ELECTRICAL & COMPUTER ENGINEERING

Lab 5: SEQUENTIAL LOGIC AND FINITE STATE MACHINES

The University of Washington | The Remote Hub Lab | Last Revised: March 2022

Summary

This lab introduces sequential logic and Finite State Machines (FSMs) both in theory and in SystemVerilog. The lab walks through a full system design problem, analyzing a problem statement and demonstrating behaviors on FPGAs using SystemVerilog.

Table of Contents

What is a Finite State Machine (FSM)?

Assume we have a digital system which is an automatic door with two states: open and closed. If there is movement near the door (input is true), we want the system to be in the “open” state. When there is no movement nearby (input is false), we want the output to be “closed.” We can describe this system visually with a “state diagram”, as in Figure 1.

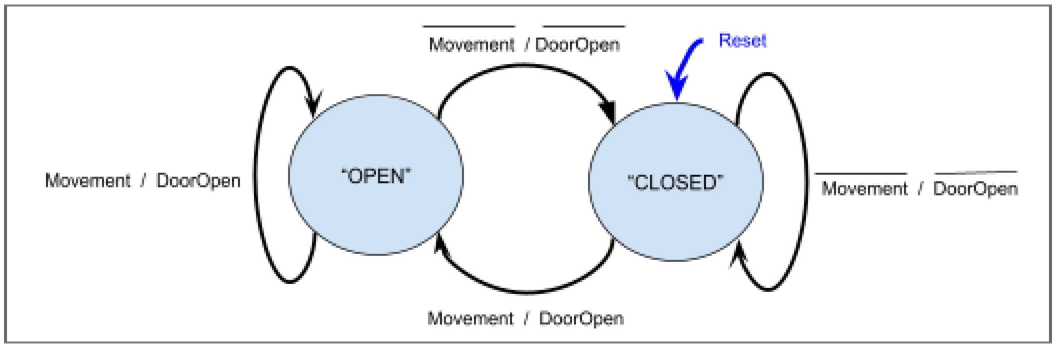

$\overline{\text{Movement}}$ / $\overline{\text{DoorOpen}}$
Reset
Movement / DoorOpen
"OPEN"
"CLOSED"
$\overline{\text{Movement}}$ / $\overline{\text{DoorOpen}}$
Movement / DoorOpen

***Figure 1**: Automatic Door State Diagram*

This example depicts the Finite State Machine (FSM): a state diagram with a finite number of states (two in this case). There are four important things to consider in the FSM:

1. **There is a Reset signal which indicates where our system should start after a reboot.**

2. **Each state has an arrow for every input possibility.**
   - In Figure 1, the arrows between states describe when changes occur. Because there is only one significant input signal (Movement), there are only two possibilities in each state: ON (Movement is true) and OFF (Movement is false).
   - As a result, there are two arrows which originate from the “open” state and two arrows which originate from the “closed” state.

3. **Every arrow is labeled with an Input/Output (I/O) behavior.**
   - In Figure 1, the arrow starting from the “open” state pointing back to itself demonstrates the following behavior: when there is movement, the door stays open (when Movement is true, the output DoorOpen is true).
   - The other arrow starting from the “open” state going to the “closed state” demonstrates the other behavior: when there is no movement, the door closes (when Movement is false, the output DoorOpen is false and we change states).

4. **Because we define there to be only one input (Movement) and one output (DoorOpen), the I/O pattern can be expressed with 1-bit binary. Because there are only two states, we can also assign distinct encodings to the states in binary.**
   - Figure 2 is a simpler (and more common) way to draw an equivalent FSM.

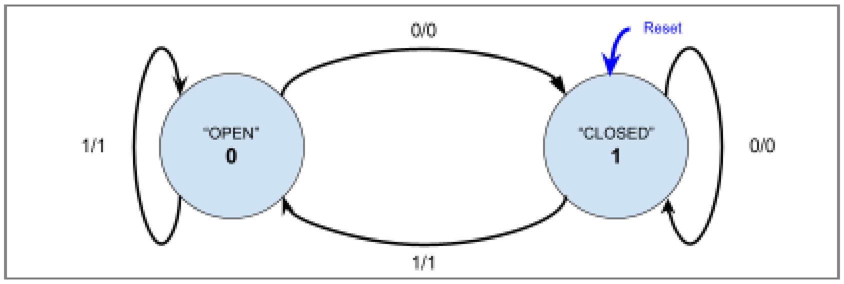


***Figure 2**: Simpler Automatic Door State Diagram*

How does a system remember the information it needs to transition between states? In Labs 0-4, we focused on combinational logic which is good for expressing output behaviors as one Boolean expression. Recall the "mystery box" model from Lab 1, provided in Figure 3.

