EECE2323 **Digital Logic Design Lab** **Lab 4**

Adding Register File to ALU

1 Objective

In this lab experiment you will learn about *sequential* logic; that is, logic that holds a *state* and is *timed* with a clock signal. You will design and build the processor datapath using an ALU, a zero register and a *register file*. You will create a register file and connect it to your ALU from Lab 3. You must complete Lab 3 before you start this experiment.

2 Overview

In this lab you will begin to assemble the blocks for the *datapath* of your processor including the ALU you have already designed. The ALU acts as the "brain" of your processor; it performs operations on the operands and generates new values. A datapath always contains one or more units that perform operations in this fashion. Another common element of a datapath is some sort of *memory* to store values to be operated on by other elements of the datapath. You will design a *register file* to act as the memory for your processor.

A register file is an array of registers that holds several data words. For our design, you can think of it as a table with multiple rows (data words) and a single column. Each data word corresponds to a register which in our case has 9-bits.

Another element that will be introduced in this lab is the *zero register*. This is a special register which has the fixed value of zero. This register will be used to implement the *clr* instruction, which resets an arbitrary element from the register file to zero.

In addition to creating the datapath, you will design sequences of control inputs to perform operations to test your datapath, such as "write these numbers to the register file and do the add".

3 Register File and Connection with ALU

You will implement the register file using the array feature of the Verilog language which allows you to define a 2D array. Based on the design specification, this array should contain four *9-bit registers*. The ninth bit is the overflow value, which is saved along with its data.

Figure 1(a) shows the register file interface. This register file has one write port data input and two read port data outputs. *wr_data(8:0)* is the data input; the data word that is to be written to the register file is presented on this input. The destination register number (row) in the register file is determined by the *wr_addr(1:0)* input. The *rd0_data(8:0)* and *rd1_data(8:0)* signals are data outputs; data words that are read from the register file are presented on these outputs. For these outputs, the source register numbers (rows) in the register file are determined by *rd0_addr(1:0)*

and *rd1_addr(1:0)* inputs. If *wr_en* signal is activated, new data could be written to register file otherwise the content of register file should not be changed, and the output ports should take the values of the rows which *rd0_addr(1:0)* and *rd1_addr(1:0)* point.

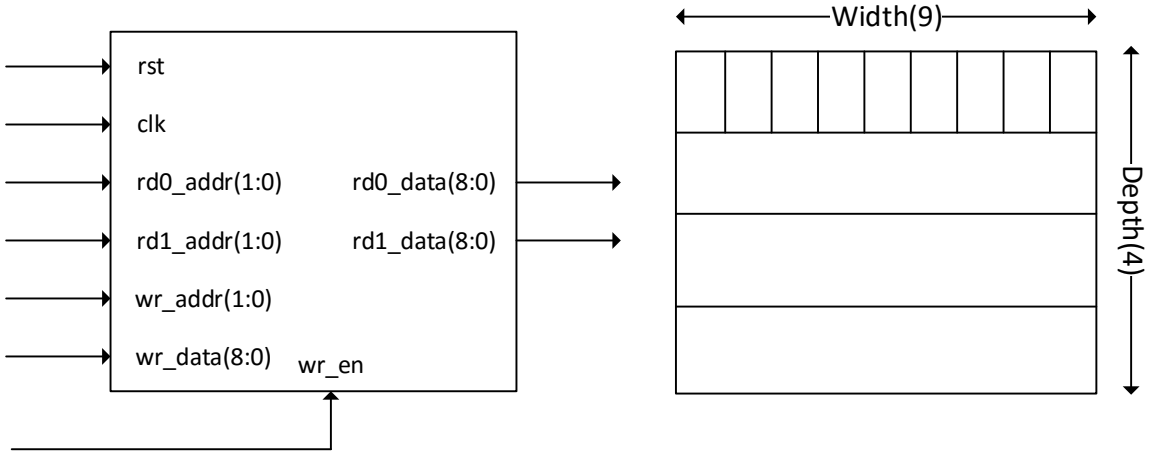

Figure 1: (a) Register file interface (b) Register file dimensions

There are also a clock input(*clk*) and a reset input(*rst*). Most memory elements can be reset, meaning that all of their data storage is cleared (to the value 0). The clock signal is essential to sequential logic, and provides a sense of time to the register file and determines when the data input will be looked at.

Register files can be described by their *width* and their *depth*; see Figure 1(b). The depth of a register file is the number of data words that it can hold. In this lab we will design a register file that can hold four data words, so it is a "four-deep" register file. A register file's width is equal to the size of each data word. In our register file each data word will be nine bits (we will store the 8-bit output of the ALU plus the overflow flag), so it will be a "nine-bit-wide" register file.

