CSE 560M Computer Systems Architecture I

Assignment 1, due Friday, Sep. 20, 2019

This goal of this lab assignment is to help familiarize you with simulating a system using `gem5` by simulating two programs with two 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 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'`. If on a PC, connect via an ssh client (e.g., `putty`, which is free). There is a client called 'SSH Secure Shell Client' on the PCs in the instructional labs.

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:

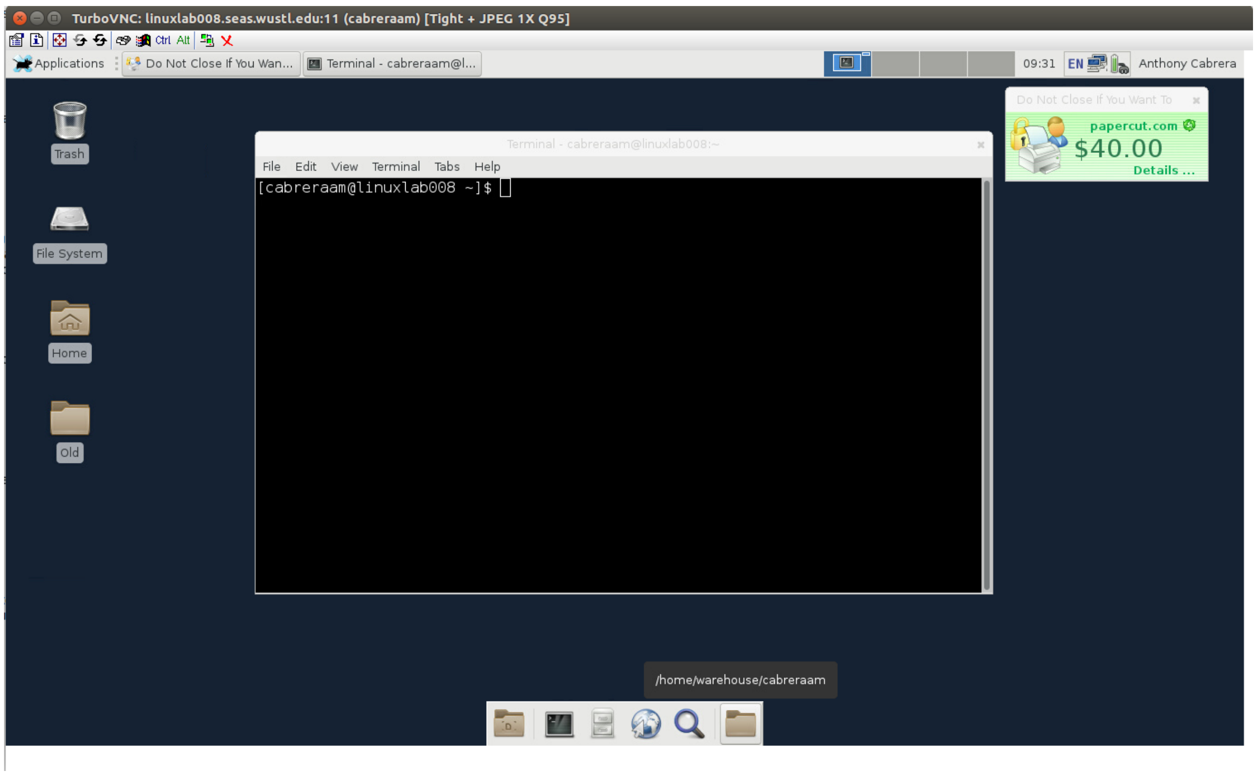
   https://linuxlab.seas.wustl.edu/equeue/

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

   Use your WUSTL key credentials to log in, click `Submit Job`, then click `Linux_Desktop`. This will bring up a list of options for your environment, but the default options are fine. Click `Submit job`. A file with the `.jnlp` extension should have begun downloading. Once finished, open the `.jnlp` file and use your WUSTL key credentials to log in. From there, you should see a Linux desktop and be able to open a terminal window. An example of the end result is shown in Figure 1.

