

ECS 154B – Spring 2009 – Lab 2

Due by midnight on May 8

Objectives

- Build and test a multi-cycle MIPS CPU that implements a subset of the MIPS instruction set
- Design a microcode control unit

Description

In this lab you will use Quartus to build a multi-cycle CPU to further increase your knowledge of MIPS and as an introduction to microcode control. To test your CPU, you will run assembly language programs and simulate them in Quartus. You will be given an ALU to help you out and an assembler that will generate a file to initialize your memory. You may find Figure 5.28 and the figures in Appendix C from your text helpful. You must implement the CPU control using microcode as discussed in class. 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 memory, make it 128 words deep. Logically (not in your design) reserve the first 64 words for instructions and the last 64 words for data.

You will be given an ALU (available on the course website) to start your lab. You must use the provided ALU in your project and you must perform the shift operations using the 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.

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

Table 1: ALUCtl Operations

When performing shift operations, the B input is shifted by the amount specified by `ShAmt`. The `ALUCtl` signals correspond to the operations listed in Table 1. These are the same values from Lab 1 with the addition of the shift operations.

Your CPU must execute the following instructions:

- All required instructions from Lab 1 (add, sub, and, or, nor, slt, addi, beq, lw, sw, j)
- `sll`, `srl`, `lui`, `andi`

To test your CPU you can use `/home/cs154b/bin/mips2mif` (you may want to add this directory to your `PATH`) to generate a `mif` file to initialize your memory. This program takes a MIPS assembly language program as an argument, for example `lab2.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 `lab2.mif` and put it in your project's root directory.

For this lab, you must use microcode to implement the main control unit. However, you may optimize the memory contents to only work for the required instructions. You must generate the initialization file for your control ROMs by hand (notes on the `mif` file format are on the course web page). All control signals must come from this control ROM, with the following possible exception - if you wish to use combination logic to generate the ALU control signals (`ALUCtl[3..0]`), as is done in the book, you may do so.

In your lab report, include the finite state machine diagram for your CPU and describe any changes you made to the CPU discussed in the textbook to implement the required instructions. Also include a brief description of how you designed the ALU control unit.

Grading

To grade your lab, provide the TA with I/O pins for all major control signals and data path elements. Include pins for the clock, `PCWriteCond`, `PCWrite`, `IorD`, `MemRead`, `MemWrite`, `MemtoReg`, `IRWrite`, `RegDst`, `RegWrite`, `ALUSrcA`, `ALUSrcB`, `ALUOp`, `PCSource`, the output of the registers `PC`, `IR`, `MDR`, `A`, `B`, `ALUOut`, and any other control signals you add to your CPU. Follow the text for signal assignment values. Additionally, create a vector waveform file in your project's root directory, named `lab2.vwf`, and include the required signals in it.

Hints

- Test and debug in steps. Start with a subset of the lab requirements, implement it, test it, and then add other requirements. Performing the design and test in steps will ease your debugging. For example, you could implement the Lab 1 instructions, then add the shift instructions, and finally add the `lui` instruction.
- 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_dff`, `lpm_rom`, `lpm_mux`, `lpm_and`, and `lpm_or`. Here are some specific recommendations:
 - Memory: `lpm_ram_dp`
 - Registers: `lpm_dff`
 - Register File: `alt3pram`
 - ROM: `lpm_rom`

Extra Credit

For up to five points extra credit, implement exception handling in your CPU. To ease your implementation, you only need to consider invalid instruction (opcode) and arithmetic overflow exceptions. To get full credit have your CPU do the following on an exception:

- Store the instruction address into a new EPC register. The EPC exists as a new register outside the Register File.
- Save the cause in a new Cause register. Like the EPC, the Cause register exists as a new register outside the Register File. The Cause register should be 32 bits wide. The green reference card at the front of your text has the values to save in the Cause register.
- Load the PC with the address of the exception handling routine.