ECE 440 - Project #5

Collin Heist

29th February 2020

Contents

Figures

Listings

1 Design

1.1 Top-Level Module

To start the design process, I created my top-level *memory reading* module. This module interacts with all pieces of the design (except the debounce circuitry); including the block memory, GCD calculator, and the SPI module. The FSM I ended up implementing was the following:

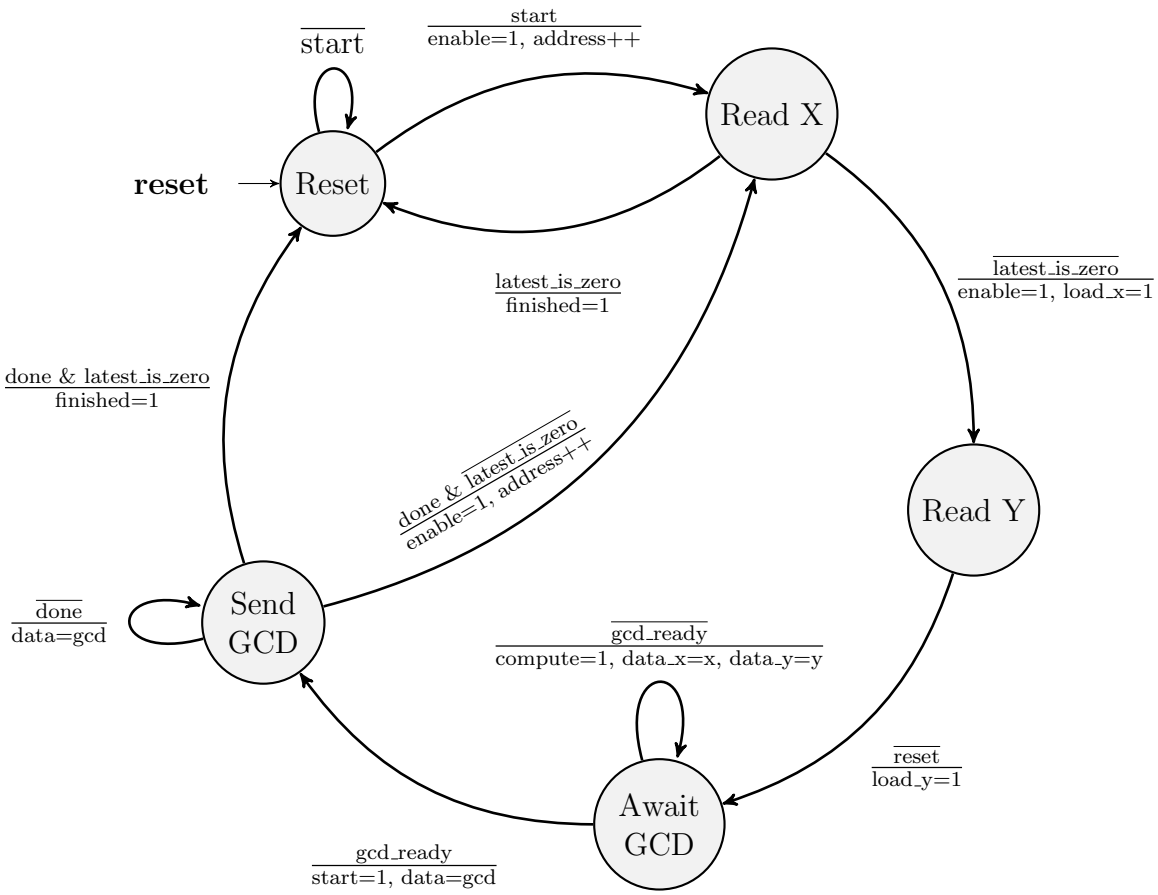

Figure 1.1: Top-Level Finite State Machine

The most important part of this sequence of operations is that the **load_x** and **load_y** signals are asserted one clock cycle after their respective states. This is required because the block memory unit has a one clock cycle delay on all reading and writing operations. Beyond that, the two most important states are the **Await GCD** and **Send GCD** states. At each of these, the FSM waits indefinitely until the respective *done* signals are asserted. When these are asserted is defined by the FSM's shown in **Figure 1.2** and **Figure 1.3**.

The read, calculate, and send sequence is repeated for each pair of X and Y data read from the memory until the last read from the memory unit is evaluated to be zero. At

this point the FSM terminates and this top-level *done* signal (called **finished**) is asserted.

The code for this module is shown in **Listing 4.1**.

1.2 GCD Calculator

I decided to approach my GCD interaction and calculation much like the GCD wrapper from **Project #3**. Rather than *bloating* the state machine of my top-level module, I decided to create a GCD Calculator module (code shown in **Listing 4.2**) which takes both X and Y as parallel inputs, and when **compute** is asserted, will sequence the GCD core module. This module's FSM is shown below.

