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

CS 150 J. Wawrzynek
Fall 1996 N. Weaver

Lab 2
Finite State Machine in Workview

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 ViewDraw and simulate it with SpeedWave and ViewTrace.

2 Prelab

- **Get a partner for your section!**
- Complete your `IN1` and `IN2` blocks.
- Write a .cmd file to test your CLB.
- Write one single .cmd file with all the test scenarios in there as specified in the checkoff sheet.
- **Do as much other things 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 she would set the two switches to `11` (`CODE1` = high, `CODE0` = high) and press `ENTER`. This would cause the circuitry 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 project. Make the project directory in your home directory named lab2. Make `RESET` and `ENTER` inputs. Simulate 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. (Refer to a later section for information on busses in Workview.) 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 entered

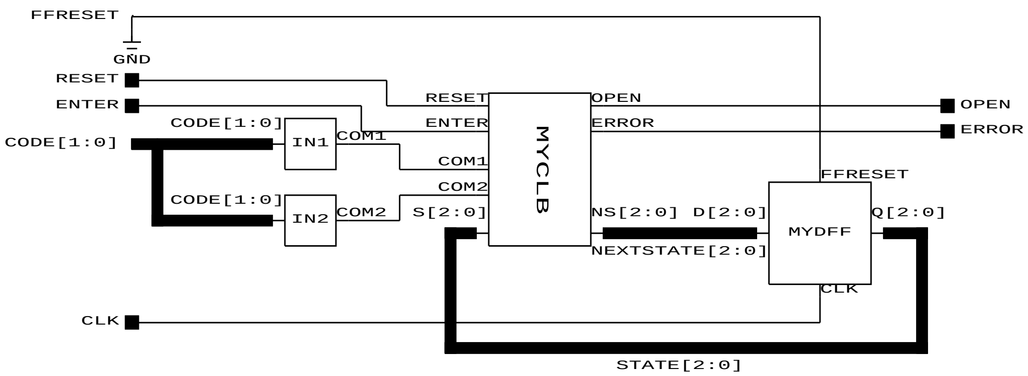

Figure 1: Controller for the Combination Lock

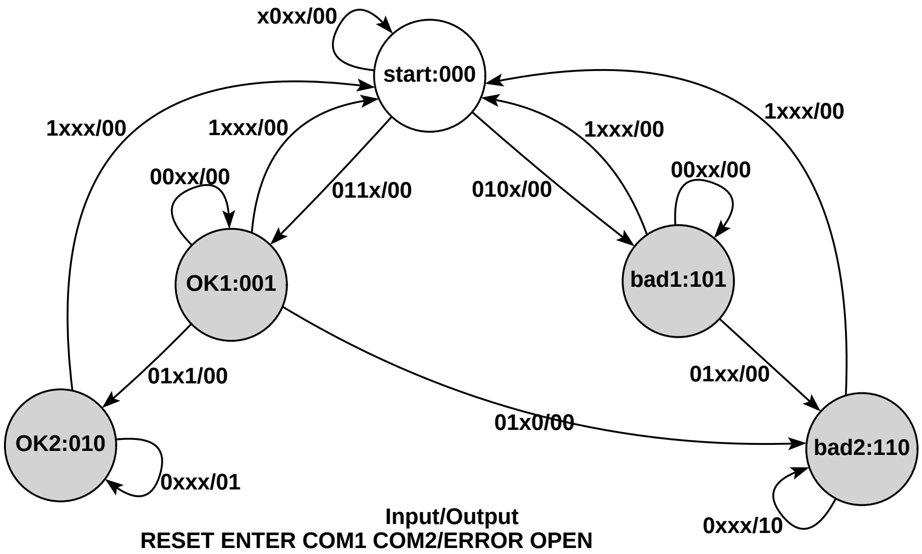

Figure 2: State Transition Diagram

4 Low-level Specification

4.1 IN1 and IN2

Blocks IN1 and IN2 process the input signals COM1 and COM2 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 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 block takes RESET, ENTER, COM1, COM2, and present state and generates OPEN 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 reset state (i.e., the state the machine starts in).

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 this, as shown in Figure 3.

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

$$
\begin{aligned}
\mathtt{ERROR} &= \mathtt{RESET'\ S2\ S1\ S0'}\\
\mathtt{OPEN} &= \mathtt{RESET'\ S2'\ S1\ S0'}\\
\mathtt{NS0} &= \mathtt{RESET'\ ENTER\ COM1'\ S2'\ S1'\ S0'} +\\
&\ \mathtt{RESET'\ ENTER\ COM1\ S2'\ S1'\ S0'} +\\
&\ \mathtt{RESET'\ ENTER'\ S2'\ S1'\ S0} +\\
&\ \mathtt{RESET'\ ENTER'\ S2\ S1'\ S0}\\
\mathtt{NS1} &= \mathtt{RESET'\ ENTER\ COM2'\ S2'\ S1'\ S0} +\\
&\ \mathtt{RESET'\ ENTER\ COM2\ S2'\ S1'\ S0} +\\
&\ \mathtt{RESET'\ S2'\ S1\ S0'} +\\
&\ \mathtt{RESET'\ ENTER\ S2\ S1'\ S0} +\\
&\ \mathtt{RESET'\ S2\ S1\ S0'}\\
\mathtt{NS2} &= \mathtt{RESET'\ ENTER\ COM1'\ S2'\ S1'\ S0'} +\\
&\ \mathtt{RESET'\ ENTER\ COM2'\ S2'\ S1'\ S0} +\\
&\ \mathtt{RESET'\ ENTER'\ S2\ S1'\ S0} +\\
&\ \mathtt{RESET'\ ENTER\ S2\ S1'\ S0} +\\
&\ \mathtt{RESET'\ S2\ S1\ S0'}
\end{aligned}
$$

