

ECE253 – Laboratory Exercise 6

Finite State Machines

This is an exercise in using finite state machines.

Preparation and In-Lab Marks

Pre-Lab: The contents of the pre-lab should contain the Verilog Code and annotated simulations for ALL parts of this lab. All questions should be answered and included in the pre-lab (see particularly Part 2, #7). The pre-lab should be handed in to your TA at the beginning of the lab. Note: Part I has two Verilog Implementations AND simulations.

In-lab: You are required to demonstrate Part III to a TA.

Part I

We wish to implement a finite state machine (FSM) that recognizes two specific sequences of applied input symbols, namely four consecutive 1s or four consecutive 0s. There is an input w and an output z . Whenever $w = 1$ or $w = 0$ for four consecutive clock pulses the value of z has to be 1; otherwise, $z = 0$. Overlapping sequences are allowed, so that if $w = 1$ for five consecutive clock pulses the output z will be equal to 1 after the fourth and fifth pulses. Figure 1 illustrates the required relationship between w and z .

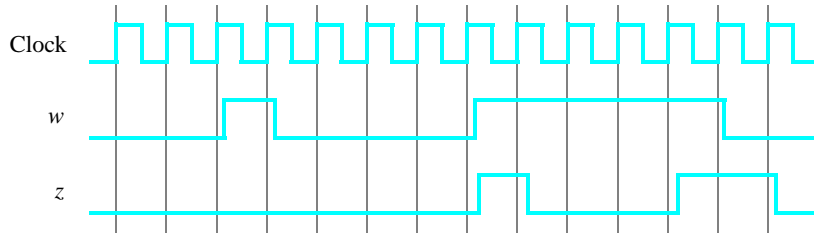


Figure 1: Required timing for the output z .

A state diagram for this FSM is shown in Figure 2. For this part you are to manually derive an FSM circuit that implements this state diagram, including the logic expressions that feed each of the state flip-flops. To implement the FSM use nine state flip-flops called y_8, \dots, y_0 and the one-hot state assignment given in Table 1.

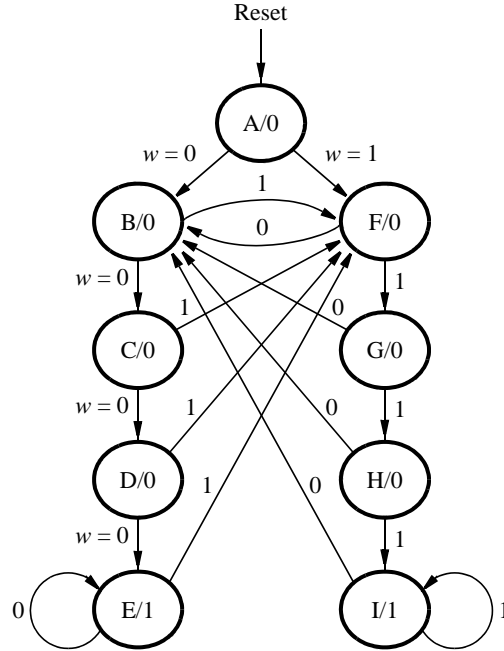


Figure 2: A state diagram for the FSM.

| Name | State Code |
|----------|-------------------------------|
| | $y_8y_7y_6y_5y_4y_3y_2y_1y_0$ |
| A | 000000001 |
| B | 000000010 |
| C | 000000100 |
| D | 000001000 |
| E | 000010000 |
| F | 000100000 |
| G | 001000000 |
| H | 010000000 |
| I | 100000000 |

Table 1: One-hot codes for the FSM.

Design and implement your circuit on the DE1-SoC board as follows:

1. Create a new Quartus project for the FSM circuit.
2. Write Verilog that instantiates the nine flip-flops in the circuit and which specifies the logic expressions that drive the flip-flop input ports. Use only simple **assign** statements in your Verilog code to specify the logic feeding the flip-flops. Note that the one-hot code enables you to derive these expressions by inspection.
Use the toggle switch SW_0 on the DE1-SoC board as an active-low synchronous reset input for the FSM, use SW_1 as the w input, and the pushbutton KEY_0 as the clock input which is applied manually. Use the red light $LEDR_9$ as the output z , and assign the state flip-flop outputs to the red lights $LEDR_8$ to $LEDR_0$.
3. Include the Verilog file in your project, and assign the pins on the FPGA to connect to the switches and the LEDs.

