FINITE STATE MACHINES INTRO

Lab Report for ECE3270
Digital Systems Design

Submitted by

Ryan Barker

Department of Electrical & Computer Engineering
Clemson University

02/20/2015

Abstract

The goal of this experiment was to take three different approaches at designing a Moore finite state machine (FSM) that recognized four ones or four zeroes at the input. After either sequence, the machine kept the output high for any overlapping sequences. The machine was designed with a synchronous reset. Design approaches one (one-hot encoding) and two (almost one-hot encoding) both involved designing the state machine manually by drawing a state diagram, converting it to a state table, and obtaining Boolean equations for each of the next-state and output variables. Approach three used a user-defined type and two process statements containing CASE-WHEN statements to specify the behavior of the FSM, which allowed Quartus to optimize it in compile time. To illustrate this, after the FSM in part three worked, compile-time parameters were changed to vary its encoding. The overall project allowed the designer to become much more familiar with FSMs.

Introduction

As mentioned in the Abstract, the goal of lab two was to design a Moore finite state machine (FSM) that matched either four zero or four one bits, allowed overlapping sequences, and had a synchronous reset. The overall project in this lab was separated into two parts, each involving a different design method for the FSM. Part one was concerned with designing the machine manually by making a state diagram for the overall FSM, converting that diagram to a state table specific to the encoding, generating Boolean equations for the next state and output variables, and implementing those equations in VHDL code. Note that part one contained two subsections that generated two state machines: One for a machine using one-hot encoding, and one for a machine using almost-one hot encoding. Part two used the same state diagram from part one, but utilized VHDL CASE WHEN statements in the machine architecture to describe the behavior of the machine. This created a much more versatile FSM, since Quartus could optimize the machine for whatever encoding the user specified during the "Analysis and Synthesis" stage of compile time.

Section 1.A: Designing the FSM State Diagram

Perhaps the most important part of the whole project was creating the state diagram, as it was the integral starting component to all three of the designed machines. Since the FSM had to match either "0000" or "1111" and these strings contained opposite bits, two paths were drawn off of the initial state for each string. Each state down each path had meaning (as indicated in quotes on the diagram) and the output flipped to 1 in the first states to match four of a particular bit. As overlapping sequences were allowed, each output state (states E and I) looped back to themselves as the bit they matched kept occurring. If a one was received at any point while matching zeroes or a zero was received at any point while matching ones, the diagram was designed so the FSM's state would enter the top state of the other path. Figure 1.1 shows the state diagram for the lab 2 FSM and a generalized state table for that machine.

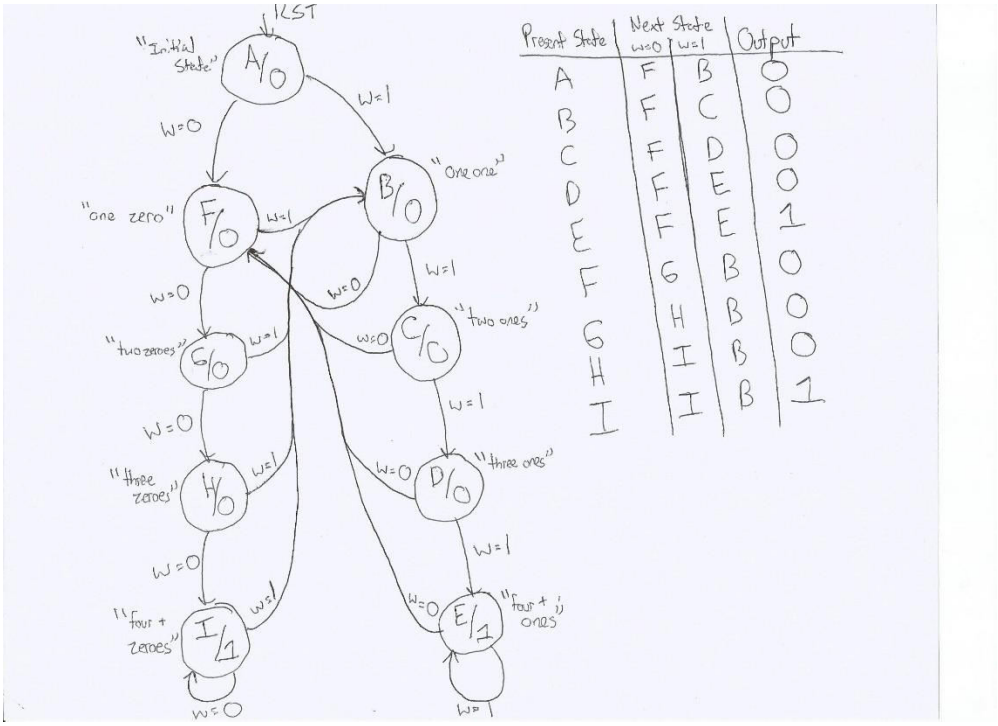

Figure 1.1. Overall FSM Diagram and General State Table.

