ECE 440 - Project #8

Collin Heist

29th March 2020

Contents

Figures

Listings

1 Design

1.1 Codon Reader

I chose to implement this project with two relatively complicated modules (although in hindsight, I should have partitioned my project more). The first one being the codon-reading module, that is responsible for reading the codon memory and storing the parsed codons for later operations. The FSM I designed for this module is shown in **Figure 1.1**.

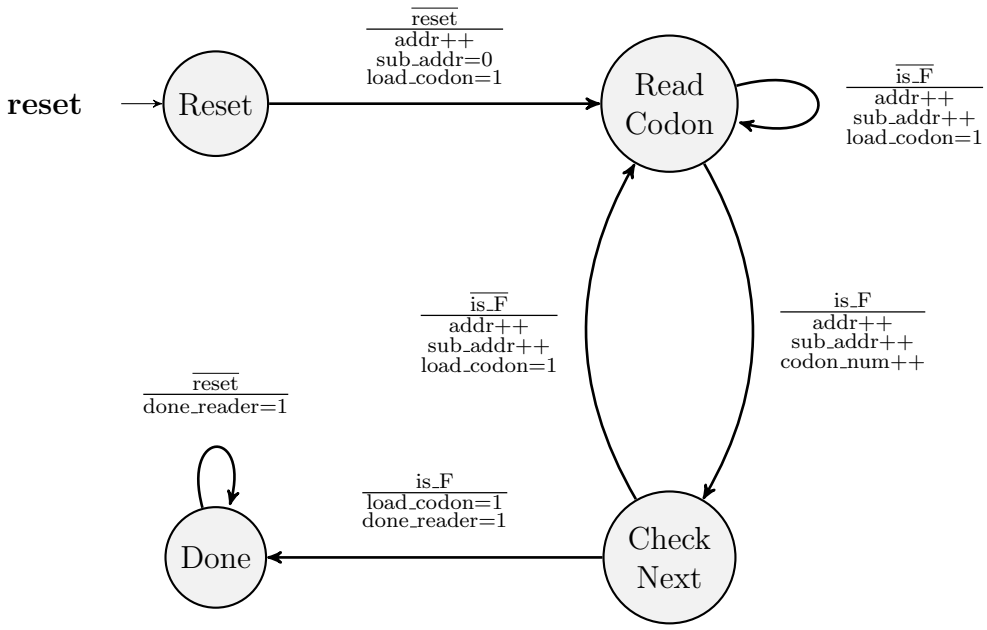

Figure 1.1: Codon Reading Finite State Machine

As evident by the FSM, this is a fairly straightforward module. The codon memory is instantiated, and then each sequential memory location is read, filling in a 3D array (called **codons**) as it goes. Whenever an **0xF** is detected–that value is skipped, **codon_num** is incremented (signifying which codon we are *filling*) and the process repeats. This is repeated until two sequential **0xF**'s are detected, at which point the **done** state is entered, and the process switches to entirely combinational logic.

The combinational logic in this module is fairly substantial, handling all of the codon-subindex output, and *end of codon* detection. This module takes in a 3-bit signal called **codon_index** which signifies which 4-bit nibble of all codons to look at (for example, nibble 1 of codon **0x16A2** would be **0x6**). Nibbles of all codons (1-5) are output at all times through signals **codonX**. The most important part (with regards to counting codons used in the next module) is the end of codon detection. To aid in this, the codon memory is created with one extra nibble per codon (so they are all technically 6 nibbles long), and the entire memory is initialized with 1's. Then, a 5-bit (one bit for each codon) output **end_of_codon** looks at the *next* nibble of all codons (so **codon_index**+1) to see if that contains **0xF**, signifying the current nibble is the last one in memory.

1.2 Codon Counter

The second module created for this project is the codon counter. This is fairly substantial, and should have probably been done in two or more separate modules. This module waits for the reader module to finish reading (signified by **done_reader** going high), and then it starts going through the gene memory, counting instances of each codon. The FSM for this is shown below in **Figure 1.2**.

