CS 152B Lab 2 Report

Introduction

This lab aims to design a traffic light controller using a finite state machine (FSM) in Verilog. The controller manages main and side street signals along with pedestrian walk requests, following standard traffic procedure while dynamically responding to pedestrian and vehicle presence.

Engineering standards such as reliability, real-time responsiveness, and safety must be upheld to ensure the system functions correctly under all conditions. Constraints include timing requirements, power efficiency, and maintainability. The design must ensure accurate sensor readings, minimize delays, and follow safety protocols to prevent conflicting signals.

Implementation

Schematics

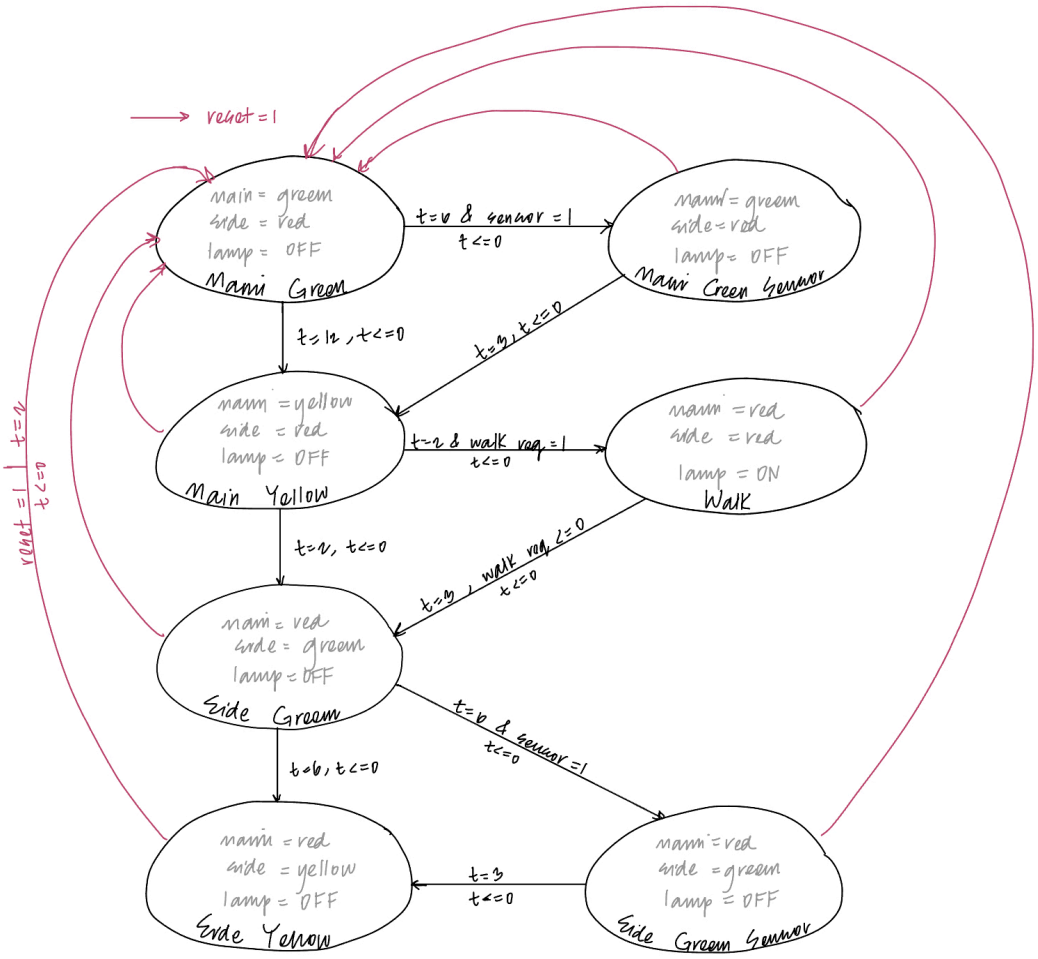


Figure 1. Finite state machine graph. Directed edges are labeled with event variables triggering the transition along with <= notation if a variable's value is modified.