Section 1.B: General Simulation Strategies

The lab was very explicit in the expected behavior of the FSM for all three FSMs and provided the timing diagram in Figure 1.2. Therefore, to aptly test each machine, a test bench was created that reset the machine and then duplicated the input in this diagram as closely as possible, and the output was checked against the given diagram. The state outputs for each machine were also checked at each stage of testing to ensure the transitions matched the expected pattern of A → F → G → H → B → F → G → H → I → B → C → D → E → E → F → G → H → I.

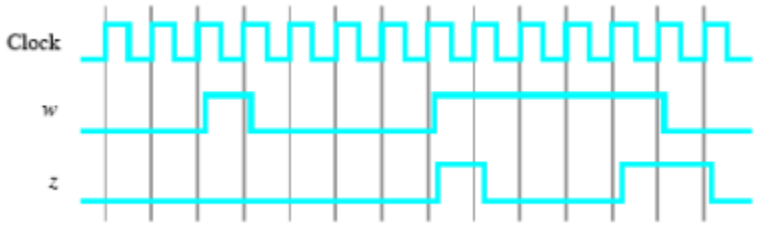

Figure 1.2. Expected FSM Behavior [1].

Section 2.A: Manually Designing the FSM with One-Hot Encoding

The first design of the FSM was a brute force approach utilizing one-hot encoding. This meant each of the nine states in Figure 1.1 was represented with nine bits. Further, each state was a unique bit set to one with the others at zeroes, starting with "000000001" up to "100000000". Figure 2.1 shows the state table for the first design.

Present										Next (w = 0)									Next (w = 1)								Output
Name	y8	y7	y6	y5	y4	y3	y2	y1	y0	Y8	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	Y8	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	(z)
A	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
B	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
C	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
D	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
E	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1
F	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
G	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
H	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
I	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1

Figure 2.1. Design 1 State Table.

Normally, this point in the design requires the engineer to draw a Karnaugh map and derive the equations for Y0-Y8 and z, but because one hot encoding represents each state uniquely as a Boolean minterm, it is possible to sight the necessary equations from Figure 2.1 by noting which state bits made each output a one and, for the next state signals, what the state of the input was. Figure 2.2 shows the equations for each output variable.

Logic
Y0 = 0
Y1 = w(y0 + y5 + y6 + y7 +y8)
Y2 = wy1
Y3 = wy2
Y4 = w(y3 + y4)
Y5 = w'(y0 + y1 + y2 + y3 + y4)
Y6 = w'y5
Y7 = w'y6
Y8 = w'(y7 + y8)
z = y4 + y8

Figure 2.2. Design 1 Output Equations.

Once all of the above prep work was done, writing the VHDL code for design 1 was simple. The machine used SW0 for the reset, SW1 for the input, LEDG0 for the output, and

LEDR's eight to one to show the current state of the machine during testing. Signals for all of these were declared in the entity of the code, but the state variable itself, y, was declared in the architecture. The main trick to the design was placing all code in the next state process inside of a VHDL IF statement with no ELSE to imply memory on the state variable. Inside of this IF statement, the reset signal was checked: If it was high, the machine was sent back to state A. Otherwise, the next state was calculated based on the first nine equations in Figure 2.2. Lastly, outside of the next-state process, the output was calculated and y was mirrored to an output for the LEDRs. All VHDL code for the first FSM design is seen in appendix A.1.1.

Section 2.B: Testing the First FSM

The general strategies in Section 1.B were followed to produce a test bench that followed the behavior in Figure 1.2. When run in ModelSim, this test bench produced the behavior in Figure 2.3, which almost exactly matches Figure 1.2: After a sequence of three zeroes, a one, and four zeroes, the output goes high. Then, on the next one, the output goes low, but after four successive ones, it raises again and stays high for a fifth one. The states follow the expected behavior described in Section 1.B. At this stage, the bitmap file for the design was loaded on the board, and the same testing was repeated. Apart from some switch debouncing issues with KEY0, the results were identical. The test bench used in simulation is shown in appendix A.1.2.

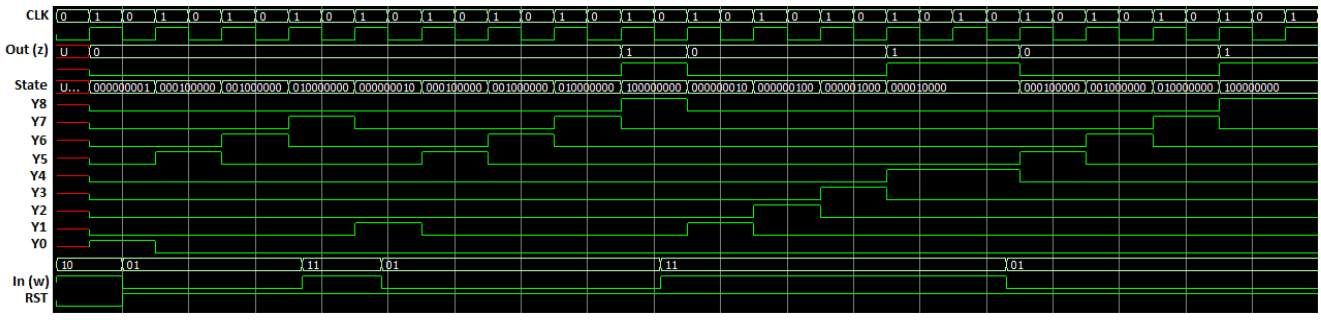

Figure 2.3. Design 1 Simulation Wave.

