ECE 316: Digital Logic Design

Tokyo Yamanote Line Simulator - Implementation Phase

20.08.2026

Rachit Kakkar

Summer 2026

Contents

1. Part 1 — Translating the Paper Design to Verilog

1.1. All Verilog source files for the module hierarchy

- `ad_sign_top.v`

Verilog

```verilog
`timescale 1ns / 1ps

module ad_sign_top (
    input wire clk,
    input wire rst,
    input wire BTNU,
    input wire [0:0] sw0,
    output wire [6:0] seg,
    output wire [3:0] an,
    output wire [4:0] leds,
    output wire outedge_led
);
    wire outedge;
    // assign outedge_led = outedge; // map led to outedge (debug)
    rising_edge_detector u_red (
        .clk(clk),
        .rst(rst),
        .BTNU(BTNU),
        .sw0(sw0),
        .outedge(outedge)
    );

    // controller + datapath signals
    wire dir_ld;
    wire start_ld;
    wire end_ld;
    wire leds_ld;
    wire counter_clr;
    wire counter_en;
    wire station_addr_sel;
    wire [1:0] pos_ld, comp_ld;
    wire [2:0] msg_ld;
    wire counter_done, ride_done;

    controller u_ctrl (
        .clk(clk),
        .rst(rst),
        .outedge(outedge),
        .sw0(sw0),
        .counter_done(counter_done),
        .ride_done(ride_done),
        .dir_ld(dir_ld),
        .start_ld(start_ld),
        .end_ld(end_ld),
        .pos_ld(pos_ld),
        .leds_ld(leds_ld),
        .msg_ld(msg_ld),
        .counter_clr(counter_clr),
        .counter_en(counter_en),
        .comp_ld(comp_ld),
        .station_addr_sel(station_addr_sel)
    );

    datapath u_dp (
        .clk(clk),
        .rst(rst),
        .dir_ld(dir_ld),
        .start_ld(start_ld),
        .end_ld(end_ld),
        .pos_ld(pos_ld),
        .leds_ld(leds_ld),
        .msg_ld(msg_ld),
        .counter_clr(counter_clr),
        .counter_en(counter_en),
        .comp_ld(comp_ld),
        .station_addr_sel(station_addr_sel),
```

```
67          .seg(seg),
68          .an(an),
69          .leds(leds),
70          .counter_done(counter_done),
71          .ride_done(ride_done)
72      );
73  endmodule
```

