

Project 2

Computer Architecture: Spring 2017

Due Date: May 25, 11:30 PM, Email to: homeworkecnu@163.com

You are not allowed to take or give help in completing this project. **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 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.

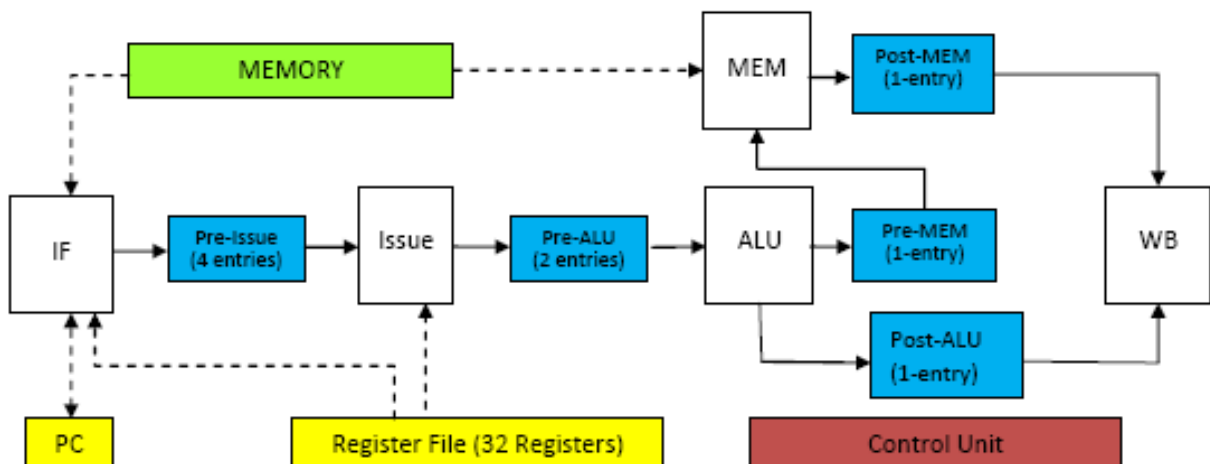
You can use either Java or C/C++ to develop your simulator. Please follow the Submission Policy (see the last page) to avoid any penalty. Your MIPS simulator (with executable name as MIPSsim) should accept an input file (inputfilename.txt) in the following command format and produce output file (simulation.txt) that contains the simulation trace.

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.

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. 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” in the 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 instructions 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 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 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 last cycle, the branch does not introduce any penalty.

There are two possible scenarios when a branch instruction (J, BEQ, 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.

We have provided two fields when printing simulation output for branch instruction in IF unit. “Waiting Instruction” shows the branch instruction that is waiting for its operand to be ready. “Executed Instruction” shows the branch instruction that is executed in that cycle.

Note that the register accesses are **synchronized**. The value read from register file in 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 and BREAK instruction will not be written to Pre-issue queue. However, 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 one can store one instruction. The instructions are sorted by their program order, the entry 0 always contains the oldest instruction 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 up to two instructions out-of-order per cycle. 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-ALU has empty slots at the end of 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 last cycle.
- The load instruction must wait until all the previous stores are issued.
- The stores must be issued in order.

Pre-ALU queue: The Pre-ALU queue has two entries. Each entry can store one instruction with its operands. The queue is managed as FIFO (in-order) queue.

ALU: ALU handles the calculation of address for memory (LW and SW) instructions and all non-memory instructions. ALU can fetch one instruction each cycle from the Pre-ALU queue, removes it from the Pre-ALU queue (at the beginning of the current cycle) and computes it. If this is a memory instruction, then the computed address along with other relevant information will be written into the Pre-MEM queue. If this is a non-memory instruction, then the computed result along with other relevant information will be written into the Post-ALU queue. All the instructions take one cycle. Note that ALU starts execution even if the Post-MEM (or Post-ALU) queue is occupied (full) at the beginning of the current cycle. This is because MEM (or WB) is guaranteed to consume (remove) the entry from the Post-MEM (or Post-ALU) queue before the end of the current cycle.

Post-ALU 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 (one from Post-ALU and another from Post-Mem) in one cycle. It updates the Register File based on the content of both Post-ALU Queue (any instruction except LW or SW) and Post-MEM Queue (LW). The update is finished before the end of the cycle. The new value will be available at the beginning of next cycle.

PC: It records the address of the next instruction to fetch. It should be set to 128 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. Fetch unit reads Register File for branch instruction with register operands whereas Issue unit read Register File for any non-branch instructions with register operands.

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 Instruction” field).
2. No data forwarding.
3. No delay slot will be used for branch instructions.
4. Different instructions takes different stages to be finished.
 - a. Branch, BREAK: only IF;
 - b. SW: IF, Issue, ALU, MEM;
 - c. LW: IF, Issue, ALU, MEM, WB;
 - d. Other instructions: IF, Issue, ALU, 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 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: [branch instruction waiting for its operand]

<tab>Executed Instruction: [branch 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-ALU Queue:

<tab>Entry 0: [instruction]

<tab>Entry 1: [instruction]

Pre-MEM Queue: [instruction]

Post-MEM Queue: [instruction]

Post-ALU Queue: [instruction]

< blank_line >

Registers

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

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

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

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

<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.

Sample Data


The course project webpage contains the following sample programs/files to test your simulator.

- sample.txt : This is the input to your program.
- disassembly.txt : This is for your reference only.
- simulation.txt : This is what your program should output as simulation trace.

Please follow the submission policy outlined below. There can be significant **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.

- Download your submission from eLearning (ensures your upload was successful).
- Remove “.txt” extension (e.g., MIPSsim.c.txt should be renamed to MIPSsim.c)
- 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 command lines option.
- Execute to generate simulation file and test with the correct one
 - ./MIPSsim inputfilename.txt
 - diff -w -B generated_simulation.txt correct_simulation.txt