Lab 2: MIPS 5 Stage Pipeline Simulator

In this Lab assignment, you will implement a cycle-accurate simulator for a 5-stage pipelined MIPS processor in C++. The simulator supports a subset of the MIPS instruction set and should model the execution of each instruction cycle by cycle.

An example MIPS program is provided for the simulator as a text file “imem.txt”, which is used to initialize the Instruction Memory. Each line of the file corresponds to a Byte stored in the Instruction Memory in binary format, with the first line at address 0, the next line at address 1, and so on. Four contiguous lines correspond to one whole instruction. Note that the words stored in memory are in “Big-Endian” format, meaning that the most significant byte is stored first.

The Data Memory is initialized using the “dmem.txt” file. The format of the stored words is the same as the Instruction Memory. As with the instruction memory, the data memory addresses also begin at 0 and increment by one in each line.

The instructions that the simulator supports and their encodings are shown in Table 1. Note that all instructions, except for “halt”, exist in the MIPS ISA. The *MIPS Green Sheet* defines the semantics of each instruction.

For the purposes of this lab, we use the *bne* instruction, rather than the beq instruction from Lab1. bne jumps to the branch address if $R[rs]$!= $R[rt]$ and jumps to PC+4 otherwise, i.e., if $R[rs] = R[rt]$. This is to make writing loops slightly easier.

Name	Format Type	Opcode (Hex)	Func (Hex)
addu	R-Type	00	21
subu	R-Type	00	23
lw	I-Type	23	-
sw	I-Type	2B	-
bne	I-Type	05	-
halt	J-Type	3F	-

Table 1. Instruction encodings for a reduced MIPS ISA

Pipeline Structure

Your MIPS pipeline has the following 5 stages:

1. **Fetch (IF):** fetches an instruction from the instruction memory. Updates PC.
2. **Decode (ID/RF):** reads from the register RF and generates control signals required in subsequent stages. In addition, branches are resolved in this stage by checking for the branch condition and computing the effective address.
3. **Execute (EX):** performs an ALU operation.
4. **Memory (MEM):** loads or stores a 32-bit word from data memory.
5. **Writeback (WB):** writes back data to the RF.

Your simulator can make use of the same RF, IMEM, and DMEM classes that you used for Lab1. Complete implementations of these classes are provided for you in the skeleton code.

*Note that we have not defined an ALU class since, for this lab, the ALU is simple and only needs to perform adds and subtracts.

Each pipeline stage takes inputs from flip-flops. The input flip-flops for each pipeline stage are described in the tables below.

IF Stage Input Flip-Flops

Flip-Flop Name	Bit-width	Functionality
PC	32	Current value of PC
Nop	1	If set, IF stage performs a nop

IF/ID Stage Flip-Flops

Flip-Flop Name	Bit-width	Functionality
Instr	32	32b instruction read from Imem
Nop	1	If set, ID stage performs a nop

ID/EXE Stage Flip-Flops

Flip-Flop Name	Bit-width	Functionality
Read-data1, Read_data2	32	32b data values read from RF
Rs, Rt	5	Addresses of source registers rs, rt. Note these are defined for both R-type and I-type instructions
Wrt_reg_addr	5	Address of the instruction's destination register. Don't care is the instruction doesn't update RF
Alu_op	1	Set for addu, lw, sw; unset for subu

Is_I_type	1	Set if the instruction is I-type
Wrt_enable	1	Set if instruction updates RF
Rd_mem, Wr_mem	1	Rd_mem set for lw and wrt_mem set for sw instructions.
Nop	1	If set, EXE stage performs a nop

EXE/MEM Stage Flip-Flops

Flip-Flop Name	Bit-width	Functionality
ALU_result	32	32b ALU result, don't care for beq
Store_data	32	32b value to be stored in DMEM for sw instruction. Don't care otherwise
Rs, Rt	5	Addresses of source registers rs, rt. Note these are defined for both R-type and I-type instructions
Wrt_reg_addr	5	Address of the instruction's destination register. Don't care is the instruction doesn't update RF
Wrt_enable	1	Set if instruction updates RF
Rd_mem, Wr_mem	1	Rd_mem set for lw and wrt_mem set for sw instructions.
Nop	1	If set, EXE stage performs a nop

WB Stage Input Flip-Flops

Flip-Flop Name	Bit-width	Functionality
Wrt_data	32	32b value to be written back to RF. Don't care for sw and beq.
Rs, Rt	5	Addresses of source registers rs, rt. Note these are defined for both R-type and I-type instructions
Wrt_reg_addr	5	Address of the instruction's destination register. Don't care is the instruction doesn't update RF
Wrt_enable	1	Set if instruction updates RF
Nop	1	If set, EXE stage performs a nop

Dealing with Hazards

Your processor must deal with two types of hazards.