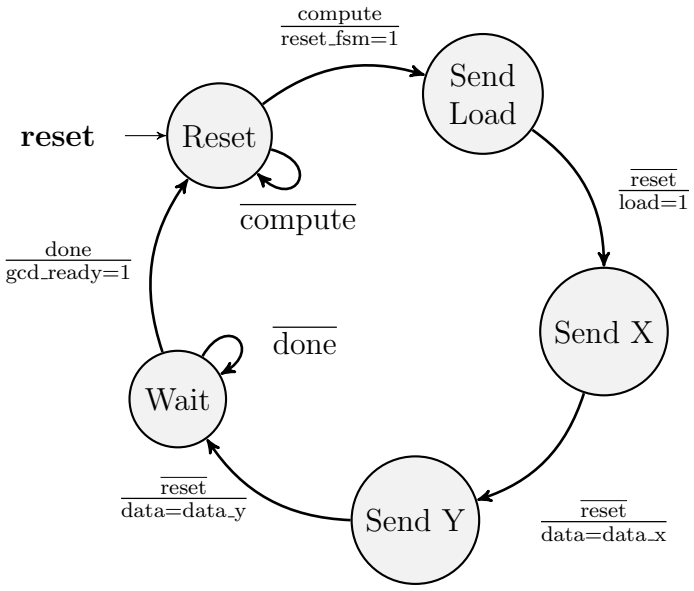

Figure 1.2: GCD Calculation Finite State Machine

This is nearly identical to the previous project's sequencing, so not a lot of design thought had to go into this. This module blocks until the GCD Core asserts the done signal, at which point the **gcd_result** signal is routed into the previously shown top-level module's **data** register. In order to reuse the GCD Core over and over, at the transition between the **Reset** state and the **Send Load** state the Core is reset – however, this is unnecessary.

1.3 SPI Module

The majority of my design process was spent on creating the SPI module. As per my top-level module, I wanted the SPI FSM to be triggered be the internal signal **start**, and will communicate back to the reading module with the assertion of **done**.

By the end of my design, the most difficult component to implement was the SPI clock generation; which I ended up separating from my control FSM completely. I discuss my problems further in **Section 2**.

