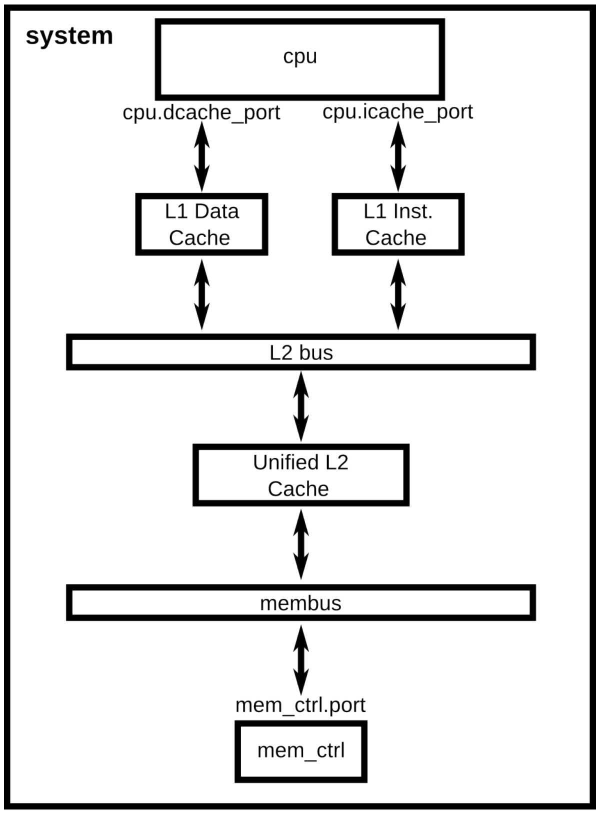
CSE 560M Computer Systems Architecture I

Assignment 2, due Fri., Nov. 1, 2019

In this lab assignment we will explore using the `gem5` simulator to look at cache simulation and program statistics.

1. In this exercise, we will be creating the following system:



   This may look somewhat similar to your previous system but with some extra building blocks added. Instead of a simple unified memory system there is now a hierarchy of caches for the CPU to query for memory. Here, we've added a L1 data and instruction cache along with a unified L2 cache to the configuration file. These caches will be extensions of the `Cache` class and will be instantiated in a similar way to the previous lab.

   Of course, this base class does not have every parameter instantiated for us so lets walk through instantiating one of these SimObjects. We'll start with the L1 instruction cache to begin and leave the rest as an exercise. First we'll need to create a `Caches.py` file alongside our `gem5` configuration file from last lab. To start we'll need to create the file `Caches.py` by running the following command in a `cse560m/hw2` folder:

```
touch Caches.py
```

Then opening the file in the text editor of your choice begin by typing out the code to import the gem5 building blocks.

```
from m5.defines import buildEnv
from m5.objects import *
```

Then, on the next line, we'll start be declaring the class `L1Cache` and setting some default parameters:

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

Now we need to define an initialize function and connections for the cache:

```
     def __init__(self, options=None):
         super(L1Cache, self).__init__()
         pass
     def connectBus(self, bus):
         self.mem_side = bus.slave
     def connectCPU(self, cpu):
         raise NotImplementedError
```

NOTE: These functions are within the scope of the `L1Cache`, take note of the indents.

Now we can move on to actually defining a cache that will be used. On the next line we will define the actual L1 instruction cache.

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

     def __init__(self, opts=None):
         super(L1ICache, self).__init__(opts)
         if opts.l1i_size:
             self.size = opts.l1i_size
         if opts.l1i_assoc:
             self.assoc = opts.l1i_assoc

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

A couple things to note with this configuration: This cache is defined as read only meaning that we can't write values back to the cache, which makes sense as we wouldn't want to corrupt our instructions. Next, the `writeback_clean` option is set to true denoting that writebacks will happen when evicting clean lines (As a fun exercise think about why this is specified here). After defining the size we add options for size and whether or not the cache is associative. Following that, the initialize function is defined which will load in the size and association options for the cache. Finally, the connections are defined stating that the connection between itself and the CPU is through the CPU's icache port.

Now, starting with what we've defined here, implement a L1D cache and create a L2 cache which will be its own extension of the `Cache` (not `L1Cache`) class.

