System Synthesis and Modeling (CSCE 3953)

Project 6

Zack Fravel

10/19/16

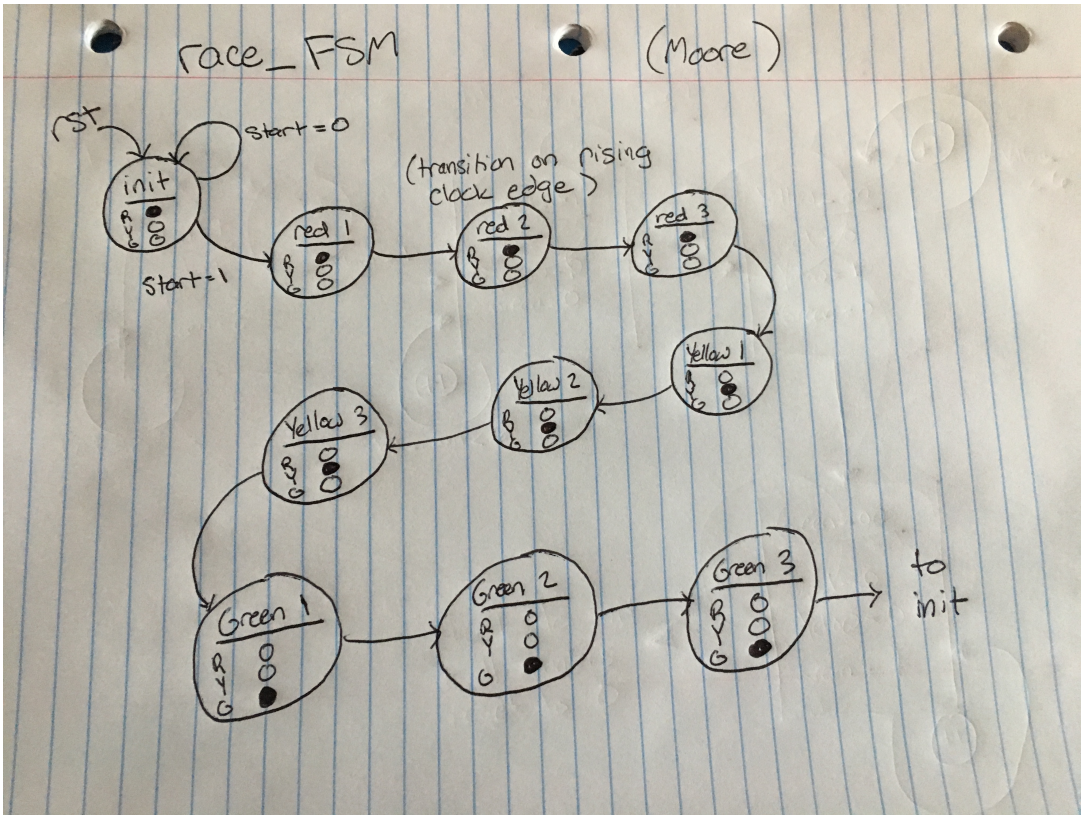
zpfravel@uark.edu

Parent Objective

The parent objective of project six was to design and implement a finite state machine in verilog that acts as a race light controller. That is, we have a device that, upon initialization, cycles through red, yellow, and green lights. We want these lights to stay on for at least three clock cycles in our design. Once the controller cycles through the three colors it goes back to its initial state with only the red light being lit, which is also the reset state.

Design Methodology / STG

We were left to make our own assumptions in our design methodology for this project. For my design, I went with a Moore FSM. That is, a finite state machine whose outputs are totally dependent on only the current state. In order to ensure the lights stay on for at least three clock cycles after the start signal is introduced, I created three separate states for each light with each state transition set to happen on the positive edge of the clock. This ensures the lights stay on for three total clock cycles. Below is the state transition graph for my race light FSM.



To implement this design in verilog, I created two four bit registers in order to store the current state and next state of the FSM and set parameters with their names respectively for all 10 states. (e.g. init = 4'b0000, red_one = 4'b0001, red_two = 4'b0010…) After that, the meat of the implementation went into three separate always blocks. The first always block is sensitive to the positive edge of the clock and the positive edge of the reset signal. This first block is responsible for handling the current state transition. If reset is high, the current state is set to "init" otherwise, the current state is set to whatever "next_state" is set to. The second always block handles the next state logic. This block is sensitive to the current state and start signals. Every time the current state changes from the first block, the second block runs and runs through a case statement to determine what the next state should be. In the init state, the circuit waits until the start signal is introduced and then sets the next state to red_one. After that, each state sets the next state to the one following it until at the very end green_three sets the next state back to init.

