CS3339 Project 3

Description:

In this project, you will create a simulator for a pipelined processor with cache. Your simulator will support the instruction set described in Project 1, and must be able to load a binary ARM file and execute it. Furthermore, your simulator will produce the disassembled program code (exactly as you did in Project 1), and will produce a cycle-by-cycle simulation showing the processor state at each cycle. The processor state includes the contents of registers, buffers, cache, and data memory at each cycle. You do not need to implement exception/interrupt handling.

Implementation: You must implement this program in Python 2.7

It is highly recommended that in each cycle, your program execute each pipeline stage in **REVERSE** order. That is, first handle the WB stage, then the MEM/ALU stages, then the ISSUE stage, then the IF stage. By executing the pipelines in this order, you will ensure that the cache is updated in the proper order, and you will not have collisions in the buffers between pipeline stages. This might give you some hint as how to structure your code.

Execution:

Your program must accept command line arguments for execution. The following arguments must be supported:

> $ python team#_project2.py -i test1_bin.txt -o team#_out
>
> NOTE: I must be able to change the output file names using the -o argument
>
> assuming test1_bin.txt is the input file at that moment

Your program will produce 2 output files. One named team#_out_pipeline.txt, which contains the simulation output, and one named team#_out_dis.txt, which contains the disassembled program code for the input ARM machine code.

Your program will be graded both with the sample input and output provided to you, and with input and output that is not provided to you. It is recommended you construct your own input programs for testing.

Expected Output:

Some sample inputs and expected outputs will be provided for testing. I am going to give you the dis output. You can extract the machine code input files that generated

the dis file. The most simple program is **t1_dis.txt** and they are progressively more complex in the following order: **t1, t2, t3.** You will also get the sim output for each.

Things we haven't covered: Yes there are terms and concepts in this spec that we haven't explicitly covered. I have full confidence that you all can google, bing, library, ect. them and figure them out. Often in your future engineering endeavors you will have specs and use cases that are sufficient but not explicit. If there is a gaping hole, and you all identify it, as your customer I will run it through the engineering chain of command (me, me and me) and get you a memo of understanding.

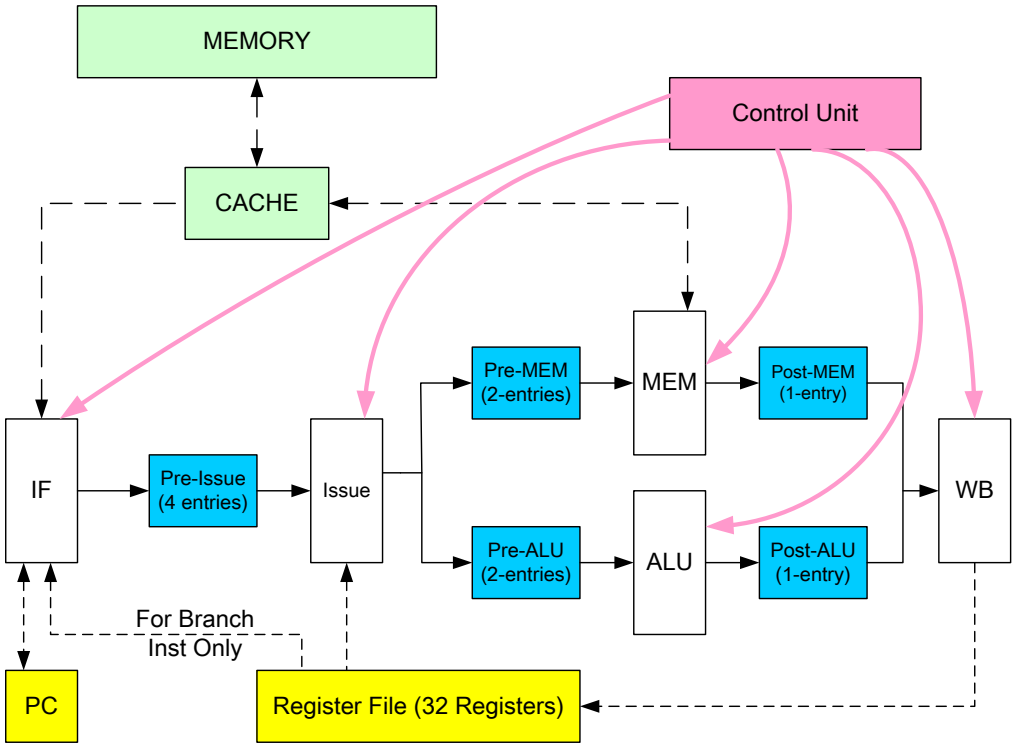
Additional Thoughts: I want you all to create a class for each of the functioning units and have a method that executes the function independently of other functioning units. That way you can execute them in any order. You will be working backwards which solves a lot of problems. Simulators often work this way. We will cover cache in detail soon.