Section 3.A: Manually Designing the FSM with Almost One-Hot Encoding

In a practical design, traditional one hot encoding is not preferable, because it is much easier to design a FSM where the reset state is all zeroes (In this case, the reset line can be hooked directly into the reset ports of all flip-flops in the FSM). The last portion of part one reflected this fact, and re-did the manual re-design with almost one hot encoding. This meant that each state was represented with eight bits, with the first state as "00000000", but the preparation for the design was otherwise identical. Figure 3.1 shows the state table and equations for the second design.

Present									Next (w = 0)								Next (w = 1)								Output
Name	y7	y6	y5	y4	y3	y2	y1	y0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	(z)
A	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0
B	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
C	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
D	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
E	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	1
F	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
G	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
H	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
I	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

Logic
Y0 = w(y4 + y5 + y6 + y7)
Y1 = wy0
Y2 = wy1
Y3 = w(y2 + y3)
Y4 = w'(y0 + y1 + y2 +y3)
Y5 = w'y4
Y6 = w'y5
Y7 =w'(y6 + y7)
z = y4 + y8

Figure 3.1. Design 2 State Table and Output Equations

The VHDL code for design two was very similar to the code for design one. The only major difference was a consequence of almost one hot encoding. Note that state A is "00000000" under almost one hot encoding. This means that the Boolean equations from Figure 2.2 always evaluate false for state A, which is incorrect. Thankfully, fixing this is as simple as specifying the behavior for state A independent of the Boolean logic, which is exactly what is done in the VHDL code for part two. All VHDL code for the second FSM design is seen in appendix A.2.1.

Section 3.B: Testing the Second FSM

Simulation at this point in the project was almost a mirror image of the simulation in Section 2.B because the encoding methods were so similar. Figure 3.2 shows the simulation waveform. The only difference was the state variables, since there was one less LED necessary. However, the state transitions themselves were exactly the same as Section 2.B describes.

Sending the bitmap file to the board yielded a circuit that behaved exactly the same as before. The test bench used in simulation is shown in appendix A.2.2.

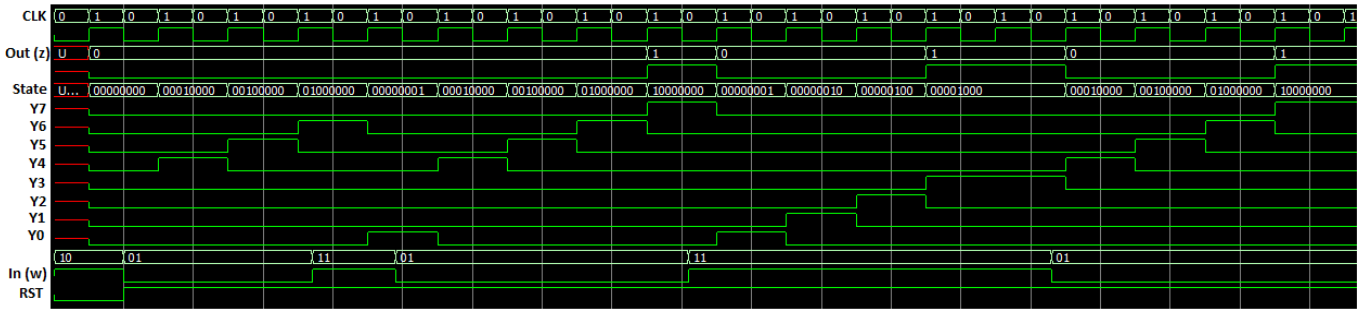

Figure 3.2. Design 2 Simulation Wave.

Section 4.A: Designing the FSM with CASE-WHEN Statements

The designs in part one of the lab were functional, but they weren't necessarily effective, since changing the encoding meant changing the code. Part two of this lab involved designing the FSM with CASE-WHEN statements, which rectified this shortcoming by allowing the compiler to set up the encoding. The other nice part about this type of design is that it only requires the state diagram in Figure 1.1 and otherwise does not require any preparation work.

Designing the FSM with CASE-WHEN statements in the architecture first required a type definition for the state variable, which had one possible value for each different state. When the machine was user-encoded, ATTRIBUTE statements were used to encode the values specified in Figure 4.1 as specified in the lab instructions. These statements were otherwise commented out to allow the user to specify the encoding via a compile time parameter in the "Assignments → Settings" menus of Quartus. Then, a process statement for the state variable was entered, which used a CASE-WHEN statement wrapped in an IF statement with no ELSE to describe the next state of the FSM based on the input and current state. Finally, another process statement was entered, which used a second CASE-WHEN statement to specify the output value in each state and the LEDRs for the current state. All VHDL code for the last FSM design is seen in appendix A.3.1.