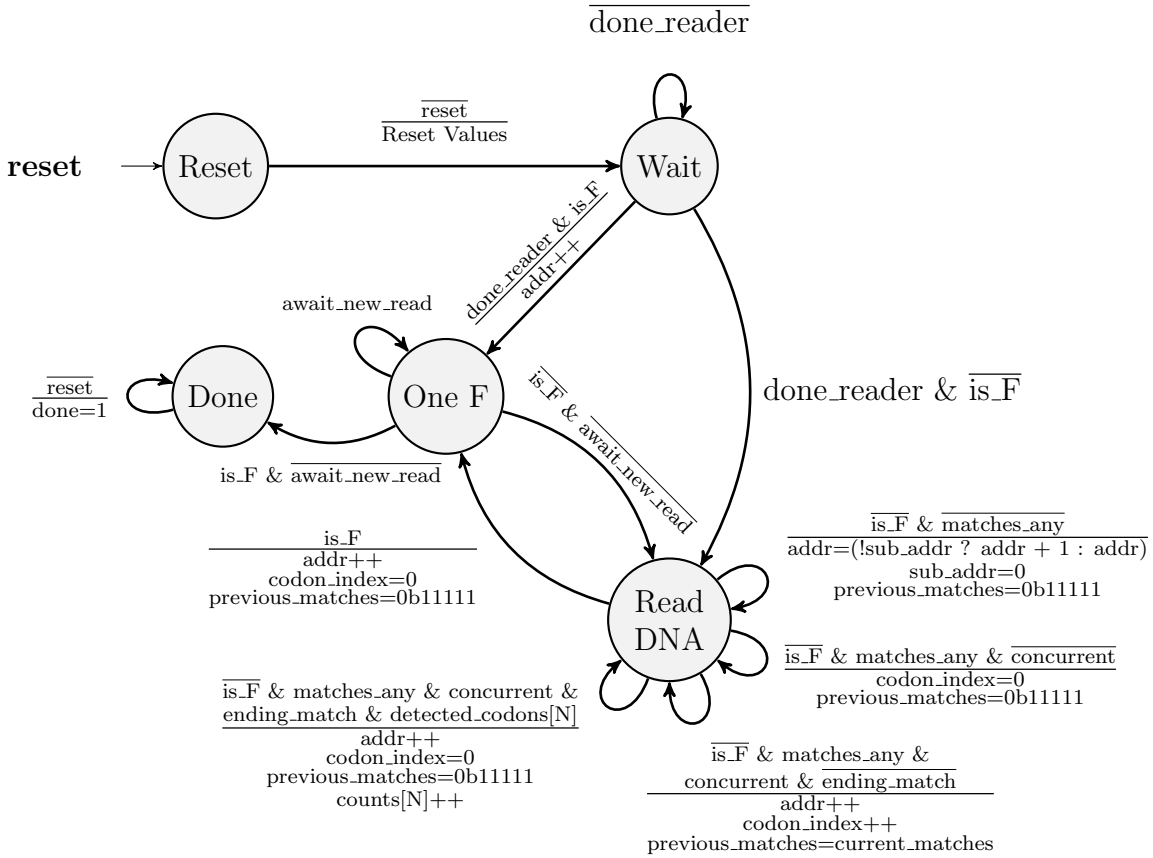

Figure 1.2: Codon Counter Finite State Machine

When reset is no longer asserted, the values used in this module are reset. The notable one here is that **previous_matches** is reset to **0b11111** as opposed to zeros. This is done so that during the detection of full codons, single-nibble codons are detected (I'll go into more detail later). I could technically remove the **Wait** state by and-ing **reset** and **done_reader** to start the FSM. However, as soon as the codon reading is finished, the gene memory is read. Not shown in the FSM (at least on the **Read DNA** state) is the relevance of **await_new_read**. This is a signal used to account for the one clock delay in all reads, and is toggled during every clock cycle while in the **Read DNA** and **One F** states.

The internal control signals being used for most of this logic are the **is_F**, **matches_any**,

concurrent_matches (labeled concurrent), **ending_match**, and **detected_codons**.Their assignments are shown in **Listing 1.1**.

