Project 2
Computer Architecture: Spring 2022, SEI, ECNU
Due Date: 12/16/2016 11:55 PM

For this project you will create a simulator for a pipelined processor. Your simulator should be capable of loading a specified MIPS binary (text) file and generate the cycle-by-cycle simulation of the MIPS code. It should also produce/print the contents of registers, buffers, and memory data for each cycle.

You do not have to implement any exception/interrupt handling during simulation for this project. You can use C, C++ or Java to implement your project. In any case, TA should be able to build and run your simulator in linux environment (e.g., Ubuntu). Please provide (on top of your source code as comments) any special notes/assumptions you made about the project that the TA should take into consideration during compilation/simulation. If you have multiple source files:

- Please provide a **Makefile** which will allow TA to correctly build your project on CISE linux machines.
- Please submit the **source files** necessary to build and run your project, in a single jar/tar file. Please do not submit any executables or intermediate files such as .exe or .class
- You should make a separate directory to work on your project (e.g., P2) and then use jar -cvf P2.jar * to create your jar file.

Your MIPS simulator (with executable name as MIPSsim) should provide the following option to users:

MIPSsim inputfilename outputfilename

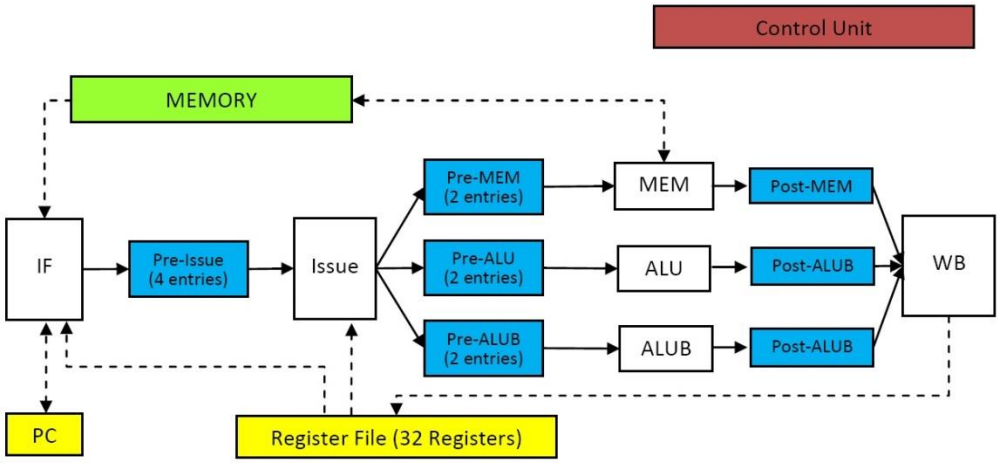
where,

* Inputfilename – The file name of the text version binary input file (e.g., sample.txt).
* Outputfilename – The file name for printing the simulation output (e.g., output.txt).

The disassembly output is not required. The executable name must be MIPSsim.

You are strongly recommended to test your program before submission with following commands. The last command ensures that the file produced by your simulator and the sample file provided by us are identical.

> Compile to build MIPSsim
> Simulate → MIPSsim sample.txt output.txt
> diff –b output.txt simulation.txt



Instruction Format:

The instruction format remains exactly the same as the project 1. Please refer to the documents of Project 1.

Pipeline Description:

This is a 4-stage pipeline. The entire pipeline is synchronized by a single clock signal. The white boxes represent the functional units, the blue boxes represent buffers/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/buffers/memory in detail. We use the terms "the end of some cycle" and "the beginning of some cycle" interchangeably in following discussion. Both of them refer to the rising edge of the clock signal.

Instruction Fetch (IF):

Instruction Fetch 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 previous 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 buffer at the end of the previous cycle, no instruction can be fetched at the current cycle.
- If there is only one empty slot in the Pre-issue buffer at the end of the previous cycle, only one 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 buffer 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 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 previous cycle, the branch does not introduce any penalty.

If one branch instruction (J, JR, BEQ, BLTZ) is fetched with its next (in-order) instruction, the next instruction will be discarded immediately (needs to be re-fetched again based on the branch outcome). **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, we stop the simulation at this cycle. In the case when BREAK is fetched as the second instruction following a branch instruction, we still consider BREAK is fetched and stop the simulation.

All branch instructions, BREAK instruction and NOP instruction will not be written to Pre-issue buffer. However, it is important to note that we still need free entries in the pre-issue buffer at the end of the previous cycle before the fetch unit fetches them, because the fetch cannot predict the decode result without fetching them.

Pre-issue Buffer:

Pre-Issue Buffer 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 issue instructions. It can issue at most **two** instruction **out-of-order** per cycle. When an instruction is issued, it is removed from the Pre-issue Buffer before the end of current cycle. The issue unit searches from entry 0 to entry 3 (in that order) of Pre-issue buffer and issues instructions if:

- No structural hazards (the corresponding queue, e.g., Pre-ALU has empty slots at the end of the previous cycle);
- No WAW hazards with active instructions (issued but not finished, or earlier not-issued instructions);
- No WAR hazards with earlier not-issued instructions;
- For MEM instructions, all the source registers are ready at the end of the previous cycle.
- The load instruction must wait until all the previous stores are issued.
- The stores must be issued in order.

As mentioned earlier, the register accesses are synchronized. For example, for the following two instructions, the second instruction should NOT be issued at the same cycle during which the first instruction writes back its result. The new value R10 will be only available at the end of this cycle.

```
SLL R10, R1, #2
LW R3, 172(R10)
```

Pre-ALU queue:

The Pre-ALU queue has two entries. Each entry can store one instruction with its 2 operands. The queue is managed as **FIFO** (in-order) queue.