Name	State Code $y_3y_2y_1y_0$
A	0000
B	0001
C	0010
D	0011
E	0100
F	0101
G	0110
H	0111
I	1000

Figure 4.1. User Encoding for FSM 3 [1].

Section 4.B: Testing the Final FSM

The first step in testing the final state machine was verifying proper operation of the machine. This was done in much the same manner as the previous two machines; utilizing the same test bench. Figure 4.2 shows the output from simulation, which passed the same as the previous machines did. Additionally, the circuit was also sent to the board, had the same tests repeated on it, and behaved identically. The test bench used in simulation is shown in appendix A.3.2.

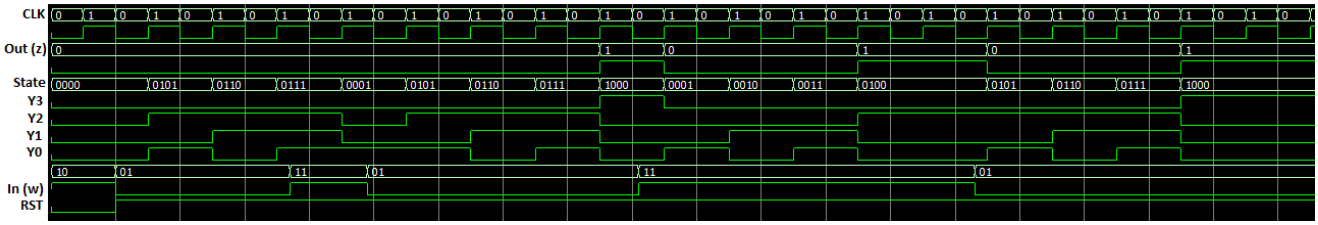

Figure 4.2. Design 3 Simulation Wave.

More interestingly, however, is the second stage of testing. Now that the machine was designed with the Quartus standard for declaring state machines, it was finally possible to verify that Quartus actually built a state machine that varied encoding with the compiler settings. This was first done using the RTL Viewer tool to open the netlist constructed by the complier in a visual format and examine the state diagram shown for the state machine that was built. Figure

4.3 shows this state diagram. Note that it exactly matches the diagram in Figure 1.1, which means the VHDL code correctly built the machine.

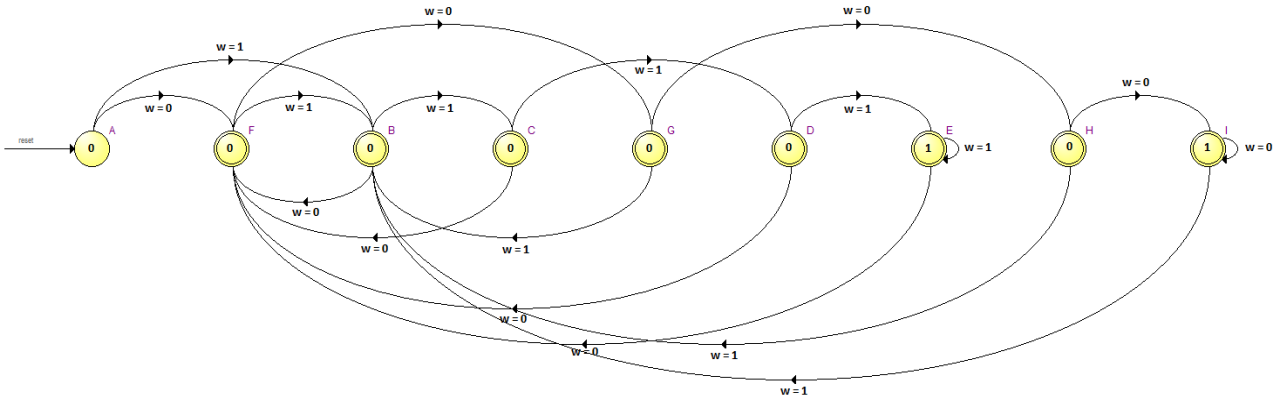

Figure 4.3. Quartus State Diagram for FSM 3.

The last part of this testing showed off the power of the Quartus complier. When a state machine is built by the compiler, after the "Analysis and Synthesis" section of the compilation report includes a "State Machines" folder, which contains a state table for the FSM that shows off the encoding used. Figure 4.4 shows two instances of this table. The version on the left was built when the user encoding in Figure 4.1 was specified. The second version was built when "One Hot" was specified in the "State Machine Processing" parameter of "Assignments → Settings → Compiler Settings → Advanced" menu of Quartus. Notice how the state variable changes size and type depending on the encoding type. The reason the "One Hot" encoding is not true one hot encoding is because it is easier to build a FSM when the reset state is all zeroes, as discussed in Section 3.A of this report.

