ECS 154B – Winter 2014 – Lab 4 Due by 11:55 PM on Sunday March 2, 2014

Objectives

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

Description

In this lab you will use Logisim to build and test a pipelined MIPS CPU that implements a subset of MIPS instrunction set alongwith data hazard resolution. You can reuse the ALU and Register File from Lab3. You must support forwarding and hazard detection as outlined later in the lab description.

Details

You can use separate Instruction and Data Memory. Your CPU must execute all required instructions from Lab 2 **except** beq and J. Specifically: add, sub, addi, lw, sw, and, or, nor, andi, ori, slt, sll, srl. Section 4.6 and 4.7 of the textbook will be very helpful to design your pipelined CPU and Forwarding Logic respectively. To test your CPU you can generate a MIF file to initialize your instruction memory by executing /home/cs154b/bin/mips2mif (you may want to add this directory to your PATH) as you have been doing in the previous labs. For this lab, you may implement the CPU control any way you wish. A completely combinational, completely microcode, or a mixture are all possibilities. Document your control design in your README file at a high level (you dont need to include equations unless you feel they improve readability).

Hazards

As you have learned in lecture, pipelining a CPU introduces the possibility of hazards. Your CPU must handle some of the hazards covered in lecture.

1. Read before Write Hazards. Your CPU must perform forwarding on both ALU inputs to prevent read before write data hazards. For example, your CPU must handle the hazards present in the following three code listings without stalling.

`add $4, $5, $6`	`add $4, $5, $6`	`add $4, $5, $6`
`add $7, $4, $4`	`add $8, $9, $10`	`add $4, $4, $4`
	`add $5, $4, $4`	`add $8, $4, $4`

   If you don't understand how the above code examples are different, review the textbook and class notes.

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. Example code follows.

`lw $4, 32($0)`	`lw $4, 32($0)`
`add $8, $4, $4`	`or $5, $8, $9`
	`add $8, $4, $4`

3. `sw` Forwarding and Stalling. Your CPU must handle the read before write data hazards associated with the sw instruction. No stalling is allowed for the following two cases as forwarding can resolve them completely. Both of these hazards can be resolved with a single mux and a single new control signal (`MEM/WB_ForwardMem`) in combination with the methods used to handle previous hazards. Describe in your README how you implemented the forwarding in these cases.

`add $4, $5, $6`	`lw  $4, 16($6)`
`sw  $4, 24($6)`	`sw  $4, 24($6)`

   You must handle the following case with stalling due to the load-use hazard as described in 2 above.

`lw $4, 24($6)`
`sw  $5, 24($4)`

   You **DO NOT** have to handle the following case where the loaded value is used in the address computation and stored on the immediately following cycle. You can assume this situation will never occur in the code I use to test your CPU.

`lw $4, 24($6)`
`sw  $4, 24($4)`

Grading

Pipelined design is 70 percent credit for this Lab. Each of the hazard resolution to implement will carry its own credit. If your CPU doesn't solve one of the above mentioned hazards correctly, you will lose credit. In order to facilitate the grading of your lab, ”LABEL” the following signals in your circuit so that the TA can identify any of the following signals in your assignment. These signals are specific to the 5 pipeline stages.

1. Fetch:
   (a) `IF_PC[31..0]`: The address of the instruction being fetched.
   (b) `IF_Instruction[31..0]`: Machine code for the instruction being fetched.
2. Decode:
   (a) `IF/ID_Hazard`: High when your hazard unit detects a load-use hazard.
3. Execute:
   (a) `ID/EX_ALUCtl[3..0]`: Four-bit input to control ALU operation.
   (b) `ID/EX_ALUResult[31..0]`: Result of the ALU operation.
   (c) `ID/EX_Branch, ID/EX_RegDst, ID/EX_ALUSrc, ID/EX_Zero`
4. Memory:
   (a) `EX/MEM_ForwardMem`: High when the value to write to Data Memory is a forwarded value.
   (b) `EX/MEM_MemRead, EX/MEM_MemWrite`
5. WriteBack:
   (a) `MEM/WB_RegWrite, MEM/WB_MemtoReg`

Make sure to include the following in your README:

- names and SIDs
- An indication of any difficulties you faced, varying in length from one sentence for no problems to multiple paragraphs if you faced serious difficulties. For example, if you have a known issue remaining in your final submission, any details you can give me about what the problem is will help me be able to give you partial credit.
- A description of how you implemented your main Control Unit and ALU Control Unit.
- The details on how you implemented sw forwarding. Include the conditions you test to determine when forward must occur.

Hints

- Design and test in steps. For example, you could follow this process:
  - Start with a basic pipeline that does no hazard detection or forwarding and test to make sure it works. Add `nop` instructions as appropriate to remove any hazards at this stage. `mips2mif` does not recognize the keyword `nop`. A good instruction to use in place is `sll $0, $0, 0`.
  - Then, add the logic for forwarding and test it.
  - Finally, add the logic for hazard detection and handling and test it.
- Think about the hardware you are creating before trying it out. The text is necessarily vague and leaves out details, so do not simply copy the figures and expect your CPU to work. Additionally, you will implement instructions not covered in the text, so your lab will have hardware not shown in the text.