4. Simulate your Verilog using a set of inputs that demonstrates your code is correct.
5. Once you are confident that the circuit works properly as a result of your simulation, download the circuit into the FPGA chip. Test the functionality of your design by applying the input sequences and observing the output LEDs. Make sure that the FSM properly transitions between states as displayed on the red LEDs, and that it produces the correct output values on $LEDR_9$.
6. Finally, consider a modification of the one-hot code given in Table 1. It is often desirable to set all flip-flop outputs to the value 0 in the reset state.

Table 2 shows a modified one-hot state assignment in which the reset state, A, uses all 0s. This is accomplished by inverting the state variable y_0 . Create a modified version of your Verilog code that implements this state assignment. (*Hint*: you should need to make very few changes to the logic expressions in your circuit to implement the modified state assignment.)

7. Simulate your Verilog using a set of inputs that demonstrates your code is correct.
8. Compile your new circuit and test it on the board.

| Name | State Code |
|----------|-------------------------------|
| | $y_8y_7y_6y_5y_4y_3y_2y_1y_0$ |
| A | 000000000 |
| B | 000000011 |
| C | 000000101 |
| D | 000001001 |
| E | 000010001 |
| F | 000100001 |
| G | 001000001 |
| H | 010000001 |
| I | 100000001 |

Table 2: Modified state codes for the FSM.

Part II

For this part you are to write another style of Verilog code for the FSM in Figure 2. In this version of the code you should not manually derive the logic expressions needed for each state flip-flop. Instead, describe the state table for the FSM by using a Verilog **case** statement in an **always** block, and use another **always** block to instantiate the state flip-flops (as demonstrated in the lectures). You can use a third **always** block or simple assignment statements to specify the output z . To implement the FSM, use four state flip-flops y_3, \dots, y_0 and binary codes, as shown in Table 3.

| 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 |

Table 3: Binary codes for the FSM.

A suggested skeleton of the Verilog code is given in Figure 3.

```
module part2 ( ... );
    ... define input and output ports

    ... define signals
reg [3:0] y_Q, Y_D;    // y_Q represents current state, Y_D represents next state
parameter A = 4'b0000, B = 4'b0001, C = 4'b0010, D = 4'b0011, E = 4'b0100,
            F = 4'b0101, G = 4'b0110, H = 4'b0111, I = 4'b1000;

always @(w, y_Q)
begin: state_table
    case (y_Q)
        A: if (!w) Y_D = B;
           else Y_D = F;
        ... remainder of state table
        default: Y_D = 4'bxxxx;
    endcase
end // state_table

always @(posedge Clock)
begin: state_FF
    ...
end // state_FFS

    ... assignments for output z and the LEDs
endmodule
```

Figure 3: Skeleton Verilog code for the FSM.

Implement your circuit as follows.

1. Create a new project for the FSM.
2. Include in the project your Verilog file that uses the style of code in Figure 3. Use the same switches, pushbuttons, and lights that were used in Part I.
3. Simulate your Verilog using a set of inputs that demonstrates your code is correct.
4. Before compiling your code it is necessary to explicitly tell the Synthesis tool in Quartus that you wish to have the finite state machine implemented using the state assignment specified in your Verilog code. If you do not explicitly give this setting to Quartus, the Synthesis tool will automatically use a state assignment of its own choosing, and it will ignore the state codes specified in your Verilog code. To make this setting, choose **Assignments > Settings** in Quartus, and click on the **Compiler Settings** item on the left side of the window, then click on the **Advanced Setting (Synthesis)** button. As indicated in Figure 4, change the parameter **State Machine Processing** to the setting **User-Encoded**.
5. Compile your project. To examine the circuit produced by Quartus open the RTL Viewer tool. Double-click on the box shown in the circuit that represents the finite state machine, and determine whether the state diagram that it shows properly corresponds to the one in Figure 2. To see the state codes used for your FSM, open the **Compilation Report**, select the **Analysis and Synthesis** section of the report, and click on **State Machines**.
6. Download the circuit into the FPGA chip and test its functionality.

