Laboratory Exercise #11
A Simple Digital Combination Lock

ECEN 248: Introduction to Digital Design

Department of Electrical and Computer Engineering
Texas A&M University

1 Introduction

A classic approach to access control is the rotary combination-lock (Figure 1), which can be found on safes, doorways, post-office boxes, etc. This typically mechanical device requires that a person wishing to gain access to a particular entryway rotate a knob left and right in a given sequence. If the sequence is correct, the entryway will unlock; otherwise, the entryway will remain locked, while conveying no information about the required sequence. In lab this week, we will attempt to recreate this conventional mechanism in digital form on the Spartan 3E board. To accomplish such a feat, we will introduce the *Moore machine*, which is a type of Finite State Machine (FSM), and the state diagram, which is a convenient way to represent an FSM. To prototype our combination-lock, we will make use of the rotary knob and LCD display on the Spartan 3E board. The exercises in this lab serve to reinforce the concepts covered in lecture.

Figure 1: A Rotary Combination-Lock

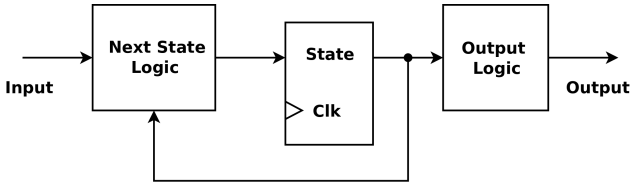
2 Background

Background information necessary for the completion of this lab assignment will be presented in the next few subsection. The pre-lab assignment that follows will test your understanding of the background material.

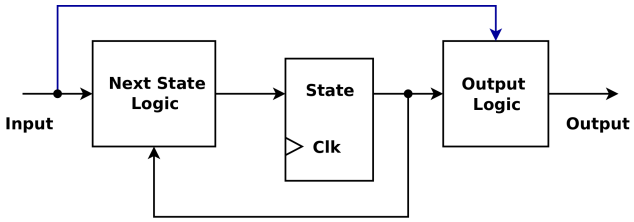
2.1 The Moore Machine

A Finite State Machine (FSM) is an abstract way of representing a sequential circuit. Often times the terms, FSM and sequential circuit, are used interchangeably; however, an FSM more formally refers to the model of a sequential circuit and can model higher-level systems, such as computer programs, as well. An FSM can be broadly classified into one of two types of machines, namely *Moore* or *Mealy*. Simply put, a *Moore* machine is an FSM such that the output depends solely on the state of the machine, whereas a *Mealy* machine is an FSM such that the output depends not only on the state but also the input.

Figure 2 differentiates between the *Moore* and *Mealy* machine with a blue wire, which on the *Mealy* machine, connects the input to the output logic. Other than the blue wire, the two machines are identical. As shown, combinational logic generates the next state based on the current state and input to the machine, while the flip-flops store the current state. The output logic is purely combinational as well and depends on



(a) Moore Machine



(b) Mealy Machine

Figure 2: Moore vs. Mealy Machine

the state in both machine. For this week's lab, we will design a *Moore* machine because it fits our application quite well; however, for the sake of comparison, we will design a *Mealy* machine next week.

2.2 State Diagrams and Behavioral Verilog

The operation of an FSM can be described with a directed graph such that the vertices represent the states of the machine and the edges represent the transitions from one state to the next. This sort of graphical representation of an FSM is known as a *state diagram*. Figure 3 depicts the state diagram for a 2-bit saturating counter. If you recall from previous labs, a saturating counter is one that will not roll over but rather, will stop at the minimum or maximum counter value. Let us examine the state diagram and convince ourselves that this is the case.

Each transition or edge of the graph in Figure 3 is marked by are particular input value that causes that transition. For this simple machine, there is only one input (with the exception of *Rst*) so we are able to explicitly represent all input combinations in the diagram. As we will see later on, this is not always the case. For each vertex, the state is clearly labeled (i.e. $S0$, $S1$, $S2$, and $S3$); likewise, the output for each state can be found under the state label. Remember that this is a *Moore* machine so the outputs are dependent only on the state of the machine. Given the state diagram of an FSM, the question to be answered now is how to go from a state diagram to a Verilog description? The next few paragraphs will demonstrate that exact process.