States

1. Main green
2. Main yellow
3. Main green sensor
4. Side green
5. Side yellow
6. Side green sensor
7. Walk

Event variables

- t; time
- sensor; 1/0 if vehicle presence on side street
- walk req; 1/0 if walk button is pressed
- reset; 1/0 if reset button is pressed

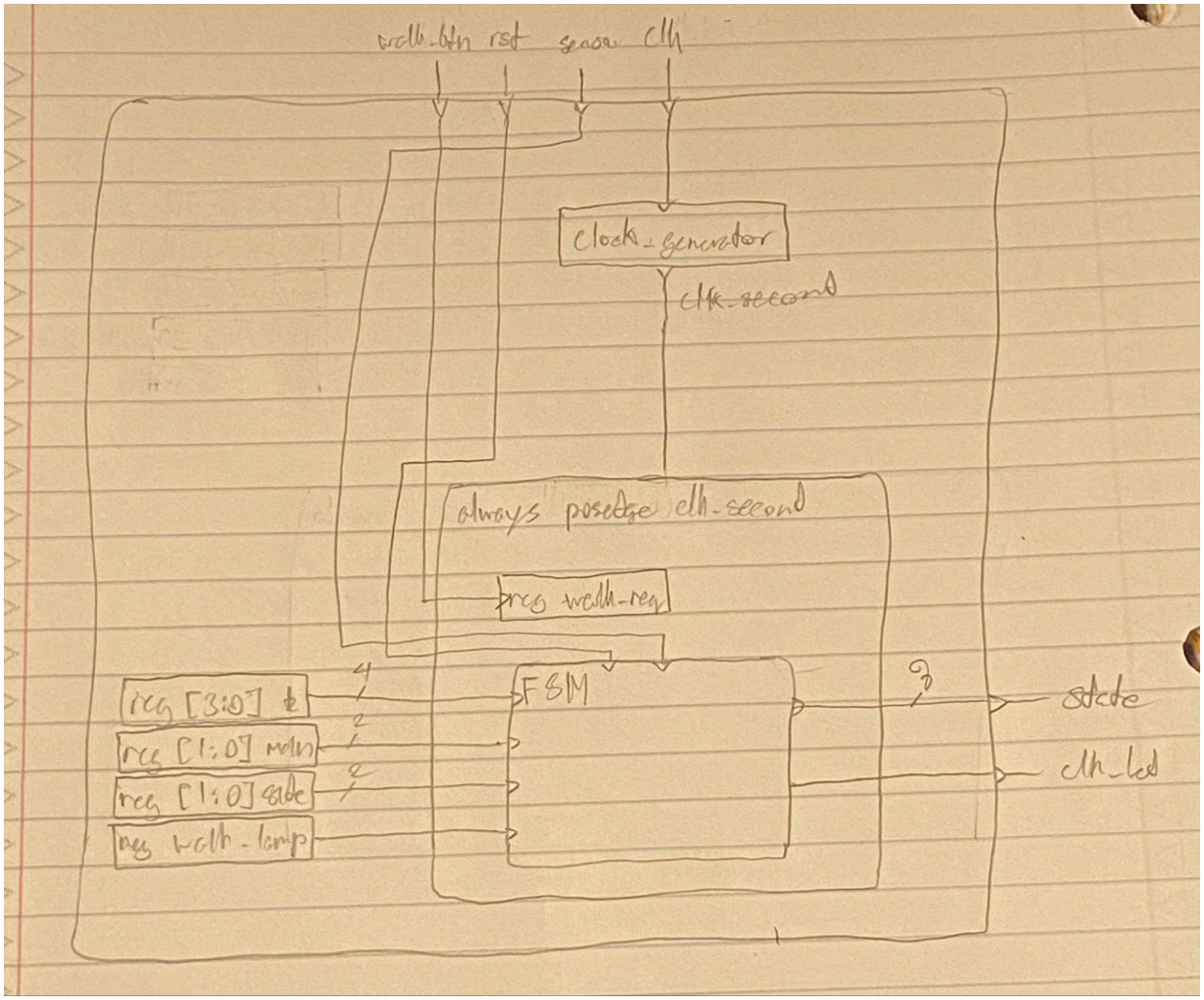


Figure 2. Traffic light controller module.

