Project 2
Computer Architecture: Fall 2017
Due Date: 11.59PM, Dec. 26

In this project you will create a simulator for a pipelined processor. Your simulator should be capable of loading a specified MIPS binary file and generate the cycle-by-cycle simulation of the MIPS binary code. It should also produce/print the contents of registers, queues, and memory data for each cycle. **Exception/interrupt handling during the simulation is not required**.

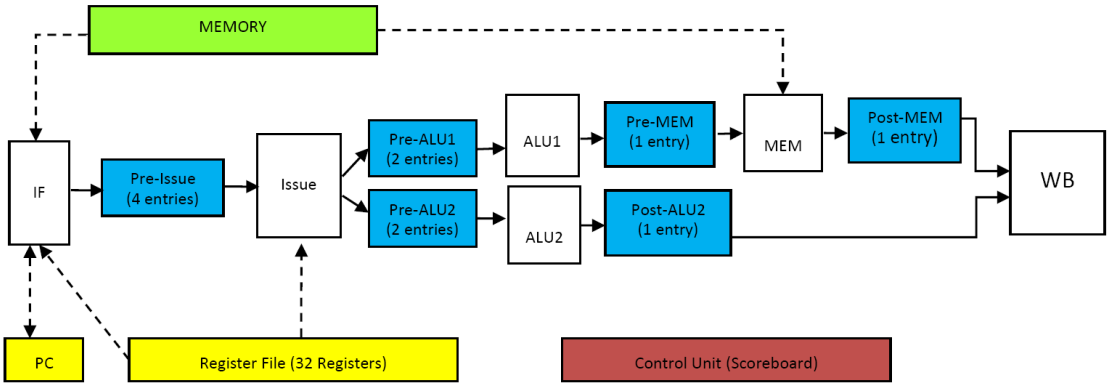
You can use either Java or C/C++ to develop your simulator. Please follow the **Submission Policy** which is at the end of this document to submit your source files. Your MIPS simulator (with executable name as **MIPSsim**) should accept an input file (inputfilename.txt) in the following command format and produce output file (simulation.txt) that contains the simulation trace.

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 with which to further test your simulator.

Instruction Format: The instruction format remains exactly the same as in Project 1. Please refer to the document of Project 1.

Pipeline Description:



The entire pipeline is synchronized by a single clock signal. 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. We use the terms "the end of cycle" and "the beginning of cycle" interchangeably in following discussion. Both of them refer to the rising edge of the clock signal, i.e., the end of the previous cycle implies the beginning of the next cycle.

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 (J, JR, BEQ, BLTZ, BGTZ) 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.

All branch instructions, BREAK instruction and NOP 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.

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.

Issue Unit: 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 **two** instruction **out-of-order** per cycle. It can send at most **one** load or store (LW or SW) instructions per cycle to the Pre-ALU1 queue, and at most **one** non load/store (except LW or SW) instructions per cycle to the Pre-ALU2 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 or Pre-ALU2 has empty slots at the end of the last 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 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.

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 two entries. Each entry can store one ALU (any instruction except LW or SW) instruction with its operands. The queue is managed as **FIFO** (in-order) queue.

ALU1: ALU1 handles the calculation of address for memory (LW and SW) instructions. ALU1 can fetch one instruction each cycle from the Pre-ALU1 queue, 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 non-memory instructions. All the instructions take one cycle. The ALU 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.

Post-ALU2 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.

MEM Unit: 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.

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

WB Unit: WB unit can execute up to **two** writebacks in one cycle consisting of at most one from Post-MEM queue and at most one from Post-ALU2 queue. It updates the Register File based on the content of both Post-ALU2 Queue (any instruction except LW or SW) 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.

PC: It records the address of the next instruction to fetch. It should be set to 256 at the initialization.

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.

Notes on Pipelines:

1. 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 Instruction" field).
2. No data forwarding.
3. No delay slot will be used for branch instructions.
4. Different Instructions takes different stages to finish.
    a. NOP, Branch, BREAK: only IF;
    b. SW: IF, Issue, ALU1, MEM;
    c. LW: IF, Issue, ALU1, MEM, WB;
    d. Other instructions: IF, Issue, ALU2, WB.

Output format

For each cycle, you should output 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: [instruction waiting for its operand]

<tab>Executed Instruction: [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:

<tab>Entry 0: [instruction]

<tab>Entry 1: [instruction]

Post-ALU2 Queue: [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>

Data

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

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

No need to handle exceptions of any kind. 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.

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 or MIPSsim.java.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 */.

2. Please test your submission. These are the exact steps we will follow too.

- Please compile to produce an executable named **MIPSsim**.
  gcc MIPSsim.c –o MIPSsim **or** javac MIPSsim.java **or** g++ 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
  diff –w –B simulation.txt sample_simulation.txt