EECE2323	Digital Logic Design Lab	Lab 3

Arithmetic and Logic Unit

1 Objective

In this lab experiment you will add the shift and branch operations to your partial ALU to create a complete ALU. You will also test your circuit on the board using the switches, buttons, and LEDs.

This experiment extends your design from Lab 2. You must complete Lab 2 before you start this experiment.

2 Overview

In this lab, you will complete your ALU design from Lab 2 by adding the shift and branch operations. The ALU has two 8-bit data inputs, one 8-bit data output, one overflow flag output, one zero output and a 3-bit selector for choosing among the following operations: ADD(i), INV ($\sim$), AND(i) ($\cdot$), OR(i) ($|$) arithmetic shift right ($>>>$), logical shift left ($<<$), branch if equal (beq) and branch if not equal (bne).

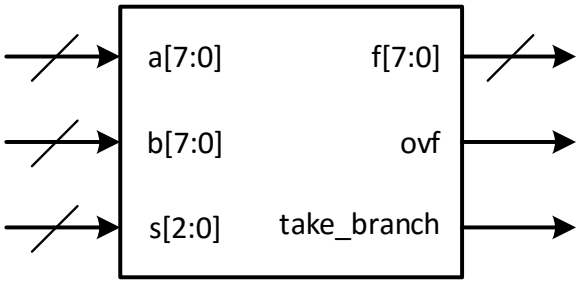

Figure 1: ALU interface

Figure 1 shows the complete ALU interface. *a(7:0)* and *b(7:0)* are the data inputs. Each input is eight bits wide and is interpreted as a two's complement number. *s(2:0)* is the select control input. The function performed can be changed by setting *s* to a different value, as shown in Table 1. The result of the operation is produced on *f(7:0)*, which is also an eight-bit two's complement number. A flag bit indicating *overflow* is produced on the *ovf* output. Note that logical operations cannot overflow, so the *ovf* output is always set to 0 in these operations. Here we introduce another output for the ALU which is called the *take_branch* output. This signal represents the branch operation result. If the branch condition is true this signal should be 1 and if it is not true it should be 0. The condition for `beq` is true if the ALU inputs are equal. The condition for `bne` is true if the ALU inputs are not equal.

s[2:0]	f[7:0]	ovf	take_branch	Description
0 0 0	$a + b(I)$ (add(i))	overflow	0	a plus b (or an 8-bit Immediate)
0 0 1	b (inv)	0	0	Bitwise inversion of b
0 1 0	$a \cdot b(I)$ (and(i))	0	0	Bitwise AND of a and b (or an 8-bit Immediate)
0 1 1	$a \mid b(I)$ (or(i))	0	0	Bitwise OR of a and b (or an 8-bit Immediate)
1 0 0	$a >>> 1$ (sra)	0	0	Arithmetic shift right
1 0 1	$a << 1$ (sll)	0	0	Logical shift left
1 1 0	0	0	$a == b$ (beq)	Branch if Equal
1 1 1	0	0	$a! = b$ (bne)	Branch if not Equal

Table 1: ALU operations

3 Adding the Shift and Branch Operations

In this lab, you will extend your partial ALU from Lab 2 by adding the shift and branch operations.

In the complete ALU you will have two shift operations, arithmetic shift right by 1($>>> 1$) and logical shift left by 1($<< 1$). Choose your own way to implement these operations in Verilog.

Furthermore, we are going to add branch if equal (beq) and branch if not equal (bne) operations to our partial ALU. These operations compare the values of the ALU inputs and based on the result of the comparison set the take_branch signal to either 1 or 0, where 1 indicates the branch condition is true and 0 indicates the brancy condition is false.

3.1 Prelab

Download the prelab assignment for this lab from the course webpage. The TAs will specify how to hand in solutions to the prelab.

3.2 Entering Your Design

Start Vivado IDE. Create a new project called `lab3`. Make sure that the device type is set correctly in the project settings.

Rename your verilog file from lab 2 (eightbit_palu.v) to eightbit_alu.v. Be sure to also rename the module within the file. Add the new functionality (shift and branch instructions) to your verilog code. To do this, you will need to extend the `case` statement in the partial ALU. Don't forget to also add the *take_branch* output to your ALU design. This output should always be zero except for branch operations when it could either 1 or 0, based on the result of a comparison.