I would prefer that you DO NOT create multiple files unless you absolutely must (irritate me)

Details:

Instruction format: The instruction format is exactly the same as in Project 1. This is the current configuration that I am coding to. I want you to contemplate what would happen if lets say I added an additional ALU or MEM Unit to the pipeline. Never know what might happen!

Pipeline Description:



White boxes represent functional units, blue boxes represent buffers (structured as queues) between the functional units, yellow boxes represent registers, and green boxes represent memory units. The various components are discussed in detail below:

Instruction Fetch (IF):

Instruction fetch can **fetch and decode** up to **two** instructions at each clock cycle. Instructions are always fetched and decoded in **program order**. The following conditions must be met before instructions can be fetched:

1. If the fetch unit is stalled, no instruction can be fetched at the current cycle (no pre-fetching). The fetch unit can be stalled due to a branch instruction or a cache miss (cache is described below)
2. If there is no room in the pre-issue buffer, no instructions can be fetched at the current cycle
3. If there is only one empty slot in the pre-issue buffer, only one instruction will be fetched

The IF unit can only fetch and decode an instruction if it is in the cache. If the instruction to be fetched is in the cache, the entire fetch and decode process takes only 1 clock cycle. If the instruction to be fetched is not in the cache, the cache unit will fetch the instruction from memory and it will be in the cache at the next clock cycle. If the first instruction to be fetched is not in the cache, the second instruction cannot be fetched even if it is in the cache.

If a branch instruction (B, CBZ, CNBZ) is fetched along with its next (in order) instruction, the next instruction will be discarded (it needs to be re-fetched based on the branch outcome).

If a branch instruction is fetched, the fetch unit will try to read all the argument registers in order to calculate the target address. If all registers are ready, or the target is immediate, the PC will be updated at the end of the cycle. Otherwise, the IF unit stalls until all argument registers are available. Therefore, if all registers are ready for a branch instruction, then no stalls are introduced.

A register can be written and read in the same clock cycle. Assume writes take place in the first half of the cycle, and reads take place in the second half of the cycle.

When a BREAK instruction is fetched, no more instructions will be fetched.

Branch, BREAK, and NOP instructions will all be fetched, but will not be written into the Pre-Issue Buffer (They are completely handled by IF).

Pre-Issue Buffer:
The pre-issue buffer has 4 entries, each entry can store a single instruction. The instructions are sorted in their program order (entry 0 always contains the oldest instruction and entry 3 contains the newest).

Issue Unit:
The issue unit follows the basic scoreboarding algorithm detailed below, to issue instructions. It can issue up to **two** instructions, **out of order**, per clock cycle. When an instruction is issued, it moves out of the pre-issue buffer and into either the pre-mem buffer or the pre-ALU buffer. The issue unit searches from entry 0 to entry 3 (IN THAT ORDER) of the pre-issue buffer and issues instructions if:

1. No structural hazards exist (there is room in the pre-mem/pre-ALU destination buffer)
2. No WAW hazards exist active instructions (issued but not finished, or earlier no-issued instructions)
3. No WAR hazards exist with earlier not-issued instructions
4. No RAW hazards exist with active instructions (all operands are ready)
5. A load instruction must wait for all previous stores to be issued
6. Store instructions must be issued in order

Remember, registers can be written and read in the same cycle.

