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

EECS 150
Spring 2001

Original Lab By: J.Wawrzynek and N. Weaver
Later revisions by R. Fearing, and J. Shih
Xilinx Foundation 3.0 version: Laura Todd

Lab 2
Finite State Machine

1 Objectives

You will enter and debug a Finite State Machine (FSM). Using our definition of the problem and logic equations specifying the FSM's operation, you will enter your design in the schematic editor and simulate it with the logic simulator.

2 Prelab

- **Get a partner from your lab section!**
- Complete your **IN1** (INput 1) and **IN2** (INput 2) blocks
- Write a **.cmd** (command) file to test your CLB (Combinational Logic Block).
- Write one single **.cmd** file with all the FSM test scenarios specified in the check-off sheet.
- **Do as much as possible before your scheduled lab time.** There is much to do in this lab. Some can be done on paper; the remainder can be done on the computers outside of lab time.

3 High-level Specification

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 "lab2". Make **RESET** and **ENTER** inputs. Simulate by pressing the **ENTER** button by forcing it high for a clock cycle. 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.

Figure 1 shows a decomposition of the combination lock controller, whose inputs and outputs are:

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

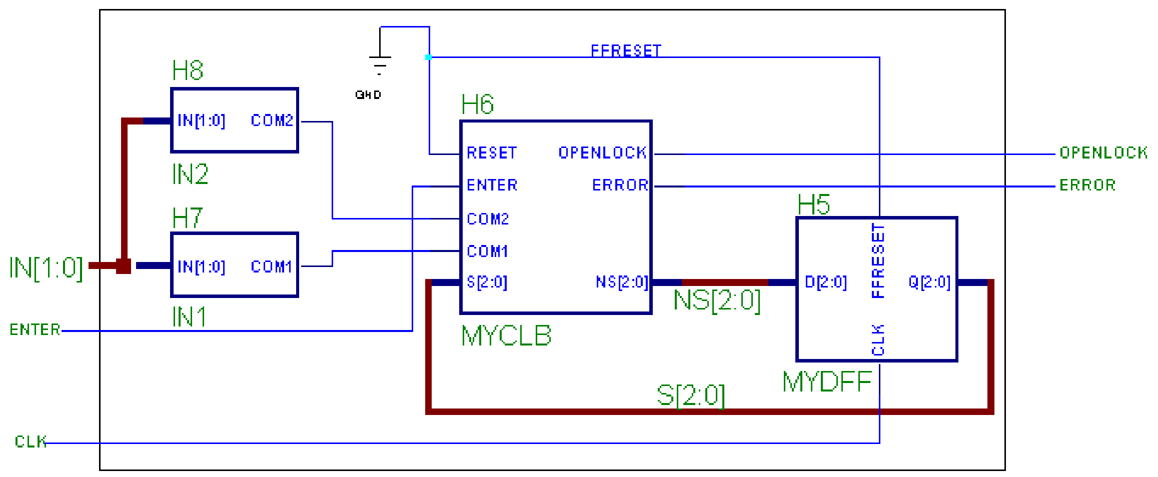

Figure 1: Controller for the combination lock

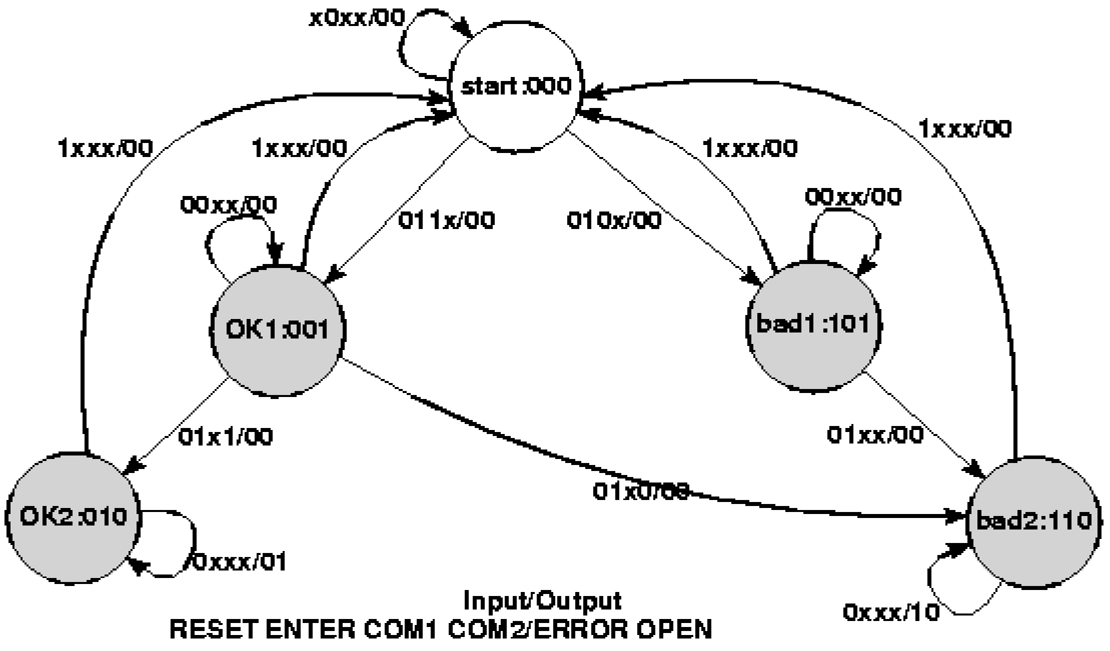

Figure 2: State Transition Diagram

4 Low-level specification

4.1 IN1 (INput 1) and IN2 (INput 2)

Blocks **IN1** and **IN2** process the input signals **COM1** (COMpare 1) and **COM2** (COMpare 2) into a simpler form for the FSM. Specifically, **COM1** is asserted when **CODE[1:0]** is the combination's first number. Similarly, **COM2** is asserted for the second number. Partitioning the circuit in this way makes the combination easy to change.

Choose your own combination; the two numbers must be different.

This should be a simple block. Use a few AND gates and inverters.

4.2 MYCLB

The MYCLB (MY Combinational Logic Block) block takes **RESET**, **ENTER**, **COM1**, **COM2**, and present state and generates **OPENLOCK** and **ERROR**, as well as the next state. Figure 2 shows the state transition diagram, a Mealy machine since the transitions are labeled with both inputs and outputs. The white circle denotes the rest state (i.e., the state the machine starts in).

