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

Comp 541 Digital Logic and Computer Design
Prof. Montek Singh
Spring 2017

Lab #8: A Basic Datapath and Control Unit
Issued Wed 3/22/17; Due Wed 3/29/17 (11:59pm)

This lab assignment consists of several steps, each building upon the previous. Detailed instructions are provided. Verilog code is provided for almost all of the designs, but some portions of the code have been erased; in those cases, it is your task to complete and test your code carefully. Submission instructions are at the end.

You will learn the following:

- Specifying memories in SystemVerilog
- Initializing memories
- Designing a multi-ported memory (3-port register file)
- Integrating ALU, registers, memory, etc., to form a full datapath
- Encoding instructions
- Designing the control unit for a processor
- More practice with test fixtures

Part 0: Understand how memories are specified in SystemVerilog

Memory Specification

A typical single-ported RAM module is described in SystemVerilog as shown in the file **`ram.sv`**. The number of "ports" in a memory is the number of distinct read/write operations that can be performed concurrently. Thus, the number of ports typically is the number of distinct memory addresses that can be provided to the memory. A single-ported RAM takes a single address as input, and can perform a read or write (determined by a write enable signal) to that address. Below is the main part of a memory specification:

```
logic [Dbits-1:0] mem [Nloc-1:0];      // The actual storage where data resides
always_ff @(posedge clock)             // Memory write occurs on clock tick
    if(wr) mem[addr] <= din;           // … but only if write is enabled
assign dout = mem[addr];               // Memory read occurs asynchronously
```

Please refer to **`ram.sv`** for the complete description of the RAM module.

Memory Initialization

(Note: You will not need to initialize memory in this lab assignment, but will need it for the next one.)

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 "**logic [Dbits-1 : 0] mem [Nloc-1 : 0];**"):

```
initial $readmemh("mem_data.mem", mem, 0, Nloc-1);
```

(Remember to put the initialization line *after* the declaration of the logic type **mem**, and remember to replace **mem** with the actual name of your memory storage.)

The first argument to **$readmemh** is a string that is the name of the file to be read, line by line, during compilation and synthesis, and its contents are used to initialize memory values. The second argument is the name of the variable that is the memory storage. 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 `mem_data.mem` in the project folder (using an external editor), and add it to your project using *Add Source…* and selecting its type as *Memory Initialization File.* Add values, one per line, in hex. Do not prefix each value by 'h'. Thus, if your memory has 8-bit data, your initialization 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):

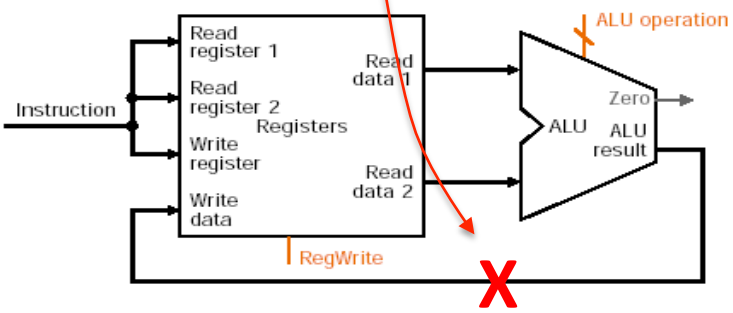
```
0000_0101  // Underscore can be used for clarity
1010_0000
1100_0001
…
```

Remember that if your datapath uses 32 bits, then the initialization values in `mem_data.mem` will have to be 32-bit values as well.

IMPORTANT: You must select the type of the file containing memory values as ***Memory File***. Otherwise, the tool may not be able to access it properly.

Part 1: Register File

Today you begin to implement the MIPS subset that we will use for this class. You will implement the portion of the CPU datapath shown in the following diagram, and test it. However, since we do not yet have a source for instructions, and to aid in testing, we will slightly modify this part of the datapath to provide more controllability and visibility. In particular, we will cut the feedback from the ALU to the write port of the register file, and instead allow *Write data* to be directly supplied by a test fixture. A modified picture is shown on the next page.



NOTE: We will NOT implement the datapath of this figure in this part. See figure on next page.