Listing 1.1: Internal Logic Signal Assignment

```
always_comb begin : codon_match_detection
  current_matches[0] = ((memory_out == codon1) && (codon1
     != 4'hF));
  current_matches[1] = ((memory_out == codon2) && (codon2
     != 4'hF));
  current_matches[2] = ((memory_out == codon3) && (codon3
     != 4'hF));
  current_matches[3] = ((memory_out == codon4) && (codon4
     != 4'hF));
  current_matches[4] = ((memory_out == codon5) && (codon5
     != 4'hF));

  ending_match = | (current_matches & end_of_codon);
  concurrent_matches = | (current_matches &
     previous_matches);
  matches_any = | current_matches;
  detected_codons = (current_matches & previous_matches &
     end_of_codon);
end : codon_match_detection
```

Although the **Read DNA** state looks complicated, it really only handles five conditions. These five conditions, and the action taken, are as follows:

- The current gene nibble is **0xF**– move to the **One F** state.
- No matches on the current gene to any of the codons, shown by $\overline{\text{matches\_any}}$ – increment the address only if we were looking at the start nibble of all codons, otherwise return to the start codon index.
- There was a match, but not a concurrent one – don't increment the address, restart on the first nibble of all codons.
- There is a concurrent match, but not an end-of-codon one – store the current matches, go to the next codon index and address.
- There's an ending, concurrent match on codon[**N**] (1-5) – increment that codon's count, reset the codon index and previous match values, and finally go to the next address.

1.3 Hardware Wrapper

The final *module* I wrote was the hardware wrapper – it takes the clock, reset, switches, and LED inputs and takes care of the output LED assignment. There is nothing of note here, however I was unable to implement the **.xdc** file portion of this project because I couldn't access the lab and was unable to run *Vivado* on my computer.

2 Tribulations

I initially planned on having all bits of all codons as inputs to the codon counting module, but I realized after beginning that design process that this would require at least 100-bits of input to the module *just for the codons*, at that seemed quite *messy*. Although this is not a direct problem, it did require a redesign, and I ended up implementing the nibble-indexing (with **codon_index**) instead.

I initially did not account for the one clock cycle delay on reading from the gene memory, and the first instance of **0xF** would terminate the counting. Because it would be **0xF** on the first cycle, and had not yet updated by the time it was evaluated for the **One F** state. I fixed this by adding the **await_new_read** signal.

3 Simulations

I was also unable to perform any simulations with the synthesized design (however I did verify it synthesized without error) – but the behavioral simulation went as expected, and the last few cycles of the codon reader are shown below in **Figure 3.1**.