4. Figure out what shell is currently active with:

   ```
   echo $0
   ```


TurboVNC: linuxlab008.seas.wustl.edu:11 (cabreraam) [Tight + JPEG 1X Q95]
Applications
Do Not Close If You Wan...
Terminal - cabreraam@l...
09:31
EN
Anthony Cabrera
Trash
File System
Home
Old
Terminal - cabreraam@linuxlab008:~
File
Edit
View
Terminal
Tabs
Help
Do Not Close If You Want To
papercut.com
$40.00
Details ...
/home/warehouse/cabreraam

```
[cabreraam@linuxlab008 ~]$
```

Figure 1: Example Linux environment.

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

```
setenv GEM5 /project/linuxlab/gem5/gem5_dev/
```

Else, if using `bash`, open `.bash_profile` (it might be `.bashrc`) in your home directory and add:

```
export GEM5=/project/linuxlab/gem5/gem5_dev
```

In order for the terminal to see these changes without restarting your session, use the following command:

`source ~/.cshrc` or `source ~/.bash_profile` 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 `cse560m` and, underneath it, `hw1` in your home directory by issuing the following command:

```
mkdir -p cse560m/hw1
```

   Navigate to this directory using:

```
cd cse560m/hw1
```

   and create the file `x86_vs_arm.py` with:

```
touch x86_vs_arm.py
```

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

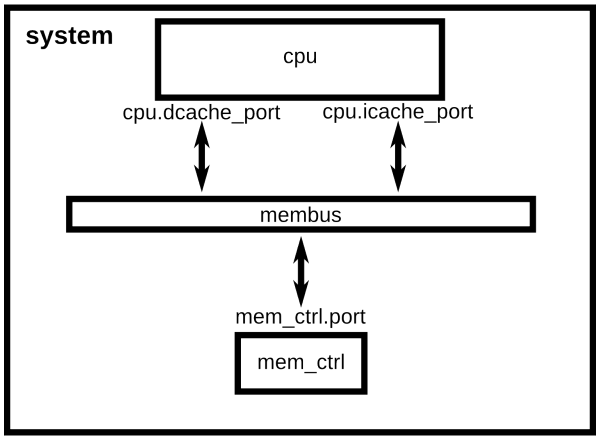


Figure 2: System to be simulated.

   In this assignment, we will be using both the `x86` and `ARM` ISAs.

   All systems in `gem5` are created by `Python` scripts. `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`. 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.

   Now, using your favorite text editor (I prefer `vim`, but you can use whatever you are comfortable with), open the `x86_vs_arm.py` file you created. The first thing we will do is make visible all of the built `SimObjects` associated with a target ISA available to us with the following lines:

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

Keeping with the LEGO theme, we are importing all of the LEGOs that we get to play with.

8. The `gem5 x86 and ARM` simulators have already been built for you. In order to use either simulator binary, add the follow lines to your configuration file:

```
import os
gem5_path = os.environ["GEM5"]
```

9. We want to be able to specify which program to run at the command line, so we will instantiate an option parser and add an option to specify the program to run. Add the following lines to your configuration file.

```
import optparse
parser = optparse.OptionParser()
parser.add_option("--prog", type="str", default=None)
(options, args) = parser.parse_args()
program = options.prog
```

10. For our first piece, we will instantiate a `System` object. The `System` object will be the parent of all the other objects in our simulated system. The `System` object contains a lot of functional information, like the physical memory ranges, the root clock domain, the root voltage domain, etc. To create the system `SimObject`, we simply instantiate it like a normal `Python` class:

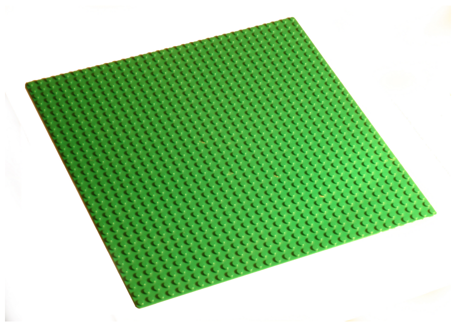
```
system = System()
```

This is analogous to the base LEGO piece that all of our other LEGOs will be placed on.