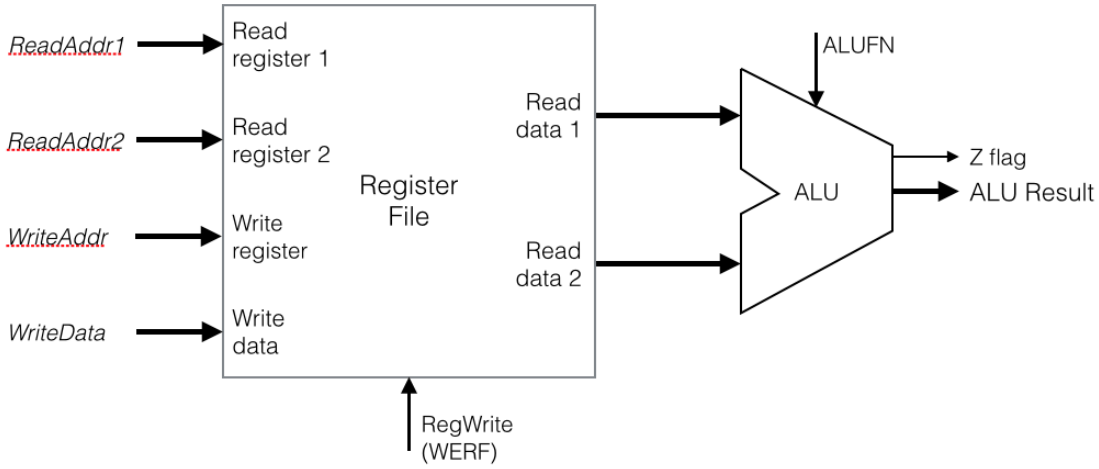
First, you will design a 3-port register file. We call it 3-port because three different addresses can be specified at any time: *Read Address 1, Read Address 2,* and *Write Addr.* These are required to access (up to) two source operands and one destination operand required for MIPS instructions.

Do the following:

- Start with the **`ram_module`** from the website. Modify it to create a new module called **`register_file`**, which has the enhancements specified below. A skeleton is provided on the website (`register_file.sv`).
    - three address inputs instead of just one (e.g., *ReadAddr1, ReadAddr2* and *WriteAddr*).
    - two data outputs instead of just one (e.g., *ReadData1* and *ReadData2*).
    - the write enable and clock stay the same.
    - when writing, *WriteAddr* is used to determine the memory location to be written.
    - when *reading* register 0, the value read should always be 0 (it does not matter what value is written to it, or whether you write any values into it).
    - use parameters for number of memory locations (`Nloc`), number of data bits (`Dbits`), and the name of the file which contains initialization values (`initfile`).
- While in the final CPU design, the three addresses will come from the register fields in the instruction being executed, for now you will use a Verilog test fixture (in Part 2) to provide these addresses, the data to be written, and the *RegWrite* signal. The test fixture does a few different reads and writes so you can see via simulation that your register file is working.

Part 2: Putting the datapath together

Design a top-level module that contains the register file and your ALU (from Lab 3). Name the Verilog source `datapath.sv`. This module must exactly correspond to the block diagram below.

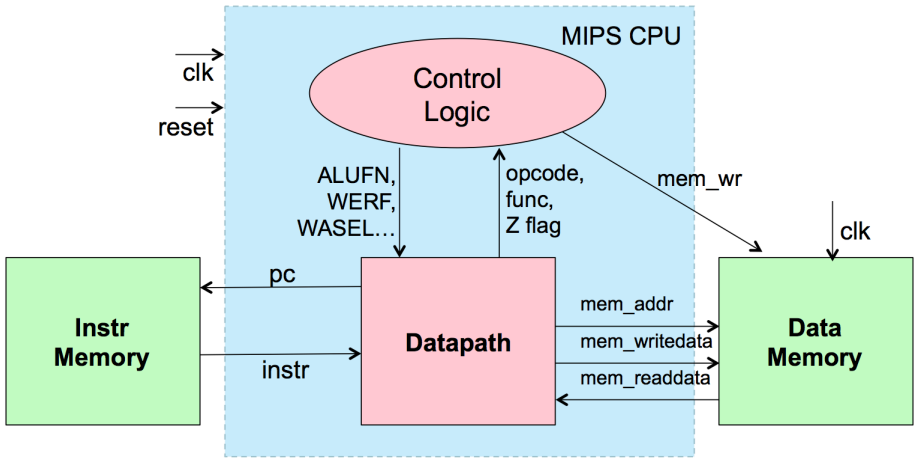


Note the following:

