Project 2

CDA 4102 / CDA 5155: Fall 2023

Due Date: November **1**, 2023 at **11:30** pm

You are not allowed to take or give help in completing this project. Taking help of ChatGPT or similar software will be treated as plagiarism. Submit the source file in eLearning before the deadline. Late submission (by email attachment to prabhat@ufl.edu) is allowed (up to 24 hours) with a 20% penalty (irrespective of whether it is late for 10 minutes or 10 hours). No grades for late submissions after 24 hours from the deadline. Please include the following sentence on top of your source file (as a comment):

On my honor, I have neither given nor received any unauthorized aid on this assignment.

In this project you will create a simulator for a pipelined processor. Your simulator should be capable of loading a RISC-V text file and generate the cycle-by-cycle simulation of the RISC-V code. It should also produce/print the contents of registers, queues, and memory data for each cycle. Please see the sample input file (sample.txt) and simulation output (sample_simulation.txt) from the project 2 assignment.

You do not have to implement any exception or interrupt handling for this project. We will use only valid testcases that will not create any exceptions. For example, test cases will not try to execute data (from data segment) as instructions, or load/store data from instruction segment. Similarly, there will not be any invalid opcodes, less than 32-bit instructions, etc. Please go through this document first, and then view the sample input/output files in the project assignment, before you start implementing the project.

Please develop your project **in one source file** (written in **C**, **C++**, **Java,** or **Python**) to avoid the stress of combining multiple files before submission and making sure it still works correctly. Please follow the **Submission Policy** which is at the end of this document to submit your source file. Your RISC-V simulator (with executable name as **Vsim**) should accept an input file (inputfilename.txt) in the following command format and produce output file (simulation.txt) that contains the simulation trace. In this project, you do not have to produce disassembly file.

Vsim inputfilename.txt

Correct handling of the sample input file (with possible different data values) will be used to determine 60% of the credit. The remaining 40% will be determined from other test cases that you will not have access prior to grading. Please construct your own sample input files to further test your simulator.

1. Instruction Format

The instruction format and other details (e.g., starting address) remain the same as **Project 1**.

2. Pipeline Description

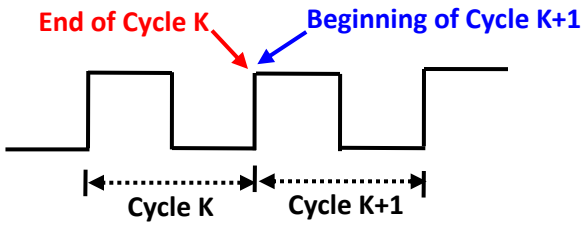

Figure 1: The end of the previous (last) clock cycle and the beginning of the current (next) clock cycle point to the same rising edge.

The entire pipeline is synchronized by a single clock signal as shown in **Figure 1**. We use the terms "the end of the current (previous) cycle" and "the beginning of the next (current) cycle" in the following discussion. Both refer to the rising edge of the clock signal, i.e., the end of a cycle is followed immediately by the beginning of the next cycle.

Figure 2 shows the pipeline. The white boxes represent the functional units, the blue boxes represent queues between the units, the yellow boxes represent registers and the green one is the memory unit. In the remainder of this section, we describe the functionality of each of the units/queues/memories in detail.

