Part 1. Designing Generic N-bit Modules

Step 1-1: Modify your 2-bit *Ripple Carry Adder* design from Lab 1 to operate on N-bit wide operands, using *generic* and *generate* statements.

Step 1-2: Create a testbench that stimulates the inputs to all possible combinations and observes the outputs. Generate a simulated timing diagram and a truth table for a 2-bit instantiation of the Unit Under Test (UUT).

Step 1-3: For the subtract function, you will need to first create a full subtractor. You will need this full subtractor to create an N-bit Ripple Carry Subtractor design. Follow the same design and testbench practices as in the N-bit Ripple Carry Adder design. Generate a simulated timing diagram and a truth table for a 2-bit instantiation of UUT.

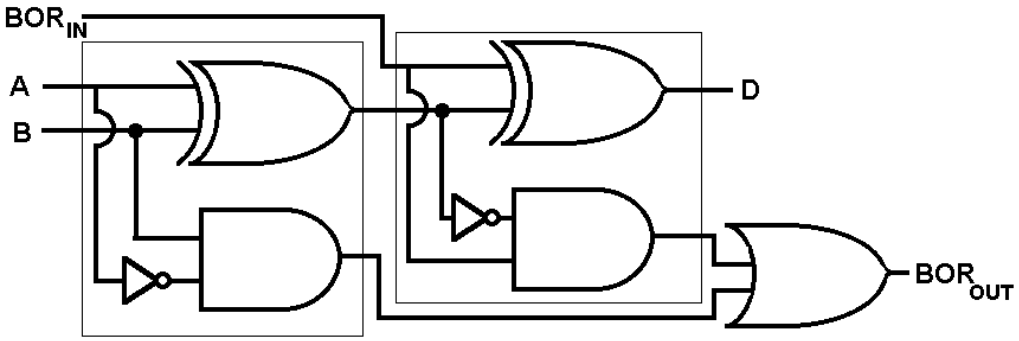

Figure 1: Full Subtractor Diagram

Step 1-4: You should notice after simulating the *Subtractor*, that the output of negative numbers will produce the binary representation of a negative number. For example, 0010 – 0100 should produce 1110 with a borrow out, or overflow, of 1. Unfortunately, this won't display as '-2' automatically on your seven segment display. Instead you should see 'E', with the overflow LED on for the negative sign (-). The simplest way to fix this is to use two's complement.

You have been provided a *Two's Complement* function to use in your design, however you are still expected to understand how it has been implemented and what it does. Figure 3 shows a circuit diagram of the XOR/OR gate network that outputs the two's complement of a 5-bit number. The resulting logic is simple, the first bit is always the same, then the following bits will either remain unchanged or be inverted, depending on if any of the bits preceding it are 1's. This design follows the "***starting from the least significant bit, going towards the most significant bit: copy each 0 until you get to the first 1, copy the first 1, and then invert all other bits***" two's complement conversion method and implements it using a cascaded XOR/OR gate chain.

As it can be seen from Figure 2, the implementation of this function results in a repeatable hardware structure starting from the output of the second bit. Each following bit uses exactly one more OR and one more XOR gate.

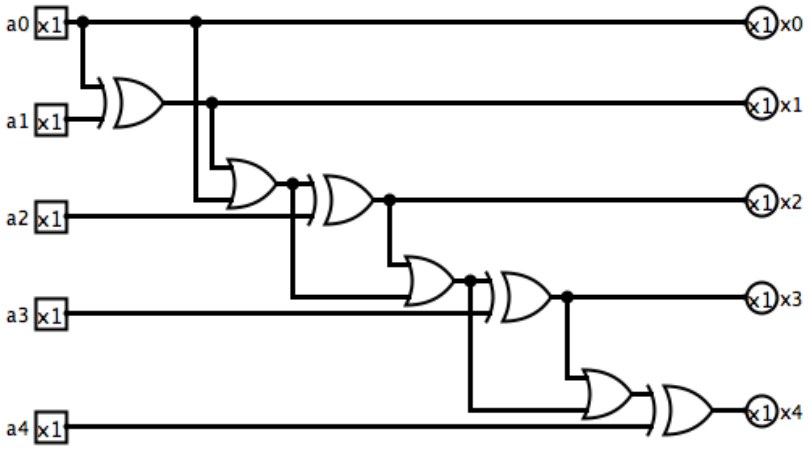

Figure 2: Two's Compliment Implemented using a series of XOR and OR gates.

Step 1-5: Use a testbench to generate a simulated timing diagram and a truth table for a 4-bit instantiation of the *Two's Complement* function.