- `controller.v`

Verilog

```
 1  `timescale 1ns / 1ps
 2
 3  module controller (
 4      input wire clk,
 5      input wire rst,
 6      input wire outedge, // Unlike the skeleton, I chose to detect the rising edge OUTSIDE the
    controller
 7      input wire sw0,
 8      input wire counter_done,
 9      input wire ride_done,
10      output reg dir_ld,
11      output reg start_ld,
12      output reg end_ld,
13      output reg [1:0] pos_ld,
14      output reg leds_ld,
15      output reg [2:0] msg_ld,
16      output reg counter_clr,
17      output reg counter_en,
18      output reg [1:0] comp_ld,
19      output reg station_addr_sel
20  );
21      reg [2:0] state, next_state;
22      // reg win_reg; // Remembers if player won or lost at exit
23
24      // localparams (inspired by FSM slides from lectures) to keep everything clear
25      localparam OFF = 3'b000;
26      localparam SELECT = 3'b001;
27      localparam RESET1 = 3'b010; // mainly to assert counter_clr
28      localparam RIDE = 3'b011;
29      localparam RESET2 = 3'b100; // again to clear clock counter
30      localparam CHECK_WIN = 3'b101;
31
32      // always @(posedge clk or posedge rst) begin
33      //      if (rst) begin
34      //          win_reg <= 1'b0;
35      //      end else if (state == RESET2) begin
36      //          win_reg <= ride_done;
37      //      end
38      //  end
39
40      // Next-state logic: each button-edge pulse advances one mode in the cycle
41      always @(*) begin
42          case (state)
43              OFF: next_state = outedge ? SELECT : OFF;
44              SELECT:next_state = outedge ? RESET1 : SELECT;
45              RESET1: next_state = RIDE;
46              RIDE: next_state = outedge ? RESET2 : RIDE;
47              RESET2: next_state = CHECK_WIN;
48              CHECK_WIN: next_state = (counter_done || outedge) ? OFF : CHECK_WIN;
49              default: next_state = OFF;
50          endcase
51      end
52
53      always @(posedge clk or posedge rst) begin
54          if (rst)
55              state <= OFF;
56          else
57              state <= next_state;
58      end
59
60      // Output logic. Moore except for pos_load, which is asserted on the
61      // button edge so the new mode starts at window position 0.
62      // To do: double check msg_ld addresses
```

```
63      always @(*) begin
64          dir_ld = 1'b0;
65          start_ld = 1'b0;
66          end_ld = 1'b0;
67          pos_ld = 2'b11;
68          leds_ld = 1'b00;
69          msg_ld = 3'b000;
70          counter_clr = 1'b0;
71          counter_en = 1'b0;
72          comp_ld = 2'b00;
73          station_addr_sel = 1'b0;
74
75          case (state)
76              OFF: begin
77                  dir_ld = 1'b1;
78                  start_ld = 1'b1;
79                  end_ld = 1'b1;
80                  pos_ld = 2'b00;
81                  msg_ld = 3'b000; // off
82                  counter_clr = 1'b1;
83                  comp_ld = 2'b00;
84              end
85
86              SELECT: begin
87                  msg_ld = 3'b001; // ride to
88                  // counter_en = 1'b1;
89                  counter_en = !sw0; // add pause functionality
90                  station_addr_sel = 1'b0;
91                  comp_ld = 2'b00;
92                  if (counter_done)
93                      counter_clr = 1'b1;
94              end
95
96              RESET1: begin
97                  msg_ld = 3'b010; // Station name
98                  pos_ld = 2'b01;
99                  counter_clr = 1'b1;
100                 comp_ld = 2'b01;
101             end
102
103             RIDE: begin
104                 msg_ld = 3'b010; // station name
105                 leds_ld = 1'b1;
106                 counter_en = !sw0;  // add pause functionality
107                 station_addr_sel = 1'b1;
108                 comp_ld = 2'b01;
109                 pos_ld = 2'b11;
110
111                 if (counter_done) begin
112                     counter_clr = 1'b1;  // Resets counter for the next station stop
113                     pos_ld = 2'b10;
114                 end
115             end
116
117             RESET2: begin
118                 counter_clr = 1'b1;
119                 comp_ld = 2'b10;
120                 msg_ld = ride_done ? 3'b011 : 3'b100;
121             end
122
123             CHECK_WIN: begin
124                 msg_ld = ride_done ? 3'b011 : 3'b100;
125                 counter_en = !sw0;
126                 comp_ld = 2'b10;
127                 if (counter_done)
128                     counter_clr = 1'b1;
129             end
130         endcase
131     end
132 endmodule
```

- `rising_edge_detector.v`

Verilog

```verilog
`timescale 1ns / 1ps

module rising_edge_detector(
    input wire clk,
    input wire rst,
    input wire BTNU,
    input wire sw0,
    output reg outedge
);
    reg [1:0] state, next_state;
    wire signal;
    assign signal = btn_db && !sw0;

    // Combinational logic: next state + output.
    always @(*) begin
        // Hint: use a case statement on the current state.
        case (state)
            2'b00: outedge = 0;
            2'b01: outedge = 1;
            2'b10: outedge = 0;
            default: outedge = 0;
        endcase

        // Next state
        case ({signal, state})
            3'b000: next_state = 2'b00;
            3'b100: next_state = 2'b01;
            3'b001: next_state = 2'b00;
            3'b101: next_state = 2'b10;
            3'b010: next_state = 2'b00;
            3'b110: next_state = 2'b10;
            default: next_state = 2'b00;
        endcase
    end

    // Sequential logic on the slow clock with asynchronous reset.
    always @(posedge clk or posedge rst) begin
        // Hint: don't forget about the reset signal, which can be handled with an if-statement
        if (rst)
            state <= 2'b00;
        else
            state <= next_state;
    end

    // Debounce logic
    reg btn_db;
    reg [19:0] db_count;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            btn_db <= 0;
            db_count <= 0;
        end
        else if (BTNU == btn_db) begin
            db_count <= 0;
        end
        else begin
            db_count <= db_count + 1;
            if (db_count == 20'hFFFFF) begin
                btn_db <= BTNU;
                db_count <= 0;
            end
        end
    end
endmodule
```