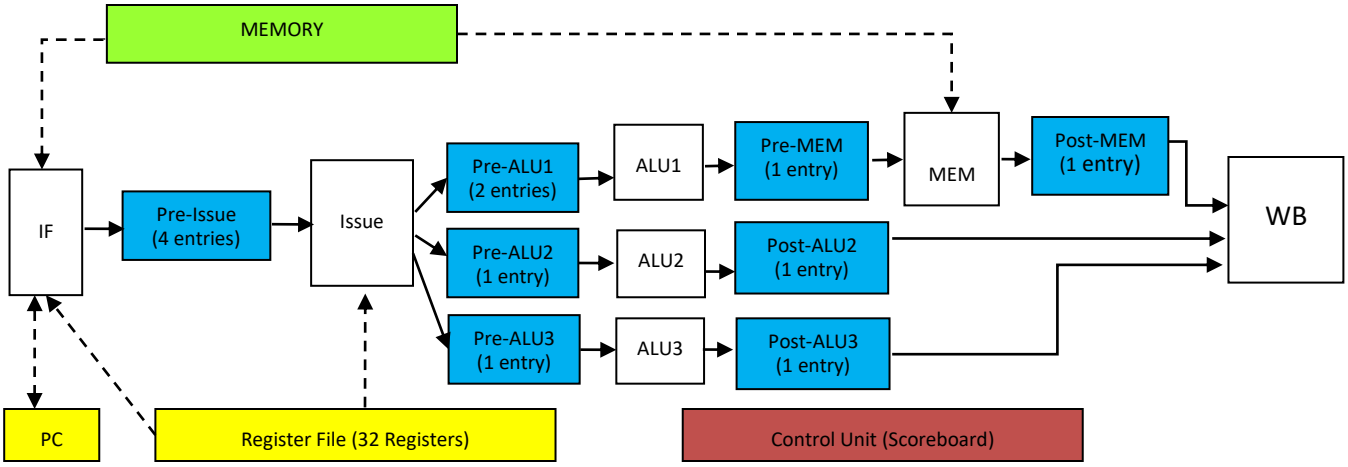

Figure 2: Pipelined Architecture (number of queue entries are shown in brackets)

2.1 Functional Units

Instruction Fetch/Decode (IF): Instruction Fetch/Decode unit can **fetch and decode** at most **two** instruction at each cycle (in program order). The unit should check all the following conditions before it can fetch further instructions.

- If the fetch unit is stalled at the end of last cycle, no instruction can be fetched at the current cycle. The fetch unit can be stalled due to a branch instruction.
- If there is no empty slot in the Pre-issue queue at the end of the last cycle, no instruction can be fetched at the current cycle.

Normally, the whole fetch-decode operation can be finished in 1 cycle. The decoded instruction will be placed in Pre-issue queue before the end of the current cycle. If a branch instruction is fetched, the fetch unit will try to read all the necessary registers to calculate the target address. If all the registers are ready (or target is immediate), it will update PC before the end of the current cycle. Otherwise the unit is stalled until the required registers are available. In other words, if registers are ready (or immediate target value) at the end of the last cycle, the branch does not introduce any penalty.

There are two possible scenarios when a branch instruction (jal, beq, bne, blt) is fetched along with another instruction. The branch can be the first instruction or the last instruction in the pair (remember, up to two instructions can be fetched per cycle). When a branch instruction is fetched with its next (in-order) instruction (first scenario), the next instruction will be discarded immediately (needs to be re-fetched again based on the branch outcome). When the branch is the last instruction in the pair (second scenario), both are decoded as usual.

Note that the register accesses are synchronized. The value read from register file in the current cycle is the value of corresponding register at the end of the previous cycle. In other words, any functional units cannot obtain the new register values written by WB in the same cycle.

When a break instruction is fetched, the fetch unit will not fetch any more instructions.

The branch instructions and break instruction will not be written to Pre-issue queue. It is important to note that we still need free entries in the pre-issue queue at the end of last cycle before the fetch unit fetches them, because the fetch cannot predict the types of instructions before fetching and decoding them.

Issue: Issue unit follows the basic Scoreboard algorithm to read operands from Register File and issue instructions when all the source operands are ready. It can issue at most **three** instructions **out-of-order** per cycle. It can send at most **one** load or store (lw, sw) instruction per cycle to the Pre-ALU1 queue, at most **one** arithmetic (add, sub, addi) instructions per cycle to the Pre-ALU2 queue, and at most **one** logical (and, or, andi, ori, sll, sra) instructions per cycle to the Pre-ALU3 queue. When an instruction is issued, it is removed from the Pre-issue queue before the end of current cycle. The issue unit searches from entry 0 to entry 3 (in that order) of Pre-issue queue and issues instructions if:

- No structural hazards (the corresponding queue, i.e., Pre-ALU1, Pre-ALU2, or Pre-ALU3, has empty slots at the end of the last cycle). The issue unit should not speculate whether there will be an empty slot at the end of the current cycle.
- No RAW or WAW hazards with active instructions (issued but not finished, or earlier not-issued instructions).
- If two instructions are issued in a cycle, you need to make sure that there are no WAW or WAR hazards between them.
- No WAR hazards with earlier not-issued instructions;
- For MEM instructions, all the source registers are ready at the end of the last cycle.
- The load instruction must wait until all the previous stores are issued.
- The stores must be issued in order.

ALU1: ALU1 handles the calculation of address for memory (lw, sw) instructions. ALU1 can fetch one instruction each cycle from the Pre-ALU1 queue (the oldest one), removes it from the Pre-ALU1 queue (at the beginning of the current cycle) and computes it. The instruction and its result will be written into the Pre-MEM queue at the end of the current cycle. Note that ALU1 starts execution even if the Pre-MEM queue is occupied (full) at the beginning of the current cycle. This is because MEM is guaranteed to consume (remove) the entry from the Pre-MEM queue before the end of the current cycle.