In lecture, you have learned how to implement an FSM by mapping the design into state tables. Then

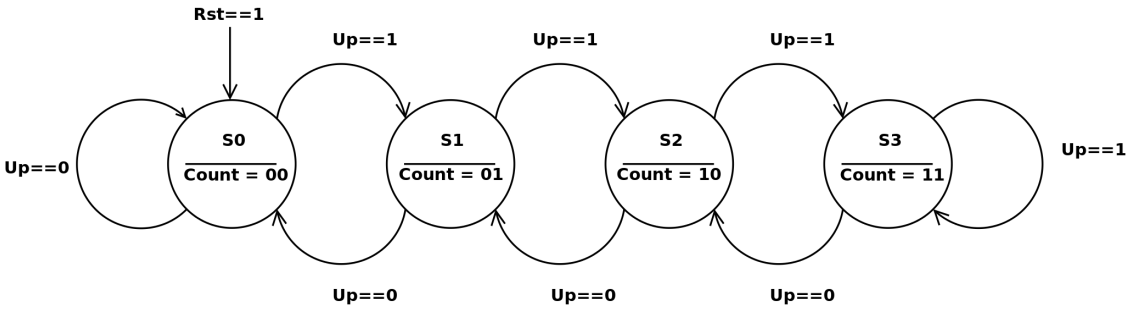


Figure 3: 2-bit Saturating Counter State Diagram

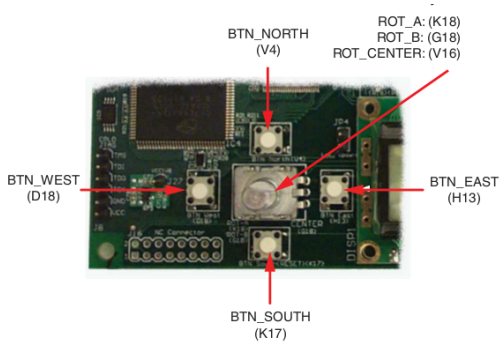
by performing logic minimization on the next-state and output expressions, you are able to realize your FSM with gates and flip-flops. In lab, however, we will take a different approach to implementing an FSM. Because we will be describing our final circuit in Verilog and synthesizing it for the Spartan 3E FPGA, the process would be much more efficient and less error-prone if we simply described the functionality of our FSM directly in Verilog using behavioral constructs. The flip-flops for holding the state of the machine can be described with a positive-edge triggered **always** block as done in previous labs, while the next-state and output logic can be eloquently described with **case** statements contained within **always@(*)** blocks. The code below demonstrates the use of these behavioral constructs to describe the FSM in Figure 3.

```
1 `timescale 1ns / 2 ns
  `default_nettype none
3
  /*This is a  behavioral Verilog description of*
5  *a 2-bit saturating counter.                  */
  module saturating_counter(
7     /*output and input are wires*/
      output wire [1:0] Count; //2-bit output
9     input wire Up; //input bit asserted for up
      input wire Clk, Rst; //the usual inputs to a synchronous circuit
11
      /*we haven't talked about parameters much but as you can see *
13     *they make our code much more readable!                      */
      parameter S0 = 2'b00,
15              S1 = 2'b01,
                S2 = 2'b10,
17              S3 = 2'b11;

19    /*intermeidate nets*/
      reg [1:0] state; //4 states requires 2-bits
21    reg [1:0] nextState; //will be driven in always block!

23    /*describe next state logic*/
      always@(*) //purely combinational!
```

