Project 2

CDA 4102/5155: Fall 2021

Due: November 24, 11:30 PM **Total Points:** 10 points

You are not allowed to take or give help in completing this project. It is okay to reuse your Project 1 code to develop your Project 2. **No late submission will be accepted**. Please include the following sentence on top of your source file: **On my honor, I have neither given nor received 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 specified MIPS text (0/1) file and generate the cycle-by-cycle simulation of the MIPS code. It should also produce/print the contents of registers, queues, and memory data for each cycle.

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 or less than 32-bit instructions in the input file, 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 simulator in one (C, C++, Java or Python) **source file** to avoid the stress of combining multiple files before submission and making sure it still works correctly. Please follow the **Submission Policy** (see the last page) to avoid 10% score penalty. Your MIPS simulator (with executable name as **MIPSsim**) 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.

```
MIPSsim 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. It is recommended that you construct your own sample input files to further test your simulator.

1. Instruction Format

The instruction format remains the same as in Project 1.

2. Pipeline Description

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.

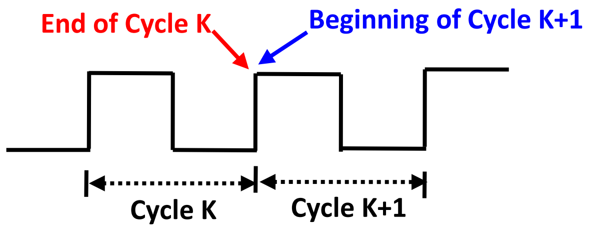

Figure 1: The end of the previous (last) clock cycle and the beginning of the current (next) clock cycle point to the same rising edge.

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. In the remainder of this section, we describe the functionality of each of the units/queues/registers/memory in detail.

2.1 Functional Units

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

- If the fetch unit is stalled at the end of the last cycle, no instruction can be fetched at the current cycle. The fetch unit can be stalled due to a branch instruction in the waiting stage.
- If there is no empty slot in the pre-issue queue (Buf1) at the end of the last cycle, no instruction can be fetched at the current cycle.

Normally, the fetch-decode operation can be finished in 1 cycle. The decoded instruction will be placed in the Pre-issue queue (Buf1) before the end of the current cycle. If a branch instruction is fetched, the fetch unit will try to read all the necessary operands (from Register File) to calculate the target address. If all the operands 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 operands are available. In other words, if operands are ready (or immediate target value) at the end of the last cycle, the branch does not introduce any penalty.

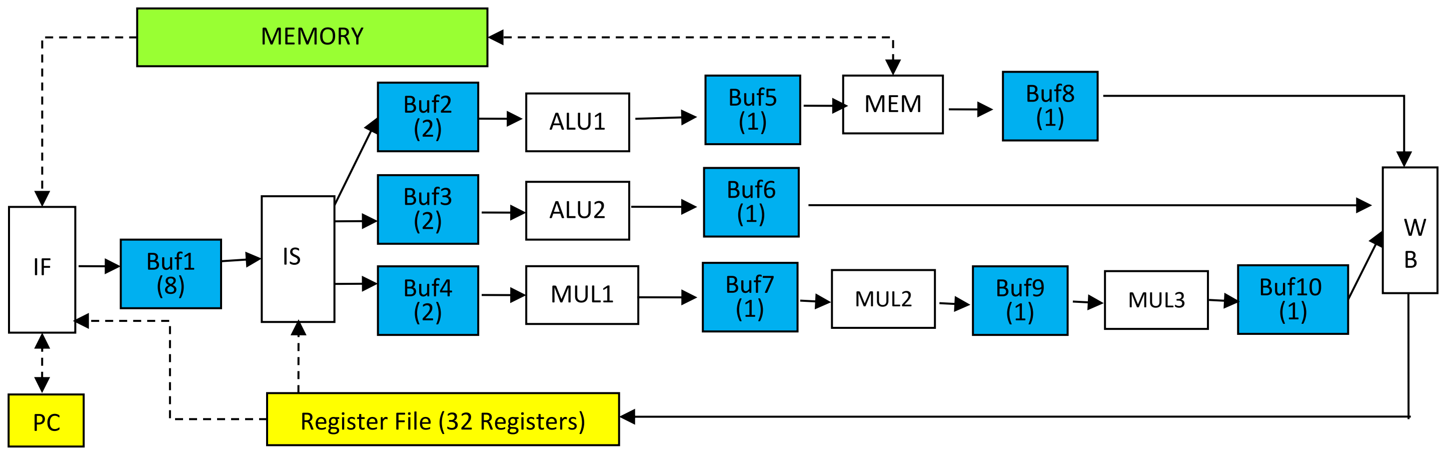

Figure 2: Pipelined Architecture (number of queue entries are shown in brackets)

There are four possible scenarios when a branch instruction (J, BEQ, BNE, BGTZ) is fetched along with another instruction. The branch can be the first, second, third, or the last instruction in the sequence (remember, up to four instructions can be fetched per cycle). When a branch instruction is fetched with its next (in-order) instruction (first three scenarios), the subsequent instructions will be discarded immediately (needs to be re-fetched again based on the branch outcome). When the branch is the last instruction in the sequence (last scenario), all four are decoded as usual. We have provided two fields when printing simulation output for branch instruction in IF unit. “Waiting” shows the branch instruction that is waiting for its operand to be ready. “Executed” shows the branch instruction that is executed in the current cycle.

Note that the register accesses are synchronized. The value read from register file in the current cycle is the value of the corresponding register at the end of the previous cycle. In other words, a unit **cannot** read 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 *Buf1*. However, it is important to note that we still need free entries in the *Buf1* at the end of the last cycle before the fetch unit fetches them in the current cycle, because the fetch cannot predict the types of instructions before fetching and decoding them.

Issue Unit (IS): Issue unit follows the basic Scoreboard algorithm to read operands from Register File and issues instructions when all the source operands are ready. It can issue up to six instructions **out-of-order** per cycle. Please note that it can issue up to two instructions to each of the output queues (Buf2, Buf3, and Buf4) in each cycle. Please see the description of these queues to understand what type of instructions can be issued to them. When an instruction is issued, it is removed from the Buf1 before the end of the current cycle. The issue unit searches from entry 0 to entry 7 (in that order) of Buf1 and issues instructions if:

- No RAW hazards.
- No structural hazards (the corresponding output queue should have empty slots at the beginning of the current cycle). The issue unit does not speculate whether there will be an empty slot at the end of the current cycle.
- No 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 LW/SW 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: This unit performs address calculation for LW and SW instructions. It can fetch one instruction each cycle from Buf2, removes it from Buf2 (at the beginning of the current cycle) and computes it. The computed address along with other relevant information will be written to Buf5 (for your simulation, write the same instruction in the queue). All of the instructions take one cycle. Note that ALU1 starts execution even if Buf5 is occupied (full) at the beginning of the current cycle. This is because MEM is guaranteed to consume (remove) the entry from Buf5 before the end of the current cycle.

