University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Science

EECS 150
Spring 2002

Original Lab By: J.Wawrzynek and Mark Feng

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

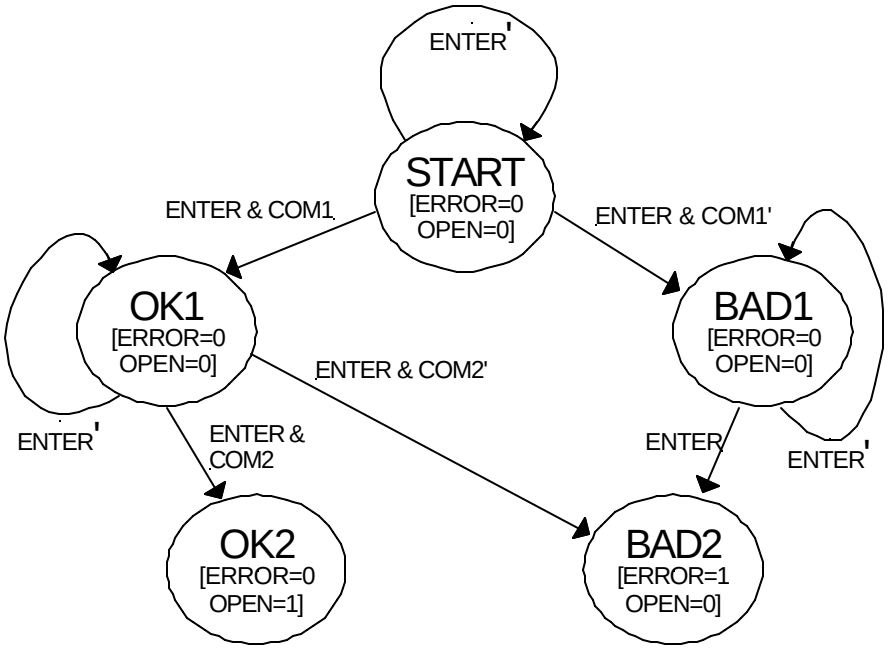

Figure 1: Combination Lock State Transition Diagram

Detailed Specification

Combination Comparitor.
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. Dipswitch[1] and Dipswitch [2] correspond to CODE[1:0]. Choose your own combination; the two numbers must be different. This should be a simple block. Use a few AND gates and inverters, but write them in Verilog.

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.

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 3rd cycle and set low on the positive edge of the 4th cycle. The output of your debouncer will be used to drive the rest of

your lock as the **ENTER** signal. You can build this in via FFCEs and logic gates, but **make sure to ground your reset signal for the FFCE blocks**. Finally, include your code for the debouncer 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 top.v file and synthesize locktop.v (locktop.v is the IO module given to you). Please take a look at locktop.v to see what are the inputs and output ports for top.v. During synthesis, use lab5contraint.exc as your constraint file.

Here is the proper synthesis procedure:

1. Go to synthesis.
2. Check the Edit Synthesis/Implementation constraints box.
3. Click on the ports tab.
4. Import the lab5constraint.exc file. If you have trouble doing this, just manually type in the corresponding pads for each pin (refer to Pin Assignment).
5. Run synthesis and implementation.

Finally, you will interface your design to the Xilinx board via the locktop.v file given to you. 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 take advantage of this debug tool. For example, if you have state 00100, you should encode this state to 011. The light LEDs will display **ERROR** or **OPENLOCK**. The right most LED correspond to **OPENLOCK** and the 2nd most right LED correspond to **ERROR**. The other light LEDs is always on.

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 31st wire being the highest-order bit, and the 0th wire being the lowest-order bit. Order of the number matters: if the data bus were called ADDR[0:31], the 31st wire being the lowest-order bit, and the 0th wire being the highest-order bit.

Bussing related signals makes the circuit easier to read and simulate. When using the command window or writing a command file using the script editor, as you did in lab1, writing:

```
vector data DATA[7:0]
```

(or: `v data DATA[7:0]`) makes the signals **DATA7**, **DATA6**, … **DATA0** into a vector called **data**, which can be treated like any other signal: you can watch vectors and set their values. Use the **`assign`**

command to set a vector's value. E.g.: **`a data 3e\h`** (hexadecimal) or **`a data 001111110\b`** (binary).

Forcing Internal Signals.

In addition to inputs, the logic simulator allows you to force internal signals, those normally driven by components, to particular values. This trick lets you set the state to anything you want. Simply set **NEXTSTATE[2:0]** to the state you want, clock the FSM, and then release **NEXTSTATE[2:0]**.

Clocks.

You can define a clock (an input signal that changes periodically) in your script file or in the command window. For example,

```
clock clk 0 1
```

makes the clock signal **clk** oscillate as the circuit is simulated. To simulate for a single clock period, use **cycle** instead of **sim.** Do not mix **cycle** and **sim**, as it can lead to some very interesting software behavior.

Command and Log Files.

File → Run Script File… loads a script file and runs each line of it as if you were typing each of those lines in the command window. The most convenient way to create a script file is to use the **Tools → Script Editor**. If you want to use another editor you can, but if you do, make sure to save the file as plain text and that the file extension is **.cmd**.

To save a log of the commands you use in your session, you can create a transcript of your work using the command **log**. Start a log with log **filename.log**, and end a log by typing **log** alone.

Naming.

- The Xilinx software, DOS, and Windows is case-*in*sensitive, somewhat, although we've used all caps for signals throughout this handout.
- You may use letters, numbers, and underscores (_) in filenames. The period (.) may only appear in certain places (e.g., **yourfile.cmd**, etc). Avoid other punctuation.

Prelab

Write all necessary Verilog code before coming to lab. This code includes the combination-comparitor, the FSM module, the "debounce" circuit, and your top level module.

Think about how to test your design. Here is a hint for testing FSM – make sure you follow long 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 test procedure and inspection of test command script.

   TA:_____(20%)

2. Working simulation of lock with binary encoded states.

   TA:_____(20%)

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

   TA:_____(20%)

4. Binary encoded lock working on board.

   TA:_____(20%)

5. One-hot encoded lock working on board.

   TA:_____(20%)

50% off for lab turned in one week late TA:_____(-50%)

Total Score: _____

TA:_____