1. RAW Hazards: RAW hazards are dealt with using either only forwarding (if possible) or, if not, using stalling + forwarding. You must follow the mechanisms described in Lecture to deal with RAW hazards.

2. Control Flow Hazards: You will assume that branch conditions are resolved in the ID/RF stage of the pipeline. Your processor deals with bne instructions as follows:

   a) Branches are always assumed to be NOT TAKEN. That is, when a bne is fetched in the IF stage, the PC is speculatively updated as PC+4.

   b) Branch conditions are resolved in the ID/RF stage.
   To make your life easier, we will ensure that every bne instruction has no RAW dependency with its previous two instructions. In other words, you do NOT have to deal with RAW hazards for branches!

   c) Two operations are performed in the ID/RF stage: (i) Read_data1 and Read_data2 are compared to determine the branch outcome; (ii) the effective branch address is computed.

   d) If the branch is NOT TAKEN, execution proceeds normally. However, if the branch is TAKEN, the speculatively fetched instruction from PC+4 is quashed in its ID/RF stage using the nop bit and the next instruction is fetched from the effective branch address. Execution now proceeds normally.

The nop bit

The **nop** bit for any stage indicates whether it is performing a valid operation in the current clock cycle. The nop bit for the IF stage is initialized to 0 and for all other stages is initialized to 1. (This is because in the first clock cycle, only the IF stage performs a valid operation.)

In the absence of hazards, the value of the nop bit for a stage in the current clock cycle is equal to the nop bit of the prior stage in the previous clock cycle.

However, the nop bit is also used to implement stalls that result from a RAW hazard or to squash speculatively fetched instructions if the branch condition evaluates to TAKEN. See slides for more details on implementing stalls and squashing instructions.

The HALT Instruction

The halt instruction is a “custom” instruction we introduced so you know when to stop the simulation. When a HALT instruction is fetched in the IF stage at cycle N, the nop bit of the IF stage in the next clock cycle (cycle N+1) is set to 1 and subsequently stays at 1. The nop bit of the ID/RF stage is set to 1 in cycle N+1 and subsequently stays at 1. The nop bit of the EX stage is set to 1 in cycle N+2 and subsequently stays at 1. The nop bit of the MEM stage is set to 1 in cycle N+3 and subsequently stays at 1. The nop bit of the WB stage is set to 1 in cycle N+4 and subsequently stays at 1.

At the end of each clock cycle the simulator checks to see if the nop bit of each stage is 1. If so, the simulation halts. Note that this logic is already implemented in the skeleton code provided to you.

What to Output?

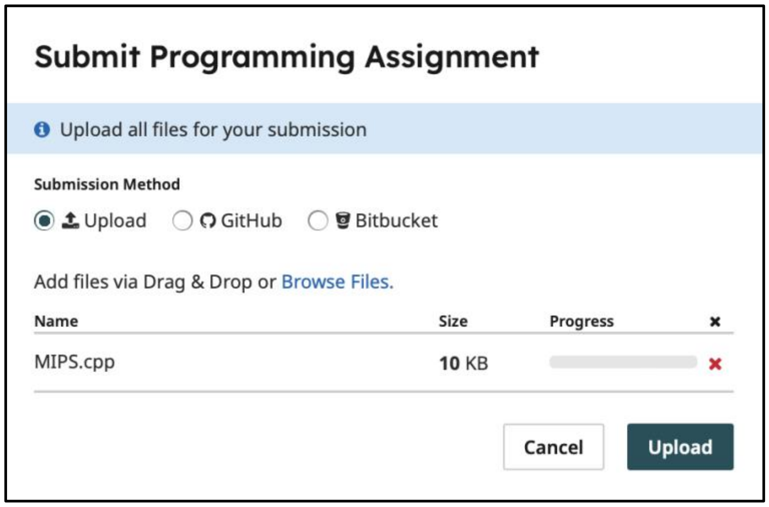
Make sure that the architectural state is updated correctly after the execution of **each** instruction. The architectural state consists of the Program Counter (PC), the Register File (RF), and the Data Memory (DataMem). We will check the correctness of the architectural state after *each* instruction. Your simulator will output the values of all flip-flops at the end of each clock cycle.

Specifically, the OutputRF() function is called at the end of each iteration of the while loop, and will add the new state of the Register File to "RFresult.txt". Therefore, at the end of the program execution "RFresult.txt" contains all the intermediate states of the Register File. Once the program terminates, the OutputDataMem() function will write the final state of the Data Memory to "dmemresult.txt". These functions have been implemented for you. Do not modify them.
(Note: **You should delete the "RFresult.txt" and "dmemresult.txt" files before re-executing your program**, otherwise the new results will append to the previous results.)