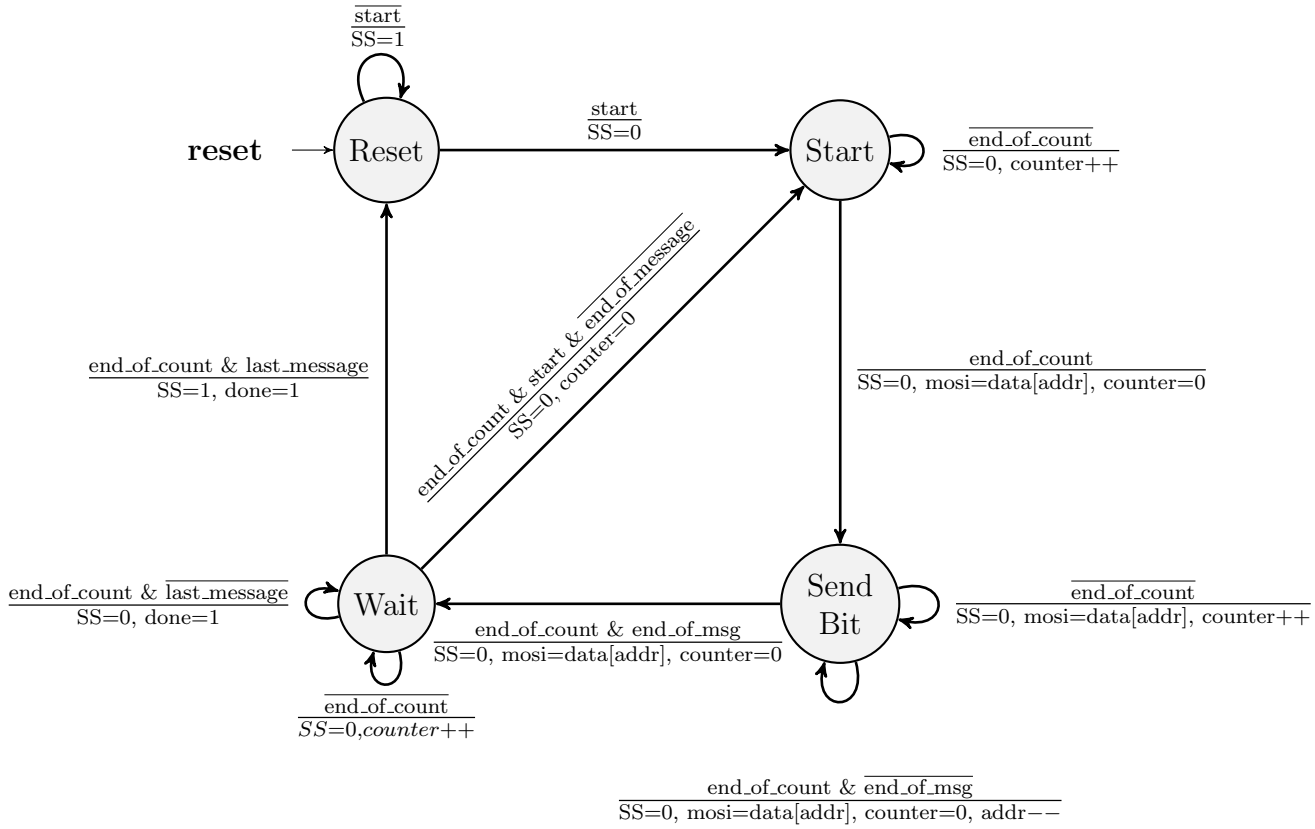

Figure 1.3: SPI Communication Finite State Machine

Despite being a very messy finite state machine, this is not as complicated as it might seem. Each state (excluding **Reset**) has at least one looping condition that accounts for the variable clock-slowing of the SPI bus. This is labeled as $\overline{\textbf{end\_of\_count}}$, and simply increments the counter for each clock cycle where **end_of_count** is not true. This functionally means that for each clock cycle that the counter is less than one-less than the desired clock delay (passed in as a parameter **clock_scale**), nothing will change on the SPI bus – *slowing* down the communication.

At each instance of the counter reaching the entered number, the counter is reset, the address variable being used to keep track of *where* in the 8-bit data line we are currently sending out over **MISO** is decremented, and then this is repeated. Once all 8 bits have been sent, the **wait** state is entered, and this state waits until start or the control signal **last_message** is asserted. This is done to keep the slave select line *low* so long as there is more data to send.

2 Tribulations

2.1 Problems

In hindsight, I should have tested each component individually. However, I ended up creating one large testbench that tested the entire integrated module. Luckily, my memory reader module and my GCD calculation module both worked as expected. The only change required between these two modules was that I did not initially account for the

one-clock delay of the results from the block memory unit – resulting in me attempting to load X and Y one cycle earlier than they were available. This was easily rectified by shifting the **load_x** and **load_y** back one state.

By far my biggest problems came from the SPI module. I began by creating the slowed clock using a counter, however I then had a much simplified FSM that was triggered by my created clock. Whether this was problematic in itself, or I simply had other fundamental errors, but I was finding that all my simulations worked as expected, except when implementing I would receive various timing errors that I could not resolve. After attempting to debug this for [what felt like] hours I ended up redoing my design completely – deciding instead to have everything triggered by the global clock, and have the counter being utilized as a control signal.

Clock implementation ended up being much easier in this new method, but took some trial and error. I ended up implementing it as:

Listing 2.1: SPI Clock Generation

```
1  assign spi_clock = ~((counter > clock_scale / 2 - 1) |
       slave_select) & (state == send_bit);
```

This is a combinational assignment that sets the SPI clock high for the first half of the count (done so that the falling edge falls in the middle of the data output as opposed to the end), is only pulsing if **slave_select** is down (we do not want the clock active while not *selecting* a device), and only during the **send_bit** state so that the clock isn't active for the required padded time at the start and end of each transmission.

The last thing I struggled with was generating the correct SPI clock frequency. I didn't realize the simulation's timescale was not representative of the board's actual frequency – leading t me incorrectly configuring the SPI clock counter. Luckily, I choose to create the SPI module as a parameterized module so I had to simply change the parameter being used for the counter to 626.

2.2 Tested Cases

To validate my design, I tested different amounts of data within my block memory, between 1 and 7 pairs of data to verify the data-end verification worked as expected. I also looked at the output of my SPI communication in both the Protocol and the Logic window of Waveforms, in the cases where I spammed, held and then released, and just pressed and released the reset button. In each instance, my design behaved as expected.

3 Simulations

3.1 Behavioral Simulation

To get more clear simulations, my behavioral simulations were done with only a $\frac{1}{2}$ scaled SPI clock. All internal signals are shown in the two figures below, where I've zoomed in on just a single GCD transmission.

