Project 2

CDA 5155: Spring 2014

Due: November 27, 11:30 PM **Total Points:** 12 points

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 main 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, buffers, and memory data for each cycle. Exception/interrupt handling during the simulation is NOT required. No need to handle interrupts or 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.

You can use C, C++ or Java to develop your simulator. 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.

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.

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. **Figure 1** shows the pipeline. The white boxes represent the functional units, the blue boxes represent buffers 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/buffers/registers/memory 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):

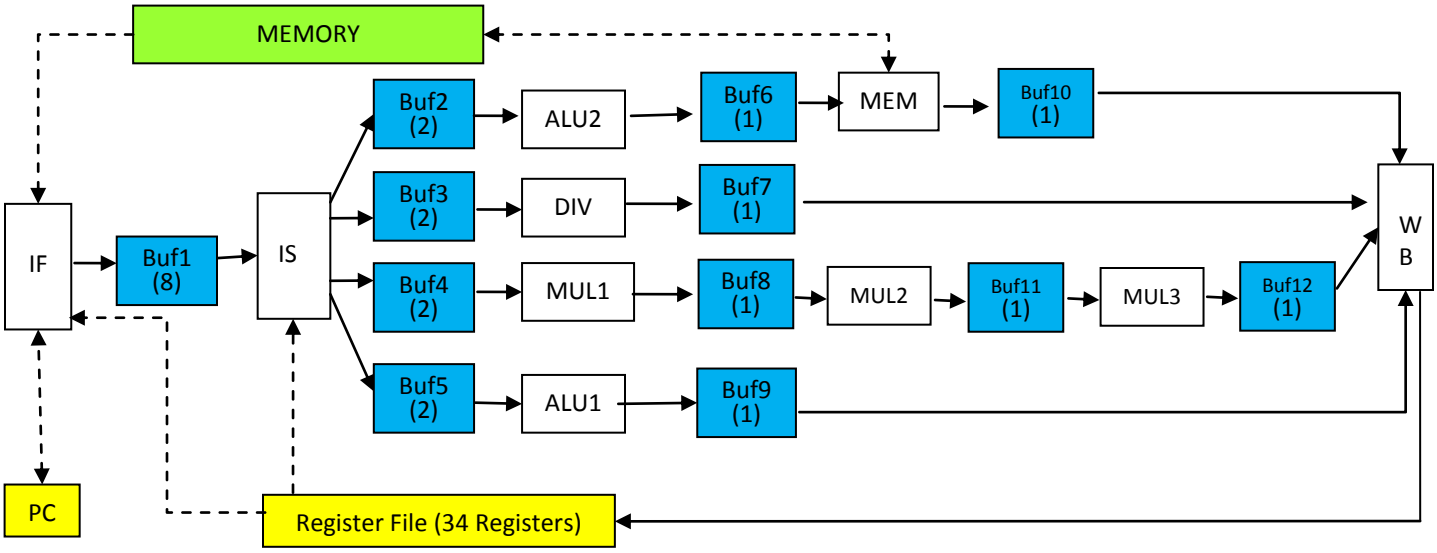
Instruction Fetch/Decode unit can **fetch and decode** at most **four** instructions at each cycle (in program order). It should check all of the following conditions before it can fetch further instructions.

- 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 buffer (**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 buffer (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 1: Pipelined Architecture



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 eight instructions **out-of-order** per cycle. Please note that it can issue up to two instructions to each of the output buffers (Buf2, Buf3, Buf4 and Buf5) in each cycle. Please see the description of these buffers 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 current cycle. The issue unit searches from entry 0 to entry 7 (in that order) of Buf1 and issues instructions if:

- No RAW hazards. Please note that some of the operands may be implicit. For example, MFLO and MFHI have to wait for implicit source of LO and HI, respectively. Similarly, DIV instruction uses implicit HI and LO destination registers. Likewise, MULT instruction uses LO as destination register [**Although MULT instruction uses both HI and LO as destination registers in MIPS instruction-set manual, in this project we assume that MULT uses only LO as the destination register. Please note that HI should not be affected due to a MULT instruction. Moreover, we will provide small operand values such that the multiplication result fits in LO (no overflow)].**
- No structural hazards (the corresponding output buffer should have empty slots);
- No WAW hazards with active instructions (issued but not finished, or earlier not-issued instructions). Again, please consider instructions (such as MULT and DIV) with implicit destination registers. For example, if a DIV instruction is ahead in the pipeline, MULT instruction should not be issued (or vice versa).
- 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 of 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-Issue buffer (**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.

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

Division Buffer (Buf3): The issue unit can send only **DIV** instruction to this buffer. It has two entries. Each entry can store one instruction with its operands. The buffer is managed as **FIFO** (in-order) queue.

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

ALU Buffer (Buf5): The issue unit issues all other instructions (ADD, SUB, AND, OR, SRL, SRA, ADDI, ANDI, ORI, MFHI, MFLO) to this buffer. It has two entries. Each entry can store one instruction with its operands. The buffer is managed as **FIFO** (in-order) queue.

