where `wustlkey` is your WUSTL key. If on a PC, connect via an ssh client (e.g., `putty`, which is free).

2. Once logged in, issue the command `'qlogin'`, which opens a shell on one of the Linux cluster machines. Always execute `gem5` simulations on one of the cluster machines, not directly on `shell.cec`, as compute-intensive jobs on `shell.cec` will be summarily terminated by the EIT staff.

3. As an alternative to the two steps above, you can use a web browser to access one of the Linux cluster machines by following the link below:

   `https://linuxlab.engr.wustl.edu`

   This gives you a windowing environment on one of the Linux cluster nodes.

   Use your WUSTL key credentials to log in, then in the Interactive Apps pulldown (look at the top menu bar), select 'Academic Linux Desktop', then 'Launch', and when it is available (it takes a bit), 'Launch Academic Linux Desktop'. An example of the end result is shown in Figure 1.

4. Figure out what shell is currently active with:

   ```
   echo $0
   ```

   If using `tcsh`, open `.cshrc` in your home directory and add:

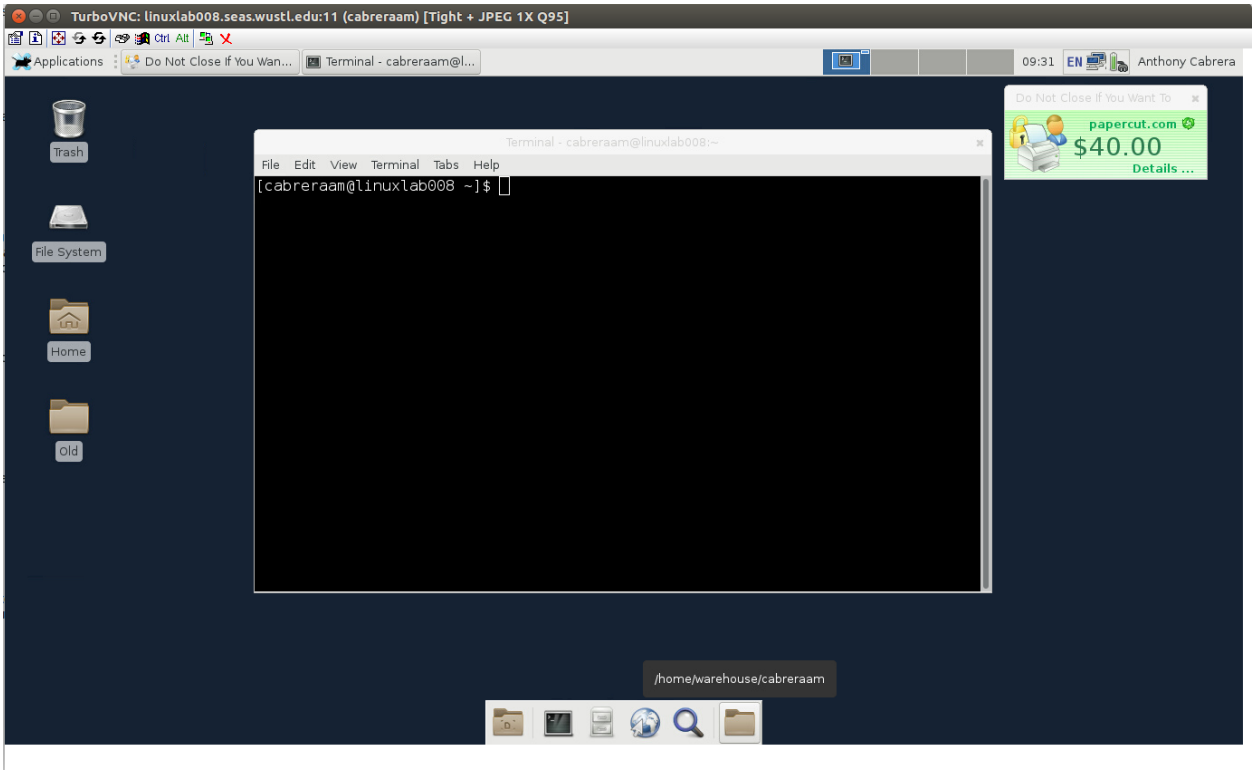


Figure 1: Example Linux environment.

```
source /project/linuxlab/gem5-24/gem5-v25.1.0.1/env.tcsh
```