Traffic light controller

`module traffic_light_fsm`

This module implements a finite state machine for a traffic light controller that manages a main street, side street, and pedestrian walk request system. The individual states are defined by a 8-bit 1-hot encoding: `{Main green light, main yellow light, main red light,`

`side green light, side yellow light, side red light, walk lamp, sensor service}`. The last bit is for the cases where 2 states are essentially equivalent (e.g., Main green and Main green sensor). The last bit is set to zero except for when the vehicle sensor is serviced in Main green and Side green states. The module takes in the system clock as input and uses a helper module to create a 1-second clock frequency. State transitions follow the schematic in Figure 1.

```
module traffic_light_fsm(
    input clk,
    input rst,
    input sensor,
    input walk_btn,
    output reg [7:0] state
);
    reg [3:0] t;
    reg [1:0] main_light;
    reg [1:0] side_light;
    reg walk_lamp;
    reg walk_req;

    wire clk_second;
    clock_generator clk_gen(
        .clk(clk),
        .clk_second(clk_second)
    );

    // States and lights definition
    // 8-bit 1-hot encoding: {Main green light, main yellow light, main red light,
       side green light, side yellow light, side red light, walk lamp, sensor service}
    parameter MAIN_GREEN = 8'b100_001_0_0;
    parameter MAIN_YELLOW = 8'b010_001_0_0;
    parameter SIDE_GREEN = 8'b001_100_0_0;
    parameter SIDE_YELLOW = 8'b001_010_0_0;
    parameter MAIN_GREEN_SENSOR = 8'b100_001_0_1;
    parameter SIDE_GREEN_SENSOR = 8'b001_100_0_1;
    parameter WALK = 8'b001_001_1_0;

    assign clk_led = clk_second;

    initial begin
        state <= MAIN_GREEN;
        walk_req <= 0;
        t <= 0;
    end

    always @(posedge clk_second) begin
```

```
if(walk_btn == 1) begin
    if(walk_req == 0) begin
        walk_req <= 1;
    end
end
if(rst == 1) begin
    state <= MAIN_GREEN;
    walk_req <= 0;
    t <= 0;
end else begin
    case(state)
        MAIN_GREEN: begin
            if(t >= 12) begin
                state <= MAIN_YELLOW;
                t <= 0;
            end else if (t >= 6 & sensor == 1) begin
                state <= MAIN_GREEN_SENSOR;
                t <= 0;
            end else begin
                t <= t + 1;
            end
        end
        MAIN_YELLOW: begin
            if(t >= 2 & walk_req == 0) begin
                state <= SIDE_GREEN;
                t <= 0;
            end else if(t >= 2 & walk_req == 1) begin
                state <= WALK;
                t <= 0;
            end else begin
                t <= t + 1;
            end
        end
        MAIN_GREEN_SENSOR: begin
            if(t >= 3) begin
                state <= MAIN_YELLOW;
                t <= 0;
            end else begin
                t <= t + 1;
            end
        end
        SIDE_GREEN: begin
            if(t >= 6 & sensor == 0) begin
                state <= SIDE_YELLOW;
                t <= 0;
            end else if(t >= 6 & sensor == 1) begin
                state <= SIDE_GREEN_SENSOR;
                t <= 0;
            end else begin
                t <= t + 1;
            end
        end
        SIDE_YELLOW: begin
            if(t >= 2) begin
```

```
                    state <= MAIN_GREEN;
                    t <= 0;
                end else begin
                    t <= t + 1;
                end
            end
            SIDE_GREEN_SENSOR: begin
                if(t >= 3) begin
                    state <= SIDE_YELLOW;
                    t <= 0;
                end else begin
                    t <= t + 1;
                end
            end
            WALK: begin
                if(t >= 3) begin
                    state <= SIDE_GREEN;
                    t <= 0;
                    walk_req <= 0;
                end else begin
                    t <= t + 1;
                end
            end
        endcase
    end
end
endmodule
```

module clock_generator

This module generates a 1 Hz clock signal from a 100 MHz system clock by implementing a frequency divider. We count 100 million clock cycles before toggling clk_second.

```
`define ONE_HUNDRED_MILLION 100000000 // 100 MHz / ONE_HUNDRED_MILLION = 1 Hz


module clock_generator(
    input clk,
    output reg clk_second
);
    integer clk_dv = 0;

    always @(posedge clk) begin
        // 1 Hz
        if (clk_dv == 0) begin
            clk_second <= 1'b1;
        end else begin
            clk_second <= 1'b0;
        end
        // If clk_dv reaches `ONE_HUNDRED_MILLION, reset clk_dv
        if (clk_dv + 1 >= `ONE_HUNDRED_MILLION) begin
            clk_dv <= 0;
        end else begin
            clk_dv <= clk_dv + 1;
        end
```

