

ECS 154B – Spring 2009 – Lab 3

Due by midnight May 20th

Objectives

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

Description

In this lab you will use Quartus to build a pipelined CPU. To test your CPU, you will run assembly language programs that you write and simulate the operation in Quartus. You will be given an ALU to help you out and an assembler that will generate a file to initialize your instruction memory. You may implement the CPU control in any way you choose. You must support forwarding and hazard detection as outlined later in the lab description. During interactive grading you will be given an initialization file to run that will use all the required instructions.

Details

It is important for this lab that you select "APEX II" as the device family when creating your project and logical blocks. If you forget to select APEX II when you create your project, you can change it by selecting Assignments→Device. Any generated logical blocks must also use the APEX II device family. You can safely ignore the message that APEX II is not a recommended family.

Your program counter (PC) must be 32 bits and it must address bytes. Any other choice will result in a loss of points. When generating your instruction memory, make its depth 128 words.

You will be given an ALU (available on the course website) to start your lab. This is the same ALU from Lab 2 and you must use this ALU. Simply download the tar file, extract the files into your project directory, and insert the ALU like any other logic block. The ALU will be listed under the Project folder in the Insert Symbol dialog box.

The ALU has the following I/O:

- **A,B**: The 32-bit data inputs to the ALU.
- **ALUCt1**: The 4-bit control input as described below. **ALUCt1[3]** is the left most bit in the table.
- **ShAmt**: The 5-bit value that controls how much the **B** input gets shifted.
- **ALUResult**: The 32-bit result of the ALU operation.
- **Zero**: A flag bit that is set when the ALU result equals zero (all bits are low).
- **Overflow**: A flag that indicates an arithmetic overflow occurred when set.

When performing shift operations, the **B** input is shifted by the amount specified by **ShAmt**. The **ALUCt1** signals correspond to the operations listed in Table 1. These are the same values from Lab 2.

ALUCtl	Operation
0010	add
0110	sub
0111	slt
0000	and
0001	or
1100	nor
1010	sll
1110	srl

Table 1: ALUCtl Signals

Your CPU must execute all instructions from Lab 2 (add, sub, and, or, nor, slt, addi, beq, lw, sw, j, sll, srl, lui, andi)

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`). This program takes a MIPS assembly language program as an argument, for example `lab3.mips`, and generates a `mif` file with the extension `mif`. When you create your memory block, you can specify a `mif` file that will initialize its contents. Name the file used to initialize your memory `lab3.mif` and put it in your project's root directory.

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 Lab report. Describe your design at a high level (you don't 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. 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. 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. Your CPU should implement forwarding to the data memory. For example, your CPU must handle the hazards present in the following code listings without stalling.

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

4. Your CPU must handle control hazards caused by the branch instruction by implementing a static branch not taken predictor. If a branch is not taken, your CPU should not stall or flush any instructions. When a branch is taken, your CPU must flush the invalid instructions and fetch the correct instruction. You must insert the minimum number of pipeline bubbles. You must handle all data hazards associated with branch instructions (e.g., if you implement branch comparison in the Decode stage, you must forward to the comparison logic when necessary).

Grading

To grade your lab, provide the TA with I/O pins for all control signals and major data path elements. Include pins for the clock, PC, instruction memory output, register file output, ALU output, data memory output, and all data path mux outputs. Additionally, create a vector waveform file in your project's root directory, named `lab3.vwf`, and include the required signals in it. Include a description of your control unit in your lab report.

Hints

- Think about your design before using Quartus! You don't have figures from the text that you can copy directly for all your CPU components. Design your CPU before you begin building it.
- 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.
 - 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.
- Use the LPM MegaFunctions whenever you need more than a simple logic gate. Some LPM blocks you may find useful in this lab include: `lpm_add_sub`, `lpm_ff`, `lpm_rom`, `lpm_mux`, `lpm_and`, and `lpm_or`. Here are some specific recommendations:
 - Data Memory: `lpm_ram_dp`
 - Registers: `lpm_dff`
 - Register File: `alt3pram`
 - ROM and Instruction Memory: `lpm_rom`

Extra Credit

For up to 5 points extra credit, implement interrupts as described in the book. You do not need to prioritize multiple interrupts that handle simultaneously.