Lab 11

FSM: Guessing Game

Updated: April 18, 2020
Version: SystemVerilog 2017

11.1 Introduction

Up to this point, we've covered combinational logic and *regular* sequential logic, meaning the output exhibits a regular or repeating pattern (e.g. counter and shift register). But not all solutions involve regular patterns. Thus, we need a design element that can handle irregular, non-repeating conditions. This is the purpose of a finite state machine (FSM). A FSM comprises a finite number of internal "states" (hence the name) and the logic required to switch between them, based on the inputs. The output can be of two different types (Mealy or Moore). This lab will examine how to design a synchronous (clocked) FSM.

11.2 Objectives

After completing this lab, you should be able to:

- Explain the role of finite state machines in digital systems design
- Explain the difference between a Mealy output and a Moore output
- Implement a state machine in Verilog

11.3 Background

A simple example of a FSM is a "debounce" circuit. We first have to explain what "bounce" means and why we need it, then we can look at the code.

In the real, physical world, signals are never as clean as we would like them to be. Both pushbuttons and switches exhibit a characteristic called "bounce," where the output cycles on and off a few times when the button is pressed or switch is moved. This happens so fast that for many devices it doesn't matter. However, digital circuits *can* change that fast, so we need to eliminate this behavior. We can do this with external circuitry or by building it into our digital circuit within the FPGA. Figure 11.1 shows the input (button) behavior and the clean, debounced output.

The typical method to debounce is to ensure the value of the input is held for a certain period of time. Although, t_{bounce} is typically 20-30 ms, we don't have to wait this full time, only longer than the longest bounce cycle. Thus, t_{wait} is shorter than t_{bounce}.

This method can be implemented with the FSM described by Figure 11.2 and implemented with the code in Listing 11.1. The FSM has four states: zero, wait1, one, and wait0. It uses a counter to determine t_{wait} in both the *wait1* and *wait0* states. The parameter N allows setting the counter bit width (and thus t_{wait}) when instantiating this module. Your instructor will provide more explanation and the files for your use.

11.4 Procedure

11.4.1 Debounce testing

Run a simulation of the `debounce_test` module. Verify the behavior of the debounce circuit as described above and in Figure 11.1. Record the time at which it changed to each of the four states (*zero, wait1, one, wait0*).

11.4.2 Guessing game

In this lab, you will be implementing a simple game, in which the player tries to guess which LED is illu-

minated and press the corresponding button. Figure 11.3 shows the state diagram for this Guessing Game. States *s0* through *s3* correspond to each illuminated segment position. *swin* represents the"correct guess" condition, and *slose* represents the "wrong guess" condition.

Create a new project named `Lab11_guess` and a new Verilog file named `guess_FSM`. Using the debounce module as a guide, develop this module to implement the state diagram in Figure 11.3. Note that square brackets have been left off for brevity, so `y0` means `y[0]`.

11.4.3 Top module

Once you have the state machine done, create a new top-level module `guessing_game`. Implement the diagram shown in Figure 11.4. Here are some notes:

Use the upper four segments of one of the 7-seg digits to display a moving target. You can decide what order they move in, but one LED should illuminate for each state (e.g. `seg[0]` for *s0*).

Use the 4 pushbuttons as the user guess input. For example, if `seg[1]` is illuminated and the user presses `btnR`, they win!

Use `btnC` as the reset input.

The user should know when they win or lose, so assign one (or more) of the LED's to the *win* output from the FSM and an LED to the *lose* output.

We want a "difficulty" option. We can do this by switching between a slow clock and fast clock. Use the main (master) `clk` for the fast one, and instantiate a counter module for the slow clock. Choose a reasonable divider value (parameter N). Use `sw[0]` to choose between clocks.

Optional: If you want to have multiple levels of difficulty, implement a MUX using a *case* statement to choose different bits of the counter output value. For example, if no switch is on, use the highest bit for the slowest clock speed. If `sw[1]` is on, use bit N-2. If `sw[2]` is on, use bit N-4. Something like that.

11.4.4 Testing (i.e. playing!)

Implement your game on the Basys3 board and play! Keep a record of 10 games played on each clock speed setting including whether you won and if so, which segment was illuminated. Determine your win percentage for each setting.

11.5 Deliverables

Submit a report containing the following:

1. Verilog source files (`guess_FSM`, `guessing_game`, and their test modules)
2. Three simulation waveforms (`debounce`, `guess_FSM`, and `guessing_game`)
3. A video of you playing a game on each speed setting
4. A list of 10 games played on each speed setting (20 total), showing if you win and which segment was illuminated, and your win percentage for each speed setting
5. Answers to the following:
   (a) At what time in the simulation did the debounce circuit reach each of the four states (*zero, wait1, one, wait0*)?
   (b) Why can this game not be implemented with regular sequential logic?
   (c) What type of outputs did you use for your design (Mealy or Moore)? Explain.

