

**Lab 5**  
**Finite State Machine in Verilog**

## Objective

You will design, debug, and implement a Finite State Machine (FSM). The FSM for this exercise is the combination lock example presented in class. Using our definition of the problem and our state transition diagram, you will derive a behavioral Verilog description of the state machine, enter your design in the HDL editor, synthesize the circuit, map it to the Xilinx FPGA, simulate the netlist, and finally download the design onto the Xilinx board and test it there.

## Introduction

You are building the controller for a 2-bit serial lock used to control entry to a locked room. The lock has a **RESET** button, an **ENTER** button, and two two-position switches, **CODE1** and **CODE0**, for entering the combination. For example, if the combination is 01-11, someone opening the lock would first set the two switches to 01 (**CODE1** = low, **CODE0** = high) and press **ENTER**. Then s/he would set the two switches to 11 (**CODE1** = high, **CODE0** = high) and press **ENTER**. This would cause the circuit to assert the **OPEN** signal, causing an electromechanical relay to be released and allowing the door to open. Our lock is insecure with only sixteen different combinations; think about how it might be extended.

If the person trying to open the lock makes a mistake entering the switch combination, s/he can restart the process by pressing **RESET**. If s/he enters a wrong sequence, the circuitry would assert the **ERROR** signal, illuminating an error light. S/he must press **RESET** to start the process over.

In this lab, you will enter a design for the lock's controller in a new Xilinx project. Name this lab "lab5". Make **RESET** and **ENTER** inputs. Use a two-bit wide input bus called **CODE[1:0]** for the two switches. (Information on how to use buses will be given later in this handout). The outputs are an **OPEN** signal and an **ERROR** signal.

The table below summarizes the combination lock inputs and outputs:

Input Signal	Description
RESET	Clear any entered numbers
ENTER	Read the switches (enter a number in the combination)
CODE[1:0]	Two binary switches
Output signal	Description
OPEN	Lock opens
ERROR	Incorrect combination

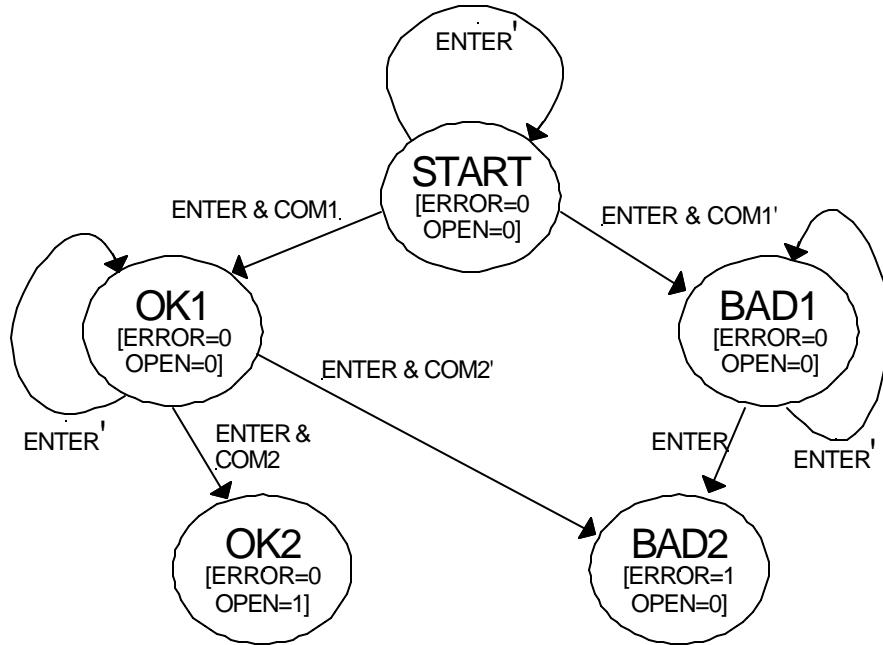


Figure 1: Combination Lock State Transition Diagram

## Detailed Specification

### Combination Comparator.

You will need to design and implement a compare block based on the lock combination. This block processes the CODE[1:0] input signals into a simpler form for the FSM number, namely COM1 (COMpare 1) and COM2 (COMpare 2). Specifically, COM1 is asserted when CODE[1:0] is the combination's first number and COM2 is asserted for the second number. Partitioning the circuit in this way makes the combination easy to change. 2 dipswitches on the board correspond to CODE[1:0]. Choose your own combination and fix them in your Verilog module; the two numbers must be different. This should be a simple block. Use a few AND gates and inverters, but write them in Verilog. Call this module **compare.v**