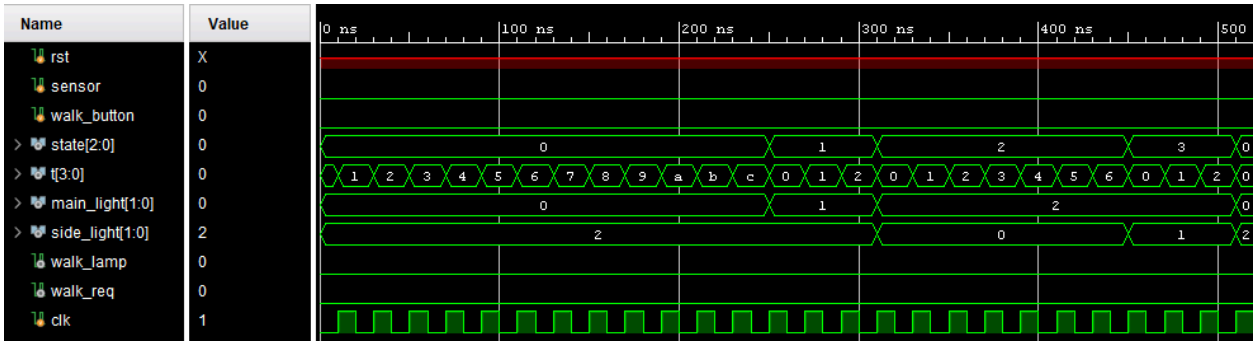
```
    end
endmodule
```

Testing

Demo

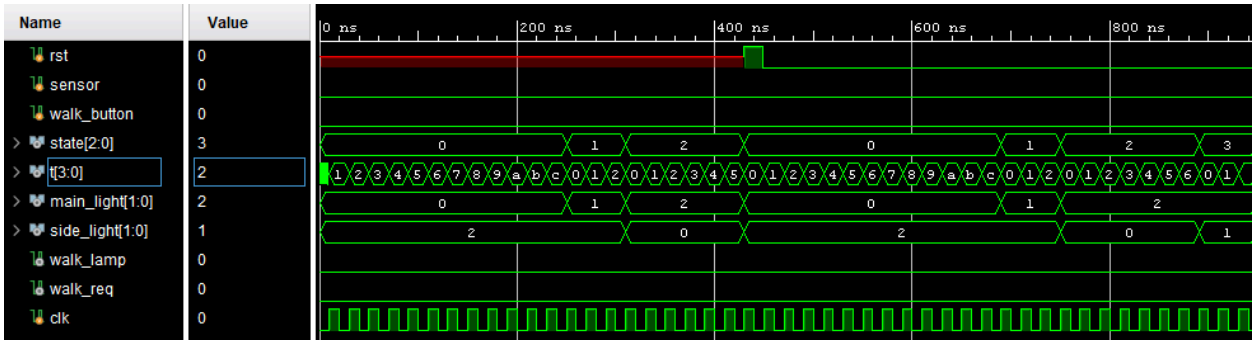
The traffic light FSM and controller was demonstrated in person using the Basys3 FPGA board. The 8-bit 1-hot encoding of states was visually represented by eight LEDs on the board, where each lit LED corresponded to a ‘hot bit’ for a specific state of the FSM. A physical switch was used to represent the vehicle sensor. The reset and walk buttons were mapped to the board’s physical push-buttons. The FSM controlled traffic flow dynamically and serviced walk and sensor requests at the correct states.

Waveforms



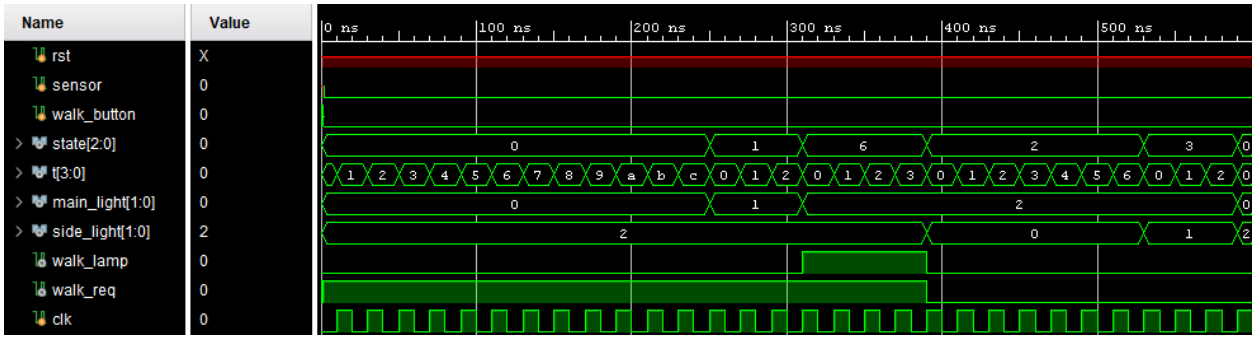
- This waveform demonstrates the standard traffic light cycle with no interruptions by a walk or car sensor signal. For the first 12 clock cycles (equivalent to seconds), the state is at “0”, meaning the main light is 0 (green) and the side light is 2 (red). For the next 3 clock cycles, the state is at “1”, meaning the main light is 1 (yellow) and the side light is 2 (red). Now, the system will be in state 2, meaning the main light is at 2 (red) and the side light is 0 (green). Next, the system will shift to state 3, where the main light is 2 (red) and the side light is 1 (yellow). Finally, the system will loop back to main light green.

```
initial begin
    // TEST 1: Run system for full traffic cycle
    walk_button = 0;
    sensor = 0;
    #520;
    $finish;
end
```