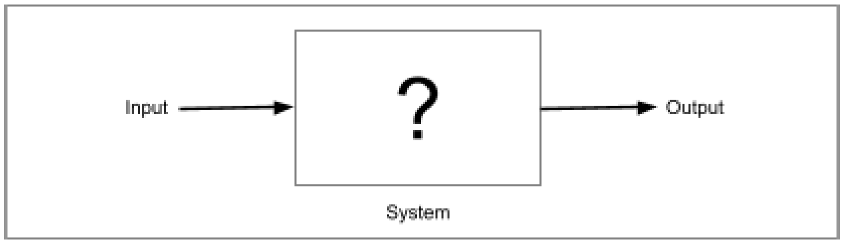


***Figure 3**: Input/Output System*

However, if we want to define a signal that has memory of what happened before a change, then we want to understand sequential logic. In this case, we have a new model in Figure 4.

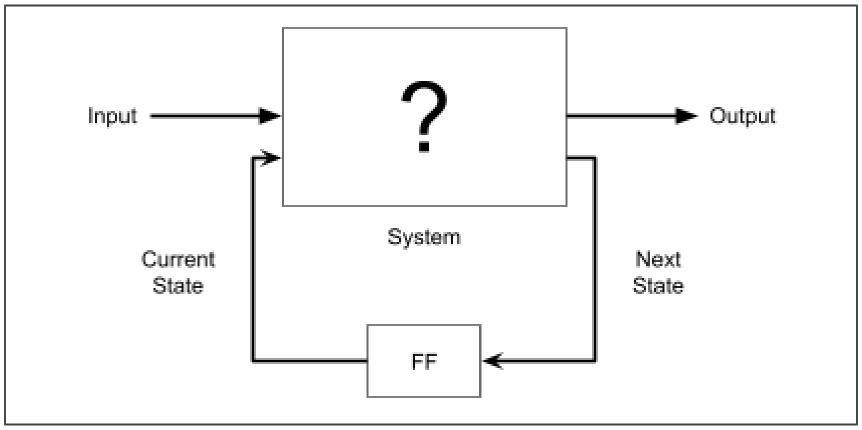


***Figure 4**: Input/Output System with a Flip Flop for Feedback*

Notice how there is a new box called "FF" (representing a flip-flop) which allows the system to remember states even as time progresses. To express this added information with the Truth Table, we must add a Present State input column and a Next State output column in addition to the normal I/O signals. A State Table for the automatic door FSM is provided in Table 1.

***Table 1**: Automatic Door State Table*

PresentState	Movement	DoorOpen	NextState
0	0	0	1
0	1	1	0
1	0	0	1
1	1	1	0

Table 1 is a direct translation of the state diagram in Figure 2. First, note that our “Truth Table” is now called a “State Table.” Also, notice that there is an input column “Movement” and an output column “DoorOpen” for our input and output signals respectively. The “Present State” input column refers to our state encoding: 0 for “Open” and 1 for “Closed.” So what does each row in Table 1 mean?

// Abbreviations: PS = Present State; NS = Next State

1. When the door is open → // PS = Open = 0
   and there is no movement, → // Movement = 0
   the door should close and → // DoorOpen = 0
   go to the “Closed” state. → // NS= Closed = 1

2. When the door is open → // PS = Open = 0
   and there is movement, → // Movement = 1
   the door should be open → // DoorOpen = 1
   and stay in the “Open” state. → // NS= Open = 0

3. When the door is closed → // PS = Closed = 1
   and there is no movement, → // Movement = 0
   the door should be closed → // DoorOpen = 0
   and stay in the “Closed” state.→ // NS = Closed = 1

4. When the door is closed → // PS = Closed = 1
   and there is movement, → // Movement = 1
   the door should open → // DoorOpen = 1
   and go to the “Open” state. → // NS = Open = 0

Creating K-Maps for the DoorOpen and NextState signals is left as an exercise for the reader. However, it is important to note that this process will allow you to arrive at Boolean expressions for the output signals, and these expressions are programmable in SystemVerilog code.