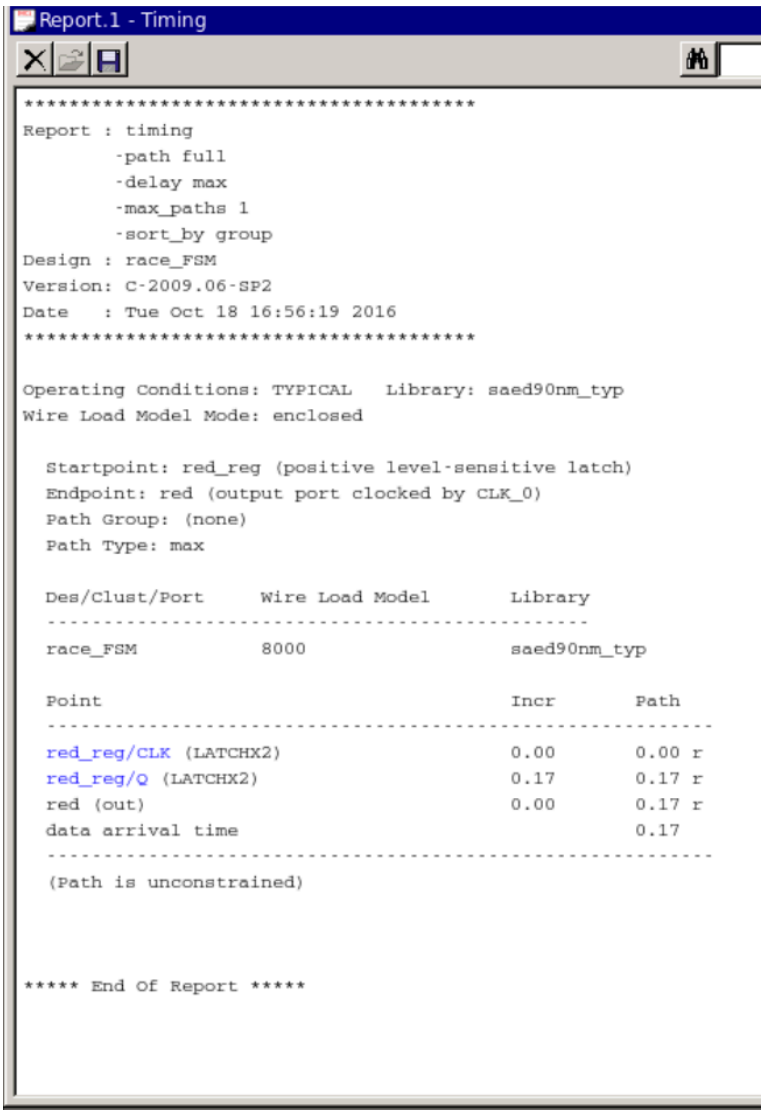
Finally, in the third always block I handle the outputs. This block is sensitive to the current state signal and contains a case statement for each case of current state. For each of the 10 states I set the outputs accordingly (e.g. yellow_two: begin red <= 1'b0; yellow <= 1'b1; green <= 1'b0; end). The source code for all the behavior described is included at the end of the report.

Simulation Method

For my simulation method I decided to write a testbench file. In my testbench I declare my inputs as registers and outputs as wires and instantiate my race_FSM module with the inputs and outputs I just declared. After that, I created an initial block that sets all three inputs (CLK,