Else, if using `bash`, open `.bashrc` in your home directory and add:

```
source /project/linuxlab/gem5-24/gem5-v25.1.0.1/env.sh
```

This will set up your environment for running the `gem5` simulator. In order for the terminal to see these changes without restarting your session, use the following command:

```
source ~/.cshrc  or  source ~/.bashrc
```

depending on which shell you are using. (Most folks will be using `bash`.)

5. If you are unfamiliar with the Linux command line, the following link has a tutorial:

   `http://clusters.engineering.wustl.edu/guide/`

6. Create directories `cse5600` and, underneath it, `assign1` in your home directory by issuing the following command:

```
mkdir -p cse5600/assign1
```

Navigate to this directory using:

```
cd cse5600/assign1
```

7. Now we will begin to construct our simple system. In this exercise, we will be creating the system shown in Figure 2:

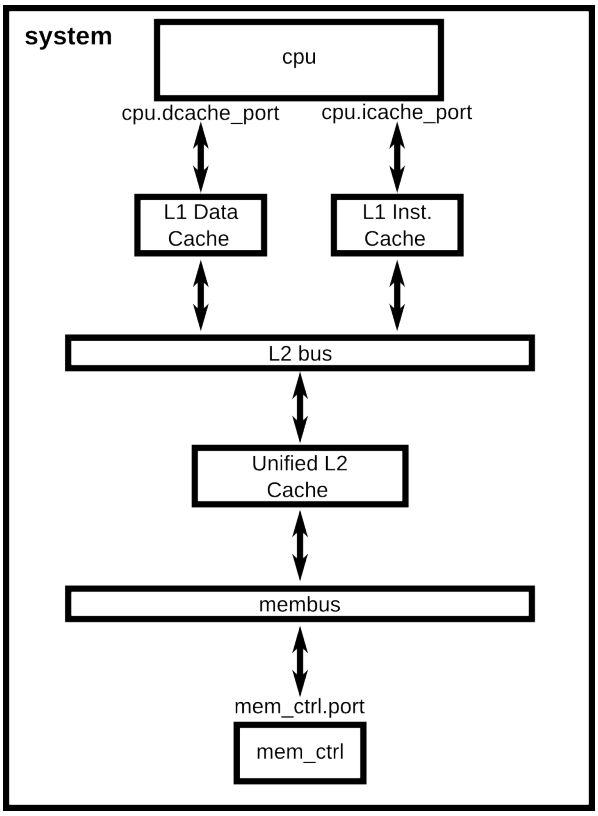
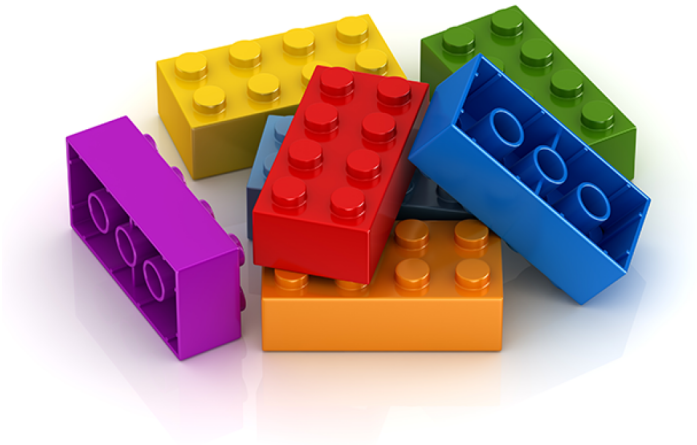


Figure 2: System to be simulated.

