Com Sci 152B Digital Design
Lab 2: State Machine Design

Michael Hale	004-620-459
Matthew Nuesca	904-440-067
Shilin Patel	904-569-866
Bingxin Zhu	704-845-969

1 Overview

In this lab we implemented a simulated traffic intersection on the FPGA development board. The design revolved around a state machine which would transition based on input signals from the street sensor and the walk button.

2 Design Requirements

The design implements the traffic and walk signals for a four way intersection with no protected turns. The traffic light pattern repeats the following cycle:

1. Main St begins with a 12-second green light.
   (a) If the traffic sensor is high after 6 seconds, move to the next step after only 3 additional seconds.
2. Main St has a 2-second yellow light.
3. If the walk button has been pressed and not serviced, illuminate the walk signal for 3 seconds.
4. Side St has a 6-second green light.
   (a) If the traffic sensor is high at the end of 6 seconds, remain green for an additional 3 seconds.
5. Side St has a 2-second yellow light.

3 Implementation Details

3.1 State Machine

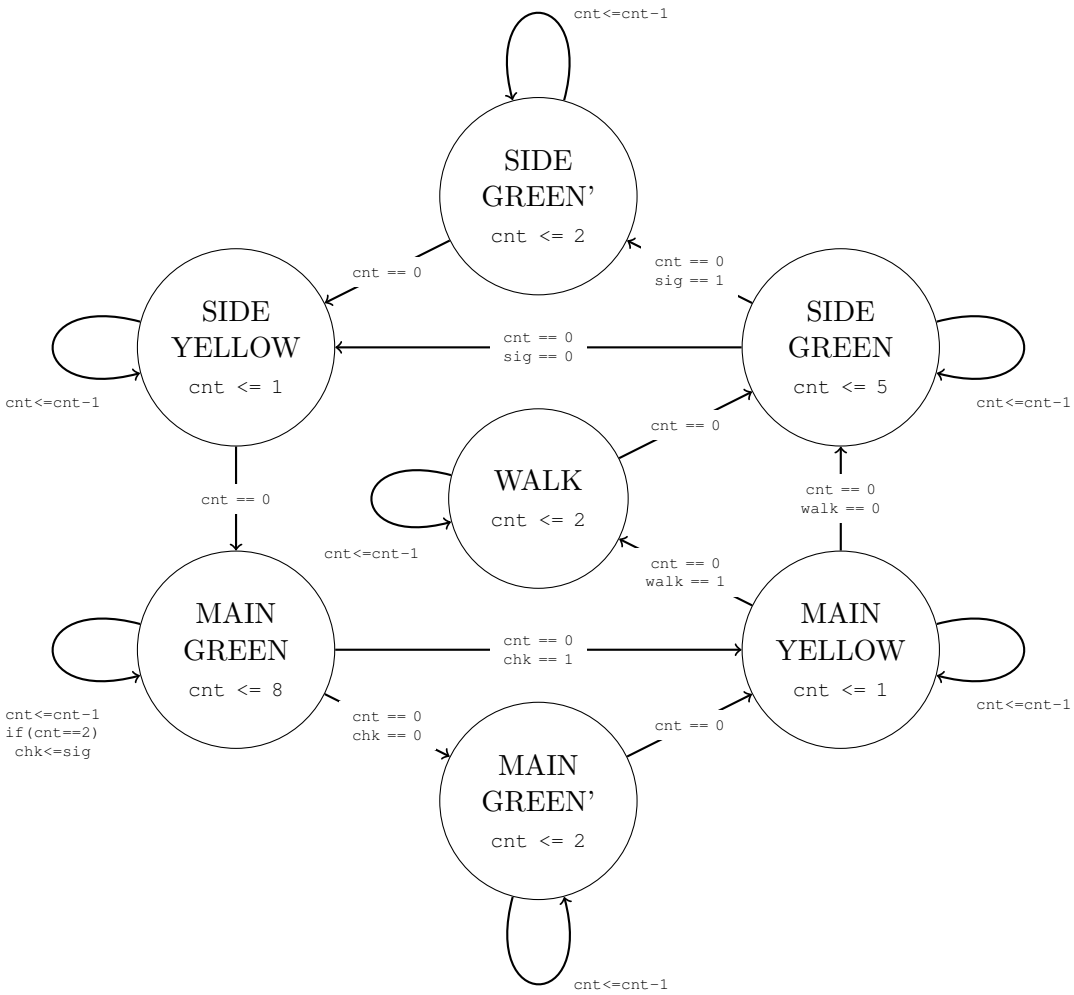
Our state machine breaks up each green light into two parts each. The first represents the minimum amount of time a green light can remain on and the second represents the extra time.

In general, we initialize a counter upon state transition with a value determined by the destination state. For each clock edge (speaking of a 1 Hz clock), we decrement the counter until it reaches zero, at which point we transition to the next state. The first green state for Main St is unique in that it must check the status of the traffic signal without changing states.

Below is an illustrated state diagram where each state has an explicit counter init value. We describe the variables in the picture as follows:

- `cnt`: Counter variable.

- `sig`: Side St traffic signal.
- `chk`: Checked value of traffic signal after Main St has been on for 6 seconds.
- `walk`: Independently updated walk register.



The Verilog code to implement the state machine was rather trivial, except for when the machine is in the MAIN GREEN state. Here we must check the traffic signal when the counter reaches 2–six seconds after the counter was initialized to 8.

```
case(state)
  MAIN_GREEN: begin
    if(second_cnt == 0) begin
      /* Cut the green light short */
      if (sensor_check == 1) begin
        state <= MAIN_YELLOW;
        second_cnt <= 1;
      /* Continue for 3 more seconds */
      end else begin
        state <= MAIN_GREEN_EXTRA;
        second_cnt <= 2;
      end
```

```
    /* Check the sensor after 6 seconds */
    end else if (second_cnt == 2) begin
      sensor_check <= side_sensor;
    end
  end
  ...
```

Since these states don't correspond to individual traffic light bulbs, we must implement that logic independently:

```
/* Main St lights (G,Y,R) */
assign led[0] = state == MAIN_GREEN || state == MAIN_GREEN_EXTRA;
assign led[1] = state == MAIN_YELLOW;
assign led[2] = state != MAIN_GREEN && state != MAIN_GREEN_EXTRA
  && state != MAIN_YELLOW;
/* Side St lights (G,Y,R) */
assign led[3] = state == SIDE_GREEN || state == SIDE_GREEN_EXTRA;
assign led[4] = state == SIDE_YELLOW;
assign led[5] = state != SIDE_GREEN && state != SIDE_GREEN_EXTRA
  && state != SIDE_YELLOW;
/* Walk signal */
assign led[6] = state == WALK;
```

3.2 Clock Divider

To implement our traffic light in hardware, we needed to generate two clock signals. A 1 Hz clock to run our state machine and a 32 Hz clock for the button debouncer. We implemented our clock divider in verilog using two counter registers. The first generated clock signal inverts after 50 million clock edges and the second inverts after 1,562,500 clock edges.

3.3 Button Debouncer

We wanted to ensure our walk button did not generate any noisy signals by filtering the raw button input with a debouncing circuit. Essentially we only check the raw input on a slow clock edge, generating a sampled version of the input signal. We implement this in the following Verilog code:

```
module debouncer(
  input button,
  input clk,
  output reg deb
);

always @(posedge clk) begin
  deb <= button;
end

endmodule
```

4 Testbech

5 Challenges & Solutions