	Name	y~5	y~4	y~3	y~2
1	y.A	0	0	0	0
2	y.B	0	0	0	1
3	y.C	0	0	1	0
4	y.D	0	0	1	1
5	y.E	0	1	0	0
6	y.F	0	1	0	1
7	y.G	0	1	1	0
8	y.H	0	1	1	1
9	y.I	1	0	0	0

	Name	y.I	y.H	y.G	y.F	y.E	y.D	y.C	y.B	y.A
1	y.A	0	0	0	0	0	0	0	0	0
2	y.B	0	0	0	0	0	0	0	1	1
3	y.C	0	0	0	0	0	0	1	0	1
4	y.D	0	0	0	0	0	1	0	0	1
5	y.E	0	0	0	0	1	0	0	0	1
6	y.F	0	0	0	1	0	0	0	0	1
7	y.G	0	0	1	0	0	0	0	0	1
8	y.H	0	1	0	0	0	0	0	0	1
9	y.I	1	0	0	0	0	0	0	0	1

Figure 4.4. Quartus State Tables for FSM 3 with Two Encoding Types.

Conclusions

Overall, this lab illustrated the essential steps in designing a state machine. Part one stepped through the manual design process, which requires making a state diagram, then a state table with a set encoding, and then deriving equations for each output. This is a good way to check work while designing a finite state machine. Part two showed that, thanks to the power of Quartus, all that is really necessary to design a state machine is a state diagram. However, in large designs, a table should still be generated for comparison with the Quartus-generated design to verify correctness. The biggest lesson in this lab is that you must be careful with syntax in VHDL: While building a state machine with CASE-WHEN statements, it is important to set up the state variable correctly with a user-defined type. If this is not done correctly, the compiler will not recognize the built circuit as a state machine, and will not provide a state diagram or table for the overall design. Syntax matters.

References

[1] M. Smith. *ECE327 Digital System Design, Lab 2: Finite State Machines Intro.* [Online]. Available: https://bb.clemson.edu/bbcswebdav/pid-1887657-dt-content-rid-17699569_2/courses/smithmc-ece-327-DSD/Lab2_FSM_partA.pdf

APPENDIX

Appendix A.1: FSM 1 – Manual, One Hot Encoding

A.1.1: VHDL Code

```vhdl
-- Ryan Barker --

LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- Declare state machine --
ENTITY FSM IS
   PORT (clk     : IN std_logic;
         w, rst : IN std_logic;
         state  : OUT std_logic_vector(8 DOWNTO 0);
         z       : OUT std_logic);
END FSM;

-- Architecture of state machine: Manual --
ARCHITECTURE Behavioral OF FSM IS
SIGNAL y : std_logic_vector(8 DOWNTO 0);
BEGIN
    logic: PROCESS (clk)
    BEGIN
        IF rising_edge(clk) THEN
            IF rst = '0'
               -- Synchronous Reset --
                THEN y <= "000000001";
            ELSE
               -- Find intermediate logic --
                y(0) <= '0';
                y(1) <= w AND (y(0) OR y(5) OR y(6) OR y(7) OR y(8));
                y(2) <= w AND y(1);
                y(3) <= w AND y(2);
                y(4) <= w AND (y(3) OR y(4));
                y(5) <= (NOT(w)) AND (y(0) OR y(1) OR y(2) OR y(3) OR y(4));
                y(6) <= (NOT(w)) AND y(5);
                y(7) <= (NOT(w)) AND y(6);
                y(8) <= (NOT(w)) AND (y(7) OR y(8));
            END IF;
        END IF;
    END PROCESS logic;

    -- Find output --
    state <= y;
    z <= y(4) OR y(8);
END Behavioral;
```

```
52
53  LIBRARY ieee;
54  USE ieee.std_logic_1164.all;
55
56  -- Declare Entity for mapping state machine to physical ports --
57  ENTITY Lab2a IS
58      PORT(KEY   : IN std_logic_vector(0 DOWNTO 0);
59           SW    : IN std_logic_vector(1 DOWNTO 0);
60           LEDR  : OUT std_logic_vector(8 DOWNTO 0);
61           LEDG  : OUT std_logic_vector(0 DOWNTO 0));
62  END Lab2a;
63
64  -- Port map for state machine --
65  ARCHITECTURE Structural of Lab2a IS
66  COMPONENT FSM
67      PORT (clk     : IN std_logic;
68            w, rst  : IN std_logic;
69           state    : OUT std_logic_vector(8 DOWNTO 0);
70            z       : OUT std_logic);
71  END COMPONENT;
72
73  BEGIN
74      Lab2a : FSM
75          PORT MAP(clk=>KEY(0), w=>SW(1), rst=>SW(0), state=>LEDR, z=>LEDG(0));
76  END Structural;
77
```