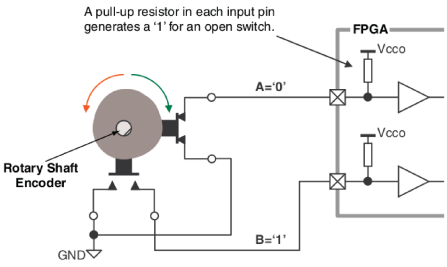
```
25          case(state)
                S0: begin
27                  if(Up) //count up
                        nextState = S1;
29                  else //saturate
                        nextState = S0;
31              end
                S1: begin
33                  if(Up) //count up
                        nextState = S2;
35                  else //count down
                        nextState = S0;
37              end
                S2: begin
39                  if(Up) //count up
                        nextState = S3;
41                  else //count down
                        nextState = S1;
43              end
                S3: begin
45                  if(Up) //saturate
                        nextState = S3;
47                  else //count down
                        nextState = S2;
49              end
                //do not need a default because all states
51              //have been taken care of!
            endcase
53
        /*describe the synchronous logic to hold our state!*/
55      always@(posedge Clk)
            if(Rst) //reset state
57              state <= S0;
            else
59              state <= nextState;

61      /*describe the output logic, which in this case *
         *happens to just be wires                      */
63      assign Count = state;

65  endmodule //that's it!
```

2.3 Rotary Knob and Character LCD on the Spartan 3E

For interacting with the user, we will utilize of the rotary knob and character Liquid Crystal Display (LCD) on the Spartan 3E board. This subsection serves as a high-level introduction to these components. For more information, consult the Spartan 3E User Guide and the Rotary Encoder Interface Guide; however, an in-



(a) The Rotary Shaft



(b) The Rotary Encoder

Figure 4: Rotary Shaft and Encoder, courtesy of Xilinx®

depth understanding is not required because you will be provided with modules that directly interact with these components. We will begin by discussing the rotary knob.

The rotary knob, mounted atop a rotary shaft (Figure 4a) on our FPGA board, has an encoder, which facilitates angular rotation detection. The encoder is made up of a cam fixed on the rotary shaft and two push-buttons. As the shaft rotates, the push-buttons open and close based on the position of the cam as shown in Figure 4b. The outputs of the encoder, A and B, are quadrature square waves, meaning they are 90° out of phase from each other. See Figure 5. One cycle of A or B indicates that the encoder has been rotated by one click or 18°. Because there are 360° in one rotation of the knob, there are 20 clicks per rotation. When the encoder is not rotating, it is in detent mode and both the A and B signals are LOW. The angular direction of the knob can be easily determined based on which switch opens first.

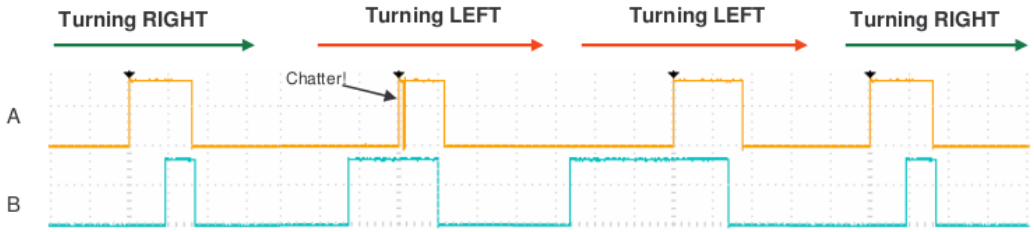


Figure 5: Output of the Rotary Encoder

As with the push-buttons used in previous labs, the rotary encoder is an electo-mechanical device, which is susceptible to switch bounce or electrical chatter. Thus, the rotary encoder module provided for you in lab and depicted in Figure 6 must first filter out electrical chatter. As seen in the diagram, the inputs to the module are the rotary signals, *rotA* and *rotB*, and the outputs are *Right* and *Left*. When the knob is rotated to

the left, the *Left* signal will pulse every 18°. Similarly, when rotated to the right, the *Right* signal will pulse.

To display the angular position of the knob and the status of the combination lock (i.e. locked or unlocked), we will make use of the 2x16 character LCD on the Spartan 3E board. As illustrated in Figure 6, the LCD driver, which is provided for you, generates the necessary signals to interact with the LCD. The interface signals allow the FPGA to communicate ASCII encoded characters to the LCD. We will now take a moment to describe the specific functions of each of these signals.