Another element that we introduce is the *zero register*. It is convenient to have a source of zeros in the circuit. The *zero register* plus the ALU's "AND" operation will be used to implement an instruction called "clr" or "clear". This instruction will clear an arbitrary register from the register file to zero; it could be used for initializing or resetting register values.

3.1 Prelab

Download the prelab assignment from the course webpage and complete your prelab. Your TA will provide information regarding how the prelab should be handed in.

3.2 Entering Your Design

Create a new Verilog source called `reg_file`. It should have seven input ports: `rst`, `clk`, `wr_en`, `rd0_addr(1:0)`, `rd1_addr(1:0)`, `wr_addr(1:0)`, `wr_data(8:0)` and two output ports: `rd0_data(8:0)` and `rd1_data(8:0)`.

Define a 2D array for the register file using the following syntax:

```
reg [WIDTH-1:0] array_name [0:DEPTH-1];
```

Remember that you want a 4-deep, 9-bit-wide register file.

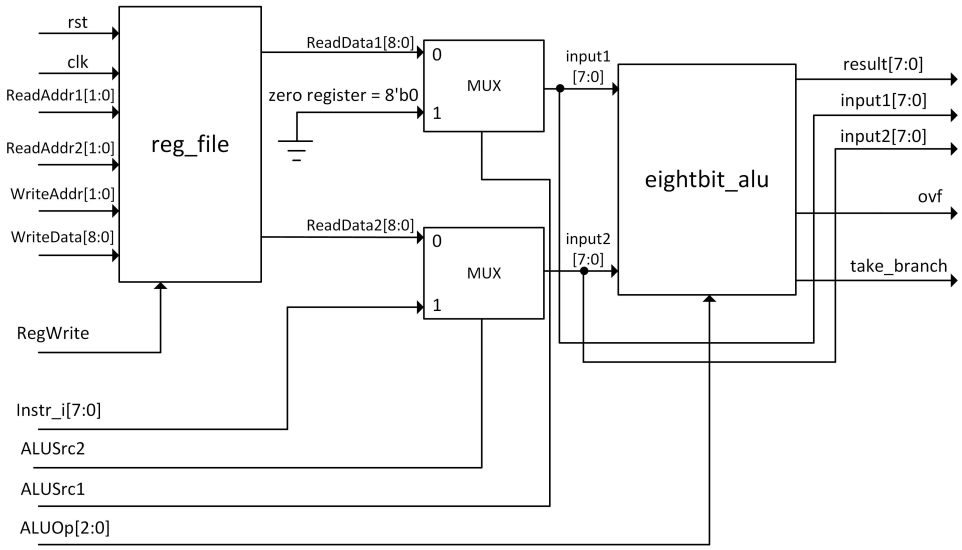

Figure 2: ALU, Register File and muxes

Next create a verilog file called `alu_regfile.v` that connects the register file with the ALU. This design is shown in Figure 2.

Create an 8-bit signal which is tied to zero as the zero register. Use an 8 bit 2-to-1 multiplexer to choose the source of the ALU's first input to be either from the regfile when the sel bit of the MUX is 0 or from the zero register when the sel bit of the MUX is 1. ALU operations only need the 8-bit value part, thus the MUXes are 8 bit wide. You can just connect the output of the register file to the input of MUX. In Verilog, remove the MSB of the 9 bit signal by using the other bits.

Create a second MUX for the ALU's second input. For now, this mux will be connected to a VIO input. In later labs we will use it as a source of immediate (constant) values from the instructions.

3.3 Simulating Your Design

Create a new testbench called `alu_reg_file_tb` to test the combined functionality of the ALU, register file, and two muxes. The inputs for this circuit are the register file inputs, mux select bits and ALU select. The outputs are the ALU outputs and mux outputs. The mux outputs allow you to see the register file outputs.

In comparison to previous labs where we only simulated combinational circuits, here we want to simulate a sequential circuitry. In sequential circuits we have a `clock` signal. The clock propagates signals in a specific period and gives a notion of time and sequence to the circuit. In the testbench

you will need to generate a clock to feed your sequential circuit. In addition, your testbench should generate a reset signal to clear the memory elements in your design, in this case the register file.

In your testbench, after reseting your register file, you should set the register file words to non-zero values to test your circuit.

Your testbench should be organized as a sequence of test vectors. Be sure to test: 1) writing and reading values to/from the register file, 2) different mux configurations and outputs, and 3) all possible ALU outputs. It is fine if you do not simulate all possible ALU operations. After generating the test vectors, save the testbench, and run it using the Vivado Simulator. Make sure that you set the `Simulation Run Time` property to an appropriate value before running. Capture your waveform as a screen shot.