ALU2: ALU2 handles the calculation of all arithmetic (add, sub, addi) instructions. All the instructions take one cycle. The ALU2 can fetch one instruction each cycle from the Pre-ALU2 queue, removes it from the Pre-ALU2 queue (at the beginning of the current cycle) and compute it. The instruction and its result will be written into the Post-ALU2 queue at the end of the current cycle. Note that ALU2 starts execution even if the Post-ALU2 queue is occupied (full) at the beginning of the current cycle. This is because WB is guaranteed to consume (remove) the entry from the Post-ALU2 queue before the end of the current cycle.

ALU3: ALU3 handles the calculation of all logical (and, or, andi, ori, sll, sra) instructions. All the instructions take one cycle. The ALU3 can fetch one instruction each cycle from the Pre-ALU3 queue, removes it from the Pre-ALU3 queue (at the beginning of the current cycle) and compute it. The instruction and its result will be written into the Post-ALU3 queue at the end of the current cycle. Note that ALU3 starts execution even if the Post-ALU3 queue is occupied at the beginning of the current cycle.

This is because WB is guaranteed to remove the entry from the Post-ALU3 queue before the end of the current cycle.

MEM: The MEM unit handles lw and sw instructions. It reads from Pre-MEM queue. For lw instruction, MEM takes one cycle to read the data from memory. When a lw instruction finishes, the instruction with destination register id and the data will be written to the Post-MEM queue before the end of the current cycle. Note that MEM starts execution even if the Post-MEM queue is occupied (full) at the beginning of the current cycle. This is because WB is guaranteed to consume (remove) the entry from the Post-MEM queue before the end of the current cycle. For sw instruction, MEM also takes one cycle to finish (write the data to memory). When a sw instruction finishes, nothing would be sent to Post-MEM queue.

WB: WB unit can execute up to **three** writebacks in one cycle consisting of one from Post-MEM queue (if any), one from Post-ALU2 queue (if any), and one from Post-ALU3 queue (if any). It updates the Register File based on the content of Post-ALU2, Post-ALU3, and Post-MEM queue (lw). The update finishes before the end of the cycle. The new value will be available at the beginning of the next cycle.

2.2 Storage Locations (Queues/PC/Registers)

Program Counter (PC): Records the address of the next instruction to fetch (should be initialized to 256)

Pre-issue Queue: Pre-Issue Queue has 4 entries; each entry can store one instruction. The instructions are sorted by their program order, the entry 0 always contains the oldest and the entry 3 contains the newest.

Pre-ALU1 queue: The Pre-ALU1 queue has two entries. Each entry can store one memory (LW or SW) instruction with its operands. The queue is managed as **FIFO** (in-order) queue.

Pre-ALU2 queue: The Pre-ALU2 queue has one entry. Each entry can store one arithmetic (add, sub, addi) instruction with its operands.

Pre-ALU3 queue: The Pre-ALU3 queue has one entry. Each entry can store one logical (and, or, andi, ori, sll, sra) instruction with its operands.

Post-ALU2 queue: This queue has one entry. This entry can store one instruction with destination register id and the result.

Post-ALU3 queue: This queue has one entry. This entry can store one instruction with destination register id and the result.

Pre-MEM queue: The Pre-MEM queue has one entry. This entry can store one memory instruction (lw, sw) with its operands.

Post-MEM queue: Post-MEM queue has one entry that can store one lw instruction with destination register id and data.

Register File: There are 32 registers. Assume that there are sufficient read/write ports to support all kinds of read write operations from different functional units. Fetch unit reads Register File for branch instruction with register operands whereas Issue unit reads Register File for any non-branch instructions with register operands.

2.3 Notes on Pipelines

1. In reality, simulation continues until the pipeline is empty but for this project, the simulation finishes when the break instruction is fetched. In other words, the last clock cycle that you print in the simulation output is the one where break is fetched (shown in the "Executed" field). In other words, there may be unfinished instructions in the pipeline when you stop the simulation (see the sample_simulation.txt to see how it ended early).
2. No data forwarding.
3. No delay slot will be used for branch instructions.
4. Issue unit checks structural hazards (does not issue instructions unless its output buffers have empty slots at the beginning of the current cycle). However, four units (ALU1, ALU2, ALU3, and MEM) ignore structural hazards (starts execution even if the output buffer is full at the beginning of the current cycle since they are guaranteed to be removed before the end of the cycle).
5. Different instructions take different stages to finish.
    a. jal, break, beq, bne, blt: only IF
    b. sw: IF, Issue, ALU1, MEM
    c. lw: IF, Issue, ALU1, MEM, WB
    d. add, sub, addi: IF, Issue, ALU2, WB
    e. and, or, andi, ori, sll, sra: IF, Issue, ALU3, WB

3. Output Format

For each cycle, print the whole state of the processor and the memory **at the end of each cycle**. If any entry in a queue is empty, no content for that entry should be printed. The instruction should be printed as in Project 1.