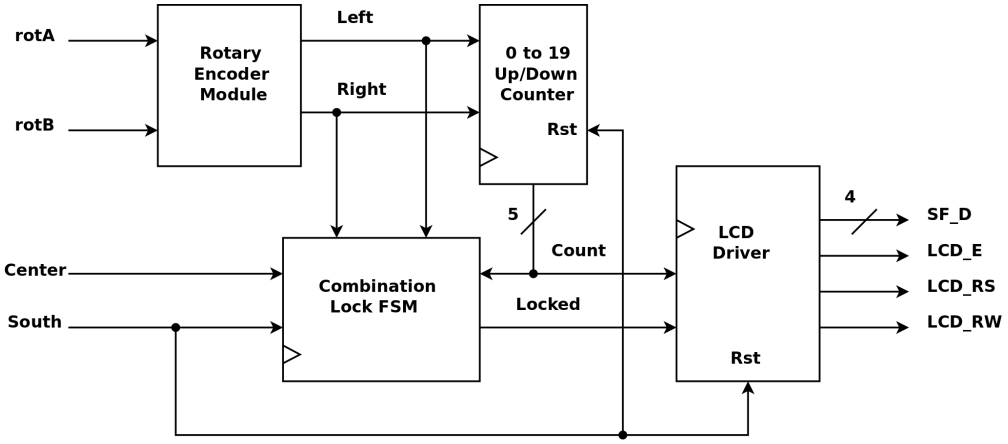


Figure 6: The Combination-Lock Diagram

The *LCD_E* is an enable signal for the LCD, while the *LCD_RW* indicates whether we are reading or writing to the memory internal to the LCD. Note that this memory is where the LCD stores 2 rows of 16 characters, encoded using ASCII, an 8-bit standard numerical representation of characters. The *LCD_RS* signal stands for register select and indicates whether the FPGA is sending character data (i.e. ASCII codes) or commands. An example command would be "clear display." Finally, *SF_D* is a 4-bit data bus for transmitting character data and instructions. Only 4-bits are used as oppose to 8-bits to reduce the number of I/O pins necessary to communicate with the LCD. Consequently, the FPGA must break each 8-bit code up into two 4-bit codes and transmit them sequentially. We would like to remind you that the details of the LCD interaction will be handled for you in the LCD driver module. The LCD driver module requires a clock, a reset, a 5-bit binary number (0 to 19), and a signal that indicates which mode the combination-lock is in (i.e. locked or unlocked).

3 Pre-lab

The objective of the pre-lab assignment is to describe in Verilog the FSM that you will load onto the Spartan 3E board during lab. Therefore, the following subsections will describe the design we are attempting to create in detail. The pre-lab deliverables at the end will state what it is you are required to have prior to

attending your lab session.

3.1 Digital Combination-Lock

In lab this week, we will design a digital combination-lock that can be used on a safe or access-controlled doorway. To do so, we will require a few components shown in Figure 6. The rotary encoder module and LCD driver have already been discussed in detail in the background section. Although these two modules are provided for you, it is imperative that you understand what signals these modules need as input and what signals they supply as output. Thus, if you have not already done so, please read over the background section.

For the remaining two components, you will be responsible for describing them in Verilog. The 0-to-19 up/down counter is a simple synchronous circuit that counts up or down in the range of 0 to 19 and rolls over in both directions. In other words, if the *Up* signal is asserted, the counter will count up until it hits 19, in which case it will roll over to 0. Likewise, when the *Down* signal is asserted, the counter will count down until it hits 0, in which case it will roll over to 19. This counter indicates the position of the rotary knob and mimics the number that the pointer on a combination-lock is pointing to. We will discuss how you would go about describing this sort of counter in Verilog during the lab session, but for now, try to understand what it is we want it to do.

