ECE 440 - Project #3

Collin Heist

28th February 2020

Contents

1 Design **1**
1.1 Wrapper Block Diagram 1
1.2 Wrapper Input Selection Finite State Machine 1
1.3 LED Encoder 2

2 Tribulations **3**

3 Simulations **4**
3.1 Behavioral Simulation 4
3.2 Post-Synthesis Timing Simulation 5

4 Source Code **6**

Figures

1.1 Input Finite State Machine 2
3.1 Behavioral Simulation. 4
3.2 Post-Synthesis Timing Simulation. 5

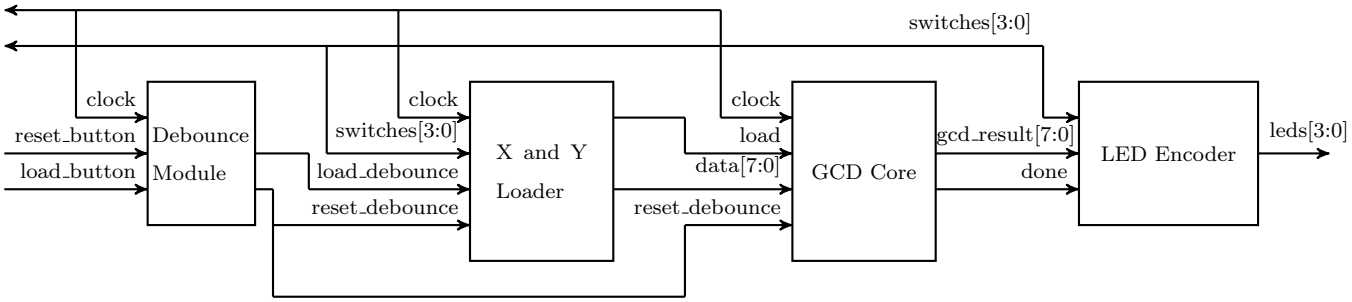
Listings

4.1 Wrapper Module 6
4.2 Testbench 8

1 Design

1.1 Wrapper Block Diagram

As part of **Homework #4**, and for the purpose of clearly outlining the necessary components for my wrapper, I designed the following block diagram for the overall wrapper module.



The debounce module was provided for us, and the logic for the **X and Y Loader** is outlined in **Section 1.2**. The GCD core is the unchanged code from **Project #2**, and the LED encoder is purely combinational and is described in **Section 1.3**.

1.2 Wrapper Input Selection Finite State Machine

I decided to implement the input logic using a very basic finite state machine. Sequentially selecting **X** and **Y** could not be done without sequential logic because the previous presses of **BTN1** must be remembered.

The FSM I implemented in **Listing 4.1** is shown below:

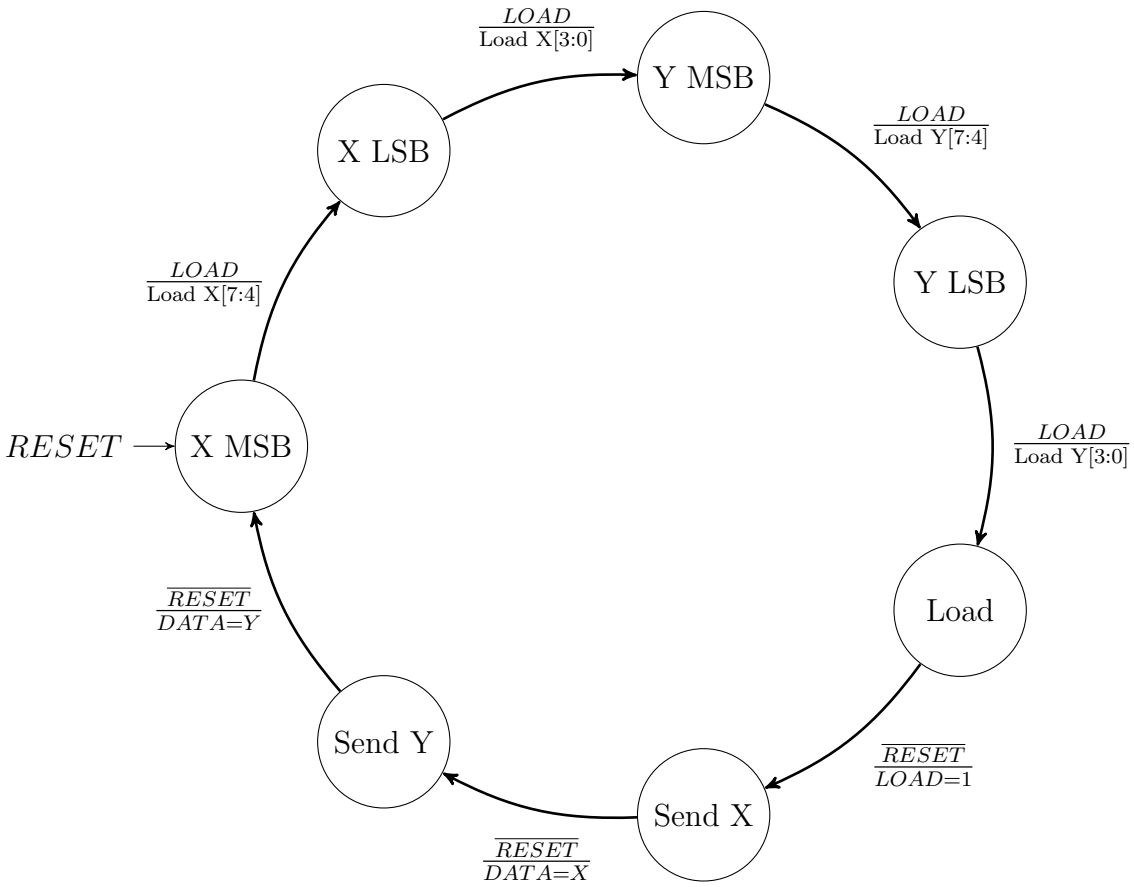


Figure 1.1: Input Finite State Machine

This FSM was implemented inside a sequential block, with the **LOAD** signal being used to begin many of the state transitions referencing the output of the button debouncer. This was necessary in order to avoid loading the registers with all of the same value (triggered by many debounces).

This state mechanism interacts with the **gcd_core** module, and the final three states are input-independent (excluding reset, obviously) that only exist to send the **Load**, **X**, and **Y** signals synchronized with the clock. Because of this, there is no need to send the debounced load signal directly into the GCD core module, as four load-button presses will automatically trigger the only required load into the module.

1.3 LED Encoder

As shown in **Figure 1.1**, the output logic is purely combinational, and is dependent upon the status of the switches and the constantly-asserted **Done** signal from the GCD Core. Because of this, the implementation for this section was very simple, and shown at the end of **Listing 4.1**, specifically lines 84-95.

2 Tribulations

My first implementation of the FSM in **Figure 1.1** had the debounced load being required for all state transitions, not just the first four *loading* states. This seemed necessary to me at first, as button presses were necessary for advancing the FSM at first, but upon simulating this with my testbench (see **Listing 4.2**), I noticed that the FSM never went past the **Send X** state. This of course makes sense, because the user is not required to press any more buttons after both X and Y are loaded, and so the transitions to send X and Y over the **Data** line never occurred. This was fixed by changing my FSM code to include a second if statement checking only if reset was not asserted, and if the state was already on the **Send Load** state, then transitions occurred on all clock edges.

The next problem I encountered occurred while I was testing my code on the hardware. I was not able to *randomly* generate suitable numbers that had a high-enough GCD to verify that toggling between seeing the MSB or LSB of the result was working. For my simulations I only tested (at that point) relatively small numbers with GCD's less than 2^4. To rectify this I wrote a small Python script to test all possible number combinations between 1 and 2^8 and output the ones with the highest GCD. This gave me the inputs of $x = 236$, and $y = 156$ with a GCD of 78 ($|0100|1110|$ in binary) - achievable in a relatively small amount of operations on the datapath. This allowed me to verify that my MSB and LSB toggle was working properly.

