Lab 2 Rotating Square Circuit

CSE 521 FPGA Design

Aaron Chamberlain, ID: 003392908

I Goals:

The following lab was designed to aid in following up the lessons learned in the first lab while also adding in new concepts. This lab helps to learn the basics behind finite state machines, asynchronous resets, and clock rate division. The visible output of this circuit is to see a square made from half of a seven segment display rotate either clock-wise or counter-clockwise depending on the direction chosen by the user.

II Design Process:

The desired output can be achieved by constructing a simple Moore finite state machine that determines both which display is enabled but also what data is display on the display. For example, in the clockwise direction, we start by lighting the top square of the left seven segment display, then moving this square to the second, the third, and forth. At this point, we simply switch the data to the bottom square and move in the opposite direction. Accordingly, there are only 8 states that we need to worry about, two (2) for each of the four (4) seven segment displays (see Figures 1&2).

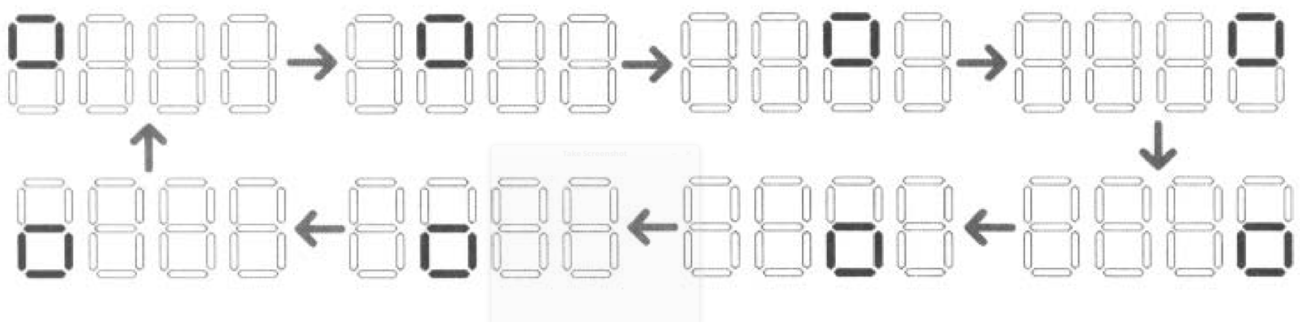

Figure 1. Example movement of the seven segment display for clockwise rotation.

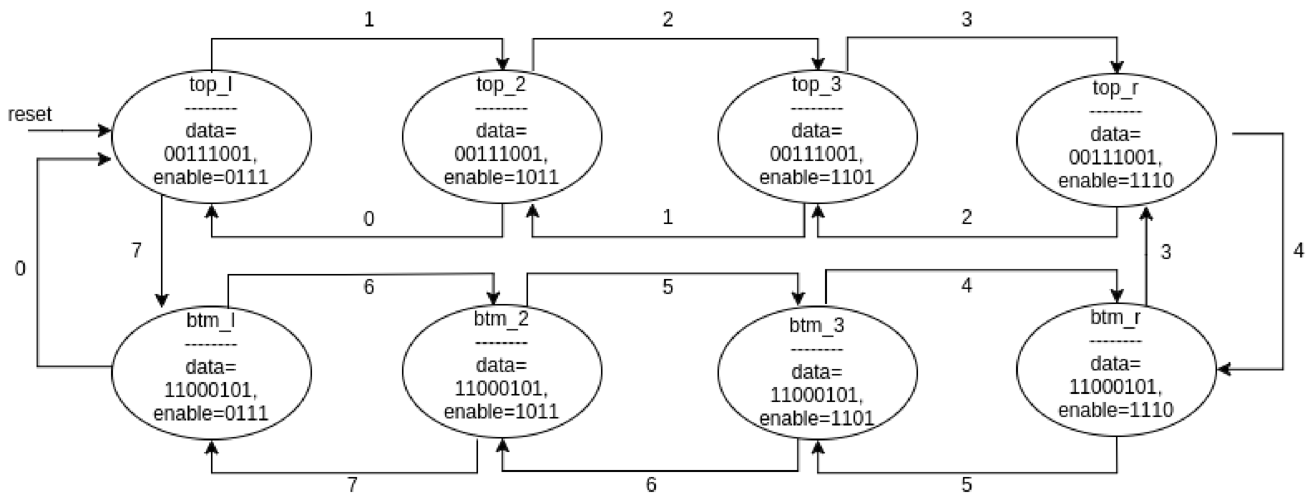

Figure 2. The circuit logic written in FSM Moore Machine format.
Note that each state has an input that will cause it to loop back on itself, but it is not shown in the diagram for simplicity.