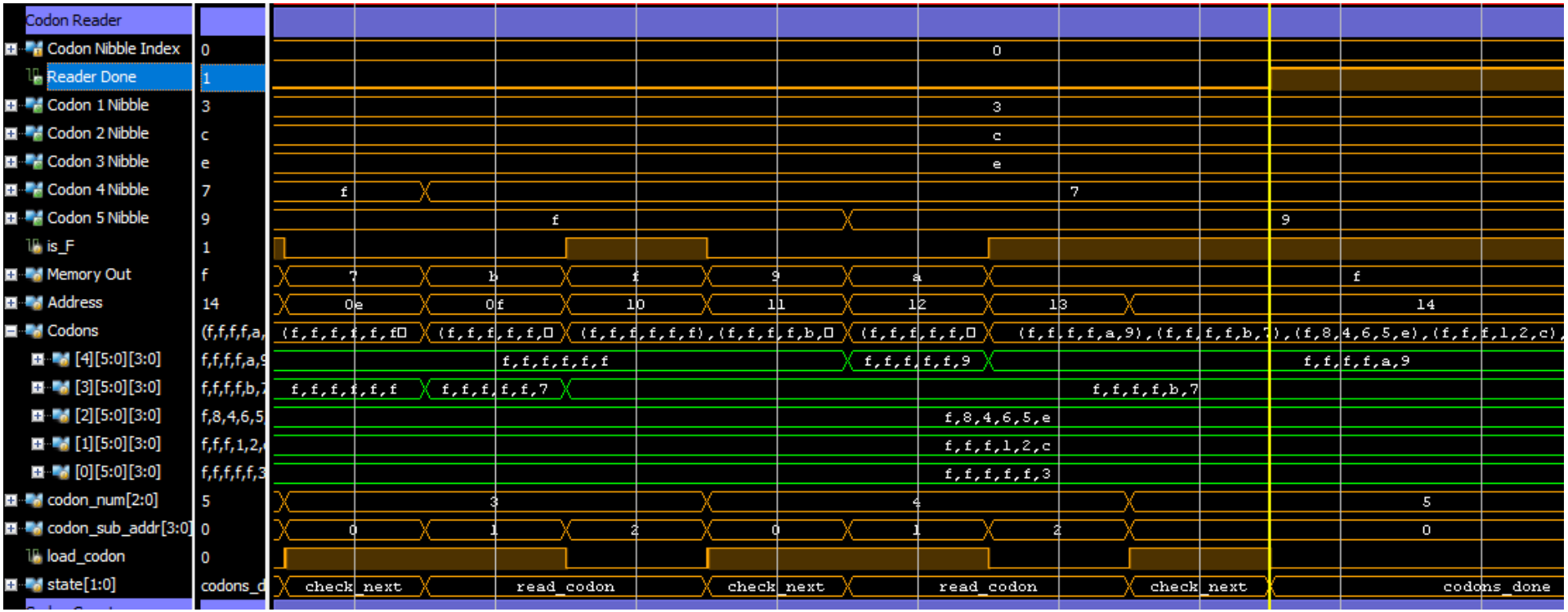

Figure 3.1: Codon Reader Behavioral Simulation.

This simulation was performed on the example codon and gene memory instance, and viewing **Codons** shows how the loading sequence works. All stored codons start off as **0xFFFFFF**, and are loaded sequentially (shown in reverse order in the simulation, for some reason). After two sequential instances of **0xF**, the done signal is asserted. I also have an image of the final count outputs, shown in **Figure 3.2**.

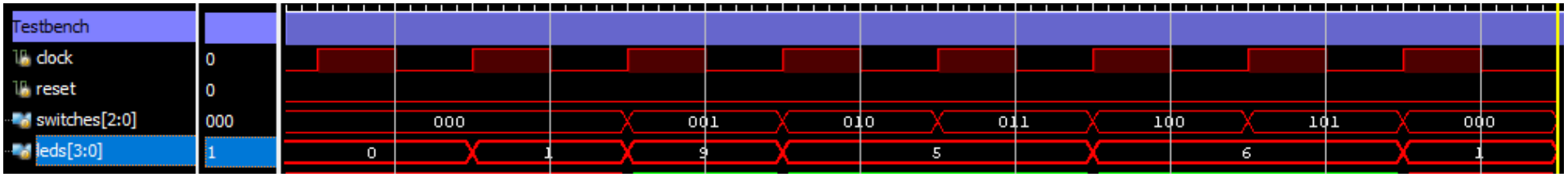

Figure 3.2: All codon counts asserted on the LEDs.

When the switches are 0, **LED0** goes high as soon as the counting is done, and then the sequence of switches cycles through the counts of each codon.