A.1.2: Test Bench

```
1   -- Copyright (C) 1991-2014 Altera Corporation. All rights reserved.
15
16  -- *****************************************************************************
22
23  -- Vhdl Test Bench template for design  :  Lab2a
27
28  LIBRARY ieee;
29  USE ieee.std_logic_1164.all;
30
31  ENTITY Lab2a_vhd_tst IS
32  END Lab2a_vhd_tst;
33  ARCHITECTURE Lab2a_arch OF Lab2a_vhd_tst IS
34  -- constants
35  -- signals
36  SIGNAL KEY : STD_LOGIC_VECTOR(0 DOWNTO 0);
37  SIGNAL LEDG : STD_LOGIC_VECTOR(0 DOWNTO 0);
38  SIGNAL LEDR : STD_LOGIC_VECTOR(8 DOWNTO 0);
39  SIGNAL SW : STD_LOGIC_VECTOR(1 DOWNTO 0);
40  COMPONENT Lab2a
41      PORT (
42      KEY : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
43      LEDG : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
44      LEDR : BUFFER STD_LOGIC_VECTOR(8 DOWNTO 0);
45      SW : IN STD_LOGIC_VECTOR(1 DOWNTO 0)
46      );
47  END COMPONENT;
```

```
48  BEGIN
49      i1 : Lab2a
50      PORT MAP (
51  -- list connections between master ports and signals
52      KEY => KEY,
53      LEDG => LEDG,
54      LEDR => LEDR,
55      SW => SW
56      );
57  init : PROCESS
58  -- variable declarations
59  BEGIN
60              -- Initialize input and toggle reset
61              SW(0) <= '0';
62              SW(1) <= '1'; wait for 10 ps;
63              SW(0) <= '1';
64
65              -- Test input
66              SW(1) <= '0'; wait for 27 ps;
67              SW(1) <= '1'; wait for 12 ps;
68              SW(1) <= '0'; wait for 42 ps;
69              SW(1) <= '1'; wait for 52 ps;
70              SW(1) <= '0';
71  WAIT;
72  END PROCESS init;
73  always : PROCESS
74  -- optional sensitivity list
75  -- (        )
76  -- variable declarations
77  BEGIN
78          -- code executes for every event on sensitivity list
79  WAIT;
80  END PROCESS always;
81  clk : PROCESS
82  BEGIN
83      KEY(0) <= '0'; wait for 5 ps;
84      KEY(0) <= '1'; wait for 5 ps;
85  END PROCESS clk;
86  END Lab2a_arch;
87
```

Appendix A.2: FSM 2 – Manual, Almost One Hot Encoding

A.2.1: VHDL Code

```
1   -- Ryan Barker --
11
12  LIBRARY ieee;
13  USE ieee.std_logic_1164.all;
14
15  -- Declare state machine --
16  ENTITY FSM2 IS
17      PORT (clk     : IN std_logic;
18            w, rst  : IN std_logic;
19            state   : OUT std_logic_vector(7 DOWNTO 0);
20            z       : OUT std_logic);
21  END FSM2;
22
23  -- Architecture of state machine: Manual --
24  ARCHITECTURE Behavioral OF FSM2 IS
25  SIGNAL y : std_logic_vector(7 DOWNTO 0);
26  BEGIN
27      logic: PROCESS (clk)
28      BEGIN
29          IF rising_edge(clk) THEN
30              IF rst = '0'
31                  -- Synchronous Reset --
32                  THEN y <= "00000000";
33              ELSIF y = "00000000" THEN
34                  IF w = '0' THEN y <= "00010000";
35                  ELSE y <= "00000001";
36                  END IF;
37              ELSE
38                  -- Find intermediate logic --
39                  y(0) <= w AND (y(4) OR y(5) OR y(6) OR y(7));
40                  y(1) <= w AND y(0);
41                  y(2) <= w AND y(1);
42                  y(3) <= w AND (y(2) OR y(3));
43                  y(4) <= (NOT(w)) AND (y(0) OR y(1) OR y(2) OR y(3));
44                  y(5) <= (NOT(w)) AND y(4);
45                  y(6) <= (NOT(w)) AND y(5);
46                  y(7) <= (NOT(w)) AND (y(6) OR y(7));
47              END IF;
48          END IF;
49
50      END PROCESS logic;
51
52      -- Find output --
53      state <= y;
54      z <= y(3) OR y(7);
55  END Behavioral;
```

```
56
57  LIBRARY ieee;
58  USE ieee.std_logic_1164.all;
59
60  -- Declare Entity for mapping state machine to physical ports --
61  ENTITY Lab2b IS
62      PORT(KEY   : IN std_logic_vector(0 DOWNTO 0);
63           SW    : IN std_logic_vector(1 DOWNTO 0);
64           LEDR  : OUT std_logic_vector(7 DOWNTO 0);
65           LEDG  : OUT std_logic_vector(0 DOWNTO 0));
66  END Lab2b;
67
68  -- Port map for state machine --
69  ARCHITECTURE Structural of Lab2b IS
70  COMPONENT FSM2
71     PORT (clk     : IN std_logic;
72              w, rst : IN std_logic;
73           state    : OUT std_logic_vector(7 DOWNTO 0);
74              z        : OUT std_logic);
75  END COMPONENT;
76
77  BEGIN
78      Lab2b : FSM2
79          PORT MAP(clk=>KEY(0), w=>SW(1), rst=>SW(0), state=>LEDR, z=>LEDG(0));
80  END Structural;
81
```