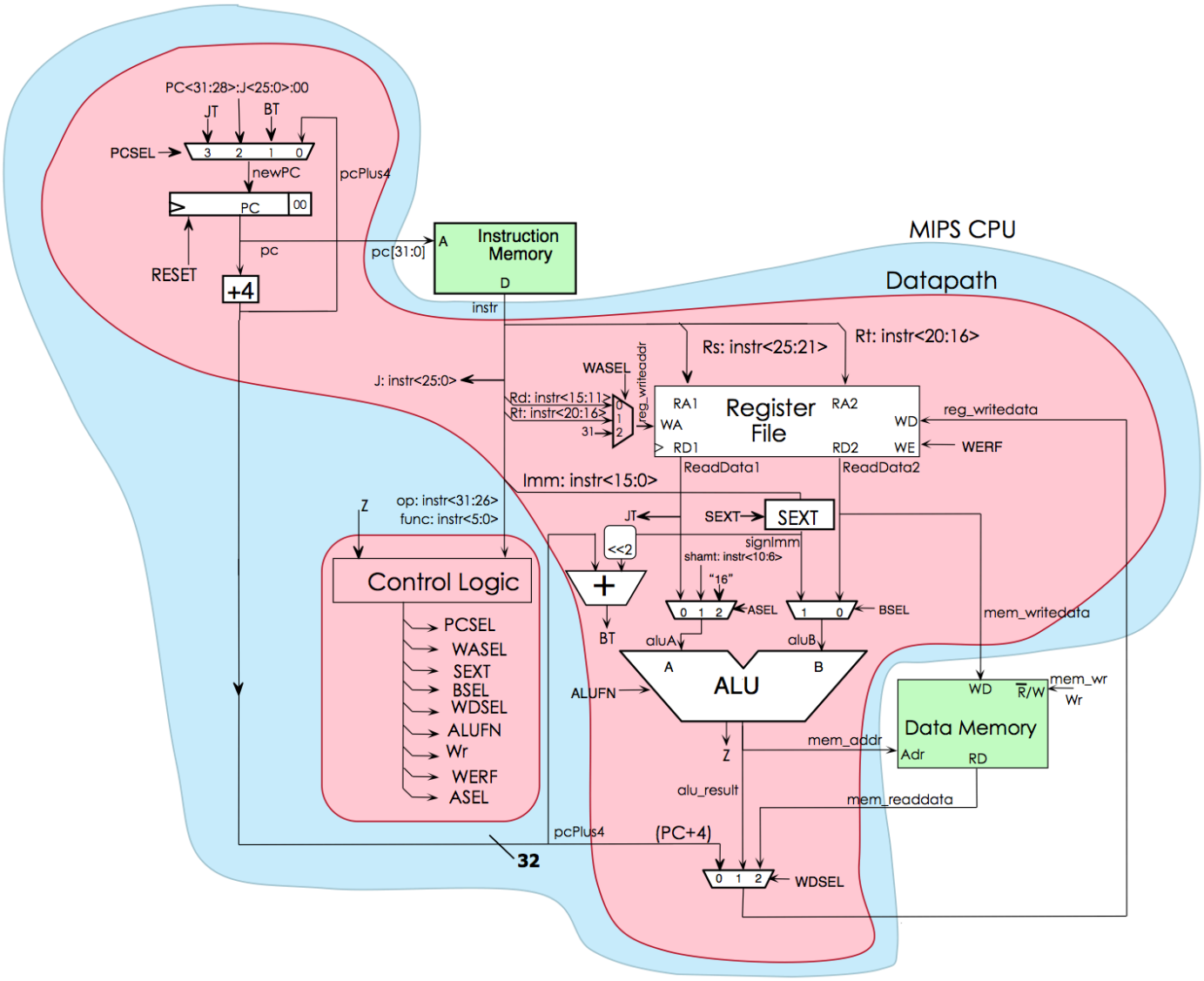
- To aid in testing your design, send "*ReadData1", "ReadData2"* and *"ALUResult"* to the output of the top-level module so they can be easily observed during simulation. The Zero flag (*Z*) must also be generated as an output from the top-level module (because branch instructions will need it).
- For now, do not feed the ALU result back to the register file. Instead, the data to be written into the register file should come in directly from the test fixture as an input to the top-level module.
- The inputs to the top-level module are: clock, *RegWrite*, the three addresses, the ALU operation to be performed (*ALUFN*), and the data to be written into the register file (*WriteData*).
- Use the Verilog test fixture provided on the website to simulate and test your design. The text fixture is self-checking, so any errors will be flagged automatically. Please use exactly the same names for the top-level inputs and outputs as used in the tester where the "unit under test" is instantiated.