In this assignment, we will be using the `x86`, `ARM`, and `RISC-V` ISAs.

All systems in `gem5` are created by `Python` scripts. `gem5`'s modular design is built around the `SimObject` type of the M5 simulator. 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`. Essentially, creating a system configuration file is analogous to playing with LEGOs, where each component in our system is one LEGO that we will combine with other LEGOs. `gem5` provides a standard library of components, features, and functionality that wrap many of the M5 `SimObjects`. We will be using the `gem5` standard library to simulate our system.



The `gem5` standard library is introduced in the `gem5` standard library overview documentation. For detailed information on components, you can navigate through the standard library directly in the `gem5` Git repository: `gem5` standard library Python package.

For each piece of the system, we will be making a separate configuration file. These files will be linked together through a main python script, which you will invoke to begin running your simulations. We do this so that in future assignments, single files can be swapped in and out to swap processor types, memory configurations, caches, or any other system component without needing to rely on modifying a single monolithic file.

Note, if you choose to copy and paste code from this document assignment, make sure you proof it because it frequently doesn't transcribe correctly and is error prone.

Start by creating a new folder inside of this directory and navigating to it:

```
mkdir config && cd config
```

then create the needed files by running this command:

```
touch args.py board.py cache.py memory.py processor.py workload.py
```

Now, using your favorite text editor (I prefer `vim`, but you can use whatever you are comfortable with), open the `memory.py` file you created so we can define a simple memory topology for this simulation. First, import the needed GEM5 component:

```
from gem5.components.memory import SingleChannelDDR3_1600
```

and then define another function which is going to construct a simple memory component for our simulator:

```
def make_memory():
    return SingleChannelDDR3_1600(size="32MiB")
```

Now save and close this file and open up the `cache.py` file. Just like before, we are going to import the GEM5 component and then we will define a construction function to make our cache for us. Add these lines to your file then save and close it:

```
from gem5.components.cachehierarchies.classic.private_l1_private_l2_cac
he_hierarchy import (
    PrivateL1PrivateL2CacheHierarchy,
)


def make_cache_hierarchy():
    return PrivateL1PrivateL2CacheHierarchy(
        l1d_size="16KiB",
        l1i_size="16KiB",
        l2_size="256KiB",
    )
```

8. Now that our memory subsystems have dedicated construction scripts, we are going to make one of our processors. Start by opening `processor.py`. For this one we need to import a few different GEM5 pieces:

```
from gem5.components.processors.cpu_types import CPUTypes
from gem5.components.processors.simple_processor import SimpleProcessor
from gem5.isas import ISA
from gem5.runtime import get_supported_isas
```

We need all of these different pieces so that we can construct processors with different ISAs.

Now, to begin, we need to make sure we request the correct ISA from the GEM5 simulator when we construct a processor. Add this function to unpack and return the ISA and ISA strings from the GEM5 runtime when it starts up:

```
def get_runtime_isa():
    supported_isas = get_supported_isas()
    isa = next(iter(supported_isas))
    isa_str = isa.value
    return isa, isa_str
```

Next we are going to need a function which takes in a given ISA from this function and constructs a processor for us. For now, regardless of the ISA we select, we are

going to keep the processor type consistent. We are going to always choose an in-order, timing-based, single-core processor model. This will model the timing aspects such as instruction latencies and dependencies, but without detailed modeling of the microarchitecture. Add this to your script:

```
def make_processor(isa):
    processor = SimpleProcessor(
        cpu_type=CPUTypes.MINOR,
        isa=isa,
        num_cores=1,
    )

    for cpu in processor.get_cores():
        core = cpu.core

        # Make it single issue
        core.decodeInputWidth = 1
        core.executeInputWidth = 1
        core.executeIssueLimit = 1
        core.executeCommitLimit = 1

    return processor
```

Now we need to make one final function to set what our clock speed is. Since we have two different types of architectures (two RISC architectures and one CISC architecture), we want to set the system clock conditioned on which architecture is being tested. We can do this simply by checking the value of the `isa` of the processor when it's constructed. Make this function to set the speed conditioned on the ISA:

```
def get_clock_frequency(isa):
    clk_freq = "1GHz"

    if isa in (ISA.ARM, ISA.RISCV):
        clk_freq = "1.2GHz"

    return clk_freq
```

Once you have this added, save and close the processor file.