- `datapath.v`

Verilog

```verilog
`timescale 1ns / 1ps

```

```
 3 module datapath (
 4     input wire clk,
 5     input wire rst,
 6     input wire dir_ld,
 7     input wire start_ld,
 8     input wire end_ld,
 9     input wire [1:0] pos_ld,
10     input wire leds_ld,
11     input wire [2:0] msg_ld,
12     input wire counter_clr,
13     input wire counter_en,
14     input wire [1:0] comp_ld,
15     input wire station_addr_sel,
16     output wire [6:0] seg,
17     output wire [3:0] an,
18     output reg [4:0] leds,
19     output wire counter_done,
20     output wire ride_done
21 );
22     wire slow_clk, disp_clk;
23     // reset clk_div count at the same time as we clear the counter (which controls message
   position) to prevent first character from scrolling too fast
24     clkdiv #(26) u_slow (.clk(clk), .rst(rst), .clr(counter_clr), .slow_clk(slow_clk));
25     clkdiv #(17) u_disp (.clk(clk), .rst(rst), .clr(1'b0), .slow_clk(disp_clk));
26
27     // counters for randomness
28     reg dir_cnt;
29     reg [4:0] start_station_cnt;
30     reg [4:0] end_station_cnt;
31
32     reg dir_reg;
33     reg [4:0] start_reg, end_reg;
34
35     always @(posedge clk or posedge rst) begin
36         if (rst) begin
37             dir_cnt <= 1'b0;
38             start_station_cnt <= 5'b00000;
39             end_station_cnt <= 5'b00000;
40         end else begin
41             dir_cnt <= dir_cnt + 1'b1; // Should toggle / overflow back to zero?
42
43
44             // i'm adding start and end by different primes to ensure they don't equal each other
45             // station counters from 0 to 29
46             if (start_station_cnt == 5'b11101) // 29
47                 start_station_cnt <= 5'b00000;
48             else
49                 start_station_cnt <= start_station_cnt + 5'b00101; // 5
50             if (end_station_cnt == 5'b11101) // 29
51                 end_station_cnt <= 5'b00000;
52             else
53                 end_station_cnt <= end_station_cnt + 5'b00111; // 7
54
55         end
56     end
57
58     always @(posedge clk or posedge rst) begin
59         if (rst) begin
60             dir_reg   <= 1'b0;
61             start_reg <= 5'b00000;
62             end_reg   <= 5'b00000;
63         end else begin
64             if (dir_ld)   dir_reg   <= dir_cnt;
65             if (start_ld) start_reg <= start_station_cnt;
66             if (end_ld)   end_reg   <= end_station_cnt;
67         end
68     end
69
70     wire [4:0] pos;
71     pos_counter u_pos (
72         .clk(clk),
73         .rst(rst),
74         .pos_ld(pos_ld),
75         .start(start_reg),
76         .dir(dir_reg),
```

```
        .pos(pos)
    );

    wire [4:0] clk_ctr;
    clk_counter u_ctr (
        .clk(clk),
        .slow_clk(slow_clk),
        .rst(rst),
        .counter_en(counter_en),
        .counter_clr(counter_clr),
        .clk_ctr(clk_ctr)
    );

    wire [4:0] station_addr = (station_addr_sel) ? pos : end_reg; // Addr mux
    wire [127:0] station_name;
    wire [3:0] str_len;

    station_name_rom u_name_rom (.addr(station_addr), .name(station_name));
    str_len_rom u_len_rom  (.addr(station_addr), .len(str_len));

    reg [4:0] comp_target;
    always @(*) begin
        case (comp_ld)
            2'b00: comp_target = str_len + 9; // "RIDE TO " + station name
            2'b01: comp_target = str_len + 1; // current station string length
            2'b10: comp_target = 5'd14; // right or wrong station message
            default: comp_target = 5'd14;
        endcase
    end

    // Instead of using mag_cmp like outlined in 4a
    assign counter_done = (clk_ctr == comp_target);
    assign ride_done = (pos == end_reg);

    wire [191:0] full_msg;
    message_mux u_msg (.msg_ld(msg_ld), .station_name(station_name), .msg(full_msg));

    wire [31:0] window_disp;
    window_select u_win (.msg(full_msg), .ctr(clk_ctr), .disp(window_disp));

    wire [6:0] seg_0, seg_1, seg_2, seg_3;
    ascii_to_7seg u_seg0 (.char(window_disp[31:24]), .seg(seg_0));
    ascii_to_7seg u_seg1 (.char(window_disp[23:16]), .seg(seg_1));
    ascii_to_7seg u_seg2 (.char(window_disp[15:8]),  .seg(seg_2));
    ascii_to_7seg u_seg3 (.char(window_disp[7:0]),   .seg(seg_3));

    time_mux_sm u_tmux (
        .disp_clk(disp_clk),
        .rst(rst),
        .seg_0(seg_0),
        .seg_1(seg_1),
        .seg_2(seg_2),
        .seg_3(seg_3),
        .seg(seg),
        .an(an)
    );

    always @(posedge clk or posedge rst) begin
        if (rst)
            leds <= 5'b00000;
        else if (leds_ld)
            leds <= pos + 1'b1;
        else
            leds <= 5'b00000;
    end
endmodule
```