### Finite State Machine Module.

The FSM module takes **RESET**, **ENTER**, **COM1**, **COM2**, and present state and generates **OPEN** and **ERROR**, as well as the next state. Figure 1 shows the state transition diagram. Note that unlike the diagram presented in the lecture notes, this diagram does not contain any arc corresponding to the **RESET** input. The FPGA flip-flops have a primitive reset option and therefore it is more efficient to let reset be part of the flip-flop specification rather than the next state logic. Also note that every output is assigned for every state. Make certain that your Verilog specification does the same. Call this module **lock.v**

Remember to separate your FSM into 3 ALWAYS blocks as mentioned in the lab lecture.

### Debouncer and Edge detector for the ENTER signal.

For correct operation of the FSM, the input signal **ENTER**, as it comes from the button press must be converted to a pulse that lasts for only one clock cycle. If your clock frequency is 16MHZ and you press the **ENTER** button for 1ms, then your system would interpret the **ENTER** signal has been asserted for thousands of cycles. In our case, since you are not changing the 2-bit input, your design might think you have entered the same combination thousands of times. For your debouncer, if its input has been asserted for 2 clock cycles (this makes sure the input signal is not some random glitch), then it will generate a pulse on the next clock cycle. This means the signal will asserted on the positive edge of the 3<sup>rd</sup> cycle and set

low on the positive edge of the 4<sup>th</sup> cycle. The output of your debouncer will be used to drive the rest of your lock as the **ENTER** signal.

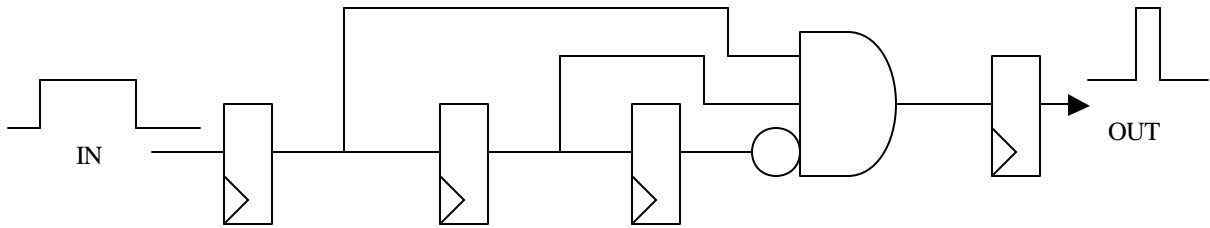


Fig.1: Debouncing and edge detection circuit

Draw the signal observed at the output given an input pulse of arbitrary length longer than 3 clock periods.

In the same way, the reset signal has to go through debouncing and edge detection circuits. However, the flip-flops used must not be reset by the reset signal. Can you explain why?

Include your code for both debouncer circuits in **locktop.v**.

## Tasks

Implement your design for the FSM in a Verilog behavioral model. Please use the sample Verilog FSM codes linked on the EECS150 website as references. You are to design two FSMs using different state assignment formats. For the first design, you must encode your states in 3 bits, but you can make up your own binary encoding. For the second design, you must use one-hot-encoding for your state assignment. Make sure you instantiate each module in the **locktop.v** file. Create a testbench and test the functionality of your design using the **locktop.v** file.

When your design is functionally correct, use the **FPGA.v** (this is the IO module given to you) file and check that the instantiation of the module **lock\_top** in this file is correct. You should take a look at **locktop.v** to see its input and output ports are correctly defined. Synthesize and implement the design using the **FPGA.v** file for implementation on the XILINX XCV2000 FG680 device.

Here is the proper synthesis/implementation procedure:

1. Go to synthesis software (Synplicity).
2. Create a new project and add the .v files of your design (don't add the testbench)
3. Choose the correct device and run the synthesis
4. After synthesis is over, open XILINX design manager and implement the design.

After synthesis and implementation, you are now ready to generate the binary (\*.bit) file that is used to program the FPGA.