9. The last piece of the system we need now is a board to bind all of our components together. Fortunately for us, GEM5 provides us with many board options. For now, we are going to go with a simple board, which takes in a processor, main memory piece, and cache hierarchy along-side a processor clock speed. First open the `board.py` file and import the required GEM5 library again:

```
from gem5.components.boards.simple_board import SimpleBoard
```

and now, just make a simple function to take in our components and create a new board with them:

```
def make_board(clk_freq, processor, memory, cache_hierarchy):
    return SimpleBoard(
        clk_freq=clk_freq,
        processor=processor,
        memory=memory,
        cache_hierarchy=cache_hierarchy,
    )
```

Once done, we have our system completed. Save and close this file.

10. The `gem5 x86, ARM, and RISC-V` simulators have already been compiled for you as well as test programs (i.e., workloads) for each ISA. In order to use either simulator binary and corresponding test program, we need to make a workload selector which chooses the correct test program depending on the ISA of the simulated processor. Open `workload.py` and import the necessary GEM5 piece:

```
import os
from gem5.resources.resource import BinaryResource
```

And just like before, we are going to make a simple stand-alone function in this file, however this time it will determine which test binary to select for execution instead of constructing pieces of our simulation:

```
def set_workload(board, args, isa_str):
    gem5_path = os.environ.get("GEM5")

    if gem5_path is None:
        raise RuntimeError(
            "The GEM5 environment variable is not set."
        )

    program_path = os.path.join(
        gem5_path,
        "testprogs",
        f"{args.prog}_{isa_str}",
    )

    if not os.path.exists(program_path):
        raise FileNotFoundError(
```

```
            f"Could not find workload binary: {program_path}"
        )

    if args.prog == "daxpy":
        workload_args = [str(args.daxpy_N)]
    elif args.prog == "queens":
        workload_args = [str(args.queens_N)]
    else:
        raise ValueError(f"Unsupported program: {args.prog}")

    board.set_se_binary_workload(
        BinaryResource(program_path),
        arguments=workload_args,
    )
```

Save and close this file.

11. We will want to be able to specify which program to run at the command line, so we will instantiate an argument parser and add an option to specify the program to run. Open `args.py` and add the following lines to your configuration file.

```
import argparse

def parse_args():
    parser = argparse.ArgumentParser()

    parser.add_argument("--prog", required=True, choices=["daxpy", "queens"])
    parser.add_argument("--daxpy-N", default=100, type=int)
    parser.add_argument("--queens-N", default=10, type=int)

    return parser.parse_args()
```

Now close this file and return to the main directory by executing

```
cd ..
```

12. The final thing we need to do is instantiate the system and begin execution. First, we are going to make a central configuration script to bind all of our construction scripts together and build our simulated system at runtime. First we create the `Simulator` object, passing in our created `board` object into the constructor. We simply call the `run()` member of this object and our simulation can begin.

    First create the new central file:

```
touch x86_arm_riscv_compare.py
```

Open it and add the following lines:

```
import atexit
import m5
from gem5.simulate.simulator import Simulator
```

Then we are going to import all of our construction scripts. Add each one as an import into this file:

```
from config.args import parse_args
from config.board import make_board
from config.cache import make_cache_hierarchy
from config.memory import make_memory
from config.processor import make_processor
from config.workload import set_workload
from config.processor import (
    get_clock_frequency,
    get_runtime_isa,
)
```