11. Now that we have a reference to the system we are going to simulate, we can set the clock on the system. We first have to create a clock domain.

```
system.clk_domain = SrcClockDomain()
```

We have to specify a voltage domain for this clock domain. Since we don't care about system power, we'll just use the default options for the voltage domain. Then we can set the clock frequency on that domain.



```
system.clk_domain.voltage_domain = VoltageDomain()
```

Setting parameters on a `SimObject` is exactly the same as setting members of an object in `Python`. Since we are using two different architectures, we want to set the system clock conditioned on what architecture we are using.

```
isa = m5.defines.buildEnv['TARGET_ISA']
if isa == "x86":
    system.clk_domain.clock = '1GHz'
elif isa == "arm":
    system.clk_domain.clock = '1.2GHz'
```

12. Once we have a system, let us 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 memory simulation. 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.

    Add the following lines to your configuration file:

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

13. Now, we can create a `CPU`. We will start with the most simple timing-based CPU in gem5, `TimingSimpleCPU`. This CPU model executes each instruction in order and takes a single clock cycle to execute except for memory requests, which flow through the memory system. To create the CPU you can simply just instantiate the object by adding the following line to your configuration file:

```
system.cpu = TimingSimpleCPU()
```

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

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

14. Now that we have a memory bus, lets connect the cache ports on the CPU to it. In this case, since the system we want to simulate does not 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.

    Add the following lines to your configuration file:

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

15. 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.

    Add the following lines to your configuration file:

    ```
    system.cpu.createInterruptController()
    if isa == 'x86':
        system.cpu.interrupts[0].pio = system.membus.master
        system.cpu.interrupts[0].int_master = system.membus.slave
        system.cpu.interrupts[0].int_slave = system.membus.master

    system.system_port = system.membus.slave
    ```

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

    Add the following lines to your configuration file:

    ```
    system.mem_ctrl = DDR3_1600_8x8()
    system.mem_ctrl.range = system.mem_ranges[0]
    system.mem_ctrl.port = system.membus.master
    ```

At this point, we have placed all of the LEGOs for this system, and we can execute test applications on it.

17. 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 will be executing `daxpy`, a double precision application that takes the form $a \times x + y$, and `queens`, which tries to find a placement for n queens on an $n \times n$ chessboard. We will execute each application on both ISAs.

    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 its workload, and finally create the functional execution contexts in the CPU. These programs have been authored and compiled for you for both the `x86` and `ARM` ISA. Depending on which ISA you specify and what program you wish to deploy

    Add the following lines to your configuration file:

    ```
    process = Process()
    apps_path = "/project/linuxlab/gem5/test_progs/"
    if program == "daxpy" and isa == "x86":
        process.cmd = [apps_path + '/daxpy/daxpy_x86']
    elif program == "daxpy" and isa == "arm":
        process.cmd = [apps_path + '/daxpy/daxpy_arm']
    elif program == "queens" and isa == "x86":
        process.cmd = [apps_path + '/queens/queens_x86']
        process.cmd += ["10 -c"]
    elif program == "queens" and isa == "arm":
        process.cmd = [apps_path + 'queens/queens_arm']
        process.cmd += ["10 -c"]

    system.cpu.workload = process
    system.cpu.createThreads()
    ```

18. 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 weve 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.

    Add the following lines to your configuration file:

    ```
    root = Root(full_system = False, system = system)
    m5.instantiate()
    print ("Beginning simulation!")
    ```

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

19. 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 4 times–2 different ISAs (`x86` and `ARM`) and 2 different programs. Note that there are 2 binaries for each ISA, i.e. there are `daxpy_x86` and `daxpy_arm` binaries for `daxpy`, and `queens_x86` and `queens_arm` binaries for `queens`.The two different programs are `daxpy`. One run for `x86` might look like this:

    ```
    $GEM5/build/X86/gem5.opt --outdir="daxpy_x86" x86_vs_arm.py --prog="daxpy"
    ```

    The ARM build is targeted by using `$GEM5/build/ARM/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 $t_{execution\_time}$:

    $$t_{execution\_time} = (number\ of\ instructions) \times (CPI) \times t_{clk} \qquad (1)$$

    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 1THz, so we can use this to determine the total number of CPU cycles simulated.

    Prepare three 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.

    Which ISA is faster? Calculate the speedup of each application for the faster ISA relative to the slower ISA. The speedup is the y axis for the third plot. Careful which mean you use for the speedup plot.

    For this assignment, turn in the configuration file you created, a screenshot of the output for each run of the simulator, and a document that includes the responses to the questions above.