20 hyphens and a new line

Cycle <value>:

<blank_line>

IF Unit:

<tab>Waiting: [instruction waiting for its operand]

<tab>Executed: [instruction executed in this cycle]

Pre-Issue Queue:

<tab>Entry 0: [instruction]

<tab>Entry 1: [instruction]

<tab>Entry 2: [instruction]

<tab>Entry 3: [instruction]

Pre-ALU1 Queue:

<tab>Entry 0: [instruction]

<tab>Entry 1: [instruction]

Pre-MEM Queue: [instruction]

Post-MEM Queue: [instruction]

Pre-ALU2 Queue: [instruction]

Post-ALU2 Queue: [instruction]

Pre-ALU3 Queue: [instruction]

Post-ALU3 Queue: [instruction]

< blank_line >

Registers

x00:< tab >< int(x0) >< tab >< int(x1) >..< tab >< int(x7) >

x08:< tab >< int(x8) >< tab >< int(x9) >..< tab >< int(x15) >

x16:< tab >< int(x16) >< tab >< int(x17) >..< tab >< int(x23) >

x24:< tab >< int(x24) >< tab >< int(x25) >..< tab >< int(x31) >

<blank line>

Data

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

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

Display all register/memory values in **signed decimal format** (e.g., 4 or -4). Immediate values in instructions should be preceded by a "#" symbol and printed in signed decimal format (e.g., #4 or #-4).

Because we will be using "**diff –w -B**" to check your output versus the expected outputs, please follow the output formatting. Mismatches will be treated as wrong output and will lead to score penalty.

4. Submission Policy

Please follow the submission policy outlined below. There can be up to **10% score penalty** based on the nature of submission policy violations.

1. **Please develop your project in one source file**. In other words, you cannot submit your project if you have designed it using multiple source files. **Please add ".txt" at the end of your filename**. Your file name must be Vsim (e.g., Vsim.c.txt or Vsim.cpp.txt or Vsim.java.txt or Vsim.py.txt). On top of the source file, please include the sentence: /* On my honor, I have neither given nor received unauthorized aid on this assignment */. **Please do not worry about if eLearning (Canvas) adds some tag to the file name when you make multiple submissions.**
2. Please test your submission. These are the exact steps we will follow too.
    - Download your submission from eLearning (ensures that your upload was successful).
    - Remove ".txt" extension (e.g., Vsim.c.txt should be renamed to Vsim.c)

- Login to **thunder**.cise.ufl.edu or **storm**.cise.ufl.edu using your Gatorlink login and password. Then you use **putty** and **winscp** or other tools to login. Ideally, if your program works on any Linux machine, it should work when we run them. However, if you get correct results on a Windows or MAC system, we may not get the same results when we run on storm or thunder. To avoid this headache and time waste, we strongly recommend that you to test your program on a CISE server (thunder or storm).
- Please compile to produce an executable named **Vsim**.
  - gcc Vsim.c –o Vsim or javac Vsim.java or g++ -std=c++17 Vsim.cpp –o Vsim
- Please do not print anything on screen.
- Please do not hardcode input filename, accept it as a command line option.
- Please hardcode your output filename as **simulation.txt**.
- Execute to generate simulation file and test with the correct/provided version.
  - ./Vsim inputfilename.txt **or** java Vsim inputfilename.txt **or** ./Vsim.py inputfilename.txt **or** python3 Vsim.py inputfilename.txt
  - diff –w –B simulation.txt sample_simulation.txt

3. *In previous years, there were many cases where output format was different, filename was different, command line arguments were different, or e-Learning submission was missing, etc. All of these led to un-necessary frustration and waste of time for the TA, instructor and students.* **Please use the exactly same commands as outlined above to avoid 10% score penalty.**

4. **You are not allowed to take or give any help in completing this project**. *Some students violated academic honesty (e.g., overlap with some online code). The students received "0" in the project, received an additional grade penalty, and their names were reported to UF Dean of Students Office (DSO). If your name is already in DSO for an earlier violation in this or another course, the penalty for second offence is determined by DSO. In the past, two students from my class were suspended for a semester due to repeat academic honesty violation (implies deportation for international students).*