Laboratory Exercise 6

Finite State Machines
October 22, 2018

Learning Objectives

The purpose of this lab is to:

1. learn how to write Finite State Machines (FSMs) in Verilog
2. learn how to use an FSM to control the sequencing of logical operations.

Preparation Before the Lab

You are required to complete Parts I to III of the lab by writing and testing Verilog code and compiling it with Quartus. Show your relevant preparation (schematics, Verilog, and simulations) for Parts I to III to the teaching assistants. You must simulate your circuit with ModelSim (using reasonable test vectors you can justify). You should also show state diagrams for Parts II and III.

In-lab Work

You are required to implement and test all of Parts I to III of the lab. You need to demonstrate them to the teaching assistants.

Part I

In this part you will implement a basic finite state machine (FSM) in Verilog. All FSMs you write in Verilog should follow this structure or you can get into lots of trouble.

We wish to implement a FSM that recognizes two specific sequences of applied input symbols, namely four consecutive 1s or the sequence 1101. There is an input w and an output z. Whenever $w = 1$ for four consecutive clock pulses, or when the sequence 1101 appears on w across four consecutive clock pulses, the value of z has to be 1; otherwise, $z = 0$. Overlapping sequences are allowed, so that if $w = 1$ for five consecutive clock pulses the output z will be equal to 1 after the fourth and fifth pulses. Figure 1 illustrates the required relationship between w and z for an example input sequence. A state diagram for this FSM is shown in Figure 2.

Figure 3 shows a partial Verilog file for the required state machine. Study and understand this code as it provides a model for how to clearly describe a finite state machine that will both simulate and synthesize properly.

The toggle switch SW_0 on the DE1-SoC board is an active-low synchronous reset input for the FSM, SW_1 is the w input, and the pushbutton KEY_0 is the clock input that is applied manually. The LED $LEDR_9$ is the output z, and the state flip-flop outputs are assigned to $LEDR_{3-0}$.

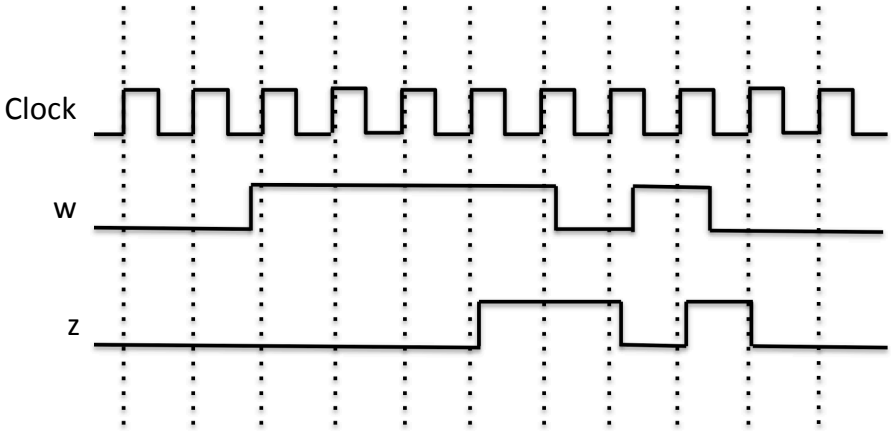

Figure 1: Required timing for the output z.

Perform the following steps:

1. Begin with the template code provided online `sequence_detector.v`. **(PRELAB)**
2. Complete the state table and the output logic. **(PRELAB)**
3. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. **(PRELAB)**
4. Compile the project. **(PRELAB)**
5. Download the compiled circuit into the FPGA. Test the functionality of the circuit on your board.