Buf6: This buffer has one entry. This entry can store one memory instruction.

Buf7: This buffer has one entry. This entry can store remainder and quotient.

Buf8: This buffer has one entry. This entry can store one multiply instruction.

Buf9: This buffer has one entry. This entry can store destination register id and the result.

Buf10: This buffer has one entry. This entry contain load value and destination register.

Buf11: This buffer has one entry. This entry can store one multiply instruction.

Buf12: This buffer has one entry. This entry can store multiplication result.

ALU2: 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 Buf6 (for your simulation, write the same instruction in the buffer). All of 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 MEM (or WB) is guaranteed to consume (remove) the entry from Buf6 before the end of the current cycle.

DIV: This unit executes DIV instruction. It can fetch one instruction each cycle from Buf3, removes it from Buf3 (at the beginning of the current cycle) and computes it. The results i.e., remainder and quotient should be written into Buf7. DIV takes four cycles. Since DIV is a multi-cycle (unpipelined) unit, a DIV instruction will keep it busy for four cycles. Note that DIV starts execution even if Buf7 is occupied (full) at the beginning of the current cycle. This is because WB is guaranteed to consume (remove) the entry from Buf7 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 Buf8 (for your simulation, write the same instruction in the buffer). MUL1 takes one cycle. Note that MUL1 starts execution even if Buf8 is occupied (full) at the beginning of the current cycle. This is because MUL2 is guaranteed to consume (remove) the entry from Buf8 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 Buf8, removes it from Buf8 (at the beginning of the current cycle) and computes it. The partial result and destination information should be written into Buf11 (for your simulation, write the same instruction in the buffer). MUL2 takes one cycle. Note that MUL2 starts execution even if Buf11 is occupied (full) at the beginning of the current cycle. This is because MUL3 is guaranteed to consume (remove) the entry from Buf11 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 Buf11, removes it from Buf11 (at the beginning of the current cycle) and computes it. The result should be written into Buf12. MUL3 takes one cycle. Note that MUL3 starts execution even if Buf12 is occupied (full) at the beginning of the current cycle. This is because WB is guaranteed to consume (remove) the entry from Buf12 before the end of the current cycle.

ALU1: This unit handles all the remaining (ADD, SUB, AND, OR, SRL, SRA, ADDI, ANDI, ORI, MFHI, MFLO) instructions. It can fetch one instruction each cycle from Buf5, removes it from Buf5 (at the beginning of the current cycle) and computes it. The computed result along with other relevant information will be written into Buf9. All the instructions take one cycle. Note that ALU1 starts execution even if Buf9 is occupied (full) at the beginning of the current cycle. This is because WB is guaranteed to consume (remove) the entry from Buf9 before the end of the current cycle.

MEM Unit: The MEM unit handles LW and SW instructions. It reads one instruction from Buf6 and removes it from Buf6. 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 Buf10 before the end of the current cycle. Note that MEM 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 the Buf10 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 Buf10.

WB: WB unit can execute up to four writebacks (up to one from each of its input buffers) in one cycle, and removes them from its input buffers. WB updates the Register File based on the content of its input buffers. The update is finished before the end of the cycle. The new values 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 34 registers. The first 32 are general purpose registers, followed by HI and LO 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.

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).
2. No data forwarding.
3. No delay slot will be used for branch instructions.
4. 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. DIV: IF, IS, DIV, WB.
    e. MULT: IF, IS, MUL1, MUL2, MUL3, WB.
    f. Other instructions: IF, IS, ALU1, WB.

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 buffer 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:
<tab>Entry 0: [instruction]
<tab>Entry 1: [instruction]
Buf6: [instruction]
Buf7: [remainder, quotient]
Buf8: [instruction]
Buf9: [result, destination]
Buf10: [result, destination]
Buf11: [instruction]
Buf12: [result]
< 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) >
HI: < tab ><value>
LO: < tab ><value>
<blank line>
Data
< firstDataAddress >:< tab >< display 8 data words as integers with tabs in between >
..... < continue until the last data word >

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.
    - Download your submission from Canvas (ensures your upload was successful).
    - Remove ".txt" extension (e.g., MIPSsim.c.txt should be renamed to MIPSsim.c)
    - Login to **thunder.cise.ufl.edu**. If you don't have a CISE account, go to https://www.cise.ufl.edu/help/account#apply and apply for one CISE class account. Then you use **putty** and **winscp** or other tools to login.
    - 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
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.**

You are not allowed to take or give any help in completing this project. *In the previous years, some students violated academic honesty (giving help or taking help in completing this project). We were able to establish violation in several cases - those students received "0" in the project and their names were reported to UF Ethics office. This year we would also impose one additional letter grade penalty. Someone could potentially lose two letter grade points (e.g., "A-" grade to "B" grade) – one for getting 0 score in the project and then another grade point penalty on top of it. Moreover, the names of the students will also be reported to UF 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).*