3 Simulations

3.1 Behavioral Simulation

After addressing those issues detailed in **Section 2**, my behavioral simulation went exactly as expected, and is shown in **Figure 3.1**.

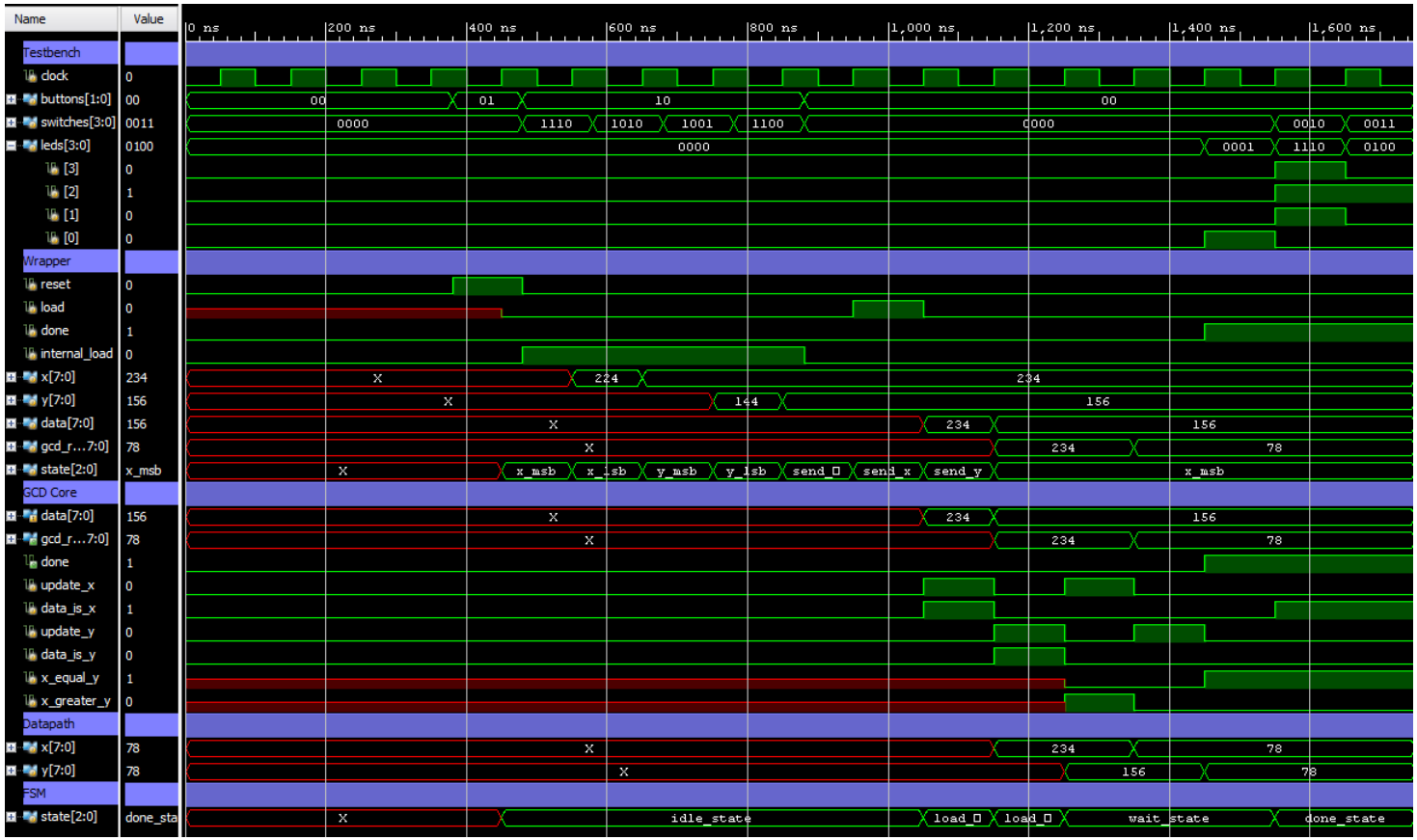


Figure 3.1: Behavioral Simulation.