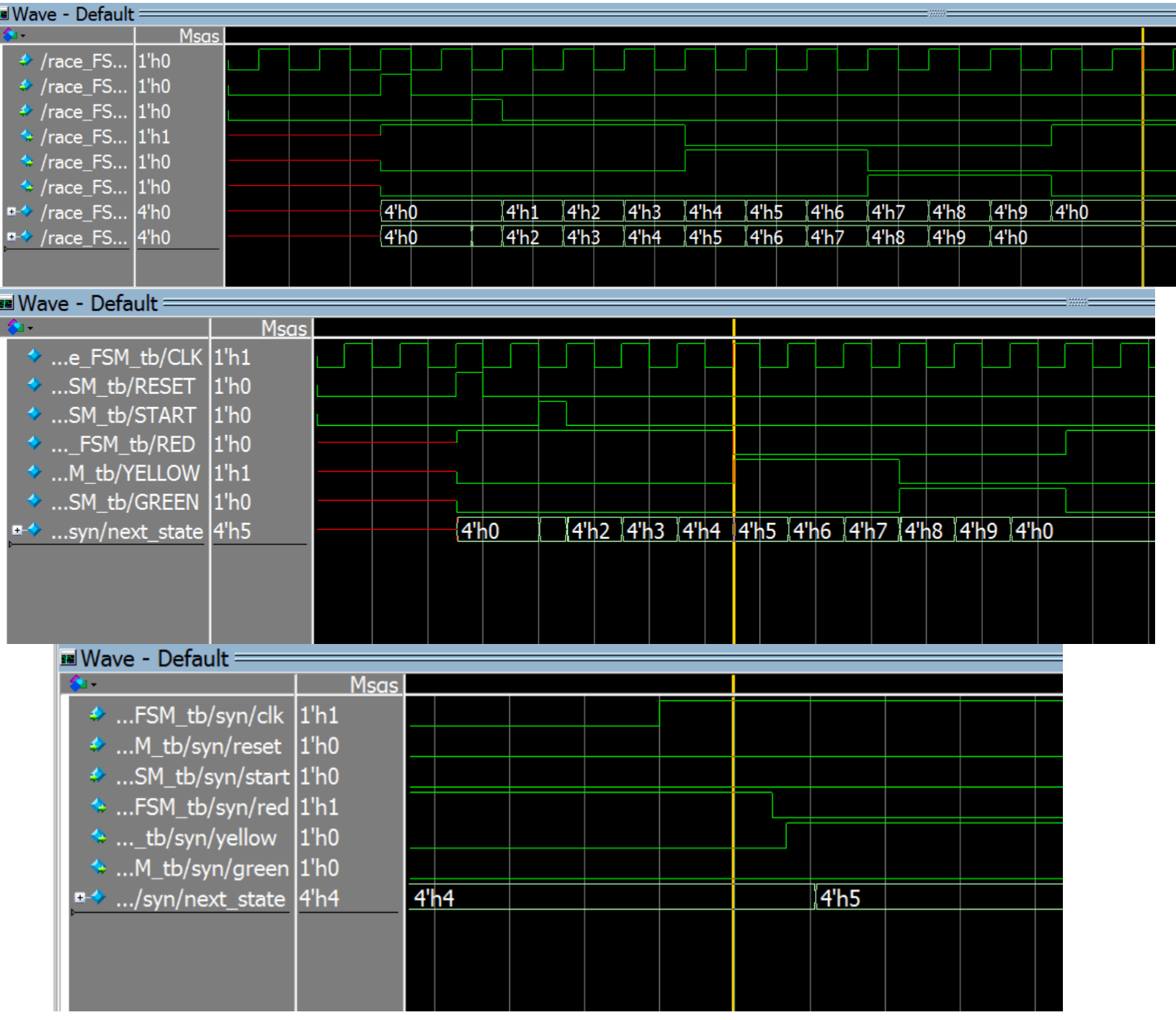
RESET, and START) to 0 and wait 50 ns. I then set the reset signal to 1, wait 10 ns and set it back to 0. Then, after 20 ns I set the START signal to 1, wait 10 ns, and set it back to 0. Other than that, the only other thing in the testbench is an always statement that sets the CLK to its compliment every 10 ns (20 ns period, 50 MHz).

To simulate the circuit with delay, I synthesized my design in Design Vision with the 90nm Standard Digital Cell Library. Once I compiled my design into the synthesis software I set my clock constraints with [create_clock -name “CLK_0” -period 50 -waveform {0.000 25.000} {clk}]. I then set the clock uncertainty and input/output delay exactly as I did in the previous project (0.14, 2.0, 0.5 respectively). Below is the timing analysis report I ran after synthesis.