- In step 3 you instructed the Quartus Synthesis tool to use the state assignment given in your Verilog code. To see the result of removing this setting, open again the Quartus settings window by choosing **Assignments > Settings**, and click on the **Compiler Settings** item, then click on the **Advanced Settings (Synthesis)** button. Change the setting for **State Machine Processing** from **User-Encoded** to **One-Hot**. Recompile the circuit and then open the report file, select the **Analysis and Synthesis** section of the report, and click on **State Machines**. Compare the state codes shown to those given in Table 2, and be prepared to discuss any differences that you observe with a TA.

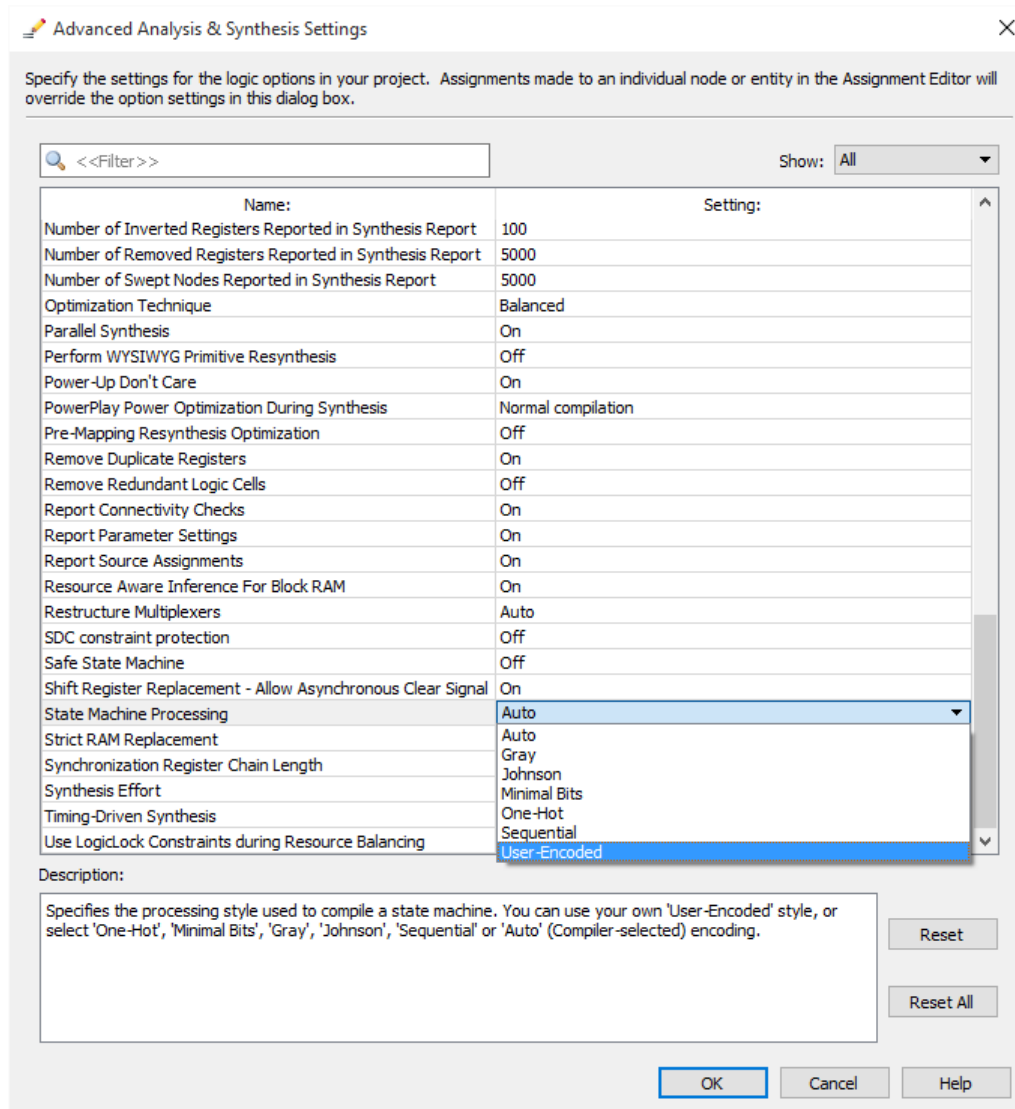


Figure 4: Specifying the state assignment method in Quartus.