3.2 Post-Synthesis Timing Simulation

With a lot of the information obfuscated, the post-synthesis simulation is a lot smaller, but shows the same results as the behavioral.

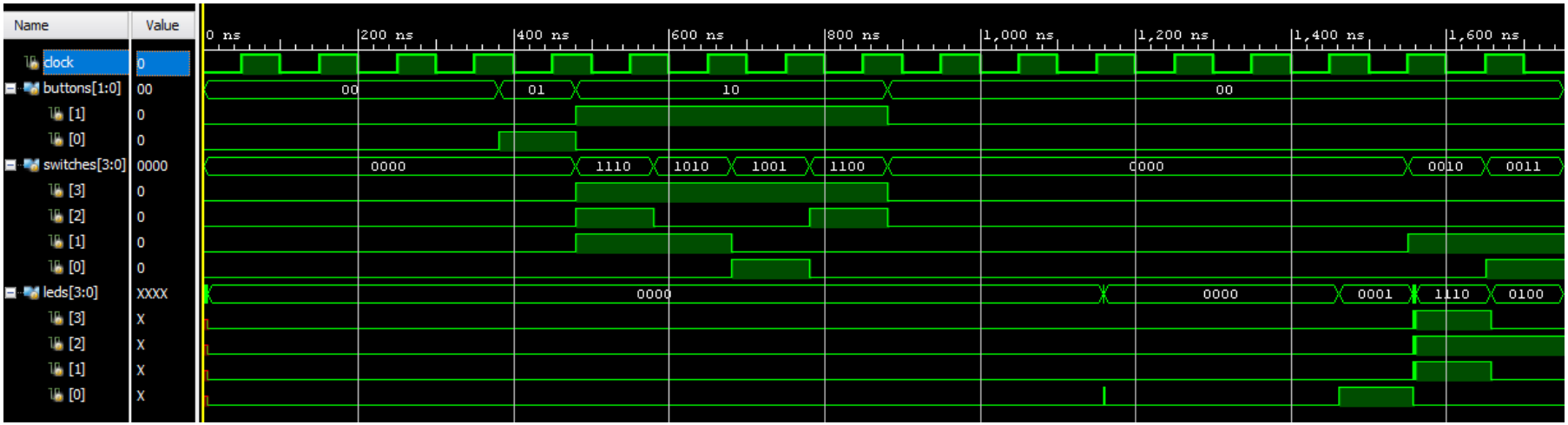


Figure 3.2: Post-Synthesis Timing Simulation.

This is as-expected, with the exception of the small *blip* on **leds[0]** for approximately 75 picoseconds. After verifying the logic was sound, and ensuring this behavior was not present when implemented on the board, I determined this was due to some combinational logic delays, and was small-enough to be ignored.