4 Source Code

Listing 4.1: Codon Reader Module

```
1  `timescale 1ns / 1ps
2
3  module codon_reader(clock, reset, codon_index, done_reader,
       codon1, codon2, codon3, codon4, codon5, end_of_codon);
4    input logic clock, reset;
5    input logic [2:0] codon_index;
6    output logic done_reader;
7    output logic [3:0] codon1, codon2, codon3, codon4, codon5
       ;
8    output logic [4:0] end_of_codon;
9
10 // Instantiate the codon memory unit
11 logic enable, write_enable, is_F;
12 logic [3:0] memory_in, memory_out;
13 logic [4:0] address;          // 32 addressable locations
14 assign enable = 1;            // Always reading
15 assign write_enable = 0;        // Never writing
16 assign memory_in = 0;          // Never writing
17 assign is_F = (memory_out == 4'hF); // Flag for if an 0xF
     has been detected
18 codon_memory codon_memory_instance (
19   .clka(clock),
20   .ena(enable),
21   .wea(write_enable),
22   .addra(address),
23   .dina(memory_in),
24   .douta(memory_out)
25 );
26
27 // Codon Register - End-of-codon detection - Codon Sub
     Addressing
28 logic [3:0] codons[4:0][5:0]; // 4-bit wide units, [
     codon_num][codon_sub_addr]
29 logic [2:0] codon_num;
30 assign codon1 = codons[0][codon_index];
31 assign codon2 = codons[1][codon_index];
32 assign codon3 = codons[2][codon_index];
33 assign codon4 = codons[3][codon_index];
34 assign codon5 = codons[4][codon_index];
35 assign end_of_codon[0] = (codons[0][codon_index + 1] == 4'
     hF); // Check if next codon value is 0xF
36 assign end_of_codon[1] = (codons[1][codon_index + 1] == 4'
     hF);
```

```
37 assign end_of_codon[2] = (codons[2][codon_index + 1] == 4'
     hF);
38 assign end_of_codon[3] = (codons[3][codon_index + 1] == 4'
     hF);
39 assign end_of_codon[4] = (codons[4][codon_index + 1] == 4'
     hF);
40
41 logic [3:0] codon_sub_addr; // Which 4-bit nibble is being
     loaded
42 logic load_codon;     // Whether or not to load the codon
     register
43
44 // FSM States
45 typedef enum logic [1:0] {reset_state, read_codon,
     check_next, codons_done} statetype;
46 statetype state;
47
48 // FSM Advancement
49 always_ff @(posedge clock) begin : fsm_advancement
50   if (reset) begin
51     state <= reset_state; // Go to reset state
52     address <= 0;      // Reset address
53     codon_sub_addr <= 0;  // Reset nibble number
54     codon_num <= 0;      // Current codon number
55     end
56   else begin
57     case (state)
58       reset_state: begin
59         state <= read_codon;
60         address <= address + 1;
61         end
62       read_codon: begin
63         address <= address + 1;
64         unique case (is_F)
65           1'b1: begin state <= check_next; codon_num <=
                codon_num + 1; codon_sub_addr <= 0; end
66           1'b0: begin state <= read_codon; codon_sub_addr
                <= codon_sub_addr + 1;        end
67         endcase
68         end
69       check_next:
70         unique case (is_F)
71           1'b0: begin state <= read_codon; address <=
                address + 1; codon_sub_addr <= codon_sub_addr
                + 1;  end
72           1'b1: begin state <= codons_done;
                                  end
```

```
73          endcase
74        codons_done:
75          state <= codons_done;
76      endcase
77      end
78  end : fsm_advancement
79
80  // FSM Outputs
81  always_comb begin : fsm_outputs
82    load_codon = 0; done_reader = 0;
83    if (~reset) begin
84      case (state)
85        reset_state:  load_codon = 1;
86        read_codon:   load_codon = ~is_F;
87        check_next:   load_codon = 1;
88        codons_done:  done_reader = 1;
89        endcase
90      end
91  end : fsm_outputs
92
93  // Codon Loading
94  always_ff @(posedge clock) begin : codon_loader
95    if (reset)
96      codons <= '{default:4'b1111}; // Codons should default
           to 0xF
97    else
98      if (load_codon) codons[codon_num][codon_sub_addr] =
           memory_out;
99  end : codon_loader
100
101 endmodule : codon_reader
```