With the FSM core out of the way, the only remaining item in the design is how to slow down the clock to make the result more visible to human eyes and how to implement the counter. The native clock rate of the FPGA board can be slowed down to a more suitable rate via a frequency divider. The general implementation of this is to count the number of positive edges of the on-board clock, at an arbitrary amount, create a pulse that goes high for one cycle. This effectively reduces the rate by a factor proportional to high the counter is set at. I.E. If we only count to 10, then the clock rate is divided by 10.

Last up, we simply need to implement a counter that will be the driver of the FSM. All it needs to is count up, which will result in a default direction of clockwise when no buttons are pressed. Otherwise, when the value of direction is changed via a button or switch, the counter counts down, resulting in a counter-clockwise movement. In both directions, the counter will go unto it reaches saturation (maximum or minimum values, then reset to the opposite). I.E. if we are counting down and reach zero (0), then we reset to seven (7). If we are counting up and reach seven (7), we reset to zero (0). Lastly, we want to add an enable signal to the counter, by setting enable to false, the counter will remain on the same value. Given that each state loops back on itself, this will essentially cause the animation to be frozen. By connecting these three simple circuit subcomponents, we can obtain the desired result. The overall circuit can be seen in Figure 3.

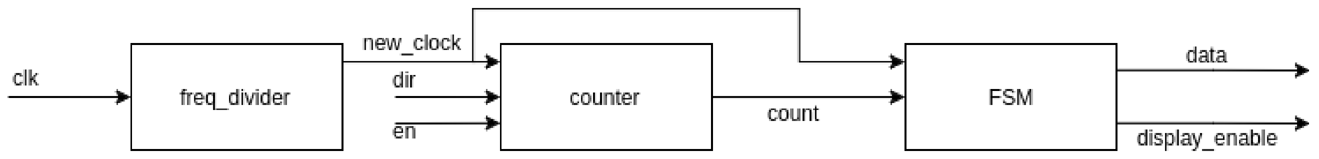

Figure 3. The overall Block-Level Design of the circuit.

III Program:

The source files follow the design outlined above to achieve re-usability and modularity. They follow similar naming schemes to those listed in the diagrams.

freq_divider:

```
module freq_divider(
   input wire clk,
          output reg new_clk
);

reg [24:0] count;

initial begin
         count = 0;
end

always@(posedge clk) begin
         if (count == 0) begin
      count <= 24999999;
      new_clk <= ~new_clk;
   end else begin
      count <= count - 1;
   end
```

```
end

endmodule
```

Counter:

```
module counter(
            input wire clk,
            input wire en,
            input wire dir, //0 is CW, 1 is counterCW
            output reg [2:0] count // 8 states
);

initial begin
            count = 0;
end

always @(posedge clk)
            if (en)
                        if (!dir)
                                    //increment until full saturation, then reset to zero
                                    count = (count==3'b111) ? 0:(count+1);
                        else if (dir)
                                    //decrement until no saturation, then reset to full saturation
                                    count = (count==3'b000) ? 3'b111:(count-1);

endmodule
```

Rotate_fsm:

```
module rotate_fsm(
    input wire clk,
             input wire [2:0] state,
             output reg [7:0] data,
             output reg [3:0] display_enable
);

always @(posedge clk)
            case(state)
                        3'b000 : begin
                                                data = 8'b00111001; // top box
                                                display_enable = 4'b0111;
                                    end
                        3'b001 : begin
                                                data = 8'b00111001; // top box
                                                display_enable = 4'b1011;
                                    end
                        3'b010 : begin
                                                data = 8'b00111001; // top box
                                                display_enable = 4'b1101;
                                    end
```

```
                3'b011 : begin
                        data = 8'b00111001; // top box
                        display_enable = 4'b1110;
                     end
                3'b100 : begin
                        data = 8'b11000101; // bottom box
                        display_enable = 4'b1110;
                     end
                3'b101 : begin
                        data = 8'b11000101; // bottom box
                        display_enable = 4'b1101;
                     end
                3'b110 : begin
                        data = 8'b11000101; // bottom box
                        display_enable = 4'b1011;
                     end
                3'b111 : begin
                        data = 8'b11000101; // bottom box
                        display_enable = 4'b0111;
                     end
        endcase

endmodule
```