RESET	ENTER	COM1	COM2	S[2:0]	NS[2:0]	ERROR	OPEN
1	X	X	X	XXX	000	0	0
0	0	X	X	000	000	0	0
0	1	0	X	000	101	0	0
0	1	1	X	000	001	0	0
0	0	X	X	001	001	0	0
0	1	X	0	001	110	0	0
0	1	X	1	001	010	0	0
0	X	X	X	010	010	0	1
0	0	X	X	101	101	0	0
0	1	X	X	101	110	0	0
0	X	X	X	110	110	1	0

Figure 3: Truth Table for the FSM

```
ERROR = RESET'                S2  S1  S0'
OPEN  = RESET'                S2' S1  S0'
NS0   = RESET' ENTER  COM1' S2' S1' S0' +
        RESET' ENTER  COM1  S2' S1' S0' +
        RESET' ENTER'       S2' S1' S0  +
        RESET' ENTER'       S2  S1' S0
NS1   = RESET' ENTER  COM2' S2' S1' S0  +
        RESET' ENTER  COM2  S2' S1' S0  +
        RESET'              S2' S1  S0' +
        RESET' ENTER        S2  S1' S0  +
        RESET'              S2  S1  S0'
NS2   = RESET' ENTER' COM1' S2' S1' S0' +
        RESET' ENTER  COM2' S2' S1' S0  +
        RESET' ENTER'       S2  S1' S0  +
        RESET' ENTER        S2  S1' S0  +
        RESET'              S2  S1  S0'
```

```
TEMP0 = RESET' S1 S0'
TEMP1 = RESET' S1'
TEMP2 = ENTER S2' S0'
TEMP3 = ENTER' S0 + TEMP2
TEMP4 = COM2' ENTER + S2
TEMP5 = ENTER S0'
TEMP6 = COM1' TEMP5 + TEMP4 S0

ERROR = S2 TEMP0
OPEN  = S2' TEMP0
NS0   = TEMP1 TEMP3
NS1   = TEMP0 + ENTER TEMP1 S0
NS2   = ERROR + TEMP1 TEMP6
```

(a) (b)

Figure 4: Logic Equations for the FSM (Unoptimized): (a) two-level (b) multi-level

By writing an "AND" term describing each row where an output is 1 and ORing these together, it's easy to obtain a set of equations implementing the table shown in Figure 3.

Translating a state transition diagram to a truth table is mechanical: Write a row for each transition. In Figure 4a, I've collapsed a few rows together by noting that when **RESET** is 1, the outputs and next state are the same for all states. Note **RESET'** means **NOT RESET**.

Implementing the equations of Figure 4a as written would be inefficient, so we introduce intermediate variables **TEMP0**, ..., **TEMP6** as shown in Figure 4b. Make symbol and schematic for **MYCLB** using the equations in Figure 4b.

4.3 MYDFF

(MY D Flip-Flops)

Create a block called **MYDFF** that contains three D flip-flops (one for each state bit) with an asynchronous clear hooked to a pin called **FFRESET** (Flip-Flop **RESET**). The flip-flops you should use are the FDR (D Flip-flop with synchronous Reset) flip-flops. Externally, connect **FFRESET** to 0 (i.e., not resetting), but during simulation you can force this to 1 to reset every flip-flop.