Listing 4.2: Codon Counter Module

```
1  `timescale 1ns / 1ps
2
3  module codon_counter(clock, reset, done_reader, count_index
       , codon1, codon2, codon3, codon4, codon5, end_of_codon,
       done_counter, codon_index, codon_count);
4    input logic clock, reset, done_reader;
5    input logic [2:0] count_index;                  // Which
       codon count to output [1->5]
6    input logic [3:0] codon1, codon2, codon3, codon4, codon5;
       // Current nibble of codon 1 to 5
7    input logic [4:0] end_of_codon;                 // Encoding
       of which codon we're on the last nibble of
8    output logic done_counter;                      // Whether
```

```
        the counter module is done
    output logic [2:0] codon_index;                 // Which
        nibble to address of all the codons
    output logic [3:0] codon_count;                 // Count of
         the current codon selected by count_index

// Instantiate the genome memory unit
logic enable, write_enable, is_F;
logic [3:0] memory_in, memory_out;
logic [7:0] address;
assign enable = 1;              // Always reading
assign write_enable = 0;          // Never writing
assign memory_in = 0;           // Never writing
assign is_F = (memory_out == 4'hF); // Flag for if an 0xF
    has been detected
gene_memory gene_memory_instance (
    .clka(clock),
    .ena(enable),
    .wea(write_enable),
    .addra(address),
    .dina(memory_in),
    .douta(memory_out)
);

// All Codon counts register and output assignment
logic [4:0][3:0] counts;              // Access with counts[
    codon_number]
assign codon_count = counts[count_index]; // Assign the
    output count

// Codon Match-detection
logic [4:0] previous_matches, current_matches,
    detected_codons;
logic ending_match, concurrent_matches, matches_any;
always_comb begin : codon_match_detection
    current_matches[0] = ((memory_out == codon1) && (codon1
        != 4'hF));  // Whether each codon matches the current
        output
    current_matches[1] = ((memory_out == codon2) && (codon2
        != 4'hF));
    current_matches[2] = ((memory_out == codon3) && (codon3
        != 4'hF));
    current_matches[3] = ((memory_out == codon4) && (codon4
        != 4'hF));
    current_matches[4] = ((memory_out == codon5) && (codon5
        != 4'hF));

```

```
43    ending_match = | (current_matches & end_of_codon);
          // T/F if a match occurred on the last of a codon
44    concurrent_matches = | (current_matches &
          previous_matches);  // T/F if two same-place
          sequential matches occurred
45    matches_any = | current_matches;                  // T/F if
          any matches occurred
46    detected_codons = (current_matches & previous_matches &
          end_of_codon);  // 5-Bit field that denotes a
          completed codon detection
47  end : codon_match_detection
48
49  // FSM States
50  typedef enum logic [2:0] {reset_state, wait_state, read_DNA
        , one_F, done_state} statetype;
51  statetype state;
52  logic await_new_read;
53
54  // FSM Advancement
55  always_ff @(posedge clock) begin : fsm_advancement
56    if (reset) begin
57      state <= reset_state;      // Go to reset state
58      address <= 0;          // Reset memory address
59      counts <= 0;           // Reset array of all codon
            counts
60      previous_matches <= 5'b11111; // Previous matches are
            default high so immediate end-matches are okay (
            through AND)
61      codon_index <= 3'b000;       // Reset codon nibble index
62      await_new_read <= 0;       // Reset 'new read' flag
63      end
64    else begin
65      case (state)
66        reset_state:
67          state <= wait_state;
68        wait_state:
69          unique casez({is_F, done_reader})
70            2'b?0: begin state <= wait_state;            end
71            2'b11: begin state <= one_F; address <= address +
                  1;  end
72            2'b01: begin state <= read_DNA;              end
73          endcase
74        read_DNA: begin
75          await_new_read <= ~await_new_read;  // Use this to
              deal with the 1 clock delay on reads
76          if (~await_new_read) begin
77            address <= address + 1;
```

```
78          unique casez({is_F, matches_any,
               concurrent_matches, ending_match,
               detected_codons})
79            9'b1????????: begin state <= one_F; address <=
                 address + 1; codon_index <= 3'b000;
                 previous_matches <= 5'b11111;
                       end // 0xF output from memory
80            9'b00???????: begin state <= read_DNA; address
                 <= (codon_index == 3'b000 ? address + 1 :
                 address); codon_index <= 3'b000;
                 previous_matches <= 5'b11111; end // No
                 matches whatsoever
81            9'b010??????: begin state <= read_DNA;
                 codon_index <= 3'b000; previous_matches <=
                 5'b11111;                                  end
                 // Was a match, but not concurrently
82            9'b0110?????: begin state <= read_DNA; address
                 <= address + 1; codon_index <= codon_index +
                  1; previous_matches <= current_matches;
                         end // Concurrent match, but not an
                 end-of-codon one
83            9'b01111????: begin state <= read_DNA; address
                 <= address + 1; codon_index <= 3'b000;
                 previous_matches <= 5'b11111; counts[4] <=
                 counts[4] + 1;      end // End-of-codon match
                  on codon5
84            9'b011101???: begin state <= read_DNA; address
                 <= address + 1; codon_index <= 3'b000;
                 previous_matches <= 5'b11111; counts[3] <=
                 counts[3] + 1;      end // End-of-codon match
                  on codon4
85            9'b0111001??: begin state <= read_DNA; address
                 <= address + 1; codon_index <= 3'b000;
                 previous_matches <= 5'b11111; counts[2] <=
                 counts[2] + 1;      end // End-of-codon match
                  on codon3
86            9'b01110001?: begin state <= read_DNA; address
                 <= address + 1; codon_index <= 3'b000;
                 previous_matches <= 5'b11111; counts[1] <=
                 counts[1] + 1;      end // End-of-codon match
                  on codon2
87            9'b011100001: begin state <= read_DNA; address
                 <= address + 1; codon_index <= 3'b000;
                 previous_matches <= 5'b11111; counts[0] <=
                 counts[0] + 1;      end // End-of-codon match
                  on codon1
88          endcase
```

```
89            end
90          end
91        one_F: begin
92          await_new_read <= 0;
93          unique casez({is_F, await_new_read})
94            2'b?1: state <= one_F;
95            2'b00: state <= read_DNA;
96            2'b10: state <= done_state;
97          endcase
98          end
99        done_state:
100         state <= done_state;
101     endcase
102     end
103 end : fsm_advancement
104
105 // FSM Outputs
106 assign done_counter = (state == done_state);
107
108 endmodule : codon_counter
```

