Lab 10 : MIPS Pipelining

Name:

Chenjie Luo

Sign the following statement:

On my honor, as an Aggie, I have neither given nor received unauthorized aid on this academic work

Chenjie Luo

1 Questions

1. Show or list all of the data dependencies in the following code fragment. Make sure that you clearly indicate which instructions and registers are involved in each dependency:

```
1        add $8, $5, $5
2        sub $8, $8, $10
3        lw $6, 4($8)
4        add $4, $6, $8
```

Instruction1 and instruction2 have dependency $8
Instruction2 and instruction3 have dependency $8
instruction3 and instruction4 have dependency $6 and $8
Instruction4 and instruction2 have dependency $8

2. The following code has pipeline hazard(s) in it.

```
1        haz:    add $5, $0, $0
2        lw $10, 1000($20)
3        addiu $20, $20, -4
4        addu $5, $5, $10
5        bne $20, $0, haz
```

Assuming you have a processor with the following specifications:

- Five pipeline stages as seen in the lecture notes (IF, ID, AL, ME, WB).
- Branch comparing done in the third stage.
- Cannot fetch instruction until branch comparison is done.
- Forwarding logic exists.
- Register write and register read can happen in the same clock cycle.

Describe the dependencies that need to be resolved, and show how the code will actually be executed (incorporating any stalls) so as to resolve the identified problems.

Instruction 2 and instruction 4 have dependency on $10

Since forwarding logic exists, only in fourth cycle the instruction will turn to noop. And because PC will not update until branch comparison is done, each time when approaching bne, there will be two stalls after bne.

2 Objective

The objective of this lab is to build and test the pipelined processor to test your program, you should use the provided testbench. Your code should run the provided testbench without errors. Ensure that your program is clearly commented. Once all tests in the testbench pass make sure to demo to your TA.

3 Processor

In this lab, you will build the MIPS pipelined processor using Verilog. You will also simulate some programs running on your pipelined processor. By the end of this lab, you should thoroughly understand the internal operation of the MIPS pipelined processor.

The lab will involve several steps. First, you are given a template of the control module in `PipelinedControl.v` that needs to be completed. You will also need modules from previous labs as well as the provided `PipelinedProc` (found in `PipelinedProc.v`) modules. Run the simulation in `PipelinedProcTest.v` to test your processor on real programs. Additionally, you will need to complete the `Hazard` modules (given in Hazard.v).

4 Hazards

In this lab, you will need to design a module which handles both control and data hazards. Since a pipelined processor has several instructions in execution at once, consecutive instructions could cause a hazard. Consider the following code segment:

```
1       lw $t0, 0($t2)
2       add $a0, $a0, $t0
3       add $a2, $a2, $t0
4       beq $a0, $0, Lend
5       addi $a1, $a2, 4
```

The instruction on line 2 requires the instruction on line 1 to have completed before it can be executed. Furthermore, the instruction on line 4 needs the instruction on line 2 to be completed before it can be executed. These are both data hazards. The instruction on line 5 cannot start until the processor knows whether the branch will be taken or not. This is a control hazard.

Both hazards will be handled in your hazard module. You can handle data hazards through setting the `Bubble` signal to 1 for the correct number of clock cycles. In the above code, 3 bubbles must be inserted between instructions 1 and 2, while only 2 are needed between 3 and 4 (to handle the dependancy between instructions 2 and 4). Your hazard module is given the list of registers both read and written by the current instruction (registers will be 0 if not used). Your module will need to hold `Bubble` high for the appropriate amount of time. You may find it useful to remember which register is being written for the past 3 instructions, and see if the current instruction needs to read any of

them. In such a case, `Bubble` will be 1, otherwise it will be 0. Remember, when you set `Bubble`, the current instruction will not executed until `Bubble` is set to 0.

Control hazards can be handled by setting the `PCWrite` and `IFWrite` signals. When a Branch instruction is decoded, `PCWrite` must be immediately set to 0, and held to 0 for 2 more clock cycles. For jump instructions, `PCWrite` must be immediately set to 0, and held to 0 for 1 more clock cycle. `IFWrite` must be 0 for one additional clock cycle than `PCWrite`, unless the instruction is a branch instruction, and the branch is **not** taken.

5 Deliverables

Once all tests have successfully completed and your design synthesizes, demo your progress to the TA.

Please turn-in the following:

- All of your verilog modules.
- Filled out PDF Form.