5 Buses

5.1 Definition and Usefulness

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.

In this class, buses will be used to conveniently associate a single name to an ordered collection of wires. They will make wiring up your Xilinx diagram much easier and your final schematics less messy!

5.2 Using Buses in the Schematic Editor

Buses are drawn by clicking on the "Draw Buses" icon and drawing as if you were drawing a wire.

Buses with the same name in the same schematic are connected implicitly. Single wires with the same name as a wire in a bus are also connected implicitly.

In Xilinx, you need to label buses with the appropriate number of bits. This bit range determines how many wires are in the bus, and what those wires are named. It doesn't matter whether you set the range from, say, for an eight-bit bus, 7 to 0, or from 0 to 7, but it is necessary to be consistent in your all of your labeling. In labs where you're using a TA schematic, it will usually be necessary to use 7 to 0.

As with wires, to end a bus you click on the right mouse button and select a method to end the bus. You have a choice of "Add Bus Terminal", "Add Bus Label", or "Add Bus End":

- "Add Bus Terminal" is how you want to end buses that will be connected to a symbol's pins (inputs and outputs) when drawing a symbol's schematic (reviewing how input and output wires were drawn in symbols in lab 1 may help make this easier to understand). Remember to specify whether the terminal is an input or output terminal. Also you will be required to input a name and how many wires the bus is made up of.
- "Add Bus Label" is how to end all other buses. As you did when adding a terminal to a bus, you will need to specify a name and range when adding a bus label.
- "Add Bus End" ends the bus without naming it. To name the bus later, edit an existing label, or change a bus's attributes, double click on the bus while in the "**Select**" mode to bring up the bus's attribute dialog.

For this lab we will not use complex buses, but they may be useful to you during later labs or the project. Complex buses allow you to combine multiple buses and single wires into one bus. Read the Xilinx help documents for more information.

To "break out" a wire from a bus, draw a wire and combine the bus name and the number of the wire to form the new wire's name. For example, if you had a bus called **DATA** that was eight bits wide, to access the wire that has the lowest bit of the bus, you would draw a wire and name it **DATA0**. It is also possible to “break out” sub buses, e.g, **DATA[5:2]**. Buses between pins with the same (bus) width do not need to be named, but any bus which you want to break a wire out of must be named.

The “Draw Bus Taps” function can also be used to break out wires, but it tends to leave “messy” schematics, and thus is not recommended.

5.3 Using Buses in the Simulator

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).

6 Forcing Internal Signals

In addition to inputs, the logic simulator allows you to force internal signals, those normally driven by components, to particular values. For example, you’ve tied FFRESET to 0, but it can be set high simply by typing:

```
h  FFRESET
```

which will reset the state bits to zero after you simulate for a step. Use

```
r FFRESET
```

to release the signal – return it to its default.

A similar 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]**.

7 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.

8 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.

9 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.
- You may use letters, numbers, and underscores in labels, but avoid other punctuation.
- **Do not use names already used by components in other libraries.** For example, do not call your combinational logic block **CLB** since there is already a component with that name in the XC4010XL library.

Name: ________________________________ Name: ______________________________________

Lab Section (Check one)

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

10 Checkoffs: Lab 2

10.1 Design the **IN1**, **IN2**, and **MYCLB** blocks. Enter it using the schematic editor using only components from the XC4000XL library (This is very important - In the next lab you will be putting this on a Xilinx chip and only components from the XC4000XL library will work). You'll probably use the **AND2**, **AND3**, **OR2**, **OR3**, **INV**, **FDC**, and **GND** components.

TA: ______________(20%)

10.2 Make a test script for your CLB and run it. Show the TA your script and that the output is correct.

TA: ______________(20%)

10.3 Design your state register using D-flip-flops. Call it MYDFF.

TA: ______________(10%)

10.4 Wire up the FSM using the CLB and state register you designed. Neatly for full credit.

TA: ______________(20%)

10.5 Write a command script to simulate the following scenarios:

(a) A successful entry of the combination.
(b) A successful entry of the combination with cycles of pauses between when **ENTER** is asserted.
(c) A sequence with the first combination number entered wrong.
(d) A sequence with the second combination number entered wrong.
(e) A sequence with the both combination numbers entered wrong.
(f) RESET is asserted after entering just the first number correctly.
(g) RESET is asserted after entering just the first number incorrectly.

Create a log file to show your TA.

TA: ______________(30%)

10.6 Turned in on time

TA: ______________
(full credit (100%))

10.7 Turned in one week late

TA: ______________
(half credit (50% x points))