FSM SystemVerilog Structure

An FSM in SystemVerilog has several new components in addition to those in Lab 4's "SystemVerilog Syntax and Main Components" section. Figure 5 shows an example SystemVerilog module for our automatic door FSM. Figure 6 and Figure 7 provide the testbench and waveform respectively.

clock and reset

- The clock and reset input signals are needed for sequential logic and flip-flops.
- In the automatic door module (Figure 5), these signals are declared as ports (line 4 and 6) and used in the always_ff block (lines 25-30).
- In the automatic door testbench (Figure 6), lines 44-48 setup the clock, and lines 51-66 simulate the reset and clock signals to test system behavior.

enumerate states

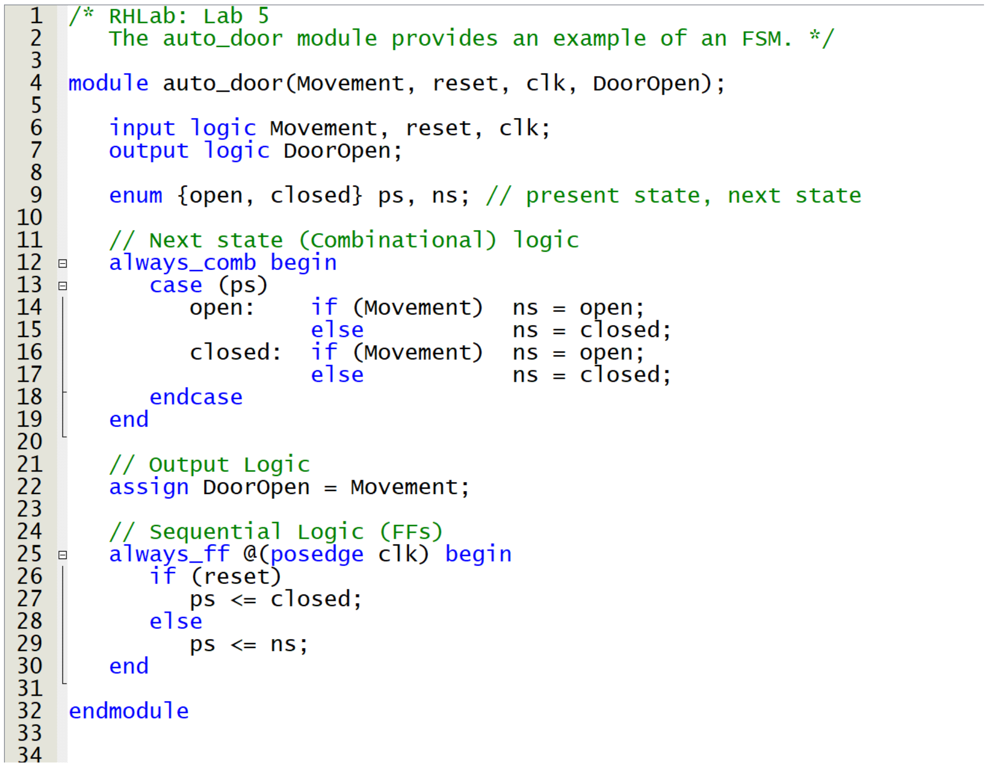
- This enumeration lists the states of your FSM.
- Line 9 in Figure 5 is an example of this enumeration. Notice the special syntax.

always_comb block

- This block encapsulates your next state logic which is combinational. As a result, this block relies on the present state and next state cases you defined in your FSM.
- Figure 5's lines 12-19 is an example of the always_comb block. Note that our convention is to include "case" and "endcase" syntax (lines 13 and 18).