- clk_div.v

Verilog

```
1 `timescale 1ns / 1ps
2
3 module clkdiv #(
4     parameter WIDTH = 26
```

```
 5 ) (
 6     input wire clk,
 7     input wire rst,
 8     input wire clr, // turns out we need to reset the clkdiv count too (which prevents the first
   character from scrolling by fast)
 9     output wire slow_clk
10 );
11     reg [WIDTH-1:0] COUNT;
12     assign slow_clk = COUNT[WIDTH-1];
13 
14     always @(posedge clk) begin
15         if (rst || clr) COUNT <= 0;
16         else COUNT <= COUNT + 1;
17     end
18 endmodule
```

- pos_counter.v

Verilog

```verilog
`timescale 1ns / 1ps

module pos_counter (
    input wire clk,
    input wire rst,
    input wire [1:0] pos_ld,
    input wire [4:0] start,
    input wire dir,
    output reg [4:0] pos
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            pos <= 5'b00000;
        else begin
            case (pos_ld)
                2'b00: pos <= 5'b00000;
                2'b01: pos <= start;
                2'b10: begin
                    if (dir == 1'b0) begin // Clockwise loop
                        if (pos >= 5'd29)
                            pos <= 5'd0;
                        else
                            pos <= pos + 1'b1;
                    end
                    else begin // Counter-clockwise loop
                        if (pos == 5'd0)
                            pos <= 5'd29;
                        else
                            pos <= pos - 1'b1;
                    end
                end
                2'b11: pos <= pos;
            endcase
        end
    end
endmodule
```

- clk_counter.v

Verilog

```verilog
`timescale 1ns / 1ps

module clk_counter (
    input wire clk,
    input wire slow_clk,
    input wire rst,
    input wire counter_en,
    input wire counter_clr,
    output reg [4:0] clk_ctr
);
    wire slow_clk_edge;

    // this is needed so the clock counter won't count millions of times while slow_clk is high
    // Instead of being level triggered, it counts just once on the rising edge of slow_clock
    rising_edge_detector u_red (
        .clk(clk),
        .rst(rst),
        .BTNU(slow_clk), // Bit confusing since BTNU != slow_clk, but I don't want to change the port list
        .sw0(1'b0), // hacky way to disable pause since its not needed for this rising_edge_detector
        .outedge(slow_clk_edge)
    );

    always @(posedge clk or posedge rst) begin
        if (rst || counter_clr) begin
            clk_ctr <= 5'b00000;
        end else begin
            // if (counter_clr) begin // clear (like rst) must take precedence
            //     clk_ctr <= 5'b00000;
            // end else
            if (counter_en && slow_clk_edge) begin
```

```
31                clk_ctr <= clk_ctr + 1'b1;
32            end
33        end
34    end
35 endmodule
```