Now we need to construct our system using all of our files. Add these lines to construct our simulator:

```
isa, isa_str = get_runtime_isa()
processor = make_processor(isa)
cache_hierarchy = make_cache_hierarchy()
memory = make_memory()
clk_freq = get_clock_frequency(isa)

board = make_board(
    clk_freq=clk_freq,
    processor=processor,
    memory=memory,
    cache_hierarchy=cache_hierarchy,
)
```

Now we need to fetch which test we want to run from the command line, and setup the system to run the test:

```
args = parse_args()
set_workload(board, args, isa_str)

simulator = Simulator(board=board)
simulator.run()
```

Since we want to collect information from our simulations, we need to set up a few more things. We first unregister the GEM5 statistics dump from the exit procedure. This ensures we will not have the run statistics duplicated, making it difficult to read the results, and we print when the simulation has finished, what caused it to exit, and what tick it exited on.

```
atexit.unregister(m5.stats.dump)
print(
    "Exiting @ tick {} because {}.".format(
        simulator.get_current_tick(),
        simulator.get_last_exit_event_cause(),
    )
)
```

Save and close this file too.

13. Now that we have created a simple simulation script we are ready to run `gem5`. `gem5` can take many parameters, but requires just one positional argument, the simulation script. In our case, however, we need to pass the output directory option to the `gem5` build and the program option to the configuration script that we created in previously. We will need to run the simulator 6 times–3 different ISAs (`x86`, `ARM`, `RISC-V`) and 2 different programs. Note that there are 3 binaries for each program, i.e. there are `daxpy_x86`, `daxpy_arm`, and `daxpy_riscv` binaries for `daxpy`, and `queens_x86`, `queens_arm`, and `queens_riscv` binaries for `queens`. One run for `x86` might look like this:

```
$GEM5/build/X86/gem5.opt --outdir="daxpy_x86" x86_arm_riscv_compare.py \
    --prog="daxpy"
```

The ARM build is targeted by using `$GEM5/build/ARM/gem5.opt`. The RISCV build is targeted by using `$GEM5/build/RISCV/gem5.opt`.

In the future, we may ask you to sweep across many microarchitectural parameters in which it would make sense to create a script. While it is not necessary for this problem, it may be good practice to create a script that runs each application using both ISAs.

Take screenshots of each simulator output.

Using the `stats.txt` file output from each simulator run, confirm the execution time (`sim_seconds`) by solving the equation for $t_{execution_time}$:

$$t_{execution_time} = (number\ of\ instructions) \times (CPI) \times t_{clk} \tag{1}$$

where CPI is cycle per instruction and t_{clk} is the CPU clock period. (By confirm, we mean to extract $t_{execution_time}$, number of instructions, and CPI from the simulation and combine it with the t_{clk} value that you specified for the simulation and ensure that the equation holds reasonably well.) Note that the total number of ticks simulated is NOT the same as the number of CPU cycles simulated, but rather the number of `gem5`

simulator ticks. However, we know that the simulator tick frequency is 1 THz, so we can use this to determine the total number of CPU cycles simulated.

What is the CPI from each simulation run? Is this as you would expect? If it is high (hint, hint), why do you think it is high?

Prepare four plots. The x axis is the same for each plot: the two applications and then the appropriate mean. For the first plot, the y axis is the execution time for the x86 ISA. For the second plot, the y axis is the execution time for the ARM ISA. For the third plot, the y axis is the execution time for the RISC-V ISA.

Let's compare ISAs. Calculate the speedup of each application for two of the ISAs relative to one ISA. You can choose which of the three ISAs to be the baseline. The speedup is the y axis for the fourth plot. Careful which mean you use for the speedup plot.

For this assignment, turn in the configuration files you created, a screenshot of the output for each run of the simulator, and a document that includes the responses to the questions above, all as a submission in Canvas. You can either combine everything into one file or upload multiple files. Please don't just combine via putting them into one zip file, that's more work for grading. If you are submitting multiple files, just upload each of them separately and ensure that the files are reasonably titled so that we can understand what file is what.