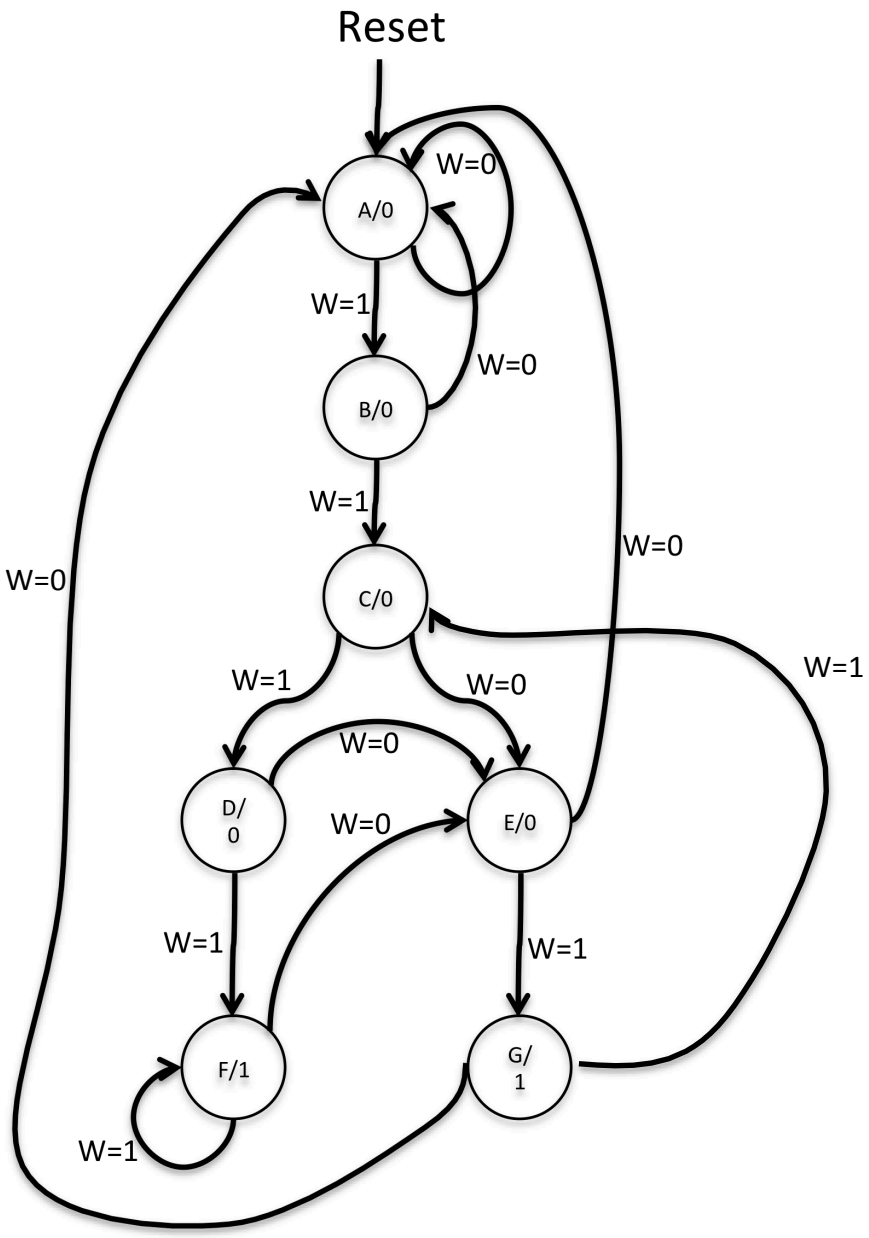

Figure 2: A state diagram for the FSM.

```
//SW[0] reset when 0
//SW[1] input signal

//KEY[0] clock signal

//LEDR[3:0] displays current state
//LEDR[9] displays output

module sequence_detector(SW, KEY, LEDR);
    input [9:0] SW;
    input [3:0] KEY;
    output [9:0] LEDR;

    wire w, clock, reset_b;

    reg [3:0] y_Q, Y_D; // y_Q represents current state, Y_D represents next state
    wire out_light;

    parameter A = 4'b0000, B = 4'b0001, C = 4'b0010, D = 4'b0011, E = 4'b0100, F = 4'b0101, G = 4'b0110;

    assign w = SW[1];
    assign clock = ~KEY[0];
    assign reset_b = SW[0];

   // State table
   // The state table should only contain the logic for state transitions
   // Do not mix in any output logic.  The output logic should be handled separately.
   // This will make it easier to read, modify and debug the code.

   always @(*)
     begin: state_table
        case (y_Q)
            A: begin
                   if (!w) Y_D = A;
                   else Y_D = B;
               end
            B: begin
                   if(!w) Y_D = A;
                   else Y_D = C;
               end
            C: ???
            D: ???
            E: ???
            F: ???
            G: ???
            default: Y_D = A;
        endcase
    end // state_table


   // State Registers

    always @(posedge clock)
    begin: state_FFs
        if(reset_b == 1'b0)
            y_Q <= 4'b0000;
        else
            y_Q <= Y_D;
    end // state_FFS


   // Output logic
   // Set out_light to 1 to turn on LED when in relevant states

   assign out_light = ((y_Q == ???) | (y_Q == ???));

   // Connect to I/O

    assign LEDR[9] = out_light;
    assign LEDR[3:0] = y_Q;
endmodule
```

Figure 3: Verilog code for the FSM.