ALU:

The ALU handles the calculation all non-memory instructions except SLL, SRL, SRA and MUL. All the instructions will take one cycle in ALU. In other words, if the Pre-ALU queue is not empty at the end of cycle N, ALU processes the topmost instruction from the Pre-ALU queue in cycle N+1. The topmost instruction is removed from the Pre-ALU queue before the end of cycle N+1. (Therefore the issue unit will see at least one empty slot in Pre-ALU queue in the beginning of cycle N+2.)

The processed instruction and its result will be written into the Post-ALU buffer at the end of cycle N+1. Note that this operation will be performed regardless of whether Post-ALU is occupied at the beginning of cycle N+1. In other words, there is no need to check for structural hazard in Post-ALU buffer.

Post-ALU buffer:

The Post-ALU buffer has one entry. This entry can store one instruction with destination register id and the result.

Pre-ALUB queue:
The Pre-ALUB queue has two entries. Each entry can store one instruction with its 2 operands. The queue is managed as **FIFO** (in-order) queue.

ALUB:
The ALUB handles SLL, SRL, SRA and MUL. Due to the hardware complexity, it takes two cycles to process SLL, SRL, SRA and MUL in ALUB. In other words, if the Pre-ALUB queue is not empty at the end of cycle N, ALUB processes the topmost instruction, X, from the Pre-ALU queue in cycle N+1 and N+2. The topmost instruction X is removed from the Pre-ALU queue before the end of cycle N+2. (Therefore the issue unit will see at least one empty slot in Pre-ALU queue in the beginning of cycle N+3.) Please note that X remains the topmost instruction at the end of cycle N+1, while being processed.

The processed instruction and its result will be written into the Post-ALUB buffer at the end of cycle N+2. Note that this operation will be performed regardless of whether the Post-ALUB is occupied at the beginning of cycle N+2.

Post-ALUB buffer:
The Post-ALUB buffer has one entry. This entry can store one instruction with destination register id and the result.

MEM Unit:
The MEM unit handles LW and SW instructions in Pre-MEM queue. For LW instruction, it takes one cycle to finish. When a LW instruction finishes, the instruction with destination register id and the data will be written to the Post-MEM buffer before the end of cycle. Note that this operation will be performed regardless of whether the Post-MEM is occupied at the beginning of this cycle. For SW instruction, it takes one cycle to write the data to memory. When a SW instruction finishes, nothing would be sent to Post-MEM buffer. When a MEM instruction finishes execution at MEM unit, it is removed from the Pre-MEM queue before the end of cycle.

Post-MEM buffer:
The Post-MEM buffer has one entry. This entry can store one instruction with destination register id and data.

WB Unit
WB unit can execute up to **three** writebacks in one cycle. It updates the Register File based on the content of Post-ALU Buffer, Post-ALUB Buffer, and Post-MEM Buffer. The update is finished before the end of the cycle. The new value will be available at the beginning of next cycle.

PC:
Program Counter (PC) records the address of the next instruction to be fetched. It should be 64 at the beginning of the simulation.

Register File:

There are 32 registers. Assume that there are sufficient read/write ports to support all kinds of read write operations from different functions units.

Notes on Pipelines:

1. The simulation stops at the cycle, when a BREAK instruction is fetched. **Simulation output after that cycle is not required.**

2. No data forwarding.
3. No delay slot will be used for branch instructions.
4. Different Instructions takes different stages to be finished.
    a. NOP, Branch, BREAK: only IF;
    b. SW: IF, Issue, MEM;
    c. LW: IF, Issue, MEM, WB;
    d. SLL, SRL ,SRA and MUL: IF, Issue, ALUB, WB
    e. Other instructions: IF, Issue, ALU, WB.

As mentioned earlier, you do not have to handle any exceptions. For example, extra test cases will not try to execute data in data segment (as instructions), or load/store data within instruction segment. Similarly, we will not provide invalid binary values, and so on.

Output format

For each cycle, you should output the whole state of the processor and memory at the end of each cycle. If any entry in buffer/queue is empty, no content for that entry should be printed.

The instruction should be printed as in project 1. Note that, the register printing format has changed to print eight registers per line (instead of 16 per line in project 1). Please refer to the sample output file, if there is any confusion in the format.

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 Buffer:
<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]
Post-ALU Buffer:[instruction]
Pre-ALUB Queue:
<tab>Entry 0: [instruction]

<tab>Entry 1: [instruction]
Post-ALUB Buffer:[instruction]
Pre-MEM Queue:
<tab>Entry 0: [instruction]
<tab>Entry 1: [instruction]
Post-MEM Buffer:[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 >

Because we will be using "**diff -b**" to check your output versus ours, you need to exactly follow the output formatting. TA may not be able to debug in case of any mismatch. In other words, mismatches can be treated as wrong output.

Sample Data

The course project email contains the following sample programs/files to test your disassembler/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.

Grading Policy

The total score is 10 points.

1. source files cannot be compiled or executed -10;
2. For the available test case you can get up to 6 points (60% of the score).
   no simulation file is generated or your program crashes or your program doesn't stop in reasonable time -6;
   if the simulation file contains any significant error
   1) if correct up to 50% of the cycles, -2;
   2) if correct up to 30% of the cycles, -4;
   3) otherwise -6;
3. (For the extra test cases):
   no simulation file is generated or your program crashes or your program doesn't stop in reasonable time -4;
   if the simulation file contains any significant error
   1) if correct up to 50% of the cycles -2;

2) otherwise -4;

During re-grading, a student is allowed to make minor changes to their code to reclaim some of the lost points. A minor change is equivalent to changing a symbol such as changing “a > b” to “a <b”. Every minor change will cost you 10% of the lost score. In other words, if you make more than 10 minor changes during regarding, you will not get back any points.