- station_name_rom.v

Verilog

```
 1 `timescale 1ns / 1ps
 2
 3 module station_name_rom (
 4     input wire [4:0] addr,
 5     output reg [127:0] name
 6 );
 7     always @(*) begin
 8         case (addr)
 9             5'b00000: name = "TOKYO           ";
10             5'b00001: name = "KANDA           ";
11             5'b00010: name = "AKIHABARA       ";
12             5'b00011: name = "OKACHIMACHI     ";
13             5'b00100: name = "UENO            ";
14             5'b00101: name = "UGUISUDANI      ";
15             5'b00110: name = "NIPPORI         ";
16             5'b00111: name = "NISHI NIPPORI   ";
17             5'b01000: name = "TABATA          ";
18             5'b01001: name = "KOMAGOME        ";
19             5'b01010: name = "SUGAMO          ";
20             5'b01011: name = "OTSUKA          ";
21             5'b01100: name = "IKEBUKURO       ";
22             5'b01101: name = "MEJIRO          ";
23             5'b01110: name = "TAKADANOBABA    ";
24             5'b01111: name = "SHIN OKUBO      ";
25             5'b10000: name = "SHINJUKU        ";
26             5'b10001: name = "YOYOGI          ";
27             5'b10010: name = "HARAJUKU        ";
28             5'b10011: name = "SHIBUYA         ";
29             5'b10100: name = "EBISU           ";
30             5'b10101: name = "MEGURO          ";
31             5'b10110: name = "GOTANDA         ";
32             5'b10111: name = "OSAKI           ";
33             5'b11000: name = "SHINAGAWA       ";
34             5'b11001: name = "TAKANAWA GW     ";
35             5'b11010: name = "TAMACHI         ";
36             5'b11011: name = "HAMAMATSUCHO    ";
37             5'b11100: name = "SHIMBASHI       ";
38             5'b11101: name = "YURAKUCHO       ";
39             default:  name = "TOKYO           ";
40         endcase
41     end
42 endmodule
```

- str_len_rom.v

Verilog

```
 1 `timescale 1ns / 1ps
 2
 3 module str_len_rom (
 4     input wire [4:0] addr,
 5     output reg [3:0] len
 6 );
 7     always @(*) begin
 8         case (addr % 30)
 9             5'b00000: len = 4'd5;  // TOKYO
10             5'b00001: len = 4'd5;  // KANDA
11             5'b00010: len = 4'd9;  // AKIHABARA
12             5'b00011: len = 4'd11; // OKACHIMACHI
13             5'b00100: len = 4'd4;  // UENO
14             5'b00101: len = 4'd10; // UGUISUDANI
15             5'b00110: len = 4'd7;  // NIPPORI
16             5'b00111: len = 4'd13; // NISHI NIPPORI
17             5'b01000: len = 4'd6;  // TABATA
18             5'b01001: len = 4'd8;  // KOMAGOME
```

```
            5'b01010: len = 4'd6;  // SUGAMO
            5'b01011: len = 4'd6;  // OTSUKA
            5'b01100: len = 4'd9;  // IKEBUKURO
            5'b01101: len = 4'd6;  // MEJIRO
            5'b01110: len = 4'd12; // TAKADANOBABA
            5'b01111: len = 4'd10; // SHIN OKUBO
            5'b10000: len = 4'd8;  // SHINJUKU
            5'b10001: len = 4'd6;  // YOYOGI
            5'b10010: len = 4'd8;  // HARAJUKU
            5'b10011: len = 4'd7;  // SHIBUYA
            5'b10100: len = 4'd5;  // EBISU
            5'b10101: len = 4'd6;  // MEGURO
            5'b10110: len = 4'd7;  // GOTANDA
            5'b10111: len = 4'd5;  // OSAKI
            5'b11000: len = 4'd9;  // SHINAGAWA
            5'b11001: len = 4'd11; // TAKANAWA GW
            5'b11010: len = 4'd7;  // TAMACHI
            5'b11011: len = 4'd12; // HAMAMATSUCHO
            5'b11100: len = 4'd9;  // SHIMBASHI
            5'b11101: len = 4'd9;  // YURAKUCHO
            default:  len = 4'd5;
        endcase
    end
endmodule
```