ADD R3, R2, R1
ADD R3, R4, R5

Note that in the above instructions, the second instruction will not be issued until the first instruction has written back R3 in the write-back stage. The issue unit does not attempt to determine special cases in which WAW or RAWhazards will be avoided. We have covered in detail what these are so look them UP! It is important! What is the most common?

Pre-ALU queue:
The pre-ALU buffer has two entries. Each entry can store an instruction with its operands. The buffer is managed as a FIFO queue

ALU:
The ALU handles all non-memory instructions (everything except LDUR and STUR and branch instructions that are handled in the IF stage). All ALU operations take one clock cycle. When the ALU finishes, the instruction is moved from the pre-ALU buffer to the post-ALU buffer. The ALU can only fetch **one** instruction from the pre-ALU buffer per clock cycle.

Post-ALU buffer:
The post-ALU buffer has **one** entry that can store the instruction with the destination register ID and the result of the ALU operation

Pre-Mem queue:
The pre-mem buffer has two entries. Each entry can store an instruction with its address and data (for STUR). It is managed as a FIFO queue.

MEM Unit:
The MEM unit handles LDUR and STUR operations. Calculate the memory addresses in this unit.

For LDUR, it takes one cycle to finish if it hits in the cache. If it misses in the cache, then the operation cannot be performed and must be retried in the next cycle. In this case, the operation remains in the pre-mem buffer. When a cache hit occurs, the operation finishes and the instruction index (which gives you the destination reg for the writeback unit) and data will be written to the post-MEM buffer.

A STUR takes one cycle to finish if there is room in the cache. If it cannot write in the cache, then the operation cannot be performed and must be retried in the next cycle. In this case, the operation remains in the pre-mem buffer. When a cache write occurs, the STUR instruction just finshes. **The STUR instruction never goes into**

the post-MEM buffer. It just disappears. Everyone gets updated in the same cycle that the write happens. If the required set is full then you must figure out what to kick out on one cycle and write everything the next cycle. This may cause a stall since you are checking to see that all instructions in flight that might access the memory location have to wait. A RAW hazard.

Write Back Unit:
The WB unit can execute **two** write-backs in one cycle. It fetches the contents of the post-ALU and post-MEM buffers and updates the register file.

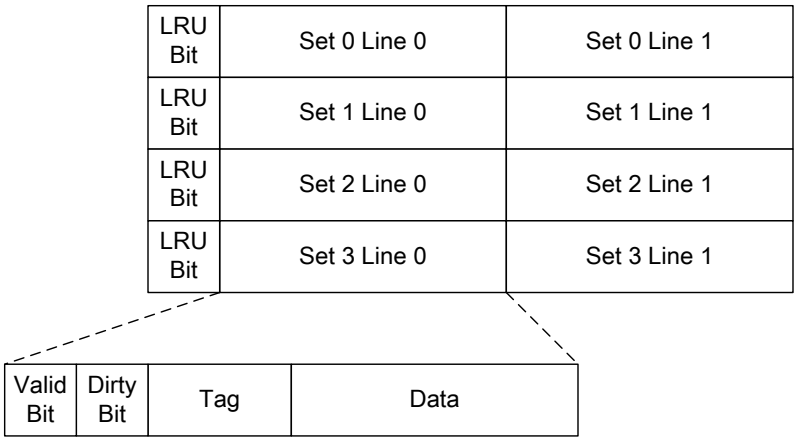
PC:
The PC holds the address of the next instruction. It should be initialized to 96.

Register File:
There are 32 registers. Assume sufficient read/write ports to perform all necessary operations during a single clock cycle.