- This waveform demonstrates the standard traffic light cycle interrupted by the reset signal. For the first 17 clock cycles, the state will switch from 0 to 1 to 2 in the standard fashion, but on the 18th clock cycle, the positive edge of the reset button occurs, resetting the entire state machine back to its initial state. The machine will continue again and behave as expected in the standard cycle.

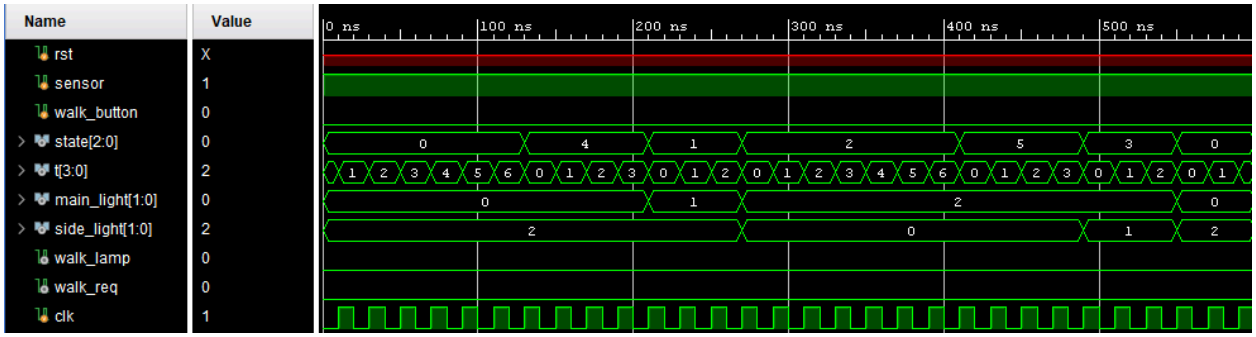
```
initial begin
    // TEST 2: Test full reset
    walk_button = 0;
    sensor = 0;
    #430;
    rst = 1;
    #20;
    rst = 0;
    #500;
    $finish;
end
```



- This waveform demonstrates the traffic light system when the walk button is pressed (at 0 clock cycles). The traffic light state will start at 0 (main light is green, side light is red), then 1 (main light is yellow, side light is red). After state 1 expires, the walk request will be serviced. The state will go to 6, where both lights are red and the walk_lamp is on, allowing pedestrians to cross the intersection. After the walk is serviced, the system will go back to its normal state and the walk_request is turned off. The system will now loop in the standard traffic light cycle.

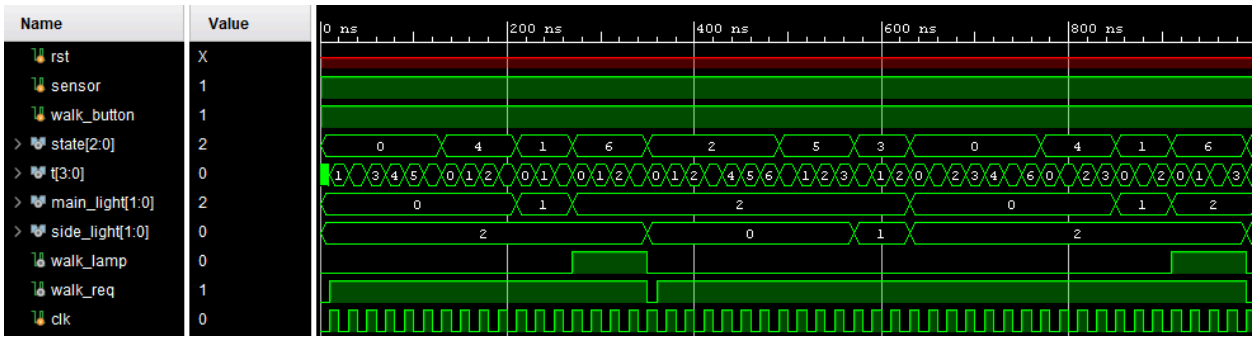
```
initial begin
    // TEST 3: Run system for full traffic cycle, walk sensor on
    walk_button = 0;
```

```
        #1;
        walk_button = 1; // Simulate button press
        #1;
        walk_button = 0;
        sensor = 0;
        #600;
        $finish;
    end
```