- message_mux.v

Verilog

```
`timescale 1ns / 1ps

module message_mux (
    input wire [2:0] msg_ld,
    input wire [127:0] station_name,
    output reg [191:0] msg
);
    always @(*) begin
        case (msg_ld)
            3'b000: msg = {" OFF", "                    "};
            3'b001: msg = {"RIDE TO ", station_name};
            3'b010: msg = {station_name, "        "};
            3'b011: msg = "RIGHT STATION           "; // Exactly 24 characters
            3'b100: msg = "WRONG STATION           "; // Exactly 24 characters
            default: msg = {" OFF", "                    "};
        endcase
    end
endmodule
```

- window_select.v

Verilog

```
module window_select (
    input wire [191:0] msg,
    input wire [4:0] ctr,
    output wire [31:0] disp
);
    wire [7:0] c0 = (ctr < 24) ? msg[(191-ctr*8) -: 8] : " ";
    wire [7:0] c1 = (ctr+1 < 24) ? msg[(191-(ctr+1)*8) -: 8] : " ";
    wire [7:0] c2 = (ctr+2 < 24) ? msg[(191-(ctr+2)*8) -: 8] : " ";
    wire [7:0] c3 = (ctr+3 < 24) ? msg[(191-(ctr+3)*8) -: 8] : " ";

    assign disp = {c3, c2, c1, c0};
endmodule
```

- ascii_to_7seg.v

Verilog

```
`timescale 1ns / 1ps

module ascii_to_7seg (
    input wire [7:0] char,
    output reg [6:0] seg
);
```

```
    always @(*) begin
        case (char)
            "0": seg = 7'b0000001;
            "1": seg = 7'b1001111;
            "2": seg = 7'b0010010;
            "3": seg = 7'b0000110;
            "4": seg = 7'b1001100;
            "5": seg = 7'b0100100;
            "6": seg = 7'b0100000;
            "7": seg = 7'b0001111;
            "8": seg = 7'b0000000;
            "9": seg = 7'b0000100;
            "A": seg = 7'b0001000;
            "B": seg = 7'b1100000;
            "C": seg = 7'b0110001;
            "D": seg = 7'b1000010;
            "E": seg = 7'b0110000;
            "F": seg = 7'b0111000;
            "G": seg = 7'b0100001;
            "H": seg = 7'b1001000;
            "I": seg = 7'b1001111;
            "J": seg = 7'b1000011;
            "K": seg = 7'b1001000;
            "L": seg = 7'b1110001;
            "M": seg = 7'b0001000;
            "N": seg = 7'b1101010;
            "O": seg = 7'b0000001;
            "P": seg = 7'b0011000;
            "Q": seg = 7'b0001100;
            "R": seg = 7'b1111010;
            "S": seg = 7'b0100100;
            "T": seg = 7'b1110000;
            "U": seg = 7'b1000001;
            "V": seg = 7'b1000001;
            "W": seg = 7'b1000001;
            "X": seg = 7'b1001000;
            "Y": seg = 7'b1000100;
            "Z": seg = 7'b0010010;
            "!": seg = 7'b0111111;
            default: seg = 7'b1111111;
        endcase
    end
endmodule
```