A.2.2: Test Bench

```
1  -- Copyright (C) 1991-2014 Altera Corporation. All rights reserved.
15
16 -- ********************************************************************
22
23 -- Vhdl Test Bench template for design  :  Lab2b
27
28 LIBRARY ieee;
29 USE ieee.std_logic_1164.all;
30
31 ENTITY Lab2b_vhd_tst IS
32 END Lab2b_vhd_tst;
33 ARCHITECTURE Lab2b_arch OF Lab2b_vhd_tst IS
34 -- constants
35 -- signals
36 SIGNAL KEY : STD_LOGIC_VECTOR(0 DOWNTO 0);
37 SIGNAL LEDG : STD_LOGIC_VECTOR(0 DOWNTO 0);
38 SIGNAL LEDR : STD_LOGIC_VECTOR(7 DOWNTO 0);
39 SIGNAL SW : STD_LOGIC_VECTOR(1 DOWNTO 0);
40 COMPONENT Lab2b
41     PORT (
42     KEY : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
43     LEDG : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
44     LEDR : BUFFER STD_LOGIC_VECTOR(7 DOWNTO 0);
45     SW : IN STD_LOGIC_VECTOR(1 DOWNTO 0)
46     );
47 END COMPONENT;
48 BEGIN
```

```
48  BEGIN
49      i1 : Lab2b
50      PORT MAP (
51  -- list connections between master ports and signals
52      KEY => KEY,
53      LEDG => LEDG,
54      LEDR => LEDR,
55      SW => SW
56      );
57  init : PROCESS
58  -- variable declarations
59  BEGIN
60              -- Initialize input and toggle reset
61              SW(0) <= '0';
62              SW(1) <= '1'; wait for 10 ps;
63              SW(0) <= '1';
64
65              -- Test input
66              SW(1) <= '0'; wait for 27 ps;
67              SW(1) <= '1'; wait for 12 ps;
68              SW(1) <= '0'; wait for 42 ps;
69              SW(1) <= '1'; wait for 52 ps;
70              SW(1) <= '0';
71  WAIT;
72  END PROCESS init;
73  always : PROCESS
74  -- optional sensitivity list
75  -- (        )
76  -- variable declarations
77  BEGIN
78          -- code executes for every event on sensitivity list
79  WAIT;
80  END PROCESS always;
81  clk : PROCESS
82  BEGIN
83      KEY(0) <= '0'; wait for 5 ps;
84      KEY(0) <= '1'; wait for 5 ps;
85  END PROCESS clk;
86  END Lab2b_arch;
87
```