Part 2. Designing the *Overflow* and *Output* Multiplexors

Step 2-1: Design a 1-bit 2-to-1 Multiplexor (MUX) using the "*with* SEL *select*" dataflow construct. The output of this mux will then be used to light an on-board LED to indicate overflow, which in the case of a subtraction means a negative number.

Step 2-2: To select a single function's output to display on the Seven Segment LED display, a multiplexer needs to be designed, that uses the *Function Select* input as the select, and the function outputs as its inputs. What needs to be done next is to design a MUX that selects which one of the 11 function outputs will pass through to the *Seven_Segment_Driver* (*SSDriver.vhd*). This is the *Output MUX* in the ALU block diagram. The simplest way to do this is with the VHDL *select* statement, which you can see an example of its use in the *Seven_Segment_Driver*. Map the Output MUX's inputs to the *Function Select* in accordance with Table 1. For all remaining unused inputs (*Function Select* values 1011 to 1111), make sure to specify the output as undefined ('*U*' std_logic). Also make sure to use a *generic* value that defines the bit-width of the operands passed in and out of the MUX.

Table 1. Output MUX Functions with Control Signals

Function Select	Function
0000	Add
0001	Subtract
0010	AND
0011	Two's Complement
0100	OR
0101	XOR
0110	NOT
0111	Shift Left
1000	Shift Right
1001	Rotate Left
1010	Rotate Right
1011	--
1100	--
1101	--
1110	--
1111	--

*Because Overflow Mux uses the rightmost bit to select between the Add and Subtract functions, it's important that the rightmost bit of the Two's Complement matches that of the Subtract function, otherwise you won't get the correct overflow for the Two's Complement output.

Part 3. ALU Mixed Modeling Design

Step 3-1: Create an *ALU.vhd* file with an *ALU* module that has the same ports as the ALU shown in Figure 1, except that the inputs A and B, and the output R should be defined as generic N-bit wide vectors. Declare components for the N-bit Ripple Carry *Adder* and *Subtractor*, the *Output MUX*, the Overflow MUX, and for the *Two's Complement* modules. Instantiate each one of the components within the *ALU* architecture once. Connect the outputs of the *Adder*, *Subtractor*, and *Two's Complement* internal functions to the inputs of the *Output MUX*. Make sure to follow the assignment order shown in Table 2, and Figure 1. Connect the overflow bits of the *Adder* and *Subtractor* modules to the 2-to-1 *Overflow MUX*, and make sure to assign the least significant bit of the *Function Select* input as the select input to the 2-to-1 *Overflow MUX* instantiation.

Step 3-2: For the remaining binary and unary operators, use dataflow assignments to internal *ALU* architecture signals, and then connect these signals to the *Output MUX* in the order shown in Table 2, and Figure 1.

Step 3-3: Write a testbench for the ALU. Verify that it works as intended before proceeding. Simulate with the test vectors shown in the *Lab Work* part 6. Save the resulting waveform and truth table and include them in your lab report.

Table 2. ALU Functions

Instruction	Function
Add	Ripple Carry Adder module
Subtract	Ripple Carry Subtractor module
AND	**Output <= A and B**
Two's Complement	Two's Complement module
OR	**Output <= A or B**
XOR	**Output <= A xor B**
NOT	**Output <= not A**
Shift Left	**Output <= A(N-2 *downto* 0) & '0'**
Shift Right	**Output <= '0' & A(N-1 *downto* 1)**
Rotate Left	**Output <= A(N-2 *downto* 0) & A(N-1)**
Rotate Right	**Output <= A(0) & A(N-1 *downto* 1)**

Part 4. Top-Level Design

Step 4-1: Create a *Lab2.vhd* file that instantiates and interconnects, as shown in Figure 1, one ALU and one 7-Segment Driver, a module which was provided to you. Map the instantiated ALU's *generic* to 4 bits. Synthesize the design. Review the elaborated schematic to make sure the RTL design synthesized mimics the RTL design shown in Figure 1.

Step 4-2: You will be using Vivado's *I/O Planning* tool in order to design your constraints file. However, before you can do this, you will need to know which pins to assign your inputs and outputs to. You have been provided two PMOD devices for this lab. The first is a set of four additional switches. The schematic for this PMOD device is shown in Figure 4.

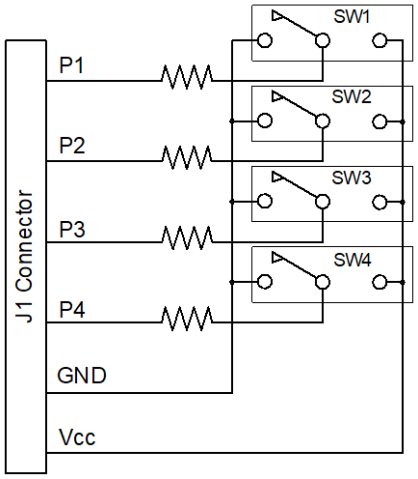

Figure 3. Four Switch PMOD Device Diagram

Refer to the PMOD section of the *ZYBO Reference Manual* for how to use the PMOD ports. This will also be how you determine the pin name and number specifics that you will need to assign in the constraints file. These switches should be used for your control input, while the onboard switches and buttons should be used for the two binary number inputs. Your constraints should match the assignments listed in Table 3.