Part II

Please note that there is a lot written here, but a lot of it is explanation and guidance, so please read carefully.

A *finite state machine* (FSM) on its own, like the one built in Part I, cannot do much and is not what you usually do with an FSM except to teach how to build an FSM. The primary use of FSMs are to act as the main control for digital systems that require functions like sequencing or responding in different ways to some stimuli. This part will show you how to use an FSM to do something more interesting than recognizing a pattern of bits. To help you focus on the FSM design, you are provided an entire datapath and example FSM that performs a computation. Your task will be to change the FSM to do a different computation.

Most non-trivial digital circuits can be separated into two main functions. One is the *datapath* where the data flows and the other is the *control path* that manipulates the signals in the datapath to control the operations performed and how the data flows through the datapath. In previous labs, you learned how to construct a simple ALU, which is a common datapath component. In Part I of this lab you have already constructed a simple FSM, which is the most common component used to implement a control path. Now you will see how to implement an FSM to control a datapath so that a useful operation is performed. This is an important step towards building a microprocessor as well as any other computing circuit.

In this part, you are given a datapath and an FSM that controls the datapath so that it computes $A^2 + C$. Observe that the example code loads all four values, despite using only two of the values in the computed result. This will give you a head start because you will need all four values loaded for the task you will do. Also, often when you inherit code, there might be parts of it that may not be used or be relevant because of how the code evolved. It is up to you to figure out what is relevant for the task you are given. Feel free to modify the code you are given, if you feel it is necessary to achieve the final result required.

Using the given datapath, you are required to implement an FSM that controls the datapath so that it performs the computation:

$$Ax + Bx^2 + C$$

The values of x, A, B and C will be preloaded by the user on the switches before the computation begins.

Figure 4 shows the block diagram of the datapath you will build. Resets are not shown, but do not forget them. The datapath will carry 8-bit unsigned values. Assume that the input values are small enough to not cause any overflows at any point in the computation, i.e., no results will exceed $2^8 - 1 = 255$. The ALU needs only to perform addition and multiplication, but you could use a variation of the ALU you built previously to have more operations available for solving other equations if you wish to try some things on your own. There are four registers R_x, R_A, R_B and R_C used at the start to store the values of x, A, B and C, respectively. The registers R_A and R_B can be overwritten during the computation. There is one output register, R_R, that captures the output of the ALU and displays the value in binary on the LEDs and in hex on the HEX displays. Two 8-bit-wide, 4-to-1 multiplexers at the inputs to the ALU are used to select which register values are input to the ALU.

All registers have enable signals to determine when they are to load new values and an active low synchronous reset.

The provided circuit operates in the following manner. After an active low synchronous reset on ***KEY***$_0$, you will input the value for R_A on switches $SW[7:0]$. When ***KEY***$_1$ is pushed and released, R_A will be loaded and then you will input the next value on the switches that will be loaded into R_B. Press and release ***KEY***$_1$ again. Likewise for R_C and R_X. Computation will start after ***KEY***$_1$ is pressed and released for loading R_X. When computation is finished, the final result will be loaded into R_R. This final result should be displayed on ***LEDR***$_{7-0}$ in binary and *HEX0* and *HEX1* in hex. You will use *CLOCK_50* as your clock.