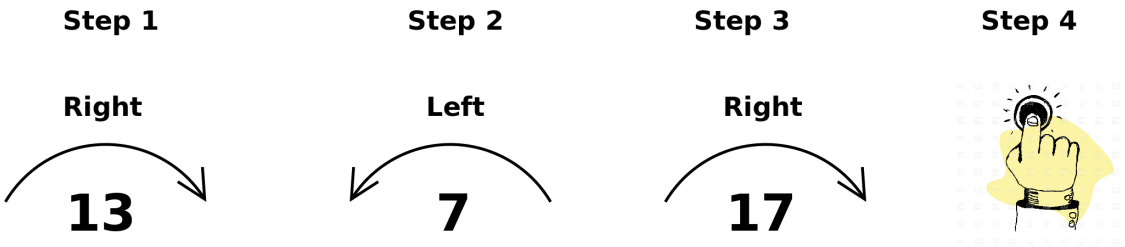


Figure 7: Rotary Combination-Lock Instructions

The combination-lock FSM is the only component we have not discussed yet. The easiest way to describe the FSM is via a *state diagram*. However, before delving into the diagram, let us take a high-level look at how our digital combination-lock is expected to operate. Figure 7 illustrates the steps outline below:

1. Rotate the knob to the right (clockwise) until the number 13 appears on the display.
2. Rotate the knob to the left (counter-clockwise) until the number 7 appears on the display.
3. Rotate the knob to the right again until the number 17 appears on the display.
4. Finally, press the knob (Center button) to unlock the entryway.
5. To lock the entryway, press the South button.

Now that we understand how we want our combination-lock to operate, we can represent the FSM for our combination-lock in a succinct manner with a state diagram. Figure 8 depicts the state diagram of the combination-lock with a some transition labels missing. Let us take a moment to examine the incomplete diagram so that you can determine how to complete it.

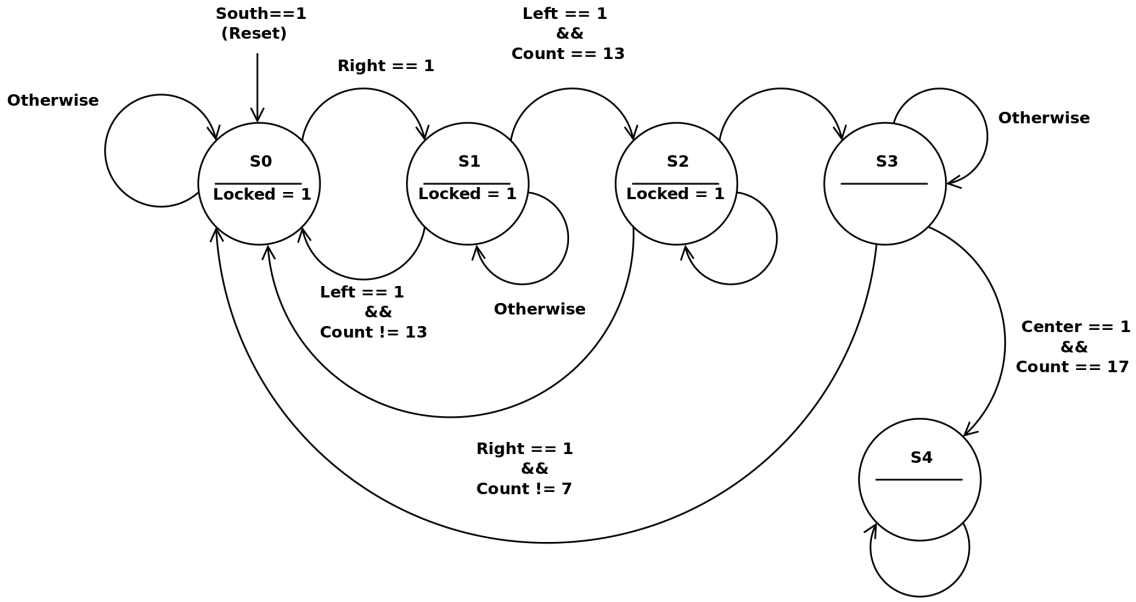


Figure 8: Rotary Combination-Lock State Diagram

The reset state is S0. This is indicated by the arrow labeled "South == 1" pointing to the S0 node. This arrow differs from the other arrows in the diagram in that it does not represent a transition from one state to the next. Rather, it signifies the fact that no matter what state the machine is in, it will return to S0 when reset is asserted (i.e. "South == 1"). We can see that the FSM will remain in state S0 until the *Right* input is equal to '1', indicating the knob is being rotated to the right. When this happens, the FSM will transition to S1, where it will remain until the knob is rotated to the left (i.e. *Left == 1*). Two transitions away from S1 are possible when *Left == 1*. If *Count == 13*, the FSM will move to state S2; otherwise, it will move back to S0. In other words, if the knob was at position 13 when it began rotating to the left, the first number combination was dialed in correctly, so the FSM will move to the next state; otherwise, it will move back to the starting state as though no combination was entered.