(a)

$$
\begin{aligned}
\mathtt{T0} &= \mathtt{RESET'\ S1\ S0'}\\
\mathtt{T1} &= \mathtt{RESET'\ S1'}\\
\mathtt{T2} &= \mathtt{ENTER\ S2'\ S0'}\\
\mathtt{T3} &= \mathtt{ENTER'\ S0} + \mathtt{T2}\\
\mathtt{T4} &= \mathtt{COM2'\ ENTER} + \mathtt{S2}\\
\mathtt{T5} &= \mathtt{COM1'\ T6} + \mathtt{T4\ S0}\\
\mathtt{T6} &= \mathtt{ENTER\ S0'}\\
\mathtt{ERROR} &= \mathtt{S2\ T0}\\
\mathtt{OPEN} &= \mathtt{S2'\ T0}\\
\mathtt{NS0} &= \mathtt{T1\ T3}\\
\mathtt{NS1} &= \mathtt{T0} + \mathtt{ENTER\ T1\ S0}\\
\mathtt{NS2} &= \mathtt{ERROR} + \mathtt{T1\ T5}
\end{aligned}
$$

(b)

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

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.

Implementing the equations of Figure 4a as written would be inefficient, so we introduce intermediate variables `T0`,...,`T5` as shown in Figure 4b.

4.3 MYDFF

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`**. The flip flops you should use are the **`FDC`** 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 Busses

ViewDraw supports busses, collections of wires drawn as one, through the bus button. An eight-bit data bus, for example, might be labeled `DATA[7:0]`. A net connected to a bus "breaks out" one of the signals—label the net with the name of the broken-out signal, e.g., `DATA5`.[1] It's also possible to break out sub-busses, e.g., `DATA[5:2]`.

All busses must be labeled so ViewDraw knows their width (how many wires), even those passing directly from one component to another where there is (seemingly) no ambiguity about the width.

Pins on symbols can accept busses: label the pins with a bus name (i.e., using the bracket notation).

Bussing related signals makes the circuit easier to read and simulate. In SpeedWave, `vector data DATA[7:0]` (or `v data`...) makes the signals `DATA7`, `DATA6`,...,`DATA0` into a vector called `data`, which can be treated like any other signal: you can watch vectors, you can use them with the `wave` command. Use the assign command to set a vector's value. E.g., `a data 3e\h` (hexadecimal) or `a data 00111110\b` (binary).

Use a bus for the inputs and outputs of the state register and the state inputs and outputs of the combinational logic block.

6 Forcing Internal Signals

In addition to inputs, SpeedWave 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

SpeedWave allows you to define clocks, input signals that change periodically. For example, `clock clk 0 1` makes the signal `clk` oscillate as the circuit is simulated. To simulate for a single clock period, use `cycle` instead of `sim`. Watched variables are not displayed with `cycle`, but those displayed with the `wave` command in ViewTrace are updated.

8 Command and Log Files

In SpeedWave, when the name of a command file is invoked, the contents of the file are interpreted as if you typed them one by one. You may create the file by going to a MS-DOS window, the equivalent of `xterm`. Double-click on the MS-DOS icon (AKA the Command Prompt) in the Program Manager. You might get a window, or the screen may turn to text with an MS-DOS prompt. To switch between the two modes, press `Alt-Enter`.

Change to your project directory with the `cd` command, and edit your command file with `edit yourfile.cmd`. In the editor, `Alt-F` brings up the file menu, from which you can open files, save files, and exit. `Ctrl-Esc` will always bring you back to the Windows desktop, even if you are in the middle of editing your file.

If you are not familiar with MS-DOS, you may want to use the windows application Notepad which is a simple text editor. The commands found in Notepad is much like the one in edit with the key exception that you can use your mouse. Ask the TA if you have any problems with Notepad.

In SpeedWave, 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.

[1] Actually, the net does not need to be connected—any net labeled `DATA5` refers to the sixth wire in the `DATA[7:0]` bus.

9 Naming

- Workview, 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`, `MYCLB.1`, etc). Avoid other punctuation. Since you will use some DOS programs later, make your file names be no more then 8 characters + the 3 letter extention. Follow this convention for naming all your components as well.
- You may use letters, numbers, and underscores in labels, but avoid other punctuation except for bus notation: `[:]`.
- **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 (xc4000) library.

Name: ______________________ Name: ______________________
Lab Section (Check one)
M: ☐11 ☐12 T: ☐13 W: ☐14 ☐15 Th: ☐16

10 Checkoffs

1. Design the `IN1`, `IN2`, and `MYCLB` blocks. Enter it in ViewDraw **using only components from the** (xc4000) **library**[2]. You'll probably use the `AND2`, `AND3`, `OR2`, `OR3`, `INV`, `FDC`, and `GND` components.

   TA: ____________(20%)

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

   TA: ____________(20%)

3. Design your state register using D-Flip Flops. Call it MYDFF.

   TA: ____________(5%)

4. Wire up the FSM using the CLB and state register you designed.

   TA: ____________(20%)

5. Write a SpeedWave command file 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 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: ____________(25%)

Turned in on time

TA: ____________(10%)

[2] This is very important—in the next lab, you will be putting this on a Xilinx chip, and only components from the xc4000 library will work.