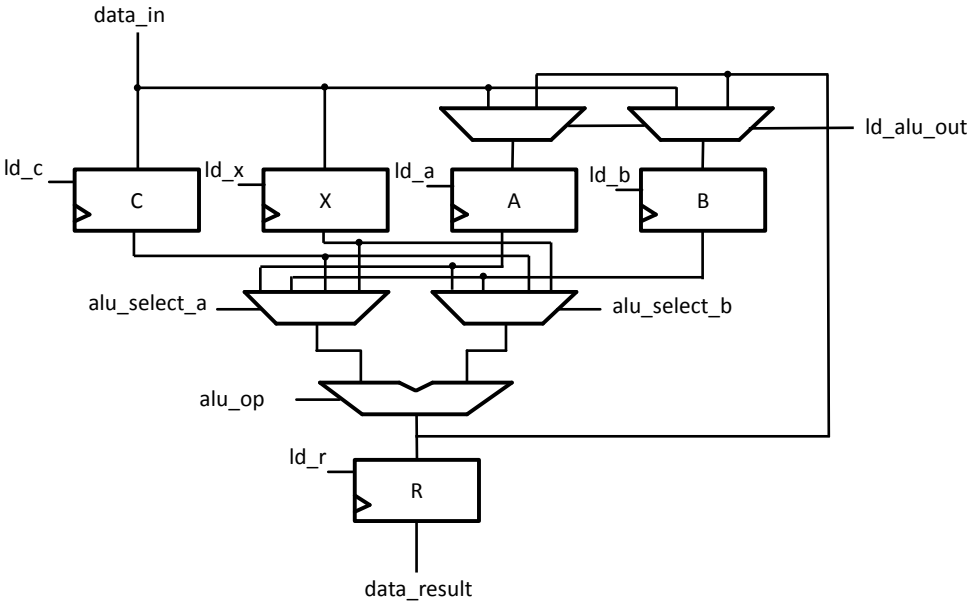

Figure 4: Block diagram of datapath.

Table 1: Register contents and control signals for computing $A^2 + C$

	Reset	1	2	3	4	5	6	7	
data_in		5	4	3	2	2	2	2	
RA		0	5	5	5	5	25	25	
RB		0	0	4	4	4	4	4	
RC		0	0	0	3	3	3	3	
Rx		0	0	0	0	2	2	2	
RR		0	0	0	0	0	0	28	
ld_a		1	0	0	0	1	0	1	
ld_b		0	1	0	0	0	0	0	
ld_c		0	0	1	0	0	0	0	
ld_x		0	0	0	1	0	0	0	
ld_alu_out		0	0	0	0	1	0	0	1 = select alu output
alu_select_a		0	0	0	0	0	0	0	0 = select A
alu_select_b		0	0	0	0	0	2	0	2 = select C
alu_op		0	0	0	0	1	0	0	0 = add, 1 = multiply
ld_r		0	0	0	0	0	1	0	

Perform the following steps:

1. Examine the Verilog code provided online in `poly_function.v`. This is a major step in this part of the lab. You will not need to write much Verilog, but you will need to fully understand the provided Verilog to make your modifications. Here's how to read the code and what to look for:

   (a) Observe that the top-level module instantiates a module called `part2` and the hex decoder modules for output. The `part2` module contains the main functionality of the design. The top-level module just connects the LEDs and HEX displays to the outputs of the `part2` module. This approach makes it clear how the actual input/outputs are connected.

   (b) The `part2` module has two modules called `datapath` and `control`, which is the explicit partitioning of control logic and the datapath logic. The code has been organized this way to make it very clear where the control logic is written versus the datapath logic. Most important is that you can clearly see what control signals coming from the control logic connect to the datapath.

   (c) Read the datapath code and identify all the components shown in Figure 4 when looking at the datapath module.

   (d) Identify all the control signals shown in Figure 4 and where they are generated in the control module. You should see that all the control signals are generated by the FSM in the control module.

   (e) Find the case statement that defines the state table for the FSM. Also, find the case statement that defines the outputs that are set in each state. It is important that the state transitions and the output logic be in separate case statements. This makes it explicitly clear what logic is being generated from your Verilog. If you start mixing the state transitions with the outputs, the code becomes very challenging to understand, with the risk that the Verilog compilation will also do something you did not intend.

   By looking at the state transitions and what outputs are set in each state, observe how the loading of the four registers is done and how the sequence of the loading is done according the specification. How does the FSM handle the pressing and releasing of the input `go` signal that is wired to KEY_1?

   You may find it helpful to draw a state diagram because you can then just modify it for Step 4.

   (f) Which states do the actual computation? Identify those states and find the sequence of control signals. Observe how the computation is done by using Figure 4 and seeing how that sequence of control signals does the required operation.

