

# ECS 154B Lab 4, Spring 2016

## Due by 11:55 PM on May 25, 2016

### Via SmartSite

## Objectives

- Build and test a pipelined MIPS CPU that implements a subset of the MIPS instruction set.
- Handle hazards through forwarding and stalling.

## Description

In this lab, you will use Logisim to design and test a pipelined MIPS CPU that implements a subset of the MIPS instruction set, along with data hazard resolution. Make sure to use the given circuit for this assignment, as the Register File included implements internal forwarding, while previous versions do not.

## Details

Your CPU must implement all instructions from Lab 3. Specifically, you must implement:

AND, ANDI, ADD, ADDI, OR, ORI, SUB, SLT, SLTU\*, XOR, LW, SW, BEQ, J, JAL, JR, SLL, SRL

**As with the previous labs, the instructions for SLTU are incorrect in the given MIPS PDF.** The instructions should read:

If  $rs < rt$  with an unsigned comparison, put 1 into rd. Otherwise put 0 into rd.

The control signals for the ALU are identical to Lab 3, and are reposted here for your convenience:

Operation	ALUCtl3	ALUCtl2	ALUCtl1	ALUCtl0
OR	0	0	0	0
SLTU	0	0	0	1
SLT	0	0	1	0
ADD	0	0	1	1
SUB	0	1	0	0
XOR	0	1	0	1
AND	0	1	1	0
...				
SLR	1	1	1	0
SLL	1	1	1	1

Your CPU should not have any branch delay slots, and should use a branch not taken strategy for branch prediction. You may implement your control signals using any method you prefer. You can use combinational logic, microcode, or a combination of the two.

## Hazards

As you have learned in lecture, pipelining a CPU introduces the possibilities of hazards. Your CPU must be able to handle **all possible** hazards. Your CPU must use forwarding where possible, and resort to stalling only where necessary. Below is a subset of the possible hazards you may encounter. This means that, while all possible hazards may not be listed here, your CPU must still be able to handle all possible hazards.

1. **Read After Write (RAW)** hazards. Your CPU must perform forwarding on both ALU inputs to prevent Read After Write hazards. Your CPU must handle the hazards in the following code segments without stalling.

ADD \$4, \$5, \$6	ADD \$4, \$5, \$6	ADD \$4, \$5, \$6	ADD \$4, \$5, \$6
ADD \$7, \$4, \$4	ADD \$8, \$9, \$10	ADD \$8, \$9, \$10	LW \$8, 0(\$4)
	ADD \$7, \$4, \$4	ADD \$7, \$4, \$8	

2. **Load Use** hazards. Your CPU must handle load-use hazards through stalling and forwarding. You may only stall when necessary. If you stall when forwarding would work, you will lose points.

LW \$8, 0(\$4)	LW \$8, 0(\$4)
ADD \$10, \$9, \$8	ADD \$4, \$5, \$6
	ADD \$10, \$9, \$8

3. **Store Word (SW)** hazards. Your CPU must handle all Read After Write hazards associated with SW using forwarding. You may need to stall in certain cases, as well.

ADD \$4, \$5, \$6	LW \$4, 0(\$0)	LW \$4, 0(\$0)
SW \$4, 0(\$4)	SW \$4, 10(\$0)	SW \$5, 10(\$4)

4. **Control Flow** hazards. Read After Write hazards can also occur with the BEQ and JR instructions. The following hazards must be solved with forwarding.

ADD \$4, \$5, \$6	ADD \$4, \$5, \$6	LW \$10, 0(\$0)	LW \$10, 0(\$0)
ADD \$8, \$9, \$10	ADD \$8, \$9, \$10	ADD \$4, \$5, \$6	ADD \$4, \$5, \$6
BEQ \$0, \$4, BranchAddr	JR \$4	ADD \$8, \$9, \$10	ADD \$8, \$9, \$10
		BEQ \$0, \$10, BranchAddr	JR \$10

The following hazards should be resolved with stalls.

ADD \$4, \$5, \$6	ADD \$4, \$5, \$6	LW \$10, 0(\$0)
BEQ \$0, \$4, BranchAddr	JR \$4	ADD \$4, \$5, \$6
		BEQ \$0, \$10, BranchAddr

LW \$10, 0(\$0)	LW \$10, 0(\$0)	LW \$10, 0(\$0)
ADD \$4, \$5, \$6	BEQ \$0, \$10, BranchAddr	JR \$10
JR \$10		

## Branch Prediction

In order to reduce the number of stall cycles in our CPU, we will be using a branch not taken prediction strategy. This means that, if a branch is taken, we will need to provide hardware to squash the incorrectly predicted instructions. For example:

BEQ \$0, \$0, BranchAddr	
ADD \$1, \$1, \$1	This instruction must be squashed.

The number of instructions that must be squashed is dependent on where in the pipeline you evaluate your branch condition. Jump instructions can be viewed as branches that are always taken and therefore are able to have their “branch” conditions evaluated in the Decode stage.

## Submission

You have 4 slip days to use with this lab. Submit your circuit on SmartSite, along with a README file. In the README, include:

- Name of Partner 1, student ID number
- Name of Partner 2, student ID number
- Whether or not your circuit works, and if it does not, what it does not work on.
- Any difficulties you had.
- Anything else you feel that the TAs should know.

## Grading

Your implementation will be tested and graded as follows:

To Be Graded	Percentage of Lab Grade	Description
<b>noForwarding.mps</b>	30%	A basic test of your pipelined CPU. No forwarding or stalling is required. Contains no control flow instructions.
<b>forwarding.mps</b>	30%	A basic test of your forwarding logic. No stalling is needed. Contains no control flow instructions. This also means that it does not test forwarding to control flow instructions.
<b>finalTestmps</b>	30%	Anything and everything possible. Requires both forwarding, stalling, and squashing. Contains control flow instructions.
Interactive Grading	10%	Making sure you understand pipelining, and making sure you did not leech off your partner.

For each test file, the TAs will look at the contents of your registers and memory to see if your CPU is performing correctly. **noForwarding.lst** and **forwarding.lst** do not have any infinite loops to terminate themselves with, so you will have to step through those programs manually.

## Hints

- The pipelined CPU diagram in the book should be used as a guide, and not a goal. It does not show everything you need to do to implement all of the instructions. Also, it is very possible to improve on their design.
- In previous editions of the book, there was an error in the forwarding logic. The current edition may contain the same error, so double check the logic.
- Build, test, and debug in parts. Build a basic pipelined CPU first. After confirming that it works, add in the forwarding logic and test again. Finally, add in the logic to stall and squash instructions. By doing work in parts, you minimize the amount of time spent debugging, and maximize the amount of points gained if you do not finish.
- After finishing a portion of the lab, save that implementation as a separate circuit so that you have something to go back to in case you need to restart.