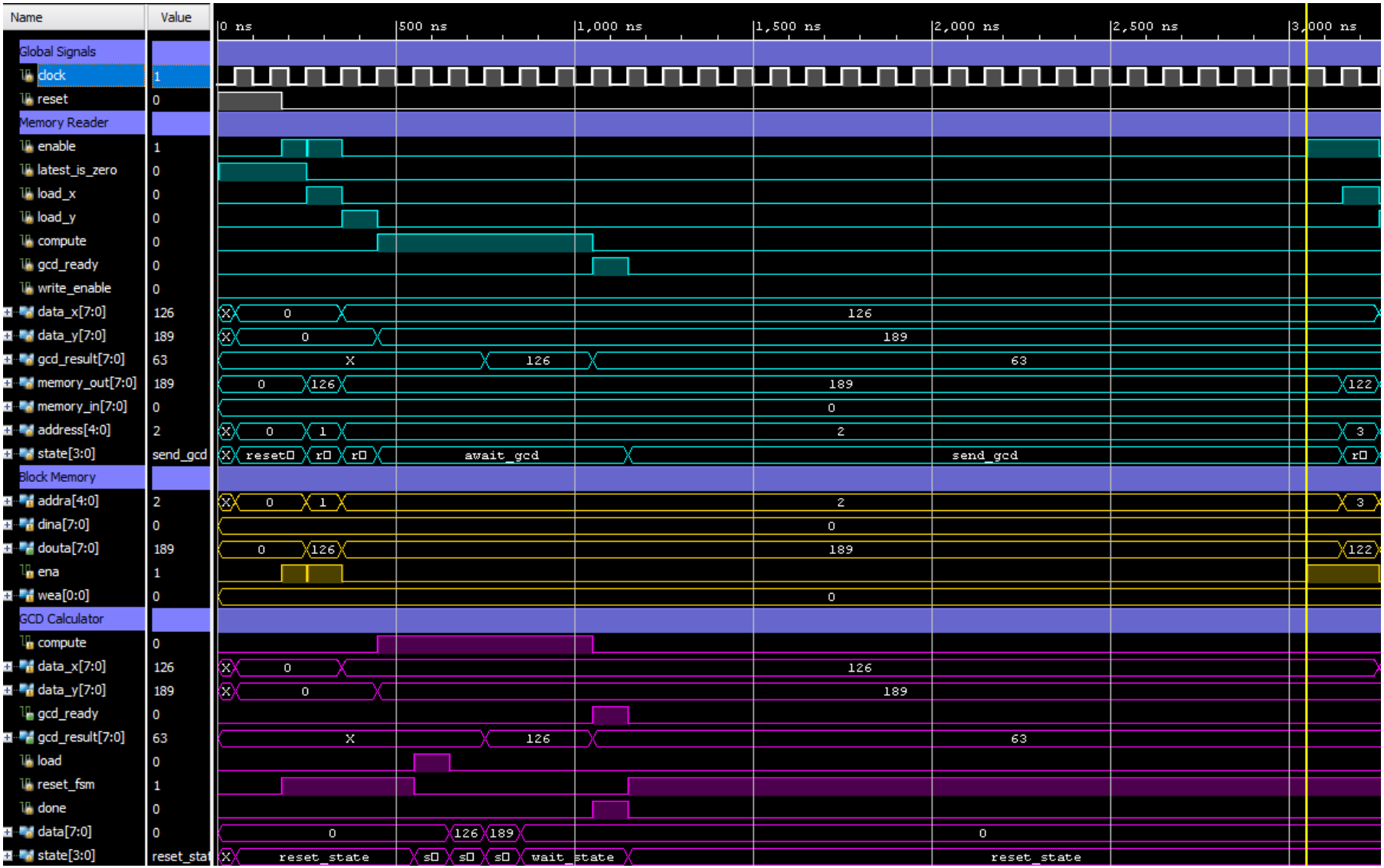

Figure 3.1: SPI Signals in Behavioral Simulation.

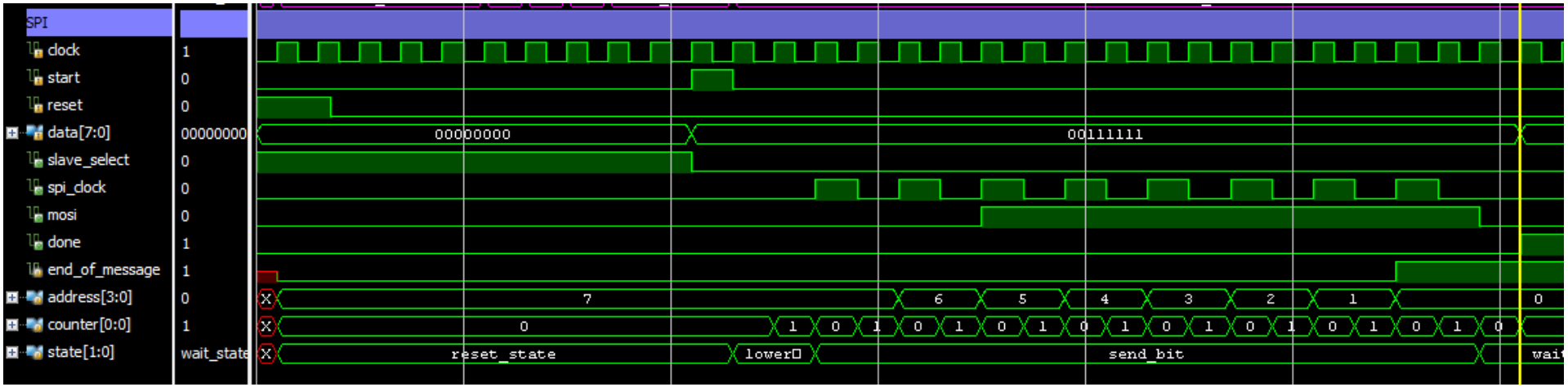

Figure 3.2: Memory Reader, Block Memory, and GCD Calculator Signals in Behavioral Simulation.