   (PRELAB)

2. Simulate the given circuit. Make sure you observe the state register of the FSM so that you can watch the state transitions. You should be able to use the same simulation script after you modify the circuit to do the new computation, except you may need to account for there being more operations being performed, so more simulation time will be needed.

   Before making changes to any design, it is always good to start from a known state, which in this case is a working example and a working simulation script. If something goes wrong, go back to when the design last worked and figure out what change caused the error. **(PRELAB)**

3. Table 1 shows a table of the sequence of register contents and control signals to do the computation of $A^2 + C$. You should be able to see the control signals changing the same way in the controller state machine output logic. Following the model of the provided design, build the table for computing $Ax + Bx^2 + C$. Constructing this table will also be discussed in the lectures. **(PRELAB)**

4. Draw a state diagram for your controller starting with the register load states provided in the example FSM Verilog code. **(PRELAB)**

5. Modify the provided FSM Verilog to implement your controller and synthesize it. You should only need to modify the control module. **(PRELAB)**

6. To examine the circuit produced by Quartus open the RTL Viewer tool (Tools > Netlist Viewers > RTL Viewer). Find (on the left panel) and double-click on the box shown in the circuit that represents the finite state machine, and determine whether the state diagram that it shows properly corresponds to the one you have drawn. To see the state codes used for your FSM, open the Compilation Report, select the Analysis and Synthesis section of the report, and click on State Machines.

   The state codes after synthesis may be different from what you originally specified. This is because the tool may have found a way to optimize the logic better by choosing a different state assignment. If you really need to use your original state assignment, there is a setting to keep it. **(PRELAB)**

7. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. It is recommended that you start by simulating the datapath and controller modules separately. Only when you are satisfied that they are working individually should you combine them into the full design. Why is this approach better? (Hint: Consider the case when your design has 20 different modules.) **(PRELAB)**

8. After you are satisfied with your simulations, download and test the functionality of the circuit on the FPGA board.

Part III

In this part, you will have to build a complete datapath and corresponding state machine. We recommend that you still have two separate modules for the datapath and the control just to help you understand the separation of what is in a datapath and what is in a controller. In Part II, the control logic just generated control signals that were inputs to the datapath. In this part, you will see that there is a signal that needs to go from the datapath into the controller so make sure you account for it accordingly.

Division in hardware is the most complex of the four basic operations. Add, subtract and multiply are much easier to build in hardware. For this part, you will be designing a 4-bit restoring divider using a finite state machine.

Figure 5 shows an example of how the restoring divider works. This mimics what you do when you do long division by hand. In this specific example, number 7 (*Dividend*) is divided by number 3 (*Divisor*). The restoring divider starts with *Register A* set to *0*. The *Dividend* is shifted left and the bit shifted out of the left most bit of the *Dividend* (called the most significant bit or MSB) is shifted into the least significant bit (LSB) of *Register A* as shown in Figure 6.

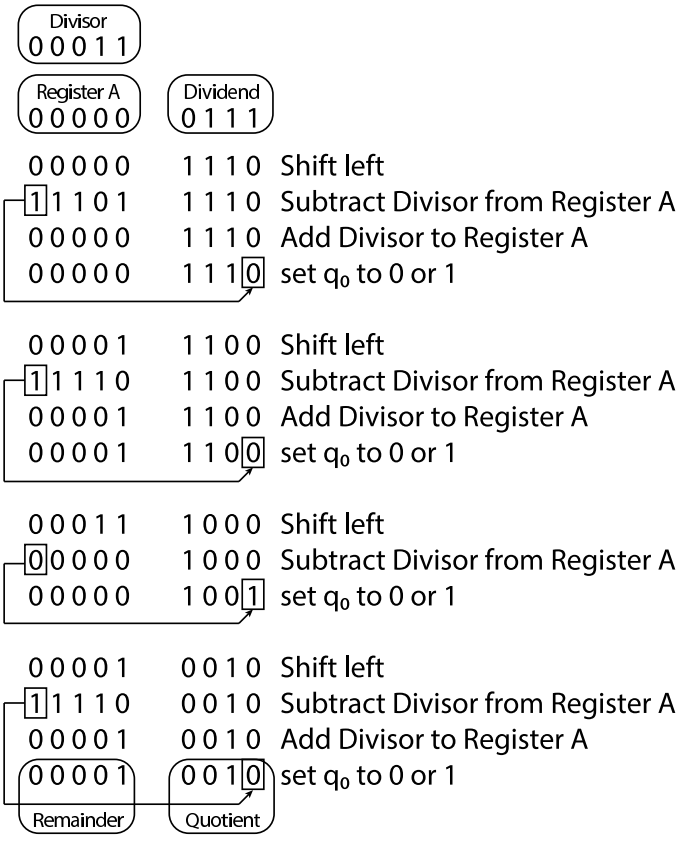

Figure 5: An example showing how the restoring divider works.

