The UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL

Comp 541 Digital Logic and Computer Design
Fall 2014

Lab #10: A Full Single-Cycle MIPS Processor

Issued Wed 10/29/14; Due Wed 11/5/14

You will learn the following in this lab:

- Integrating ALU, registers, etc., to form a full datapath
- Designing the control unit for a processor
- Integrating memory units with a processor
- Encoding instructions
- More practice with test fixtures for testing a processor

Part 0a: Understand how memories are initialized.

An uninitialized memory module contains junk (i.e., undefined values), although typically your board will initialize it to all zeros (or ones). You can, however, specify the actual values to be stored in memory upon initialization. This is done using the command **$readmemh** or **$readmemb**. The former command allows you to specify values in a file in hex format, while the latter uses binary format.

Add the following line in the register file module right after the line where the core of the storage is specified (i.e., right after "**reg [Dbits-1 : 0] register_file [Nloc-1 : 0];**"):

```
initial $readmemh("reg_data.txt", register_file, 0, Nloc-1);
```

(The actual location of this new line of code does not really matter, as long as it is within that module. Remember, all lines of code are essentially in parallel!)

This line specifies that the file `reg_data.txt` be read, line by line, during compilation and synthesis, and its contents be used to initialize the bits in the memory. The last two arguments specify the range of memory locations. In this case, they start with 0, and go up to *Nloc*-1, but you are welcome to specify a subset of the range if you do not have data to initialize the entire memory.

Create the file `reg_data.txt` in the project folder (using an external editor), and add values, one per line, in hex. You should not prefix each value by 'h'. Thus, if your memory/register file has 8-bit data, your file may look like this:

```
05   // Comments are allowed
A0
C1
...
```

You can also use the binary version of the initialization (**$readmemb** instead of **$readmemh**). In that case, the file will have a sequence of binary values, one per line (no 'b' before though):

```
0000_0101  // Underscore can be used for clarity
1010_0000
1100_0001
…
```

With these changes, re-simulate your design from Lab 9, and verify for yourself that the initialization worked correctly. You will need to modify the test bench so that some of the operations use values that were provided via initialization. Also, if your datapath uses 32 bits, then the initialization values in `reg_data.txt` will have to be 32-bit values as well.

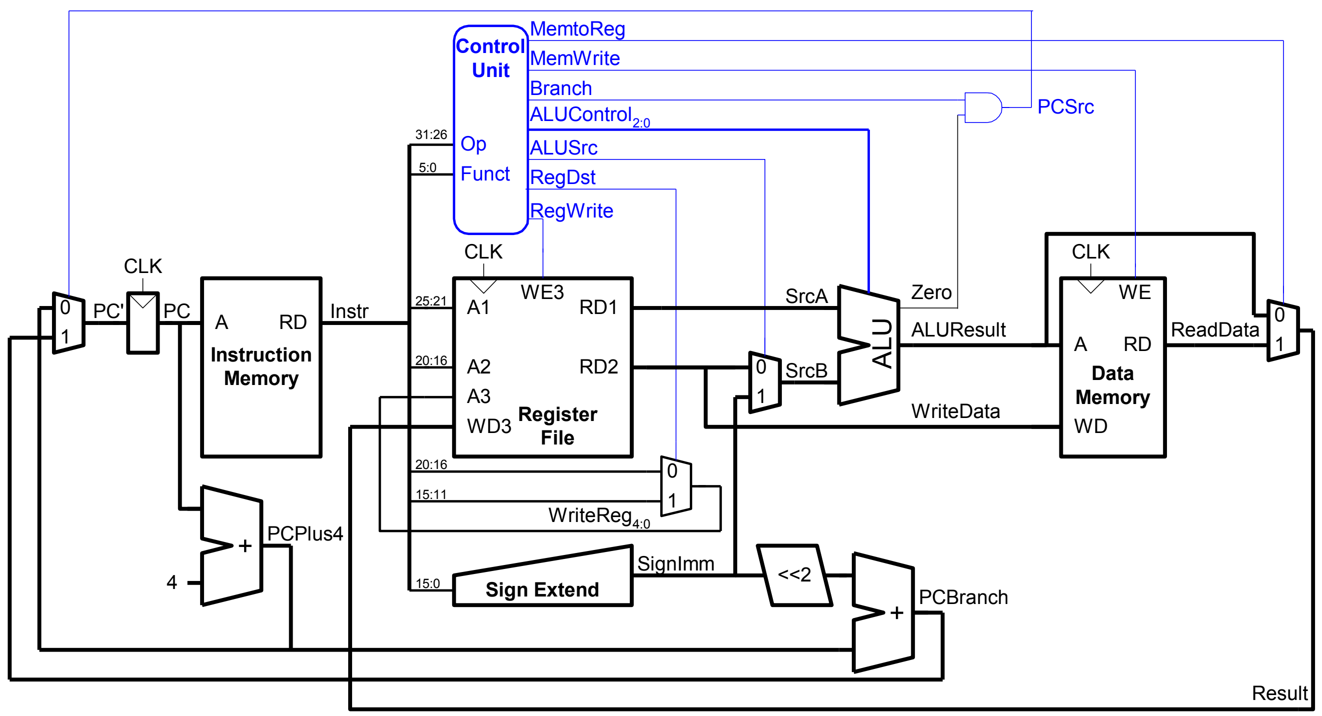
Part 0b: Carefully review the single-cycle MIPS processor design of Lecture 15.