4 Source Code

Listing 4.1: Wrapper Module

```
1  `timescale 1ns / 1ps
2
3  module wrapper(
4    input logic clock,
5    input logic [1:0] buttons,
6    input logic [3:0] switches,
7    output logic [3:0] leds
8  );
9
10 // Internal signals
11 logic reset, load, done;
12 logic internal_load, reset_debounce, load_debounce;
13 logic [7:0] x, y, data, gcd_result;
14
15 // Synthesis ONLY
16 assign reset = reset_debounce;
17 assign internal_load = load_debounce;
18
19 // Simulation ONLY
20 //assign reset = buttons[0];
21 //assign internal_load = buttons[1];
22
23 // Instantiate our modules
24 debounce debounce_inst(
25   .clock(clock),
26   .reset_button(buttons[0]),
27   .load_button(buttons[1]),
28   .reset_debounce(reset_debounce),
29   .load_debounce(load_debounce)
30 );
31
32 gcd_core gcd_core_inst(.*);
33
34 // Implementation for the Input Loading
35 typedef enum logic [2:0] {x_msb, x_lsb, y_msb, y_lsb,
     send_load, send_x, send_y} statetype;
36 statetype state;
37
38 always_ff @(posedge clock) begin
39   if (reset) begin  // On debounced-resets go to x_msb
       state
40     state <= x_msb;
41     load = 0;
```

```
42      end
43    else if (internal_load) begin // When load is asserted,
        update x, y registers
44      case (state)
45        x_msb: begin
46          state <= x_lsb;
47          x <= {switches, 4'b0000};
48          end
49        x_lsb: begin
50          state <= y_msb;
51          x <= {x[7:4], switches};
52          end
53        y_msb: begin
54          state <= y_lsb;
55          y <= {switches, 4'b0000};
56          end
57        y_lsb: begin
58          state <= send_load;
59          y <= {y[7:4], switches};
60          end
61          endcase
62          end
63      // So long as reset isn't being asserted, advance FSM
          if beyond send_load state
64    if (~reset) begin
65       case (state)
66           send_load: begin
67               state <= send_x;
68                 load <= 1;
69                 end
70             send_x: begin
71                 state <= send_y;
72                 load <= 0;
73                 data <= x;
74                 end
75             send_y: begin
76                 state <= x_msb;
77                 data <= y;
78                 end
79         endcase
80    end
81  end
82
83  // Implementation of the output logic
84  always_comb begin
85    leds = 4'b0000;
86    if (done) begin
```

```
87      if (~switches[1])
88        leds = 4'b0001;
89      else
90        if (~switches[0])
91          leds = gcd_result[3:0];
92        else
93          leds = gcd_result[7:4];
94    end
95  end
96
97  endmodule
```

Listing 4.2: Testbench

```
1  `timescale 1ns / 1ps
2
3  module testbench();
4
5  // Global Parameters
6  parameter CLK_PRD = 100;
7  parameter HOLD_TIME = (CLK_PRD * 0.3);
8  parameter MAX_SIM_TIME = (100 * CLK_PRD);
9
10 // Internal logic signals
11 logic clock;
12 logic [1:0] buttons;
13 logic [3:0] switches, leds;
14
15 // Instantiate the GCD core as a UUT
16 wrapper dut(.*);
17
18 // Prevent simulating longer than MAX_SIM_TIME
19 initial #(MAX_SIM_TIME) $finish;
20
21 // Generate Clock Signal
22 initial begin
23   clock <= 0;
24   forever #(CLK_PRD / 2) clock = ~clock;
25 end
26
27 // Main Simulation
28 initial begin
29   buttons = 2'b00; switches = 4'b0000;
30
31   // Global Reset
32   #100;
33
```

```
34    @(posedge clock); #HOLD_TIME;  // Align with clock
35
36    repeat(2) #CLK_PRD;
37    buttons[0] = 1; #CLK_PRD; buttons[0] = 0;
38
39    // Stimulate the dut
40    // Load X
41    switches = 4'b1110; buttons = 2'b10; #CLK_PRD;  // X-MSB
42    switches = 4'b1010; buttons = 2'b10; #CLK_PRD;  // X-LSB
43    // Load Y
44    switches = 4'b1001; buttons = 2'b10; #CLK_PRD;  // Y-MSB
45    switches = 4'b1100; buttons = 2'b10; #CLK_PRD;  // Y-LSB
46    switches = 4'b0000; buttons = 2'b00; #CLK_PRD;
47
48    forever begin
49      @(posedge clock);
50      if (leds) begin
51        $display("Done asserted, LEDS=%b\nLooking at result."
              , leds);
52        switches = 4'b0010; #CLK_PRD; // Look at LSB of GCD
              result
53        $display("Looking at LSB of result, LEDS=%b", leds);
54        switches = 4'b0011; #CLK_PRD; // Look at MSB of GCD
              Result
55        $display("Looking at MSB of result, LEDS=%b", leds);
56        $finish;
57      end
58    end
59
60    $finish;
61  end
62
63  endmodule
```