3.2 Post-Synthesis Timing Simulation

At the same clock scaling, the Post-Synthesis timing simulation shows all 6 GCD transmissions (for a preloaded block memory with 12 values).

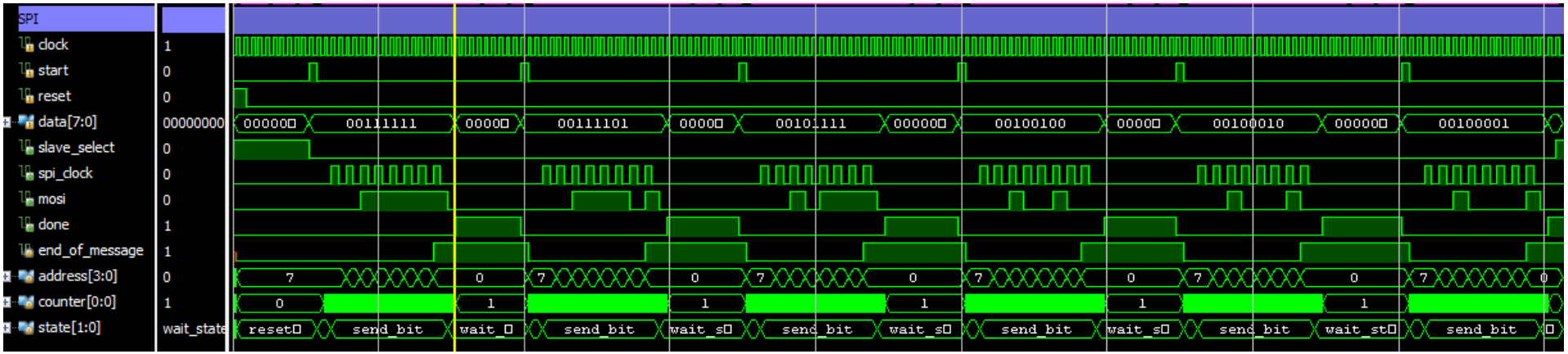

Figure 3.3: Post-Synthesis Timing Simulation with 6 transmissions.

3.3 Post-Implementation Timing Simulation

The post-implementation timing simulation was done with the correct (200kHz) SPI clock. One thing to note is that the simulation appears to have *blips* (on almost all signals) at transitions; however when zoomed in upon, these are not actually present.