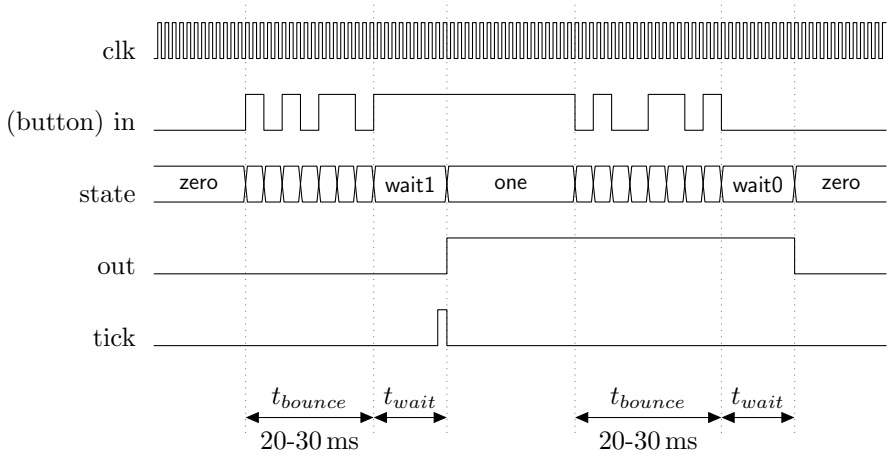

Figure 11.1: Debounce timing diagram

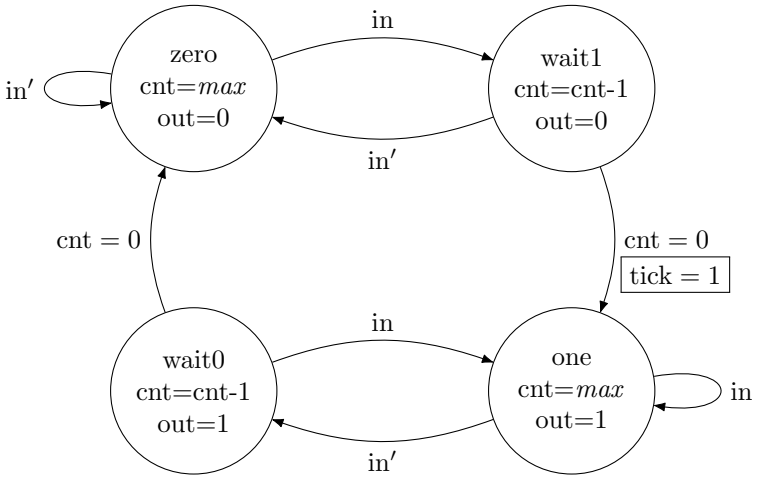

Figure 11.2: Debounce state diagram

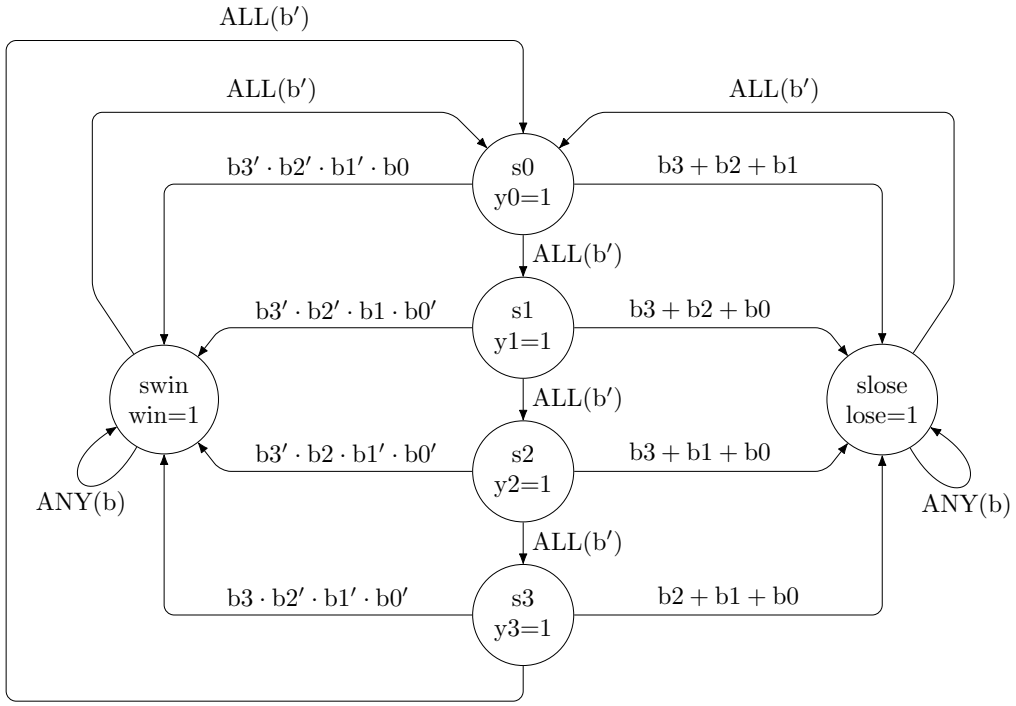

Figure 11.3: Guessing game state diagram

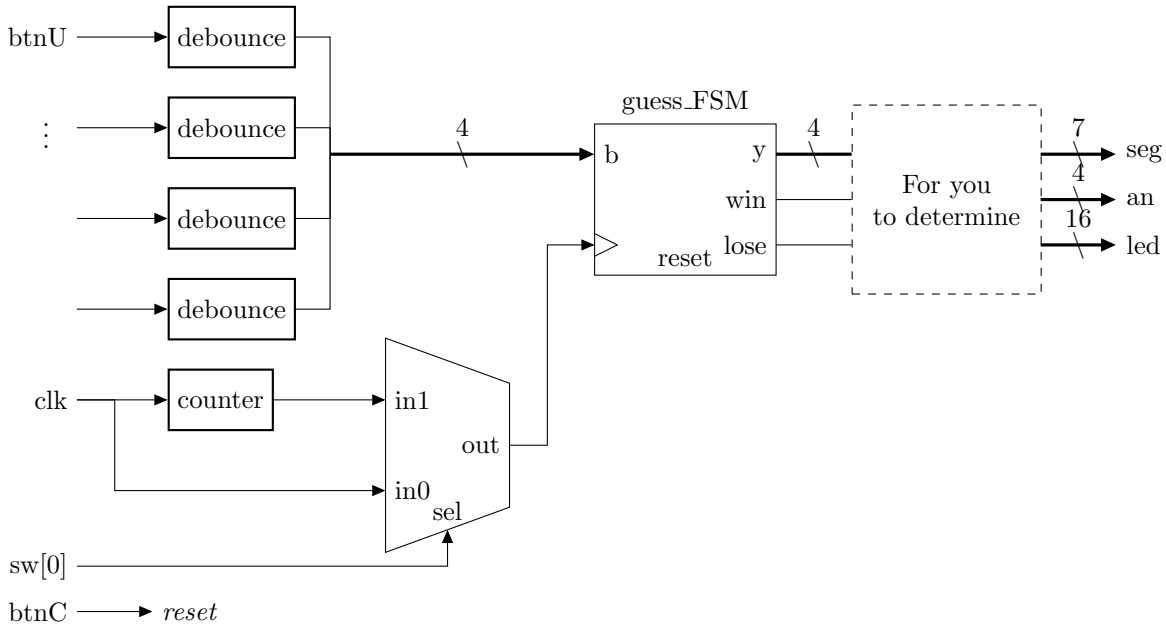

Figure 11.4: Block diagram for guessing game top module (not all connections are shown)

11.6 Code Appendix

Listing 11.1: Debounce module

```
‘timescale 1ns / 1ps
// ELC 2137, John Miller, 2019-11-08

module debounce #(parameter N=21)
   (input clk, reset,
    input  in,
    output reg out,
    output reg tick);

   // define states as local parameters
      (constants)
   localparam [1:0]
      zero   = 2’b00,
      wait1  = 2’b01,
      one    = 2’b11,
      wait0  = 2’b10;

   // internal signals
   reg [1:0] state, state_next;
   reg [N-1:0] counter, counter_next;

   // state memory (register)
   always_ff @(posedge clk or posedge
      reset)
      if (reset) begin
         state   <= zero;
         counter <= {N{1’b1}};
      end
      else begin
         state   <= state_next;
         counter <= counter_next;
      end

   // combined next-state and output
      logic
   always_comb begin
      // default behavior
      state_next   = state;
      counter_next = counter;
      tick = 0;

      case(state)
         zero: begin
            out = 0;
            counter_next = {N{1’b1}};
            if (in)
               state_next = wait1;
         end

         wait1: begin
