Lab 1: Pipeline MIPS Processor

Released: Oct 9th 2018

Due: Oct 30th 2018 Midnight [Instructions on how to upload your code to NYUClasses will be provided in a separate document]

This lab is to be performed individually. We will carefully monitor all submissions for plagiarism and any instance of plagiarism will be reported to administration.

In this Lab assignment, you will implement an 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 with cycle accuracy.

The MIPS program is provided to the simulator as a text file “imem.txt” file 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 a 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.

Instruction	Format	OpCode (hex)	Funct. (hex)
addu	R-Type (ALU)	00	21
subu	R-Type (ALU)	00	23
lw	I-Type (Memory)	23	-
sw	I-Type (Memory)	2B	-
beq	I-Type (Control)	04	-
halt	Custom instruction	FF	

Pipeline Structure

Your MIPS pipeline has the following 5 stages:

1. **Fetch (IF):** fetches an instruction from 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 Lab0. 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 stages 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

ID/RF Stage Input Flip-Flops

Flip-Flop Name	Bit-width	Functionality
Instr	32	32-bit instruction read from IMEM
nop	1	If set, ID/RF stage performs a nop

EX Stage Input Flip-Flops

Flip-Flop Name	Bit-width	Functionality
Read_data1, Read_data2	32	32-bit data values read from RF
Imm	16	16-bit immediate for I-Type instructions. Don't care for R-type instructions
Rs, Rt	5	Addresses of source registers rs, rt. Note that 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 if the instruction does not update RF

alu_op	1	Set for addu, lw, sw; unset for subu
is_I_type	1	Set if the instruction is an i-type instruction
wrt_enable	1	Set if the instruction updates RF
rd_mem, wrt_mem	1	rd_mem is set for lw instruction and wrt_mem for sw instructions
nop	1	If set, EX stage performs a nop

MEM Stage Input Flip-Flops

Flip-Flop Name	Bit-width	Functionality
ALUresult	32	32-bit ALU result. Don't care for beq instructions
Store_data	32	32-bit value to be stored in DMEM for sw instruction. Don't care otherwise
Rs, Rt	5	Addresses of source registers rs, rt. Note that 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 if the instruction does not update RF
wrt_enable	1	Set if the instruction updates RF
rd_mem, wrt_mem	1	rd_mem is set for lw instruction and wrt_mem for sw instructions
nop	1	If set, MEM stage performs a nop

WB Stage Input Flip-Flops

Flip-Flop Name	Bit-width	Functionality
Wrt_data	32	32-bit data to be written back to RF. Don't care for sw and beq instructions
Rs, Rt	5	Addresses of source registers rs, rt. Note that 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 if the instruction does not update RF
wrt_enable	1	Set if the instruction updates RF
nop	1	If set, MEM 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 6 to deal 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 beq instructions as follows:
    a. Branches are always assumed to be NOT TAKEN. That is, when a beq 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, will ensure that every beq 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 stall that result from a RAW hazard *or* to quash speculatively fetched instructions if the branch condition evaluates to TAKEN. See Lecture 6 slides for more details on implementing stalls and quashing 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 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

Your simulator will output the values of all flip-flops at the end of each clock cycle. Further, when the simulation terminates, the simulator also outputs the state of the RF and Dmem. The skeleton code already prints out everything that your simulator needs to output. Do not modify this code.

Testbenches and Grading

Test inputs for your simulator are in the form of “imem.txt” and “dmem.txt” files, and the expected outputs are in the form of “RFresult.txt”, “dmemresult.txt” and “stateresult.txt” files.

To assist you in checking your code, we will provide sample input and output files to you. However, your code will be graded on our own test inputs.

You are encouraged to develop your code in steps. A rough time frame for how long each step would take is given below.

1. Step 1: your simulator should be able to handle addu, subu, lw and sw instructions *without* any RAW hazards. (1 Week)
2. Step 2: in addition, your simulator should be able to handle addu, subu, lw and sw instructions *with* RAW hazards. (1 Week)
3. Step 3: finally, enhance your simulator to handle *beq* instructions.

The grading scheme that we will use to test your code is:

- Code compiles and executes without crashing on all test inputs [**10 Points**]
- Code handles inputs with only addu, subu, lw and sw instructions *without* any RAW hazards [**30 Points**]
- Code handles inputs with only addu, subu, lw and sw instructions *with* RAW hazards. [**30 Points**]
- Code handles all test inputs including those with beq instructions. [**30 Points**]