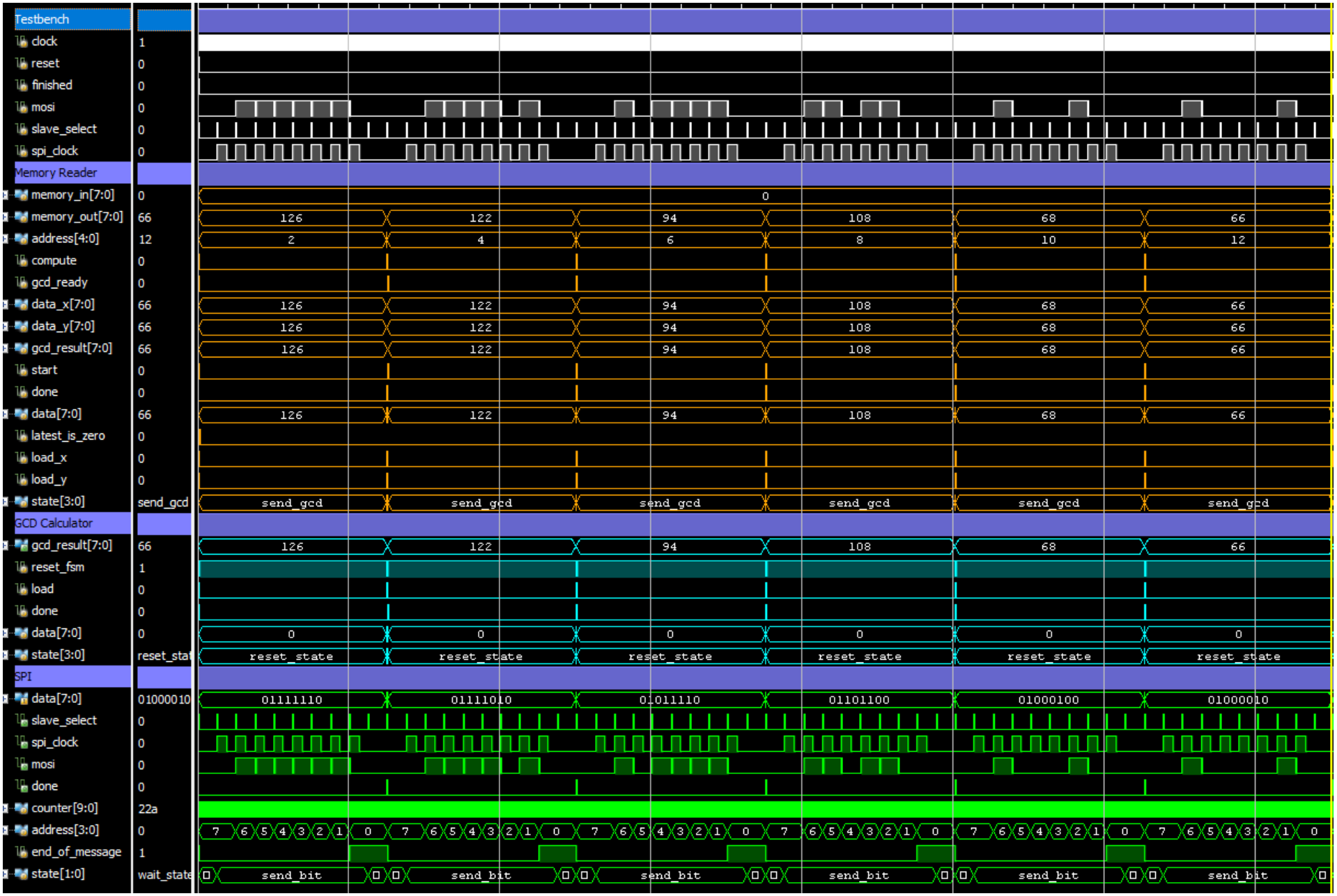

Figure 3.4: Post-Implementation Timing Simulation.

3.4 WaveForms

To verify the output of my SPI communications, I hooked up the appropriate pins on the Zybo board to the Analog Discovery, and these are the results:

```
Data: 63 | 0, 61 | 0, 47 | 0, 36 | 0, 34 | 0, 33 | 0,
Data: 63 | 0, 61 | 0, 47 | 0, 36 | 0, 34 | 0, 33 | 0,
Data: 63 | 0, 61 | 0, 47 | 0, 36 | 0, 34 | 0, 33 | 0,
Data: 63 | 0, 61 | 0, 47 | 0, 36 | 0, 34 | 0, 33 | 0,
Data: 63 | 0, 61 | 0, 47 | 0, 36 | 0, 34 | 0, 33 | 0,
Data: 63 | 0, 61 | 0, 47 | 0, 36 | 0, 34 | 0, 33 | 0,
```

Figure 3.5: WaveForms *Protocol* view of the SPI output.

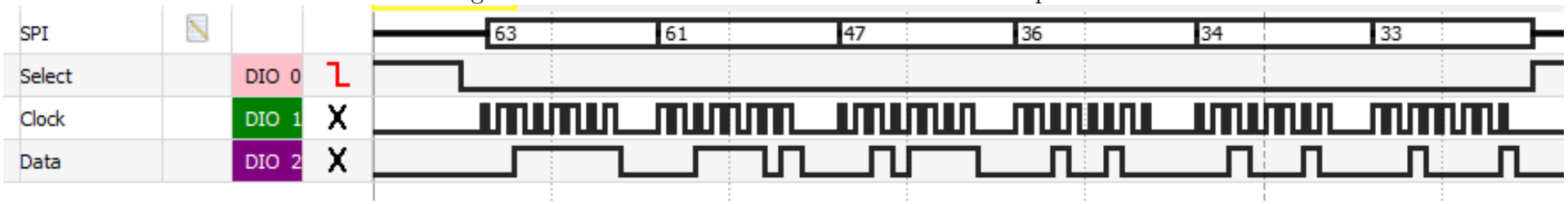

Figure 3.6: WaveForms *Signal* view of the SPI output.

In both views you can see the same data is transmitted in all instances (which is what I expected). One thing to note is that the Protocol window shows zeros between data transmissions because I believe the MISO is transmitting zeros in response – so these can be ignored.