- `time_mux_sm.v`

Verilog

```
module time_mux_sm (
    input wire disp_clk,
    input wire rst,
    input wire [6:0] seg_0,
    input wire [6:0] seg_1,
    input wire [6:0] seg_2,
    input wire [6:0] seg_3,
    output reg [6:0] seg,
    output reg [3:0] an
);
    reg [1:0] state, next_state;

    // State transition.
    always @(*) begin
        case (state)
            2'b00:   next_state = 2'b01;
            2'b01:   next_state = 2'b10;
            2'b10:   next_state = 2'b11;
            2'b11:   next_state = 2'b00;
            default: next_state = 2'b00;
        endcase
    end

    // Output multiplexer (segments).
    always @(*) begin
        case (state)
            2'b00:   seg = seg_0;
```

```
            2'b01:   seg = seg_1;
            2'b10:   seg = seg_2;
            2'b11:   seg = seg_3;
            default: seg = 7'b1111111; // all segments off (active-low)
        endcase
    end

    // Output decoder (active-low anodes).
    always @(*) begin
        case (state)
            2'b00:   an = 4'b1110;
            2'b01:   an = 4'b1101;
            2'b10:   an = 4'b1011;
            2'b11:   an = 4'b0111;
            default: an = 4'b1111;
        endcase
    end

    // State register with reset
    always @(posedge disp_clk or posedge rst) begin
        if (rst) begin state <= 2'b00; end
        else begin state <= next_state; end
    end
endmodule
```

2. Part 2 — Integration and Simulation

2.1. AI prompts for every testbench

2.1.1. Controller testbench prompt

Can you write a self-checking Verilog testbench for the following controller state machine module? The intended behavior of the controller FSM is outlined in the `Lab4a.pdf` document, which has been attached to this prompt.

Here is the full Verilog code for the module:

Verilog

```
`timescale 1ns / 1ps

module controller (
    input wire clk,
    input wire rst,
    input wire outedge, // Unlike the skeleton, I chose to detect the rising edge OUTSIDE the
  controller
    input wire sw0,
    input wire counter_done,
    input wire ride_done,
    output reg dir_ld,
    output reg start_ld,
    output reg end_ld,
    output reg [1:0] pos_ld,
    output reg leds_ld,
    output reg [2:0] msg_ld,
    output reg counter_clr,
    output reg counter_en,
    output reg [1:0] comp_ld,
    output reg station_addr_sel
);
    reg [2:0] state, next_state;
    // reg win_reg; // Remembers if player won or lost at exit

    // localparams (inspired by FSM slides from lectures) to keep everything clear
    localparam OFF = 3'b000;
    localparam SELECT = 3'b001;
    localparam RESET1 = 3'b010; // mainly to assert counter_clr
```

```
28      localparam RIDE = 3'b011;
29      localparam RESET2 = 3'b100; // again to clear clock counter
30      localparam CHECK_WIN = 3'b101;
31
32      // always @(posedge clk or posedge rst) begin
33      //      if (rst) begin
34      //          win_reg <= 1'b0;
35      //      end else if (state == RESET2) begin
36      //          win_reg <= ride_done;
37      //      end
38      //  end
39
40      // Next-state logic: each button-edge pulse advances one mode in the cycle
41      always @(*) begin
42          case (state)
43              OFF: next_state = outedge ? SELECT : OFF;
44              SELECT:next_state = outedge ? RESET1 : SELECT;
45              RESET1: next_state = RIDE;
46              RIDE: next_state = outedge ? RESET2 : RIDE;
47              RESET2: next_state = CHECK_WIN;
48              CHECK_WIN: next_state = (counter_done || outedge) ? OFF : CHECK_WIN;
49              default: next_state = OFF;
50          endcase
51      end
52
53      always @(posedge clk or posedge rst) begin
54          if (rst)
55              state <= OFF;
56          else
57              state <= next_state;
58      end
59
60      // Output logic. Moore except for pos_load, which is asserted on the
61      // button edge so the new mode starts at window position 0.
62      // To do: double check msg_ld addresses
63      always @(*) begin
64          dir_ld = 1'b0;
65          start_ld = 1'b0;
66          end_ld = 1'b0;
67          pos_ld = 2'b11;
68          leds_ld = 1'b00;
69          msg_ld = 3'b000;
70          counter_clr = 1'b0;
71          counter_en = 1'b0;
72          comp_ld = 2'b00;
73          station_addr_sel = 1'b0;
74
75          case (state)
76              OFF: begin
77                  dir_ld = 1'b1;
78                  start_ld = 1'b1;
79                  end_ld = 1'b1;
80                  pos_ld = 2'b00;
81                  msg_ld = 3'b000; // off
82                  counter_clr = 1'b1;
83                  comp_ld = 2'b00;
84              end
85
86              SELECT: begin
87                  msg_ld = 3'b001; // ride to
88                  // counter_en = 1'b1;
89                  counter_en = !sw0; // add pause functionality
90                  station_addr_sel = 1'b0;
91                  comp_ld = 2'b00;
92                  if (counter_done)
93                      counter_clr = 1'b1;
94              end
95
96              RESET1: begin
97                  msg_ld = 3'b010; // Station name
98                  pos_ld = 2'b01;
99                  counter_clr = 1'b1;
100                 comp_ld = 2'b01;
101             end
102
```

```
103             RIDE: begin
104                 msg_ld = 3'b010; // station name
105                 leds_ld = 1'b1;
106                 counter_en = !sw0;  // add pause functionality
107                 station_addr_sel = 1'b1;
108                 comp_ld = 2'b01;
109                 pos_ld = 2'b11;
110 
111                 if (counter_done) begin
112                     counter_clr = 1'b1;  // Resets counter for the next station stop
113                     pos_ld = 2'b10;
114                 end
115             end
116 
117             RESET2: begin
118                 counter_clr = 1'b1;
119                 comp_ld = 2'b10;
120                 msg_ld = ride_done ? 3'b011 : 3'b100;
121             end
122 
123             CHECK_WIN: begin
124                 msg_ld = ride_done ? 3'b011 : 3'b100;
125                 counter_en = !sw0;
126                 comp_ld = 2'b10;
127                 if (counter_done)
128                     counter_clr = 1'b1;
129             end
130         endcase
131     end
132 endmodule
```