Hopefully, it is now clear how the FSM continues until the entire combination has been entered correctly. The final event that moves the FSM into the unlocked state, S4, is when the "Center" button is pressed, at which point the FSM will remain in that state until the lock is reset by pressing the "South" button. The output logic of this FSM is quite simple. If the current state equals anything other than S4, "Locked == 1". This can be described with a simple **assign** statement. Complete the state diagram by adding the missing

labels, including the output designations for states S3 and S4.

Now that you have a completed the state diagram, use it and the example FSM implementation in the Background section to describe the combination-lock FSM in behavioral Verilog. The module interface below should get you started. Notice that we are exposing the state nets in the module interface. This is to aide in debugging!

```
/*This module describes the combination-lock   *
 *FSM discussed in the prelab using behavioral*
 *Verilog                                      */
module combination_lock_fsm(

    /*for ease of debugging, output state      */
    output reg [2:0] state,
    output wire Locked, //asserted when locked
    input wire Right, Left, //indicate direction
    input wire [4:0] Count, //indicate position
    input wire Center, //the unlock button
    input wire Clk, South //clock and reset
);
```

3.2 Pre-lab Deliverables

Include the following items in you pre-lab submission in addition to those items mentioned in the *Policies and Procedures* document for this lab course.

1. The completed state diagram for the combination-lock FSM.
2. The combination-lock FSM Verilog module.

4 Lab Procedure

The experiments below will take you through a typical design process in which you describe your design in Verilog, simulated it in ISE, and synthesize it for the Spartan 3E FPGA.

4.1 Experiment 1

The first experiment in lab will involve simulating the combination-lock FSM module you described in the pre-lab to ensure it works properly before attempting to load it on the FPGA board. Additionally, we will create a Verilog description of the Up/Down Counter discussed in the pre-lab. As with the combination-lock FSM, we will simulate this module to ensure proper operation prior to integrating it into the top-level module.

1. The following steps will guide you through the process of simulating the FSM that you created in the pre-lab.

(a) Create a new ISE project called "lab11" and add your combination-lock FSM module to that project.

(b) Copy the test bench file, "combination_lock_fsm_tb.v", from the course directory into your lab11 directory.

(c) Add the test bench file you just copied over to your ISE project and simulate its operation.

(d) Examine the console output of the test bench and ensure all of the tests passed.

(e) Likewise, take a look at the simulation waveform and take note of the tests that the test bench performs. Is this an exhaustive test? Why or why not?

2. The last component that we must describe in Verilog before we can begin creating our top-level module is the 0-to-19, Up/Down Counter, which will keep track of the position of our rotary knob. Obviously, what is unique about this counter is that its maximum value is not a power-of-two. Although the gate-level design may be significantly more involved than that of a traditional counter, behavioral Verilog does a good job of hiding the complexity. Therefore, behavioral Verilog is what we will use to create this counter! The steps below will take you through the development and simulation of this module.

   (a) Type the code below into a file called "up_down_counter.v".

```
`timescale 1ns /1ps
`default_nettype none

/*This behavioral Verilog description models *
 *an up down counter that counts between 0-19*/
module up_down_counter(
    /*output will be driven in an always block*/
    output reg [4:0] Count,
    input wire Up, Down,
    input wire Clk, South
);

    /*positive edge triggered synchronous logic*
    /*with a synchronous reset                 */
    always@(posedge Clk)
        if(South)
            Count <= 0;
        else if(Up)
          begin
            if(Count == 19)//if at top end
                Count <= 0; //roll over
            else //count up
                Count <= Count + 1;
          end
        else if(Down)
        //now do something similar for
```

```
            //down counting...
28

30

32 endmodule
```

(b) Now fill in the missing pieces and add the source file to your ISE project.