4 Source Code

Listing 4.1: Memory Reader Module

```
1  `timescale 1ns / 1ps
2
3  module memory_reader(clock, reset, finished, mosi,
       slave_select, spi_clock);
4    input logic clock, reset;
5    output logic finished, mosi, slave_select, spi_clock;
6
7  // Instantiate the block memory unit
8  logic enable, write_enable;
9  logic [7:0] memory_in, memory_out;
10 logic [4:0] address;
11 assign memory_in = 0;    // No writing to the memory
       required
12 assign write_enable = 0;  // Never writing to block memory
13 blk_mem_gen_0 block_memory(
14   .clka(clock),
15   .ena(enable),
16   .wea(write_enable),
17   .addra(address),
18   .dina(memory_in),
19   .douta(memory_out)
20 );
21
22 // Instantiate the GCD Calculation unit
23 logic compute, gcd_ready;
24 logic [7:0] data_x, data_y, gcd_result;
25 gcd_calculator gcd_calculator_instance(.*);
26
27 // Instantiate the SPI Communications unit
28 logic start, done, last_message;
29 logic [7:0] data;
30 assign last_message = finished;
31 spi #(626, 10) spi_instance(.*);  // 626x slower, this
       requires 10-bits to count
32
33 // Internal signals
34 logic latest_is_zero, load_x, load_y;
35 assign latest_is_zero = (memory_out == 0);
36
37 // FSM States
38 typedef enum logic [3:0] {reset_state, read_x, read_y,
       await_gcd, send_gcd} statetype;
39 statetype state;
```

```
40
41 // Button release-checking
42 logic button_prev;
43 always_ff @(posedge clock)
44     button_prev <= reset;
45
46 // FSM Advancement Implementation
47 always_ff @(posedge clock) begin : fsm_advancement
48   if (reset) begin
49     state <= reset_state;
50     address <= 0;
51     end
52   else begin
53     case (state)
54       reset_state: begin
55         state <= ((button_prev & ~reset) ? read_x :
              reset_state); // Advanced states on button
              releases
56         address <= ((button_prev & ~reset) ? address + 1 :
              address);
57         end
58       read_x:
59         unique case (latest_is_zero)
60           1'b1: begin state <= reset_state; address <= 0;
                  end
61           1'b0: begin state <= read_y; address <= address +
                  1;  end
62         endcase
63       read_y:   state <= await_gcd;
64       await_gcd:  state <= (gcd_ready ? send_gcd :
           await_gcd);
65       send_gcd:
66         unique casez ({done, latest_is_zero})
67           2'b0?: begin state <= send_gcd;              end
68           2'b10: begin state <= read_x; address <= address
                + 1; end
69           2'b11: begin state <= reset_state;            end
70         endcase
71     endcase
72     end
73 end : fsm_advancement
74
75 // FSM Outputs
76 always_comb begin : fsm_outputs
77   enable = 0; load_x = 0; load_y = 0; compute = 0; start =
       0; finished = 0; data = 0;
78   if (~reset) begin
```

```
79      case (state)
80        reset_state:
81          enable = (button_prev ? 1 : 0);
82        read_x:
83          unique case (latest_is_zero)
84            1'b0: begin enable = 1; load_x = 1; end
85            1'b1: begin finished = 1;     end
86          endcase
87        read_y:
88          load_y = 1;
89        await_gcd:
90          unique case (gcd_ready)
91            1'b0: begin compute = 1;          end
92            1'b1: begin start = 1; data = gcd_result; end
93          endcase
94        send_gcd:
95          unique casez ({done, latest_is_zero})
96            2'b0?: data = gcd_result;
97            2'b10: enable = 1;
98            2'b11: finished = 1;
99          endcase
100     endcase
101   end
102 end : fsm_outputs
103
104 // Syncronous loading of the data_x, and data_y register
105 always_ff @(posedge clock) begin
106   if (reset) begin
107     data_x <= 0;
108     data_y <= 0;
109     end
110   else begin
111     if (load_x)   data_x <= memory_out;
112     if (load_y)   data_y <= memory_out;
113     end
114 end
115
116 endmodule : memory_reader
```

Listing 4.2: GCD Calculator Module

```
1 `timescale 1ns / 1ps
2
3 // This module begins stimulating the gcd_core module when
     compute
4 // is asserted, and sends the value of data_x and data_y to
     the
```

```
5  // module. Then, gcd_ready and the proper gcd_result are
       asserted.
6  module gcd_calculator(clock, reset, compute, data_x, data_y
       , gcd_ready, gcd_result);
7    input logic clock, reset, compute;
8    input logic [7:0] data_x, data_y;
9    output logic gcd_ready;
10   output logic [7:0] gcd_result;
11
12 // Instantiate the GCD Core Module
13 logic reset_fsm, load, done;
14 logic [7:0] data;
15 gcd_core gcd_core_instance(
16   .clock(clock),
17   .reset(reset_fsm),
18   .load(load),
19   .data(data),
20   .gcd_result(gcd_result),
21   .done(done)
22 );
23
24 // FSM Implementation
25 typedef enum logic [3:0] {reset_state, send_load, send_x,
       send_y, wait_state} statetype;
26 statetype state;
27
28 always_ff @(posedge clock) begin : fsm_advancement
29   if (reset) begin
30     state <= reset_state;
31     end
32   else begin
33     case (state)
34       reset_state:  state <= (compute ? send_load :
           reset_state);
35       send_load:    state <= send_x;
36       send_x:     state <= send_y;
37       send_y:     state <= wait_state;
38       wait_state:   state <= (done ? reset_state :
           wait_state);
39     endcase
40     end
41 end : fsm_advancement
42
43 // FSM Output Implementation
44 always_comb begin : fsm_output_comb
45   load = 0; data = 0; gcd_ready = 0; reset_fsm = 0;
46   if (~reset) begin
```

```
47      case (state)
48        reset_state:  reset_fsm = 1;
49        send_load:    load = 1;
50        send_x:     data = data_x;
51        send_y:     data = data_y;
52        wait_state:   gcd_ready = (done == 1 ? 1 : 0);
53      endcase
54      end
55  end : fsm_output_comb
56
57  endmodule : gcd_calculator
```