Part III

In this part of the exercise you are to implement a Morse-code encoder using an FSM. The Morse code uses patterns of short and long pulses to represent a message. Each letter is represented as a sequence of dots (a short pulse), and dashes (a long pulse). For example, the middle-ish eight letters of the alphabet have the following representation:

| | |
|---|---------|
| J | • — — — |
| K | — • — |
| L | • — • • |
| M | — — |
| N | — • |
| O | — — — |
| P | • — — • |
| Q | — — • — |

Design and implement a Morse-code encoder circuit using an FSM. Your circuit should take as input one of the eight letters of the alphabet in the table above (J–Q), and display the Morse code for it on $LEDR[0]$. Use switches SW_{2-0} and pushbuttons KEY_{1-0} as inputs. When a user presses KEY_1 , the circuit should display the Morse code for a letter specified by SW_{2-0} (000 for J, 001 for K, etc.), using 0.5-second pulses to represent dots, and 1.5-second pulses to represent dashes. The “spaces” between dots/dashes should be 0.5 seconds in length. Pushbutton KEY_0 should function as an asynchronous reset.

A high-level schematic diagram of a *possible* circuit for the Morse-code encoder is shown in Figure 5.

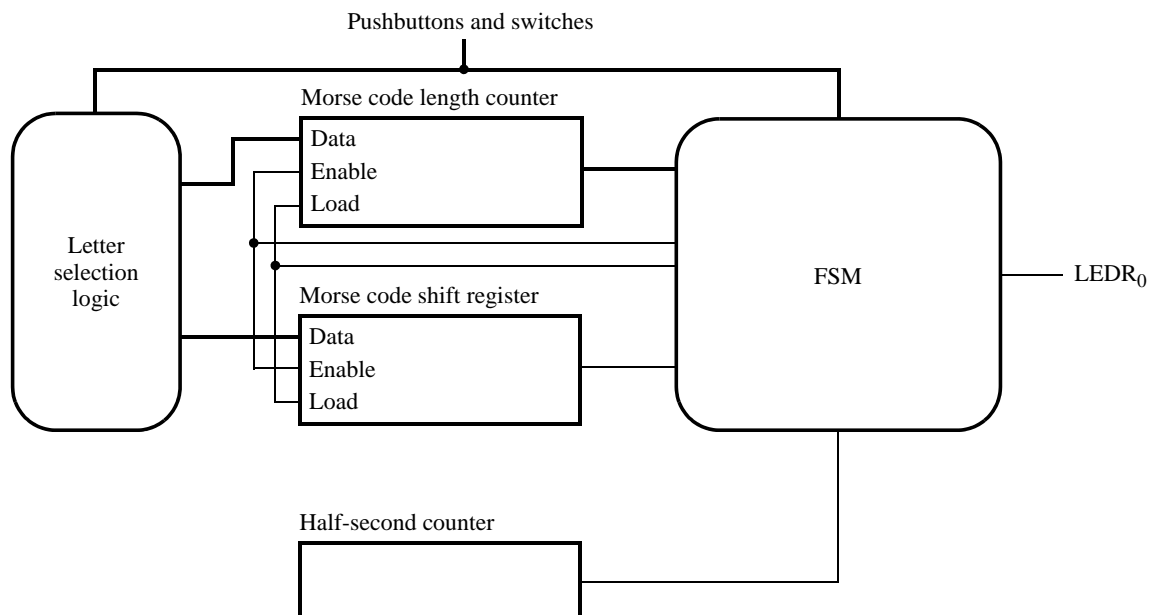


Figure 5: High-level schematic diagram of the circuit for Part III.

1. Write Verilog for the circuit, using the KEY , SW and $LEDR$ ports mentioned above.
2. Simulate your circuit to verify its correctness. Think carefully about a way to simulate the circuit, with a limited set of simulation input vectors that will allow you to verify correctness.
3. Make the necessary pin assignments needed to implement the circuit on the DE1-SoC board, and compile the code. Download and test your circuit.
4. Demonstrate the working circuit to the TA for the in-lab-marked portion of the lab.