rotateController (Top Module):

```
module sevenSegmentController(
        input wire clk, en, dir,
        output wire [3:0] display_enable,
        output wire [7:0] data
);

wire new_clock;
wire [2:0] state;

freq_divider freq1 (
        .clk (clk),
        .new_clk (new_clock)
);

counter count1 (
        .clk (new_clock),
        .en (en),
        .dir (dir),
        .count (state)
);

rotate_fsm fsm1(
        .clk (new_clock),
        .state (state),
        .data (data),
        .display_enable (display_enable)
);

endmodule
```

IV Results:

The design of the circuit as implemented was found to be correct, with functionality as described in the book. Given that the frequency divider and FSM were borrowed from other assignments where test benches were written and executed, the only module for which a test bench was written were the counter. It behaves as expected before programming the FPGA, and allowed me to continue working on another portion of the design implementation. The test bench results can be seen below in Figure 4.

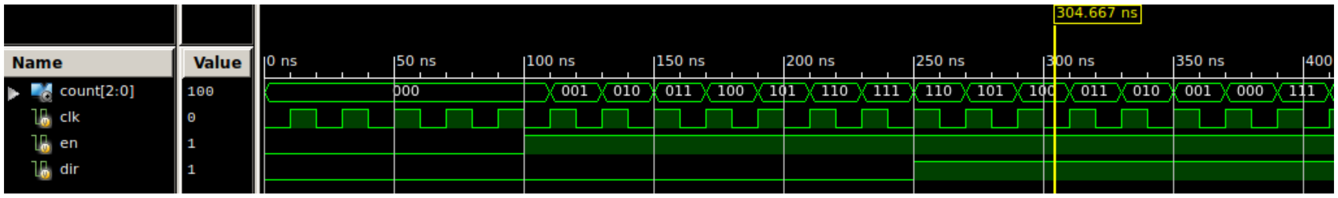

Figure 4. At #110, counts up to full saturation. At #250, dir flips and counts to zero.

Once all modules were tested via their respective test benches, the modules were wired together as per the design, and tied to the hardware. Only two switches are needed for the implementation, one to determine the enable signal, and another to determine the direction. The seven segment data pins and enable locations are also listed in the UCF file. After programming the board, the circuit was seen to be a little fast. A quick adjustment to the counter allowed the clock rate to be slowed down even further and the demo to run at the rate.

V Problem:

This lab served as a very practical way of combining the new logic that was being taught in the lectures. It did a good job at combining simple combinational logic like the counter next to a more complex state machine that implements memory and state control. The only two problems that were encountered in the lab was performing the match required to determine appropriate counter values so as to get the clock rate of the circuit to a desired speed. The other problem that was a quick and helpful reminder was determining which signals must be included in the UCF file.

VI What You Learned:

As mentioned in the previous section, one of the biggest problems I ran into in terms of time taken was simply determining behavior of the circuit if required items are not included in the UCF. I learned that in most case, the ISE software will simply infer required signals and how to route them. For example, I had never previously specified the clock signal pin in the UCF file, but given that there is only one crystal on the board, it linked it anyways. I also forgot to list the enable pins for the seven segment displays within the UCF file. This resulted in a much more unpredictable behavior because while some of the pins would light up, it was also bleed over into other areas of the bus and displays on LEDs and the like. Given this, I took the critical note to always check this hardware linking before debugging any internal Verilog code.

This lab also provided me with a more practical example of how to use a finite state machine. Previously, in our lecture it had always been an abstract problem that needed to be detailed. When this problem presented itself it was then easy to recognize that it could be easily implemented by a FSM mechanism, and that the design could thus be verified before any code was even written. Although it wasn’t that complicated, it important to note that each state has two conditions that allow for entering the node, two conditions that lead to exiting, and on that loops in place. The diagram helped to visualize these numerous changes that can occur and verify that all possibilities were considered in the circuit.