Your Assignment:

1. We have provided the skeleton code in the file MIPS_pipeline.cpp.
    a. A Makefile has been provided for you to compile the source code (do not modify the Makefile)
    b. We will be compiling your code with: `g++ version 11.3.0`. **Please make sure that your code is compatible with g++. See 4. for more detail about compiling.**
    c. You can compile your design by typing "make" which will create an executable called "MIPS_pipeline". Run the executable by calling: "./MIPS_pipeline**"**.
    d. As mentioned above, calling the binary "./MIPS_pipeline" expects both an "imem.txt" and "dmem.txt" file in the same directory and produces three output files (RFresults.txt, dmemresults.txt, and stateresult.txt).
    e. We have provided an "imem.txt" and "dmem.txt" files containing a sample program and initialized data. These files must be in the same directory as the source code.
    f. We have provided the correct output results for the provided test case in the "expected_results" directory. Make sure you pass this test case before submitting your code. However, we encourage you to write your own MIPS programs and check your design for different cases.

1. You will be submitting your assignments on Gradescope. **You must upload ONLY one file on Gradescope: "MIPS_pipeline.cpp". Do not zip the file or upload any other file.**



Submit Programming Assignment

Upload all files for your submission

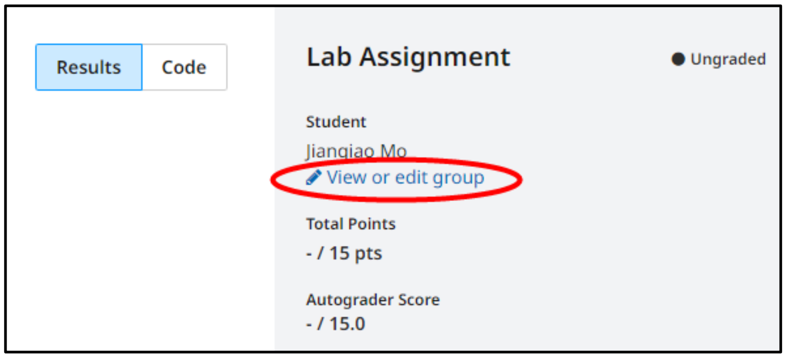
Submission Method

[✓ Upload] GitHub Bitbucket

Add files via Drag & Drop or Browse Files.

Name	Size	Progress
MIPS.cpp	10 KB	

1. You are allowed to work with one other student as a two-person group. If you choose to work with another student, only one of you needs to submit the files on Gradescope **and you must add your group partner after submission.** The screenshot below shows how to add another student to your submission after you have submitted the assignment.



Lab Assignment

Ungraded

Student

Jiangiao Mo

View or edit group

Total Points

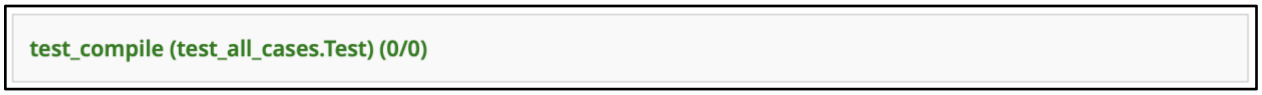
- / 15 pts

Autograder Score

- / 15.0

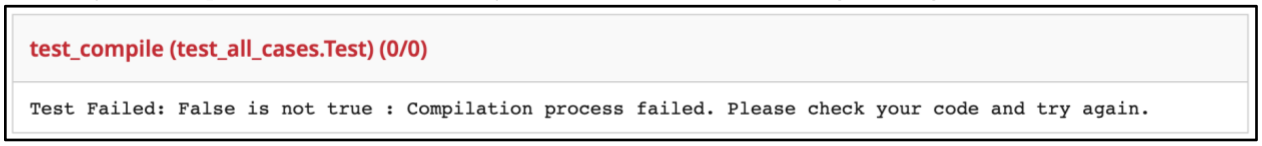
You can form your group (add a group member) after you submit your assignment.

2. When you submit your code to Gradescope, we will check that your code compiles without errors. If your code compiles, you will see the following message as the output:



```
test_compile (test_all_cases.Test) (0/0)
```

In case your compilation is unsuccessful, you will receive the following message



```
test_compile (test_all_cases.Test) (0/0)

Test Failed: False is not true : Compilation process failed. Please check your code and try again.
```

You may submit your code as many times as you like before the submission deadline, but please make sure you pass the test_compile case. Otherwise, you will receive a zero. All other test cases will be hidden until the submission deadline has passed. We will be testing your code against a series of MIPS programs that test the instructions supported in Table 1. We expect your uploaded file to be called **exactly "MIPS_pipeline.cpp"**, and your binary executable **must** output

“RFresult.txt” and “dmemresult.txt” for the test cases to run correctly. (They are already specified in the code, so don’t change them).

3. On Gradescope, there is a Similarity Test so don’t copy answers from other groups.
4. We set up 20 test cases and you earn 0.5 points by passing each.

Due Date:

Friday Oct 13, 2023, 23:59 Eastern Time.