MEM: The MEM unit handles LW and SW instructions. It reads one instruction from Buf5 and removes it from Buf5. 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 Buf8 before the end of the current cycle. Note that MEM starts execution even if Buf8 is occupied (full) at the beginning of the current cycle. This is because WB is guaranteed to consume (remove) the entry from the Buf8 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 written to Buf8.

ALU2: This unit handles the following instructions: ADD, SUB, AND, OR, SRL, SRA, ADDI, ANDI, and ORI. It can fetch one instruction each cycle from Buf3, removes it from Buf3 (at the beginning of the current cycle) and computes it. The computed result along with other relevant information will be written into Buf6. All the instructions take one cycle. Note that ALU2 starts execution even if Buf6 is occupied (full) at the beginning of the current cycle. This is because WB is guaranteed to consume (remove) the entry from Buf6 before the end of the current cycle.

MUL1: This unit executes the first stage of a pipelined MUL instruction. It can fetch one instruction each cycle from Buf4, removes it from Buf4 (at the beginning of the current cycle) and computes it. The partial result and destination information should be written into Buf7 (for your simulation, write the same instruction in the queue). MUL1 takes one cycle. Note that MUL1 starts execution even if Buf7 is occupied (full) at the beginning of the current cycle. This is because MUL2 is guaranteed to consume (remove) the entry from Buf7 before the end of the current cycle.

MUL2: This unit executes the second stage of a pipelined MUL instruction. It can fetch one instruction each cycle from Buf7, removes it from Buf7 (at the beginning of the current cycle) and computes it. The partial result and destination information should be written into Buf9 (for your simulation, write the same instruction in the queue). MUL2 takes one cycle. Note that MUL2 starts execution even if Buf9 is occupied (full) at the beginning of the current cycle. This is because MUL3 is guaranteed to consume (remove) the entry from Buf9 before the end of the current cycle.

MUL3: This unit executes the last stage of a pipelined MUL instruction. It can fetch one instruction each cycle from Buf9, removes it from Buf9 (at the beginning of the current cycle) and computes it. The result should be written into Buf10. MUL3 takes one cycle. Note that MUL3 starts execution even if Buf10 is occupied (full) at the beginning of the current cycle. This is because WB is guaranteed to consume (remove) the entry from Buf10 before the end of the current cycle.

WB: WB unit can execute up to three writebacks (up to one from each of its input queues) in one cycle, and removes them from its input queues. WB updates the Register File based on the content of its input queues. The update is finished before the end of the current cycle. The new values will be available at the beginning of the next cycle.

2.2 <u>Storage Locations (Queues/PC/Registers)</u>

Pre-Issue Queue (**Buf1):** It has 8 entries - each one can store one instruction. The instructions are sorted by their program order, the entry 0 always contains the oldest instruction and the entry 7 contains the newest instruction.

Pre-ALU1 Queue (Buf2): The issue unit can send only **LW** and **SW** instructions to this queue. It has two entries. Each entry can store one instruction with its operands. The queue is managed as **FIFO** (in-order) queue.

Pre-ALU2 Queue (Buf3): The issue unit issues the following instructions to this queue: (ADD, SUB, AND, OR, SRL, SRA, ADDI, ANDI, ORI, MFHI, MFLO). It has two entries. Each entry can store one instruction with its operands. The queue is managed as **FIFO** (in-order) queue.

Pre-MUL1 Queue (Buf4): The issue unit can send only **MUL** instruction to this queue. It has two entries. Each entry can store one instruction with its operands. The queue is managed as **FIFO** (in-order) queue.

Pre-MEM Queue (Buf5): This queue has one entry. This entry can store one memory instruction.

Pre-ALU2 Queue (Buf6): This queue has one entry. This entry can store the result and destination register.

Pre-MUL2 Queue (Buf7): This queue has one entry. This entry can store one multiply instruction.

Post-MEM Queue (Buf8): This queue has one entry. This entry contains load value and destination register.

Pre-MUL3 Queue (Buf9): This queue has one entry. This entry can store one multiply instruction.

Post-MUL3 Queue (Buf10): This queue has one entry. This entry can store the multiplication result and destination register.

Program Counter (PC): It records the address of the next instruction to fetch. It should be initialized to **260**.

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, ALU1/ALU2/MEM/MUL1/MUL2/MUL3 ignores structural hazards (starts execution even if the output buffer is full at the beginning of the current cycle).
5. Different instructions take different stages to be finished.
   a. J, BEQ, BNE, BGTZ, BREAK: only IF.
   b. SW: IF, IS, ALU2, MEM.
   c. LW: IF, IS, ALU2, MEM, WB.
   d. MUL: IF, IS, MUL1, MUL2, MUL3, WB.
   e. Other instructions: IF, IS, ALU1, WB.

3. Output format

For each cycle, please print the state of the processor and the memory **at the end of each cycle**. If any entry in 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:

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