Design logic that shifts input *a* by 1 (either to the right or to the left) and assigns the shifted value to the *f* output keeping the sign of the original input. For the branch equal and branch if not equal operations, use comparison operations to implement their functionality. The *f* output should always be 0 in branch operations.

==========

For Verilog code describing combinational circuits, you should provide a value for all outputs.

==========

3.3 Simulating Your Design

The circuit you have just created has nineteen bits of input: two eight-bit data inputs and a three-bit control input. It is possible to test all $2^{19} = 524288$ combinations, but it could take a long time and the results may be difficult to interpret.

When designing a testbench for a circuit, keep in mind that the goal is to give the observer (in this case, the TA) confidence that your circuit works. It is up to you to determine what *test vectors* you will use to test the functionality of your circuit. If you are having difficulty figuring out a good set of test vectors, ask the TAs for assistance.

Create a new testbench module called `alu_unit_tb`. Enter the test vectors that you have chosen into the testbench, save the testbench, and run it in XSIM. Make sure that you set the `Simulation Run Time` property to an appropriate value before running your simulation. Save and print your waveform. This should be part of what you submit.

3.4 Testing in Hardware

The next step is to test your design in hardware. In this lab we want to have full control over the ALU inputs, but since there are only eight switches on the PYNQ-Z2, we can not access all of the 19 bits of ALU inputs (16-bit operands and 3-bit selector bus). Instead we are going to use "Virtual Inputs" to the circuit by using the *Virtual Input/Ouput (VIO)* IP core from Xilinx.

VIO is an IP core, which provides a GUI-based software access to the ALU inputs. Basically, it provides an interface to the user so that s/he can enter the required input value and watch the output of the ALU at the same time. You can learn more about VIO here `http://www.xilinx.com/products/intellectual-property/vio.html#overview`.

To add this core design you should 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. We need 3 Input Probes for monitoring the ALU's outputs i.e. f, ovf and take_branch, and 3 Output Probes for controlling the inputs to the ALU which are a, b and s, so change the value of Input Probe Count and Output Probe Count fields to 3, in the *General Options* tab.

==========
The OUTPUTs from VIO connect to the INPUTS of your hardware design, and the INPUTs of VIO connect to the OUTPUTs of your hardware design.
==========

Next, in the Probes_IN Ports tab, set the IN probe width for IN0 to 8 and IN1 and IN2 to 1 for ALU's f, ovf and take_branch outputs respectively. In the Probes_OUT Ports tab, set the OUT probe width for OUT0 and OUT1 to 8 and OUT2 to 3 for ALU's a, b and s inputs respectively. Leave other settings unchanged. Then press `Ok`, to see the output product generation prompt. Click on `Generate` for Vivado to create the necessary files for the VIO IP core.

Next, you need a top module that connects the ALU (similar to Lab 2) along with the VIO. Use the *eightbit_alu_top.v* module from the course webpage. Remember to

set the *eightbit_alu_top.v* as the top module.
You can find the instantiation template for VIO core from Sources Pane, IP Sources tab. It is inside the vio_0.veo file under the Instantiation Template folder (double click to open it and copy the template to your design, then connect the ports properly).

Finally, you need to add a constraints file. For that you will be provided with an XDC file.

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.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.

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 zynq.

Find `hw_vio_1` and double click on it from the hardware panel to open it.

==========
If you previously had vio open and set to values, you may need to reset the values. To do this, enter the sequence of commands below on the TCL console:

```
reset_hw_vio_outputs [get_hw_vios {hw_vio_1}]
refresh_hw_vio -update_output_values [get_hw_vios {hw_vio_1}]
```

If you navigate away from the vio window, you can get back to it by clicking on Hardware Manager, Open Target.
==========

3.4.3 Testing your Design

Test your circuit using the test vectors you used for simulation. Enter the ALU's desired input values using VIO Out Probes while monitoring its outputs on the VIO IN probe interface and also on the LEDs. If you right click any input and select "Radix" you can change the input data type to binary, signed decimal, or any desired type.

Your TA will provide instructions regarding what is expected to be handed in for this lab experiment.