Notes on Pipeline:

1. The execution finishes when a BREAK instruction is fetched and the pipeline is empty (all other instructions have finished)
2. No data forwarding cause I am so nice
3. No delay slot will be used for branching - What is a delay slot?
4. Different instructions finish in different stages:
    a. NOP, Branch, BREAK only use IF stage
    b. STUR only uses IF, issue, MEM stages
    c. LDUR uses IF, issue, MEM, WB stages
    d. ALU ops use IF, issue, ALU, WB stages

Cache Description:
The cache is uniform (holds both data and instructions) and is a 2-way associative cache with 4 sets. Each cache line/block contains 2 data words (1 word = 32 bits = 4 bytes). Because each cache line/block is 64 bits wide, all memory fetch operations will return two words (64 bits). The cache is organized as follows:



Note that since all address are word aligned , the lower two bits will be ignored when calculating which word will be written to in a cache block. Furthermore, since each cache line is 2 words wide, memory is effectively two word aligned, so the lower three bits of a memory address are ignored when calculating which cache set the data should be written to.

There are sufficient ports of cache so that the IF and MEM unit can access the cache at the same cycle without any port conflict. If the two units are accessing the same set at the same time, assume MEM will be executed at the first half of the cycle and IF will be executed at the second half of the cycle.

Note that the IF unit can only execute 1 memory read per cycle. Therefore, if the IF unit is attempting to fetch the instructions as 100 and 104, and neither are in cache, then a memory read for the words at address 96 and 100 can be initiated. The memory read for the words at addresses 104 and 108 must be initiated during the next clock cycle.

All the bits will be initialized as 0. (The LRU bit is initialized to 0 also.)

When a valid line exists, the valid bit will be turned on (set to be 1).

The dirty bit will be turn on when the pipeline writes to any part of the line.

It takes 1 cycle to get data from main memory to cache. Assume bypassing is used here so that both the cache and the request unit (IF or MEM) can get the data at the same cycle. For example, if IF requests a data in cycle 5 and it is a miss the IF and cache will get the data in cycle 6.

The write-back from cache to main memory takes no time (for simplification).

All dirty blocks will be written back at the cycle the simulation is finished.

Output Format:
At the end of a clock cycle, the processor, cache, and memory state will be output.

If any entry in a buffer/queue is empty, no content should be printed. Only instructions are printed, not addresses or values computed. The instruction should be printed as it was for Project 1.

Every cache line must be printed, even if it is empty. Furthermore, the cache values should be printed as a bit string.

The output format is as follows:

20 hyphens and a new line

Cycle[value]:

<blank_line>

Pre-Issue Buffer:

<tab>Entry 0:<tab>[instruction]

<tab>Entry 1:<tab>[instruction]

<tab>Entry 2:<tab>[instruction]

<tab>Entry 3:<tab>[instruction]

Pre_ALU Queue:

<tab>Entry 0:<tab>[instruction]

<tab>Entry 1:<tab>[instruction]

Post_ALU Queue:

<tab>Entry 0 :<tab>[instruction]

Pre_MEM Queue:

<tab>Entry 0:<tab>[instruction]

<tab>Entry 1:<tab>[instruction]

Post_MEM Queue:

<tab> Entry 0:<tab>[instruction]

< blank_line >

Registers

R00:< tab >< int(R0) >< tab >< int(R1) >..< tab >< int(R7) >

R08:< tab >< int(R8) >< tab >< int(R9) >..< tab >< int(R15) >

R16:< tab >< int(R16) >< tab >< int(R17) >..< tab >< int(R23) >

R24:< tab >< int(R24) >< tab >< int(R25) >..< tab >< int(R31) >

<blank line>

Cache

Set 0: LRU=<Value>

<tab>Entry 0: [(valid bit, dirty bit, int(tag))<word0,word1>]

<tab>Entry 1: [(valid bit, dirty bit, int(tag))<word0,word1>]

…

Set 3: LRU=<Value>

<tab>Entry 0: [(valid bit, dirty bit, int(tag))<word0,word1>]

<tab>Entry 1: [(valid bit, dirty bit, int(tag))<word0,word1>]

<blank line>

Data

< firstDataAddress >:< tab >< display 8 data words as integers with tabs in between >

..... < continue until the last data word >