```
<tab>Executed: [branch or BREAK instruction executed in this cycle]
Buf1:
<tab>Entry 0: [instruction]
<tab>Entry 1: [instruction]
<tab>Entry 2: [instruction]
<tab>Entry 3: [instruction]
<tab>Entry 4: [instruction]
<tab>Entry 5: [instruction]
<tab>Entry 6: [instruction]
<tab>Entry 7: [instruction]
Buf2:
<tab>Entry 0: [instruction]
<tab>Entry 1: [instruction]
Buf3:
<tab>Entry 0: [instruction]
<tab>Entry 1: [instruction]
Buf4:
<tab>Entry 0: [instruction]
<tab>Entry 1: [instruction]
Buf5: [instruction]
Buf6: [result, destination]
Buf7: [instruction]
Buf8: [result, destination]
Buf9: [instruction]
Buf10: [result, destination]
< 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>
Data
< firstDataAddress >:< tab >< display 8 data words as integers with tabs in between >
..... < continue until the last data word >
```

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 submit only one source file. **Please add ".txt" at the end of your filename**. Your file name must be MIPSsim (e.g., MIPSsim.c.txt or MIPSsim.cpp.txt, MIPSsim.java.txt, or MIPSsim.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 Canvas (ensures your upload was successful).
   - Remove ".txt" extension (e.g., MIPSsim.c.txt should be renamed to MIPSsim.c)
   - o Login to any CISE linux machine (e.g., 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 should test your program on thunder or storm server.
   - Please compile to produce an executable named **MIPSsim**.
     - gcc MIPSsim.c –o MIPSsim **or** javac MIPSsim.java **or**
       g++ MIPSsim.cpp –o MIPSsim **or** g++ -std=c++17 MIPSsim.cpp –o MIPSsim
   - 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 correct/provided one
     - ./MIPSsim inputfilename.txt **or** java MIPSsim inputfilename.txt or
       ./MIPSsim.py inputfilename.txt **or** python3 MIPSsim.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 TA, instructor and students.* **Please use the exactly same commands as outlined above to avoid 10% score penalty.**

4. *In the previous years, some students violated academic honesty. We were able to establish violation in several cases - those students received "0" in the project, and their names were reported to Dean of Students Office (DSO). If your name is already in DSO for violation in 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).*