Listing 4.3: Hardware Wrapper

```
1  `timescale 1ns / 1ps
2
3  module hardware_wrapper(clock, reset, switches, leds);
4    input logic clock, reset;
5    input logic [2:0] switches;
6    output logic [3:0] leds;
7
8  // Instantiate the Codon Reader
9  logic done_reader;
10 logic [2:0] codon_index;
11 logic [3:0] codon1, codon2, codon3, codon4, codon5;
12 logic [4:0] end_of_codon;
13 codon_reader codon_reader_instance(.*);
14
15 // Instantiate the Codon Counter
16 logic [2:0] count_index;
17 logic [3:0] codon_count;
18 logic done_counter;
19 codon_counter codon_counter_instance(.*);
20 always_comb begin : count_assignment
21   count_index = 0;
22   unique casez(switches)  // Messy, but it works
23     3'b00?:   count_index = 0;
24     3'b010:   count_index = 1;
```

```
25      3'b011:   count_index = 2;
26      3'b100:   count_index = 3;
27      3'b101:   count_index = 4;
28      default:  count_index = 0;
29    endcase
30  end : count_assignment
31
32  always_comb begin : led_assignment
33    leds = 4'b0000;
34    if (done_counter) begin
35      if (switches == 3'b000)
36        leds = 4'b0001;
37      else
38        leds = codon_count;
39      end
40  end : led_assignment
41
42  endmodule : hardware_wrapper
```

Listing 4.4: Simulation 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 = (500 * CLK_PRD);
9
10 // Instantiate the GCD core as a DUT
11 logic clock, reset;
12 logic [2:0] switches;
13 logic [3:0] leds;
14 hardware_wrapper dut(
15   .clock(clock),
16   .reset(reset),
17   .switches(switches),
18   .leds(leds)
19 );
20
21 // Prevent simulating longer than MAX_SIM_TIME
22 initial #(MAX_SIM_TIME) $finish;
23
24 // Generate Clock Signal
25 initial begin
26   clock <= 0;
```

```
27    forever #(CLK_PRD / 2) clock = ~clock;
28  end
29
30  // Main Simulation
31  initial begin
32    reset = 1; switches = 0;
33
34    // Global Reset
35    #100; reset = 0;
36
37    @(posedge clock); #HOLD_TIME;  // Align with clock
38
39    forever begin
40      @(posedge clock);
41      if (leds) begin
42        switches = 4'b0001; #CLK_PRD; // Codon 1
43        switches = 4'b0010; #CLK_PRD; // Codon 2
44        switches = 4'b0011; #CLK_PRD; // Codon 3
45        switches = 4'b0100; #CLK_PRD; // Codon 4
46        switches = 4'b0101; #CLK_PRD; // Codon 5
47        switches = 4'b0000; #CLK_PRD;
48        $finish;
49        end
50    end
51  end
52
53  endmodule
```