The member functions for the L2 cache should include the following:

```
     def connectCPUSideBus(self, bus):
         self.cpu_side = bus.master
     def connectMemSideBus(self, bus):
         self.mem_side = bus.slave
```

The specifications for both L1 D- and L2 Cache are as follows:

	L1 Data Cache	L2 Cache
is_read_only	False	False
writeback_clean	False	False
size	64kB	256kB
assoc	default	8
tag_latency	default	20
data_latency	default	20
response_latency	default	80
mshrs	default	20
tgts_per_mshr	default	12
Connections	cpu.dcache_port	CPUSideBus:Master MemSideBus:Slave

And for both caches add the option to specify their size and associativity as we did in the L1 Cache.

2. Now lets add the caches that we've created to our system configuration file. To start, copy your configuration file from the first assignment, `x86_vs_arm.py`, into a new file with some meaningful name, in this case `assignment2.py`.

```
cp ../hw1/x86_vs_arm.py assignment2.py
```

From there open your newly copied file in a text editor of your choice.

First we will need to **remove** the following commands from the last assignment since we are going to connect the L1 caches to the L2 cache.

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

Next you'll need to **remove** the ISA specific clock settings.

```
if isa == "x86":
    system.clk_domain.clock = '1GHz'
elif isa == "arm":
    system.clk_domain.clock = '1.2GHz'
```

And replace with:

```
if options.clock_freq:
    system.clk_domain.clock = options.clock_freq
else:
    system.clk_domain.clock = '1.2GHz'
```

Now we can import your newly defined caches in the script.

```
from Caches import *
```

You'll need to use the `parser.add_option` command to add all of the options you added in your caches; otherwise, `Python` will complain. You should also add an option for varying the CPU clock speed. Note that the cache parameters will be of type `int` and clock frequency will be of type `str`.

Now we will start to connect the modules together, first instantiate the L1 caches using the following lines of code:

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

Then connect them to your cpu using the following lines:

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

Now we need to create a bus to connect our L1 caches to to L2, however, it only has one port so we need to create a bus to connect the three along with control signals from the CPU.

```
# Create a memory bus, a coherent crossbar, in this case
system.l2bus = L2XBar()
# Hook the CPU ports up to the l2bus
system.cpu.icache.connectBus(system.l2bus)
system.cpu.dcache.connectBus(system.l2bus)
```

Then create the L2Cache in the system and connect it's `CPUSideBus` to the `l2Bus` and it's `MemSideBus` to the `membus` created in the last assignment.

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

You should now have a complete system with a working L1 & L2 Cache! In this lab we are going to use the program `daxpy` to test the system.

Now in your `hw2` directory, test your `assignment2.py` configuration file using the command:

```
$GEM5/build/ARM/gem5.opt --outdir="daxpy_arm" assignment2.py --prog="daxpy"
```

Include a screenshot of the console output in your writeup.

3. Now that we have a working system let's change some parameters and get some measurements! Change your simulation so that the following parameters are implemented and run a simulation for each. Create a table for using the `ARM` ISA of the following parameters and record the value of the parameter for each cache configuration. Pick a parameter (an interesting one that is either listed here or that you've found on your own) and create a graph making it a function of the cache size, use two lines to represent the different clock and association sets. As an answer to this question show the graph and table that you have created.

Clock Speed	L1D$ Association	L1D$ Size
1.0 GHz	1	8 KB
1.0 GHz	1	16 KB
1.0 GHz	1	32 KB
1.0 GHz	1	64 KB
0.8 GHz	2	8 KB
0.8 GHz	2	16 KB
0.8 GHz	2	32 KB
0.8 GHz	2	64 KB

Parameters to record.
Instructions Committed
Average Gap Between Requests
L1D$ Overall Hits
L1D$ Number of replacements
L1D$ Overall Miss Rate
L1I$ Overall Miss Rate
L2 $ Overall Miss Rate
Number of CPU Cycles

(Note: It may be a good idea to add a `clock_speed` option to `assignment2.py` along with options to vary the cache size and associativity in the `Caches.py` file so that you can automate the data generation process. Additionally, it would help to save a copy of the output for each configuration so that you can use the `grep` utility to find the parameters that you are looking for.)

4. Pick two configurations from the previous problem and calculate their CPI as a function of the number of instructions and total number of CPU cycles. Show your work! Compare their CPI's.Is there a difference between the CPI of these two configurations? Why or why not?