EECE2323 **Digital Logic Design Lab** **Lab 6**

Adding Instruction Decoding to the Datapath

1 Objective and Overview

In this lab you will control the datapath using instructions from the lab computer. This lab mainly includes two new components. First, you should write an *instruction decoder* that translates the different bits in an instruction word to control and data signals for the datapath. Next, you will be given a debounce circuit to integrate it into your design that connects the button *BTN1* on the TUL PYNQ board to the clock so that you can single step your design. We will also add the 7-Segment displays on the PYNQ add-on board for additional monitoring of the ALU output. The displays show the values in signed decimal format.

You must complete Lab 5 before you start this experiment.

2 Decoding Instructions and Executing them on the Datapath

In this lab, you will add instruction decode logic to the datapath you completed in Lab 5. You should implement an instruction decoder as described below. The machine code for your instructions will be entered into your design using VIO. Instruction formats are described in `instruction_set.pdf` available on Canvas. Please note that the instruction decoder is a combinational logic and a clock edge is only needed for writing to the register file and writing to memory in the datapath. The second change is to add a debounce circuit that allows you to single step your datapath. This circuit will connect the *BTN1* button on the TUL PYNQ board to the clock inputs for memory and the register file. Figure 1 shows the top level design for this lab experiment.

==

2.1 Prelab

Begin by reading Sections 1 and 2 above. Then answer the following questions.

2.1.1 Instruction Decoder: Control Signal

Based on the instruction set document find out the values of control signals for all of the op-codes and complete the table below showing the control signal values corresponding to each op-code,

2.1.2 Instruction Decoder: Verilog Module

Write verilog code for the instruction decoder with the interface shown in Figure 2.

Verify your code using Vivado simulator and make sure there is no syntax error in the code.

Your response to the prelab should include a separate file containing this verilog code.

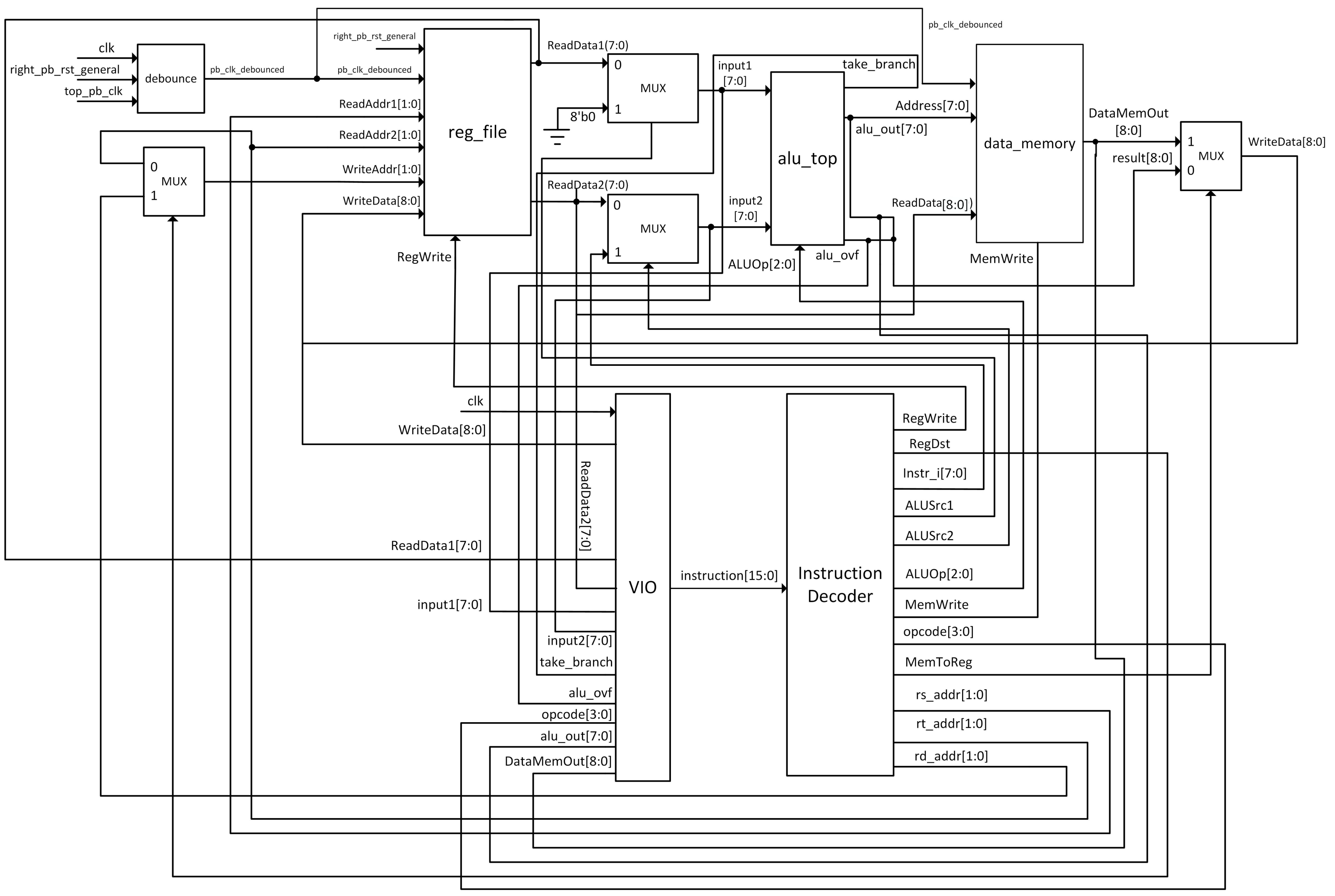

Figure 1: Datapath with Instruction Decoder Unit

Opcode	**RegDst**	**RegWrite**	**ALUSrc1**	**ALUSrc2**	**ALUOp[2:0]**	**MemWrite**	**MemToReg**
0000	1'b0	1'b1	1'b0	1'b1	3'b000	1'b0	1'b1
0001							
0010							
0011							
0100							
0101							
0110							
0111							
1000							
1001							
1010							
1011							
1100							
1101							

Table 1: Table for opcodes and their corresponding control signals

2.1.3 Translate the Steps to Instructions!

Translate the steps for the load and store example in the `load store steps.txt` file to a sequence of MIPS-like assembly instructions. You can optimize out the redundant steps or add intermediate steps while keeping the functionally and results the same after the translation.