            out = 0;     // Moore output
            counter_next = counter - 1;
            if (counter == 0) begin
               tick = 1’b1; // Mealy
                  output
               state_next = one;
            end
            else if (~in)
               state_next = zero;
         end

         one: begin
            out = 1;
            counter_next = {N{1’b1}};
            if (~in)
               state_next   = wait0;
         end

         wait0: begin
            out = 1;
            counter_next = counter - 1;
            if (counter == 0)
               state_next = zero;
            else if (in)
               state_next = one;
         end
      endcase
   end

endmodule // debounce
```

Listing 11.2: Debounce test module

```
‘timescale 1ns / 1ps
// ELC 2137, John Miller, 2019-11-08

module debounce_test();

    reg clk, reset, in;
    wire out, tick;
    integer i;

    debounce #(.N(2)) db (.clk(clk), .
        reset(reset), .in(in), .out(out),
         .tick(tick));

    always begin
        #5 clk = ~clk;
    end

    initial begin
        clk=0; reset=0; in=0; #5;
        reset=1; #10;
        reset=0; #5;
        // bounce
        for (i=0; i<10; i=i+1) begin
            #20 in=~in;
        end
```

```
        // hold input = 1 for a while
        in = 1; #200;
        // bounce
        for (i=0; i<10; i=i+1) begin
            #20 in=~in;
        end
        // hold input = 0 for a while
        in = 0; #200;
        $finish;
    end
endmodule // debounce_test
```