Here is the procedure for generation the configuration file and programming the FPGA

1. After implementation in Design Manager, the configuration file will be generated automatically. By default the configuration method will be using serial port.
2. Power up your FPGA using a 5 V supply. Make sure the power and ground are connected correctly and the voltage is set correctly
3. Connect the serial port connector from your PC to the XILINX board.
4. From Design Manager, go to Tools -> iMPACT

5. Ignore the message and click OK. Click on the tab Slave Serial, right click to add the fpga\_top.bit file. The FPGA will appear in your workspace.
6. Right click on the FPGA and program it.
7. When the configuration is ended and successful, you can do a reset, then start testing your design.
8. You can use the LEDs to check your state, the DISPLAY to check your outputs and you can enter your inputs (enter and reset) using the push buttons and your code using the DIP switches.

Find the following on the board:

DIP SWITCH : SW9, CODE (AB38, AB39)  
LED : STATE, (U36 U35 V39 V38 V37 V36 V35 W38)  
PUSH BUTTON SWITCHES: RESET (SW1), ENTER (SW2)

2 of the DIP switches are mapped to your input combination  
3 of the LEDs correspond to your present state encoded using 3 bits.  
The 2 push buttons correspond to your reset and enter inputs.

Once you downloaded your design onto the Xilinx board, the number LEDs will display which state you are in to help you to debug. Since the number LEDs only take in 3bit inputs, you must encode your 5bit state in your one-hot-encoding design to 3 bits to take advantage of this debug tool. For example, if you have state 00100, you should encode this state to 011.

The 7-segment displays will display the letters **open** or **error** or will not display anything.

Compare the two designs, and tell your TAs the CLB usage and max clock frequency of each implementation.

## More Details

### Buses.

Buses are collections of ordered wires that (for one reason or another) were collected in a group for easy reference. Examples of busses include the two input bits of our combination lock (aka. IN[1:0]), the state/nextstate of our combination lock (S[2:0], NS[2:0]), or the memory busses for address and data in your personal computer.

Quite often, a bus's wires have similar purposes; the memory address bus in your PC is used to dictate which address in the memory the CPU would like to access. To do so, it needs to send a 32-bit integer to the memory. The easiest way to do so is to connect 32 wires from the CPU to the memory. Each wire corresponds to one bit of data.

Xilinx uses a thicker wire to denote a bus, and it uses a standard naming convention. The convention is:

**NAME\_OF\_BUS [ number1 : number2 ]**

The number of bits in the bus is determined by the numbering. A bus called S[2:0] will have 3 wires; a bus called DATA[31:16] will have 16 wires. From our memory address bus example above, if the data bus were called ADDR[31:0], the wires are numbered from 31 to 0, with the 31<sup>st</sup> wire being the highest-order bit, and the 0<sup>th</sup> wire being the lowest-order bit. Order of the number matters: if the data bus were called ADDR[0:31], the 31<sup>st</sup> wire being the lowest-order bit, and the 0<sup>th</sup> wire being the highest-order bit.

## Verilog Help.

- Evoke Program -> XILINX Foundation -> Project Manager . click cancel to create project window  
Click on Tools-> Design Entry ->HDL Editor Click OK to use HDL DesignWizard and choose Verilog and enter port signals

Click on Tools -> Language Assistant

You will see the templates that XILINX provides to help in the writing of the Verilog code.

## Prelab

Write all necessary Verilog code before coming to lab. This code includes the combination-comparator, the FSM module, the “debounce” circuits, and your top level module.

Think about how to test your design. Here is a hint for testing FSM – make sure you follow each and every arc in the state transition diagram.

Name: \_\_\_\_\_ Name: \_\_\_\_\_

Lab Section (Check one)

M:  AM  PM    T:  AM  PM    W:  AM  PM    Th:  PM

## Checkoffs: Lab 5

1. Verbal description of the FSM and the test procedure, answers to the debouncing circuits, and inspection of the test bench.

TA:\_\_\_\_\_ (20%):\_\_\_\_\_ (10%)

2. Working simulation of lock with binary encoded states.

TA:\_\_\_\_\_ (20%):\_\_\_\_\_ (10%)

3. Working simulation of lock with one-hot encoded states.

TA:\_\_\_\_\_ (20%):\_\_\_\_\_ (10%)

4. Binary encoded lock working on board.

TA:\_\_\_\_\_ (20%):\_\_\_\_\_ (10%)

5. One-hot encoded lock working on board.

TA:\_\_\_\_\_ (20%):\_\_\_\_\_ (10%)

**Total Score:**\_\_\_\_\_