As an example for the test result steps of Instruction 5 in that file, we have:

```
1.  Change the regfile_read_address1 to 2
2.  Change the ALUOp to 3'b011
3.  Change the instr_i to 8'hF0
4.  Change the ALUSrc2 to 1
5.  Change the regfile_write_address to 2
6.  Change the RegWrite to 1
7.  Change the RegWrite to 0
```

You should see the output of ALU changes to 8'bFE.
The corresponding instruction for these steps is an immediate OR (ori) as below:
`ori $2, $2, 0xF0`

2.1.4 Generate the Machine Codes

Use the field coding from the Instruction Set document to convert your assembly instructions to a sequence of 16-bit instructions. Create a table and list the instructions in the first column while writing the corresponding machine codes in the second column.

Instruction	**Machine Code**
ori \$2, \$2, 0xF0	1000101011110000

Table 2: Example table for an instruction and the corresponding machine code

======================== END Prelab =====================

2.2 Entering Your Design

2.2.1 Instruction Decoder

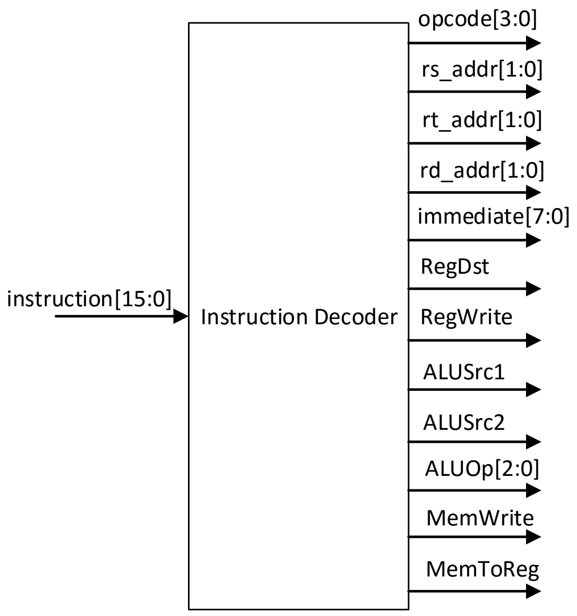

Figure 2: Instruction Decoder Interface

Design an instruction decoder in Verilog with the interface shown in Figure 2. Name your module `inst_decoder`. Your circuit should be combinational and should take in an instruction and generate the signals shown in the figure. Note that there is a new signal `RegDst` that comes from the decoder. This signal chooses whether the register write address from the instruction word is the `rd` field (for R type instructions) or the `rt` field (for I type instructions). The top level module will instantiate a mux based on this signal.

2.2.2 Simulate the instruction decoder

Before implementing the instruction decoder on FPGA, write a testbench to test your instruction decode circuitry. Your *test vectors* should cover all possible types of instruction (by varying opcode, following the same order as shown in Prelab Table 1).

Create a new testbench module called `inst_decoder_tb`.
Enter the test vectors that you have chosen into the testbench, save the testbench, and run it in Vivado simulator (XSIM). Make sure that you set the `Simulation Run Time` property to an appropriate value before running your simulation.

Save the waveform as part of the evidence that your lab works, and submit it with your report.

2.2.3 Designing the top level

Get the top level Verilog file, `pdatapath_top.v` from the course webpage. This Verilog code shows how all of your components are connected. You should have regfile, alu and instruction decode components already designed. Get the debounce circuit, `debounce.v` and the 7-Segment display modules, `adaptor_display.v` from the course webpage. These two modules have already been instantiated and set up for you in the top module. You will need to generate data memory and VIO to add to this experiment. Note that this VIO module has several new inputs and outputs.

Add the constraint file for your design. The constraint file for this design is named: `pdatapath.xdc`

2.3 Testing in Hardware

2.3.1 Implementing the Design with Vivado Synthesizer

Click on `Generate Bitstream` **under the Program and Debug section in the Flow Navigator pane to run the synthesis process. Vivado IDE will run the Synthesis and Implementation processes automatically.**

If the synthesis process runs correctly, continue by programming the FPGA board. Follow the same steps you have for previous labs.

2.3.2 Programming the FPGA with Hardware Manager

Program your hardware the way you have in previous labs. Open the hardware manager, open the target device, and program the PYNQ.

Now you should see the `hw_vio_1` window open. Click on the + sign and add all of the probes by selecting and adding them. Now you are ready to test your design in hardware.

2.3.3 Testing Your Design

For this design, we are using one reset, named `rst general` in Verilog, to reset the memory elements (memory and register file). On the board, it is connected to the right push button (`BTN0`). We have added the single step button the push button (`BTN1`). You need to press `BTN1` whenever you want to store data in either memory or the register file.

Use VIO to test your design. Use the instructions you developed in the prelab. Keep screen shots to show that you have done this.

3 What to submit on Canvas:

A single PDF report containing:

a) All your Pre-lab responses.
b) Summary of hardware/design implementation report as outlined in the rubric.
c) Simulation testbench and test vectors for your instruction decoder, and all your output screen shots demonstrating the correct operation of your design.
Note: You do not need to include a video demo.