Table 3. ZYBO Signal Assignments

PMOD Switches	Onboard Switches	Onboard Buttons
SW1 = Function_Select (3)	SW3 = A(3)	BTN3 = B(3)
SW2 = Function_Select (2)	SW2 = A(2)	BTN2 = B(2)
SW3 = Function_Select (1)	SW1 = A(1)	BTN1 = B(1)
SW4 = Function_Select (0)	SW0 = A(0)	BTN0 = B(0)

Step 4-3: The second PMOD device you have been provided is a Seven Segment Display. The schematic for this PMOD accessory is shown in Figure 5. This device actually allows for two digits to be displayed, however, for this lab you will only be using the first digit.

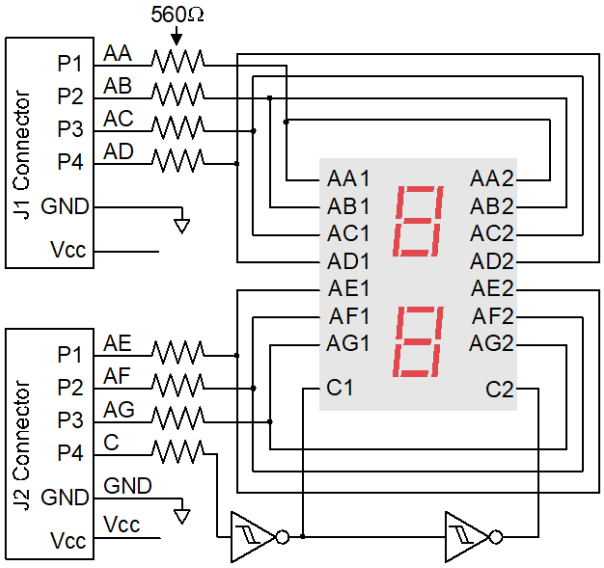

Figure 4. Seven Segment Display PMOD Device Diagram

When connecting the seven segment display to the onboard PMOD ports, make sure to use two of the High-Speed PMOD Ports, as only one standard PMOD port has been provided and this device requires two ports. This means you should use only PMOD JB, JC, or JD. As you will only be using one of the seven segment displays, you do not need to do anything with the common cathode. You only need to assign pins for P1-P4 of the J1 Connector, and P1-P3 of the J2 Connector.

Step 4-4: Finally, for the overflow LED, you should use one of the available onboard LEDs. Your board should match the one showed below if the correct PMOD ports as used in the constraints file. Make sure when plugging in the PMOD devices that they are placed in the top row of each of the PMOD ports.

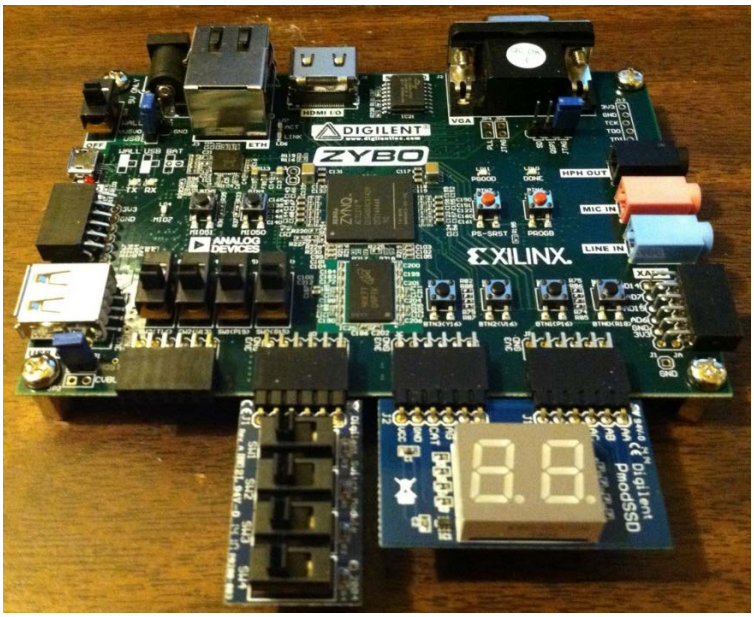

Figure 5. Board Layout for Lab 2 Design Constraints.