Appendix A.3: FSM 3 – CASE WHEN Statements

A.3.1: VHDL Code

```
1  -- Ryan Barker --
12
13 LIBRARY ieee;
14 USE ieee.std_logic_1164.all;
15
16 -- Declare state machine --
17 ENTITY FSM3 IS
18    PORT (clk     : IN std_logic;
19          w, rst : IN std_logic;
20          state   : OUT std_logic_vector(3 DOWNTO 0);
21          z       : OUT std_logic);
22 END FSM3;
23
24 -- Architecture of state machine: By case statements --
25 ARCHITECTURE Behavioral OF FSM3 IS
26 -- Define each state for FSM3 --
27 TYPE fsm_state IS (A, B, C, D, E, F, G, H, I);
28 -- To user-encode with four bit binary, uncomment the last two lines --
29 --ATTRIBUTE SYN_ENCODING : STRING;
30 --ATTRIBUTE SYN_ENCODING OF fsm_state : TYPE IS "0000 0001 0010 0011 0100 0101 0110 0111 1000";
31 SIGNAL y : fsm_state;
32
33 BEGIN
34    logic: PROCESS (clk)
35    BEGIN
36       IF rising_edge(clk) THEN
37          IF rst = '0'
38             -- Synchronous Reset --
39             THEN y <= A;
40          ELSE
41             -- Specify next state based on current state and input. --
42             -- Also, send current state to output --
43             CASE y IS
44                WHEN A =>
45                   IF w = '0' THEN y <= F;
46                   ELSE y <= B;
47                   END IF;
48                WHEN B =>
49                   IF w = '0' THEN y <= F;
50                   ELSE y <= C;
51                   END IF;
52                WHEN C =>
53                   IF w = '0' THEN y <= F;
54                   ELSE y <= D;
55                   END IF;
56                WHEN D =>
57                   IF w = '0' THEN y <= F;
58                   ELSE y <= E;
59                   END IF;
60                WHEN E =>
61                   IF w = '0' THEN y <=F;
```

```
61                          IF w = '0' THEN y <=F;
62                          ELSE y <= E;
63                          END IF;
64                      WHEN F =>
65                          IF w = '0' THEN y <= G;
66                          ELSE y <= B;
67                          END IF;
68                      WHEN G =>
69                          IF w = '0' THEN y <= H;
70                          ELSE y <= B;
71                          END IF;
72                      WHEN H =>
73                          IF w = '0' THEN y <= I;
74                          ELSE y <= B;
75                          END IF;
76                      WHEN I =>
77                          IF w = '0' THEN y <= I;
78                          ELSE y <= B;
79                          END IF;
80                  END CASE;
81              END IF;
82          END IF;
83      END PROCESS logic;
84
85      -- Find output --
86      output : PROCESS (y)
87      BEGIN
88          CASE y IS
89              WHEN A => z <= '0'; state <= "0000";
90              WHEN B => z <= '0'; state <= "0001";
91              WHEN C => z <= '0'; state <= "0010";
92              WHEN D => z <= '0'; state <= "0011";
93              WHEN E => z <= '1'; state <= "0100";
94              WHEN F => z <= '0'; state <= "0101";
95              WHEN G => z <= '0'; state <= "0110";
96              WHEN H => z <= '0'; state <= "0111";
97              WHEN I => z <= '1'; state <= "1000";
98          END CASE;
99      END PROCESS output;
100 END Behavioral;
```

```
101
102  LIBRARY ieee;
103  USE ieee.std_logic_1164.all;
104
105  -- Declare Entity for mapping state machine to physical ports --
106 ENTITY Lab2c IS
107      PORT(KEY  : IN std_logic_vector(0 DOWNTO 0);
108           SW   : IN std_logic_vector(1 DOWNTO 0);
109           LEDR : OUT std_logic_vector(3 DOWNTO 0);
110           LEDG : OUT std_logic_vector(0 DOWNTO 0));
111 END Lab2c;
112
113  -- Port map for state machine --
114 ARCHITECTURE Structural of Lab2c IS
115 COMPONENT FSM3
116     PORT (clk     : IN std_logic;
117           w, rst : IN std_logic;
118          state  : OUT std_logic_vector(3 DOWNTO 0);
119           z       : OUT std_logic);
120 END COMPONENT;
121
122  BEGIN
123      Lab2c : FSM3
124          PORT MAP(clk=>KEY(0), w=>SW(1), rst=>SW(0), state=>LEDR, z=>LEDG(0));
125 END Structural;
126
```

A.3.2: Test Bench

```
  1 -- Copyright (C) 1991-2014 Altera Corporation. All rights reserved.
15
16 -- ********************************************************************
22
23 -- Vhdl Test Bench template for design  :  Lab2c
27
28  LIBRARY ieee;
29  USE ieee.std_logic_1164.all;
30
31 ENTITY Lab2c_vhd_tst IS
32 END Lab2c_vhd_tst;
33 ARCHITECTURE Lab2c_arch OF Lab2c_vhd_tst IS
34 -- constants
35 -- signals
36  SIGNAL KEY : STD_LOGIC_VECTOR(0 DOWNTO 0);
37  SIGNAL LEDG : STD_LOGIC_VECTOR(0 DOWNTO 0);
38  SIGNAL LEDR : STD_LOGIC_VECTOR(3 DOWNTO 0);
39  SIGNAL SW : STD_LOGIC_VECTOR(1 DOWNTO 0);
40 COMPONENT Lab2c
41      PORT (
42      KEY : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
43      LEDG : OUT STD_LOGIC_VECTOR(0 DOWNTO 0);
44      LEDR : BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0);
45      SW : IN STD_LOGIC_VECTOR(1 DOWNTO 0)
46      );
47 END COMPONENT;
```

```
48  BEGIN
49      i1 : Lab2c
50      PORT MAP (
51  -- list connections between master ports and signals
52      KEY => KEY,
53      LEDG => LEDG,
54      LEDR => LEDR,
55      SW => SW
56      );
57  init : PROCESS
58  -- variable declarations
59  BEGIN
60              -- Initialize input and toggle reset
61              SW(0) <= '0';
62              SW(1) <= '1'; wait for 10 ps;
63              SW(0) <= '1';
64
65              -- Test input
66              SW(1) <= '0'; wait for 27 ps;
67              SW(1) <= '1'; wait for 12 ps;
68              SW(1) <= '0'; wait for 42 ps;
69              SW(1) <= '1'; wait for 52 ps;
70              SW(1) <= '0';
71  WAIT;
72  END PROCESS init;
73  always : PROCESS
74  -- optional sensitivity list
75  -- (        )
76  -- variable declarations
77  BEGIN
78          -- code executes for every event on sensitivity list
79  WAIT;
80  END PROCESS always;
81  clk : PROCESS
82  BEGIN
83      KEY(0) <= '0'; wait for 5 ps;
84      KEY(0) <= '1'; wait for 5 ps;
85  END PROCESS clk;
86  END Lab2c_arch;
87
```