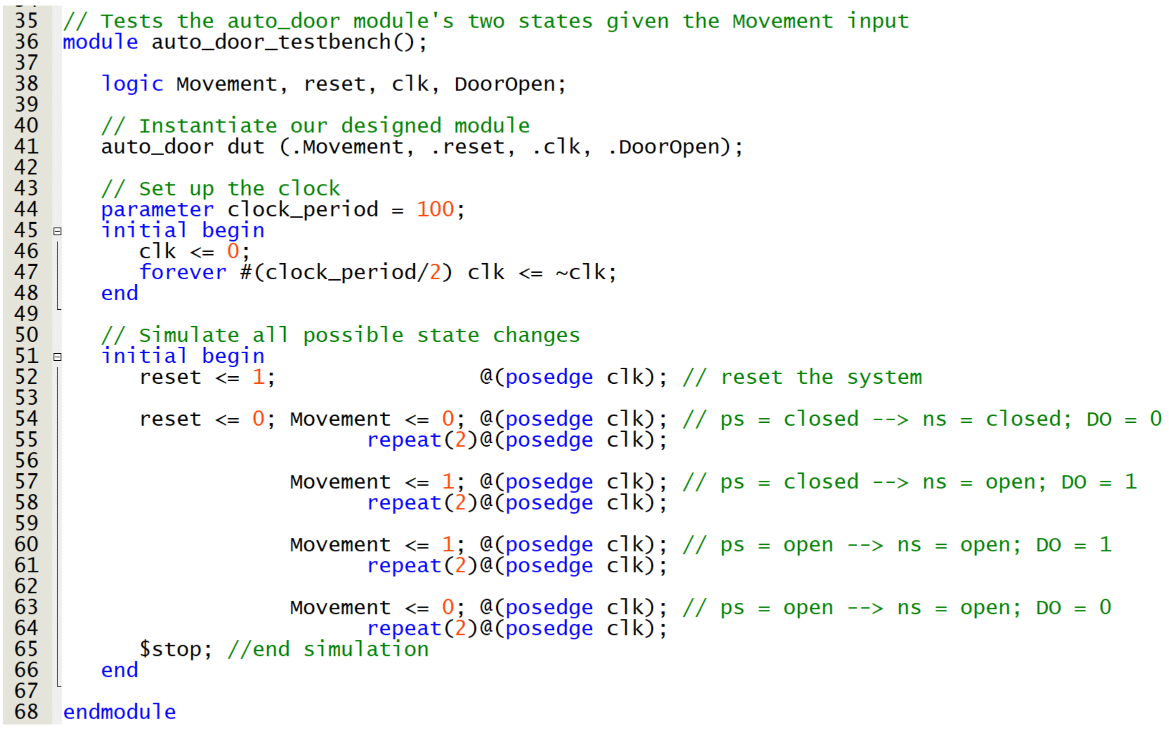
always_ff block

- This block encapsulates your sequential logic. The mechanics of this block are out of the scope of this lab series, but know that it drives the movement between present states and next states every clock cycle.
- See lines 25-30 in Figure 5.



```
/* RHLab: Lab 5
   The auto_door module provides an example of an FSM. */

module auto_door(Movement, reset, clk, DoorOpen);

   input logic Movement, reset, clk;
   output logic DoorOpen;

   enum {open, closed} ps, ns; // present state, next state

   // Next state (Combinational) logic
   always_comb begin
      case (ps)
         open:     if (Movement)  ns = open;
                   else           ns = closed;
         closed:   if (Movement)  ns = open;
                   else           ns = closed;
      endcase
   end

   // Output Logic
   assign DoorOpen = Movement;

   // Sequential Logic (FFs)
   always_ff @(posedge clk) begin
      if (reset)
         ps <= closed;
      else
         ps <= ns;
   end

endmodule

```

***Figure 5**: Automatic Door Example FSM - SystemVerilog Module*



```
// Tests the auto_door module's two states given the Movement input
module auto_door_testbench();

   logic Movement, reset, clk, DoorOpen;

   // Instantiate our designed module
   auto_door dut (.Movement, .reset, .clk, .DoorOpen);

   // Set up the clock
   parameter clock_period = 100;
   initial begin
      clk <= 0;
      forever #(clock_period/2) clk <= ~clk;
   end

   // Simulate all possible state changes
   initial begin
      reset <= 1;                @(posedge clk); // reset the system

      reset <= 0; Movement <= 0; @(posedge clk); // ps = closed --> ns = closed; DO = 0
                        repeat(2)@(posedge clk);

                  Movement <= 1; @(posedge clk); // ps = closed --> ns = open; DO = 1
                        repeat(2)@(posedge clk);

                  Movement <= 1; @(posedge clk); // ps = open --> ns = open; DO = 1
                        repeat(2)@(posedge clk);

                  Movement <= 0; @(posedge clk); // ps = open --> ns = open; DO = 0
                        repeat(2)@(posedge clk);
      $stop; //end simulation
   end

endmodule
```

