ECS 154B Lab 4, Spring 2017
Due by 11:59 PM on May 24, 2017
Via Canvas

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:

- Data instructions: `ADD, ADDI, AND, ANDI, NOR, OR, ORI, SLT, SLTI, SLTU, SUB, XOR, SLL, SRL`
- Memory access instructions: `LW, SW`
- Control flow instructions: `BEQ, J, JAL, JR`

Note that the instructions for `SLTU` are incorrect. It 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**
`XOR`	0	0	0	0
`SLTU`	0	0	0	1
`SLT`	0	0	1	0
`AND`	0	0	1	1
`NOR`	0	1	0	0
`SUB`	0	1	0	1
`OR`	0	1	1	0
`ADD`	0	1	1	0
`SRL`	1	0	0	0
`SLL`	1	0	0	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.

`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.

Grading

Your implementation will be tested and graded as follows:

Name	Percentage of Lab Grade	Description
basic.mps	15%	A basic test of your pipelined CPU. No forwarding or stalling is required. Contains no control flow instructions.
forwarding.mps	15%	A 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.
final.mps	20%	Anything and everything possible. Requires both forwarding, stalling, and squashing. Contains control flow instructions.
Interactive Grading	50%	Ensuring that you understand the lab and pipelining, and that your partner did not do everything.

For each test file, the grader will look at the contents of your registers and memory to check if your CPU is performing correctly. Partial credit is at the grader's discretion. **basic.mps** and **forwarding.mps** do not have any infinite loops to terminate themselves with, so you will have to step through those programs manually.

Submission

Warning: read the submission instructions carefully. Failure to adhere to the instructions will result in a loss of points.

- Upload to Canvas the zip/tar of your .circ file along with a README file that contains:
  - The names of you and your partner.
  - Any difficulties you had.
  - Anything that doesn't work correctly and why.
  - Anything you feel that the graders should know.
- **Copy and paste the README into the text submission box when you are submitting your assignment**, as well.
- Only one partner should submit the assignment.
- You may submit your assignment as many times as you want.
- You have 4 slip days to use for this assignment.

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.