- This waveform demonstrates the traffic light system where the car sensor is on (constant). The traffic light state will start at 0, but instead of serving for the first 12 clock cycles, it will be truncated at 6 clock cycles. Then, the traffic light state will switch to state 4, which is functionally equivalent to state 0 (main is green and side is red), but this indicates that the main light is being affected by the sensor. After 3 clock cycles, the system will switch to state 1. The system will be in state 2 for 6 clock cycles, but will then switch into state 5, which is functionally equivalent to the previous state (main is red and side is green), but this indicates that side light is being affected by the sensor. Finally, the system will switch into state 3 and loop again.

```
    initial begin
        // TEST 4: Run system for full traffic cycle, car sensor on
        walk_button = 0;
        sensor = 1;
        #600;
        $finish;
    end
```



- This waveform demonstrates the traffic light system where both the car sensor and the walk request are on. After 6 clock cycles of state 0, the state is truncated by only 3 additional clock cycles of state 4 (where main is green and side is red). Then, state 2

occurs and the main light is now yellow. Then, the walk request is serviced in state 6 for 3 clock cycles. After the request is serviced, the walk_req shuts off (it is turned back on at the end of the clock cycle) and the system switches into state 2 (where main is red and side is green). This output is “extended” by state 5 for 3 additional clock cycles. After state 5 ends, the system switches to state 3, and then begins again at state 0.

```
    initial begin
        // TEST 5: Run system for a full traffic cycle, walk & car sensor on
        walk_button = 1;
        sensor = 1;
        #9000;
        $finish;
    end
```

module traffic_light_tb

Note that clk is not taken as input in the traffic_light_fsm module and we used the system clock for testing. The main testbench code skeleton is below:

```
module traffic_light_tb;
	reg rst;
	reg sensor;
	reg walk_button;
	wire [2:0] state;
	wire [3:0] t;              // added temporary watch variables for debugging
	wire [1:0] main_light;
	wire [1:0] side_light;
	wire walk_lamp;
	wire walk_req;

	reg clk;
	initial begin
		clk = 0;
		forever #10 clk = ~clk;
	end

	traffic_light_fsm fsm(
		.clk(clk), // From basys3 constraint file
		.rst(rst), // From basys3 constraint file
		.sensor(sensor),
		.walk_btn(walk_button),
		.state(state),
		.t(t),
		.main_light(main_light),
		.side_light(side_light),
		.walk_lamp(walk_lamp),
		.walk_req(walk_req)
	);

	initial begin
	...
	end
endmodule
```

Challenges

- One challenge we faced was the unresponsiveness of the walk request button when the program was implemented on the basys3 board. Both the walk request and the reset button were passed through debouncers, but only the reset button was being serviced by the FSM. We realized that the walk_request debouncer only generates a single signal pulse, and not a constant high signal, which was required by our code to properly service the walk_request. To mitigate this, we removed the debouncer for the walk_request (for consistency, we also removed the debouncer for the reset button). Now, both the walk_request and the reset button were properly serviced.
- Another challenge we faced was the implementation of a clock that pulses at 1Hz. To solve this, we acquired the clock frequency of the basys3 board (100 MHz) and created a clock divider. We created a counter that would increment every positive edge of the basys3 clock; when the counter reached 100 million, then 1 second passed. We send a high pulse and then reset the counter back to 0.
- One coding error we fixed was the non-deterministic behavior of breaking up our state machine into multiple always blocks, i.e., one for state transitions and one for updating the walk request. Instead, we moved the code in one initial block and one always block.
- We initially had timing and synchronization issues when creating our testbench. Originally, the traffic_light_fsm module used the system clock rather than a divided 1Hz clock. So we had to figure out how long to run each test to see the full cycle + repeat, as well as add #1 buffers to simulate button pressing.

Responsibilities

105817312: Implementation of Traffic Light FSM, FSM diagram, waveforms

905699244: FSM diagram, clock generator, debugging