Part 3: The Control Unit

In preparation for designing a full MIPS CPU, we will develop the Control Unit in this exercise. Below are two diagrams of our single-cycle MIPS CPU (from Comp411), first a top-level overview, then a detailed one.



Here is the picture showing the details:



In Part 2, we put the register file and the ALU together. Now we will develop the control unit. *We will NOT yet develop the full MIPS datapath, nor will we be adding memories yet; those tasks will be the next lab.*

The control unit should handle **ALL of the following instructions:**

- `lw` and `sw`
- `addi, addiu, slti, sltiu, ori, lui`
  - **NOTE 1:** Contrary to what you may have learned earlier (e.g., in Comp411), `addiu` actually does not perform unsigned addition. In fact, it *sign-extends* the immediate. This instruction (along with `addu`) was actually misnamed! The only difference between `addiu` and `addi` is that `addiu` does not cause an exception on overflow, whereas `addi` does. Since we are not implementing exceptions, `addiu` and `addi` are identical for our purposes.
  - **NOTE 2:** Also, contrary to what you may have learned earlier, `sltiu` actually *sign-extends* the immediate, *but performs unsigned comparison*, i.e. ALUFN is "LTU".
  - Also note that `ori` should zero-extend the immediate because it is a logical operation! Finally, sign-extension for `lui` is a *don't-care* because the 16-bit immediate is placed in the upper half of the register without any need for padding.
- R-type: `add, sub, and, or, xor, nor, slt, sltu, sll, sllv, srl, sra`
- `beq, bne, j, jal` and `jr`

First study the Powerpoint slides on Single-Cycle MIPS processor. Next, fill out the table below with the values of all the control signals for the 25 basic MIPS instructions listed here.

Instr	**werf**	**wdsel**	**wasel**	**asel**	**bsel**	**sext**	**wr**	**alufn**	**pcsel** Z=1	Z=0
LW	1	10	01	00	1	1	0	0XX01		
SW										
ADDI										
ADDIU										
SLTI										
SLTIU										
ORI										
LUI										
BEQ										
BNE										
J										
JAL										
ADD										
SUB										
AND										
OR										
XOR										
NOR										
SLT										
SLTU										
SLL										
SLLV										
SRL										
SRA										
JR										

MIPS instruction decoding table

These values will depend on the Z flag.

Using the values in this table, complete the Verilog description of the control unit in the file `controller.sv` available on the website.

Use the Verilog test fixture provided on the website to simulate and test your design. The text fixture is self-checking, so any errors will be flagged automatically. Please use exactly the same names for the top-level inputs and outputs as used in the tester.

What to submit:

- **Your Verilog source for the register file (register_file.sv), datapath (datapath.sv), and control unit (controller.sv)**
- **A picture of the instruction decoding table from Part 3.**
- **A screenshot of the simulation waveform windows of Parts 2 and 3 using the self-checking testers.**

How to submit: **Please submit your work by email by 11:59pm, Mar 29 (Wed) as follows:**

- **Send email to:** `comp541-submit-s17@cs.unc.edu`
- **Use subject line: Lab 8**
- **Include the attachments as specified above**