Listing 4.3: SPI Protocol Module

```
1  `timescale 1ns / 1ps
2
3  module spi
4    #(parameter clock_scale = 4, counter_bits = 3)
5    (clock, start, reset, data, last_message, slave_select,
       spi_clock, mosi, done);
6
7    input logic clock, start, reset, last_message;
8    input logic [7:0] data;
9    output logic slave_select, spi_clock, mosi, done;
10
11 // Internal Signals needed for the clock scaling (slowing)
12 // Clock should be on only when sending bits, if slave
       select is low, and in the last half of the count
13 logic [counter_bits-1:0] counter;
14 assign spi_clock = ~((counter > clock_scale / 2) |
       slave_select) & (state == send_bit);
15
16 // SPI Addressing
17 logic [3:0] address;
18 logic end_of_message;
19 assign end_of_message = (address == 0);
20
21 // Whether the clock-slowing is done or not
22 logic end_of_count;
23 assign end_of_count = (counter == (clock_scale - 1));
24
25 // FSM States
26 typedef enum logic [1:0] {reset_state, lower_ss, send_bit,
       wait_state} statetype;
27 statetype state;
28
29 // FSM Advancement
```

```
30 always_ff @(posedge clock) begin : fsm
31   if (reset) begin
32     state <= reset_state;
33     counter <= 0;
34     address <= 7;
35     end
36   else begin
37     case (state)
38       reset_state:
39         state <= (start ? lower_ss : reset_state);
40       lower_ss:
41         unique case (end_of_count)
42           1'b0: begin state <= lower_ss; counter <= counter
                 + 1; end
43           1'b1: begin state <= send_bit; counter <= 0;
                  end
44         endcase
45       send_bit:
46         unique casez ({end_of_count, end_of_message})
47           2'b0?: begin state <= send_bit; counter <=
                counter + 1;          end
48           2'b10: begin state <= send_bit; counter <= 0;
                address <= address - 1; end
49           2'b11: begin state <= wait_state; counter <= 0;
                           end
50         endcase
51       wait_state:
52         unique casez ({end_of_count, last_message, start})
53           3'b0??: begin state <= wait_state; counter <=
                counter + 1;     end
54           3'b100: begin state <= wait_state;
                  end
55           3'b101: begin state <= lower_ss; counter <= 0;
                address <= 7;  end
56           3'b11?: begin state <= reset_state; counter <= 0;
                 address <= 7; end
57         endcase
58     endcase
59     end
60 end : fsm
61
62 // FSM Outputs
63 always_comb begin : fsm_outputs
64   slave_select = 1; mosi = 0; done = 0;
65   if (~reset) begin
66     case (state)
67       reset_state:
```

```
68          slave_select = (start ? 0 : 1);
69        lower_ss: begin
70          slave_select = 0;
71          if (end_of_count)
72            mosi = data[address];
73          end
74        send_bit: begin
75          mosi = data[address];
76          slave_select = 0;
77          end
78        wait_state:
79          unique casez ({end_of_count, last_message, start})
80            3'b0??: begin slave_select = 0;       end
81            3'b100: begin slave_select = 0; done = 1; end
82            3'b101: begin slave_select = 0;       end
83            3'b11?: begin slave_select = 1; done = 1; end
84          endcase
85      endcase
86      end
87  end : fsm_outputs
88
89  endmodule : spi
```

Listing 4.4: Hardware Wrapper

```
1  `timescale 1ns / 1ps
2
3  module hardware_wrapper(clock, reset_button, slave_select,
       spi_clock, mosi);
4    input logic clock, reset_button;
5    output logic slave_select, spi_clock, mosi;
6
7  // Intantiate the debounce module
8  logic reset_debounce;
9  debounce debounce_instance(.*);
10
11 // Instantiate the memory_reader module
12 logic reset, miso, finished;
13 assign reset = reset_debounce;
14 memory_reader memory_reader_instance(.*);
15
16 endmodule : hardware_wrapper
```