***Figure 6**: Automatic Door Example FSM - SystemVerilog Testbench*

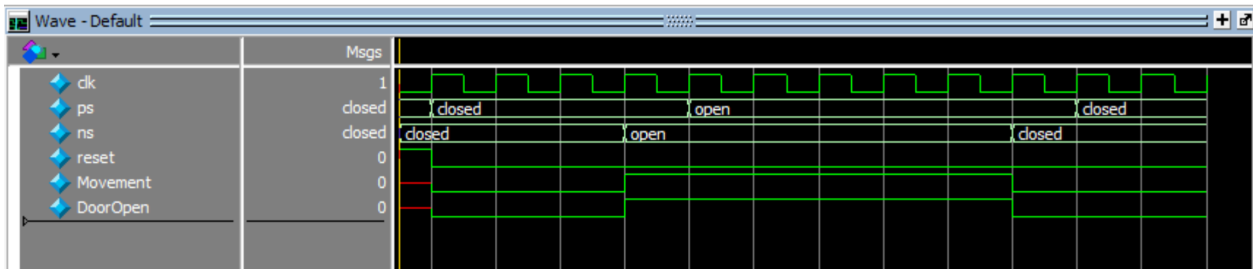


***Figure 7**: Automatic Door Example FSM - ModelSim Waveform*

Now that we can implement FSMs in SystemVerilog, let's try a guided design problem so we can put all that we've learned in Labs 0-5 together.

Problem Description

Using what you have learned in Labs 0-5, build a string recognizer which outputs a true only when the string “101” is observed. Note that a string is a sequence of characters, and our system receives a string of bits “001101010000111…”.

To solve this problem, there are five recommended steps:

1. Design an FSM to understand system behavior.
2. Describe inputs and outputs with a State Table.
3. Determine Boolean expressions for output signals using K-Maps.
4. Develop the system with SystemVerilog (Quartus) and simulate it (ModelSim).
5. Demonstrate the system on an FPGA (LabsLand).

The following sections walk you through this process.

Design a Finite State Machine to Understand System Behavior

Draw an FSM diagram corresponding to a string recognizer of ‘101’. Some hints are provided below.

- How many states do you need? What will you call them (words and binary)?
- How many inputs are there to the system? What will you call them? (Hint: think about the input signal AND present state inputs.)
- How many outputs are there to the system? What will you call them? (Hint: think about the output signal AND next state outputs.)

Describe Inputs and Outputs with a State Table

Fill in Table 2 to create a State Table for the inputs and outputs of your FSM.

***Table 2**: String Recognizer State Table*

Determine Boolean Expressions for Output Signals Using K-Maps

With the information from the State Table, use K-Maps to derive Boolean expressions for the outputs.

Verify your findings using the Boole-Web feature on LabsLand. Does the circuit schematic look like what you expected? Why or why not?

Develop and Simulate the System with SystemVerilog

Use the Boolean expressions you derived from the previous section to write and test SystemVerilog code for the string recognizer.

Quartus

Include screenshots of your code with your finished report. Use the following space to describe the syntax you used and why.

ModelSim

After simulating, draw the simulation curve you see on ModelSim. Make sure to specify your input and output signals.

Use the following space to describe why the simulation curve you see demonstrates the behavior we want the string recognizer to have.

Demonstrate the System on an FPGA

Send your code to an FPGA in LabsLand, mapping your input and output signals to the FPGA I/O (switches and LEDs). Describe what you observe in the space below.

How do you know the behavior you observed is that of a string recognizer for “101”?

Reflection and Observations

If you completed all the steps in this lab, you've just created your first digital system! Reflect on the process, things you found interesting, things you found challenging, etc. in the space below.

What did you observe about Quartus and ModelSim? Did you find the tools to be helpful?

What questions do you still have about sequential logic, FSMs, and SystemVerilog, if any?