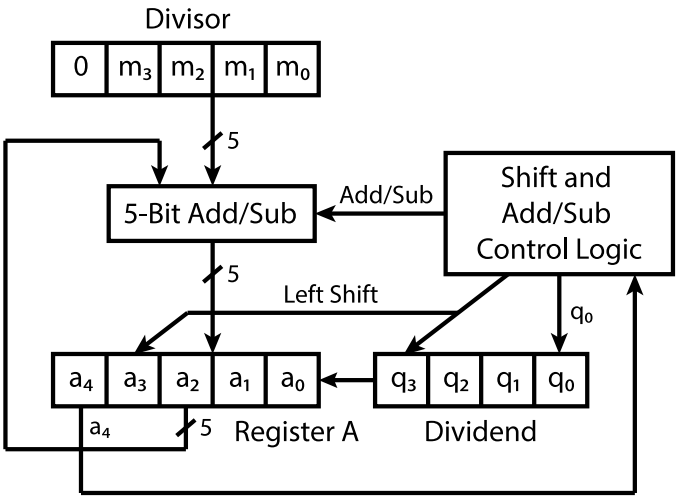

Figure 6: Block diagram of restoring divider.

The *Divisor* is then subtracted from *Register A*. If the MSB of *Register A* is a *1*, then we restore *Register A* back to its original value by adding the *Divisor* back to *Register A*, and set the LSB of the *Dividend* to *0*. Else, we do not perform the restoring addition and immediately set the LSB of the *Dividend* to *1*. You may use the subtract $(-)$ and addition $(+)$ operators in Verilog to perform the subtraction and addition. As you will see in later lectures, the *1* in the MSB of *Register A* means that the value in *Register A* after the subtraction is a negative number, meaning that the *Divisor* is larger than the original value in *Register A*. That is why *Register A* is *restored* by adding back the *Divisor*.

This sequence of steps is performed until all the bits of the *Dividend* have been shifted out. Once the process is complete, the new value of the *Dividend* register is the *Quotient*, and *Register A* will hold the value of the *Remainder*.

To implement this part, you will use SW_{3-0} for the divisor value and SW_{7-4} for the dividend value. Use *CLOCK_50* for the clock signal, KEY_0 as a synchronous active high reset, and KEY_1 as the *Go* signal to start computation. The output of the *Divisor* will be displayed on *HEX0*, the *Dividend* will be displayed on *HEX2*, the *Quotient* on *HEX4*, and the *Remainder* on *HEX5*. Set the remaining HEX displays to *0*. Also display the *Quotient* on *LEDR*.

Structure your code in the same way as you were shown in Part II.

Perform the following steps.

1. Draw a schematic for the datapath of your circuit. It will be similar to Figure 6. You should show how you will initialize the registers, where the outputs are taken, and include all the control signals that you require. Do not forget the clock and resets. **(PRELAB)**
2. Draw the state diagram to control your datapath. Check it by hand simulating the example shown in Figure 5. Hand simulation just means to work through the steps using your schematic and state diagram to check whether you can do the required operations before going through the effort of setting up the simulator. This may not catch all bugs, but it is a good step to make sure you have a design that has a chance of working. **(PRELAB)**
3. Draw the schematic for your controller module. **(PRELAB)**
4. Draw the top-level schematic showing how the datapath and controller are connected as well as the inputs and outputs to your top-level circuit. **(PRELAB)**
5. Write the Verilog code that realizes your circuit. **(PRELAB)**

6. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. Start with Figure 5 as an example because it shows you all the steps with the values that should be in the registers at each step. **(PRELAB)**

7. After you are satisfied with your simulations, download and test the functionality of the circuit on the FPGA board.