3.4 Testing in Hardware

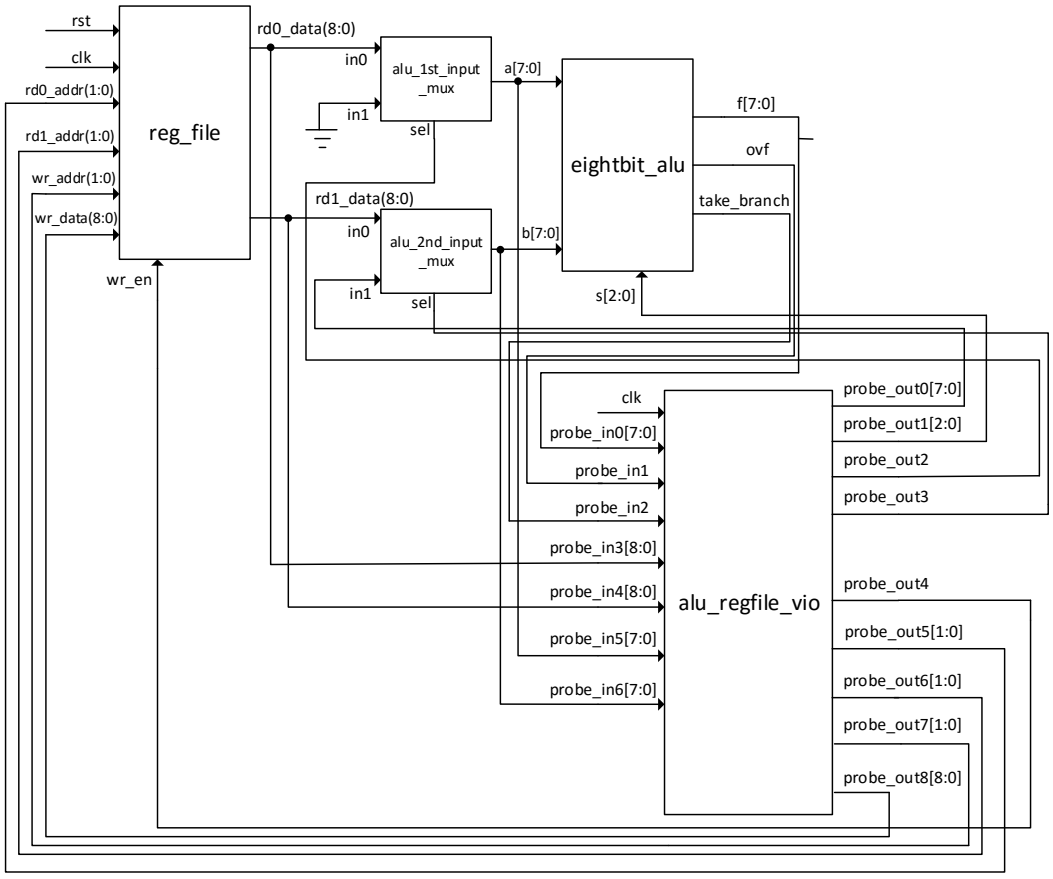

Figure 3: ALU, Register File and VIO connections

The next step is to test your design in hardware. We will use the *Virtual Input/Ouput (VIO)* IP core from Xilinx. This is the same core as in lab 3, but you will need to configure it differently for this lab. The VIO probes you will use in this lab are shown in Fig. 3.

To add this core to your design, open the IP core catalog and type "VIO" in its search bar. You should see *VIO (Virtual Input/Output)* in the list, and double click on it to open the configuration window. Configure your VIO as shown in Fig. 3 by setting up the right number of input and output probes, and the correct width for each probe. *Remember that VIO inputs are outputs from your circuit, and VIO outputs drive signals in your circuit!*

Next, you need a top module that connects the Regfile, Muxes, and ALU along with the VIO. Use the *alu_regfile_vio_top.v* module from the course webpage. You will need to write the complete top file; this time we are giving you a skeleton Verilog file for this part.

Add the XDC file by clicking on Add Source under the Project Manager section in the Flow Navigator pane and choosing `Add or Create Constraints` from the list. The name of the constraints file is eightbit_alu_regfile.xdc.

3.4.1 Implementing a 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.

3.4.2 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 PNYQ board.

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.

3.4.3 Testing Your Design

For this design there is a reset for the register file. In future designs, you will use it to reset any memory elements in your design. On the board, it is connected to the right push button (`BTN0`).

Use VIO to test your design. Use the same test vectors that you used for simulation. Remember that you need to write values to the register file, and then read them out and set the muxes appropriately to get the outputs from the ALU.

Follow instructions from the TA regarding how to show that you have completed your lab experiment.