Result

The following is a screenshot of the simulation waveform of the behavioral code as well as the waveform for my synthesized design. Both are running the same testbench described in the previous section. After the two full simulation screenshots I include another zoomed in shot to show the gate delay on the state transition between “red_three” and “yellow_one.”



Analysis and Conclusion

Initially I ran into a number of issues trying to synthesize a design that relied on counters and the combinational part being sensitive to the clock. This caused no issues in the behavioral design however when I tried to synthesize the code I ran into a number of strange random number generation errors as well as slack timing not being met. Once I changed my design to rely solely on state transitions and no counters everything went smoothly. In conclusion, we ended up with exactly what we set out to design; we have a race light Moore finite state machine that, upon receiving a start signal, transitions out of its initial state and cycles through red, yellow, and green lights with each light staying on for three clock cycles and returning to the initial state after completion.

Source Code

X:/Windows/race_FSM.v - Default

```
// Zack Fravel
// System Synthesis and Modeling
// Project 6

`timescale 1ns/1ns
module race_FSM(
        input wire clk,
        input wire reset,
        input wire start,
        output reg red,
        output reg yellow,
        output reg green
    );


reg[3:0] current_state, next_state;                    // State Signals

parameter init = 4'b0000, red_one = 4'b0001, red_two = 4'b0010, red_three = 4'b0011,
        yellow_one = 4'b0100, yellow_two = 4'b0101, yellow_three = 4'b0110,            // Set Parameters for State Signals
        green_one = 4'b0111, green_two = 4'b1000, green_three = 4'b1001;


always @(posedge clk or posedge reset)                 // Current State Transition
begin
    if(reset)
        current_state <= init;
    else
        current_state <= next_state;
end


always @(current_state or start)                       // Next State Logic
begin

    case(current_state)

    init: begin
          if(start)
            next_state <= red_one;
          else
            next_state <= init;
          end

    red_one:   begin next_state <= red_two; end
    red_two:   begin next_state <= red_three; end
    red_three: begin next_state <= yellow_one; end

    yellow_one:   begin next_state <= yellow_two; end
    yellow_two:   begin next_state <= yellow_three; end
    yellow_three: begin next_state <= green_one; end

    green_one:   begin next_state <= green_two; end
    green_two:   begin next_state <= green_three; end
    green_three: begin next_state <= init; end

    endcase
end


always @(current_state)
begin

    case (current_state)

    init: begin red <= 1'b1; yellow <= 1'b0; green <= 1'b0; end

    red_one:   begin red <= 1'b1; yellow <= 1'b0; green <= 1'b0; end
    red_two:   begin red <= 1'b1; yellow <= 1'b0; green <= 1'b0; end
    red_three: begin red <= 1'b1; yellow <= 1'b0; green <= 1'b0; end

    yellow_one:    begin red <= 1'b0; yellow <= 1'b1; green <= 1'b0; end
    yellow_two:   begin red <= 1'b0; yellow <= 1'b1; green <= 1'b0; end
    yellow_three: begin red <= 1'b0; yellow <= 1'b1; green <= 1'b0; end

    green_one:    begin red <= 1'b0; yellow <= 1'b0; green <= 1'b1; end
    green_two:   begin red <= 1'b0; yellow <= 1'b0; green <= 1'b1; end
    green_three: begin red <= 1'b0; yellow <= 1'b0; green <= 1'b1; end

    endcase

end

endmodule
```

X:/Windows/race_FSM_tb.v (/race_FSM_tb) - Default

```
Ln#
  1 // Zack Fravel
  2 // System Synthesis and Modeling
  3 // Project 6
  4
  5 `timescale 1ns/1ns
  6 module race_FSM_tb;
  7
  8 // Inputs
  9         reg CLK; reg RESET; reg START;
 10 // Outputs
 11         wire RED; wire YELLOW; wire GREEN;
 12 // Instantiate Module
 13         race_FSM syn (CLK, RESET, START, RED, YELLOW, GREEN);
 14
 15 initial
 16    begin
 17         CLK = 0;
 18         RESET = 0;
 19         START = 0;
 20         #50;
 21         RESET = 1;
 22         #10;
 23         RESET = 0;
 24         #20;
 25         START = 1;
 26         #10;
 27         START = 0;
 28    end
 29
 30 always #10 CLK = ~CLK;
 31
 32 endmodule
```