Study the lecture slides carefully, and review Chapter 7.1-7.3 of textbook. Also review the Patterson Hennessy front inside green flap for summary of the MIPS instruction set; a PDF version of this ("green card") is posted on the class website. Wherever there are differences between the Patterson Hennessy textbook and the Harris and Harris textbook, we will adhere to the former. We will do so because the MARS simulator, which you will use to assemble your MIPS assembly code follows the former.

Part 1: Design a full single-cycle MIPS.

Put all the pieces together to create a full single-cycle MIPS CPU as discussed in class. In particular, do the following:

- **Design the top-level module, with instruction and data memories and the MIPS CPU,** as shown on Slide #4 of Lecture 15. Name this file `top.v`. You may look at Slide #5 for guidance. For now, choose a reasonably small size for each memory, e.g., 8 or 16 memory locations. *Note:* The addresses to these memories will still be 32 bits long. Send full 32 bits to instruction memory; strip the last two bits inside the instruction memory to derive a word address. Similarly, send full 32 bits of address to data memory; strip the last two bits inside the data memory to derive a word address. Both of these memories should return a full 32-bit word (i.e., `Dbits` = 32, `Abits` = 32, `Nloc` = 8 or 16). (We will talk about *load-byte* and *store-byte* later.) Your instruction memory module should be in its own file called `imem.v`, and the data memory module in its own file called `dmem.v`.
- **Initialize the instruction and data memories,** using the method explained in Part 0a. The file that contains the initial values for the instruction memory will contain a 32-bit assembly coded instruction per line (either in hex or in binary). Name this file `imem_values.txt`. The file that contains the initial values for the data memory will contain some 32-bit data values, again one per line. Name this file `dmem_values.txt`.
- **Design the top-level processor, containing controller and datapath modules,** using Slides #6-7 as guidance. *Note:* There are differences between the MIPS design in the textbook and what we are designing in the lab. The lab version has a much more sophisticated ALU. Therefore, do not blindly follow the information in the book or the lecture slides, but only use it for rough guidance. The datapath should be 32-bit wide (i.e., registers, ALU, data memory and instruction memory, all use 32-bit words).
- **Complete all the little pieces,** so the design looks like that in Lecture 15, also reproduced below. (Specifically, please review Slides #19-20, #23, #25-26 carefully; some of these have been updated after the lecture.) Again, remember that our design will have some differences with respect to this figure (in terms of ALU functionality, flags, where and how PCSrc is generated, etc.).



- **Implement all of the following instructions:**
  - `lw` and `sw`
  - `addi, slti, ori`
    - NOTE: While `addi` and `slti` should sign-extend the immediate values, `ori` should zero-extend the immediate because it is a logical operation!
  - R-type (all that are supported by our ALU, including shifts and the two comparisons)
  - `beq, bne` and `j`
- **You must use the correct instruction encoding** from the Patterson Hennessy textbook, or refer to the "green card" posted on the course website. Otherwise, it will be much harder to assemble your final demo by hand.
- **Create a test fixture to test your CPU using the simulator.** Test programs are provided on the course website. These were initially written in MIPS assembly, then compiled using MARS and converted to hex machine code, which should be used to initialize your instruction memory. You will still need to write a simple test bench that instantiates your top-level design and provides the clock input (and reset).

  Store the machine code into the file that is used to initialize the instruction memory (`imem_values.txt`). Be sure to initialize the program counter (PC) inside your MIPS design to zero, so that it starts executing from the beginning of the instruction memory. If you need to initialize your data memory, put the initial values in the corresponding file (`dmem_values.txt`).
- **For now, you will not be implementing your design on the board.** You will do so next week.

- **Start thinking about what you would like to build for your final project!** Every project must use a VGA display as output. Nexys 4 boards have audio output built-in; a limited number of add-on sound cards are available to those using the Nexys 3 boards (to augment the display). Keyboard/mouse inputs will be available to every project. The NExys 4 boards have accelerometers built-in; a limited number of other input devices (joysticks, keypads, etc.) may be available to use instead. Start thinking!

Okay, good luck!

What to submit:

- **ALL of the Verilog files for Part 1**
- **A couple of sentences within the email body stating whether your design worked, any errors/bugs you couldn't resolve, any special observations, etc.**
- **Show a working demo of your design for Part 1 during lab (Nov 7).**

How to submit: **Please submit your work by email by 11:59pm, Nov 5, 2014, as follows:**

- **Send email to: `comp541submit-cs@cs.unc.edu`**
- **Use subject line: Lab 10**
- **Include all of the attachments, and your statement/observation, as specified above**