(c) Copy the test bench file, "up_down_counter_tb.v", from the course directory into your lab11 directory.

(d) Add the test bench file you just copied over to your ISE project and simulate its operation.

(e) Examine the console output of the test bench and ensure all of the tests passed.

(f) Likewise, take a look at the simulation waveform and make a note in your lab write-up about how the test bench tests the operation of your Up/Down Counter.

4.2 Experiment 2

For the final experiment, you will integrate the modules you simulated in the previous experiment with the rotary encoder and the LCD driver modules into a top-level module. You will then synthesize and implement the top-level module and load it onto the FPGA.

1. Synthesize and implement the combination-lock top-level module with the following steps:

   (a) Copy over the following files from the course directory into you lab10 directory:
   - "rotary_combination_lock.v", the top-level Verilog module
   - "rotary_combination_lock.ucf", the UCF for the top-level
   - "synchronizer.v", the synchronizer module for our asynchronous inputs
   - "lcd_driver.v", the driver module for the character LCD screen
   - "rotary_encoder_module.v", the quadrature decoding module

   (b) Add the aforementioned files along with your combination-lock FSM module and Up/Down Counter module to your ISE project.

   (c) Set the **rotary_combination_lock** module as the top-level module and kick off the synthesis process. Correct any errors with your design.

   (d) Once your design builds without errors and warnings, load it onto the FPGA board.

2. With the combination-lock design loaded onto the FPGA, test its operation.

   (a) Press the "South" button on the Spartan 3E board. You should see the following on the LCD screen:

   ```
   >00  LOCKED
   ```

If you do not see this pattern of characters on the screen, you will need to begin debugging your design.
Hint: Because we simulated the operation of the individual components, the problem is likely in the implementation phase. For example, you may have forgot to add the UCF file to your project.

(b) With the correct output on the LCD screen, run through the combination sequence shown in the pre-lab with the rotary knob. The LCD screen will display "UNLOCKED" when the correct combination has been entered. Ensure the design operates as expected. If it does not, you may want to connect the logic analyzer up to J1 and walk through the state transitions of your design. **Note:** When your design works properly, demonstrate your progress to the TA.

3. At this point, we have demonstrated our prototype to our customer, and they liked what they saw. However, they mentioned that they would like to use the lock to secure sensitive company information and would require a more secure combination. Thus, we will modify our existing design to accept a four number combination instead of just three.

   (a) Modify the Verilog source to require the input of a fourth number of your choosing in the combination. This will require the addition of one more state in your FSM.

   (b) Re-synthesize and implement your design in ISE.

   (c) Load the design on the FPGA and ensure that it works properly. As with the previous test, you may want to use the logic analyzer to aide in debugging. **Note:** When your modifications work properly, demonstrate your progress to the TA.

5 Post-lab Deliverables

Please include the following items in your post-lab write-up in addition to the deliverables mentioned in the *Policies and Procedures* document.

1. Include the source code with comments for **all** modules you simulated and/or implemented in lab. You do **not** have to include test bench code that was provided! Code without comments will not be accepted!

2. Include screenshots of all waveforms captured during simulation in addition to the test bench console output for each test bench simulation.

3. Answer <u>all</u> questions throughout the lab manual.

4. A possible attack on your combination-lock is a brute-force attack in which every possible input combination is tried. Given the original design with a combination of three numbers between 0 and 19, how many possible input combinations exist? How about for the modified design with a combination of four numbers?

6 Important Student Feedback

The last part of lab requests your feedback. We are continually trying to improve the laboratory exercises to enhance your learning experience, and we are unable to do so without your feedback. Please include the following post-lab deliverables in your lab write-up.
Note: If you have any other comments regarding the lab that you wish to bring to your instructor's attention, please feel free to include them as well.

1. What did you like most about the lab assignment and why? What did you like least about it and why?
2. Were there any section of the lab manual that were unclear? If so, what was unclear? Do you have any suggestions for improving the clarity?
3. What suggestions do you have to improve the overall lab assignment?