Requirements for the testbench:

- Generate a 100 MHz clock and proper reset
- Make sure to cycle through every state (either by pulsing outedge for one clock cycle, or in the case of the last state also test if the transition occurs when the `counter_done` control signal is asserted/the message has looped once)
- Make sure to check getting off at both the RIGHT and WRONG station (indicated by `ride_done`) to verify that a different message (indicated by `msg_ld`)

Here is the datapath logic which chooses the start station, end station, and direction so you can know when to get off:

Verilog

```
1  // counters for randomness
2  reg dir_cnt;
3  reg [4:0] start_station_cnt;
4  reg [4:0] end_station_cnt;
5 
6  reg dir_reg;
7  reg [4:0] start_reg, end_reg;
8 
9  always @(posedge clk or posedge rst) begin
10     if (rst) begin
11         dir_cnt <= 1'b0;
12         start_station_cnt <= 5'b00000;
13         end_station_cnt <= 5'b00000;
14     end else begin
15         dir_cnt <= dir_cnt + 1'b1; // Should toggle / overflow back to zero?
16 
17 
18         // i'm adding start and end by different primes to ensure they don't equal each other
19         // station counters from 0 to 29
20         if (start_station_cnt >= 5'b11101) // 29
21             start_station_cnt <= 5'b00000;
22         else
23             start_station_cnt <= start_station_cnt + 5'b00101; // 5
24         if (end_station_cnt >= 5'b11101) // 29
25             end_station_cnt <= 5'b00000;
26         else
```

```
            end_station_cnt <= end_station_cnt + 5'b00111; // 7

    end
end
```

- Also make sure to check that the state of the system “pauses” when the pause switch (`sw0`) is asserted (i.e. that no state transitions occur)
- Make it self checking pretty please :)

2.1.2. Position counter testbench prompt

I need a Verilog testbench for the following position counter module, which keeps track of a position 0 to 29. Based off the `pos_load` control signal, the counter either resets its position to 0, loads initial start value, increments or decrements based on the 1 bit input `dir`, or holds current value.

Here is the full Verilog code for the module:

Verilog

```
`timescale 1ns / 1ps

module pos_counter (
    input wire clk,
    input wire rst,
    input wire [1:0] pos_ld,
    input wire [4:0] start,
    input wire dir,
    output reg [4:0] pos
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            pos <= 5'b00000;
        else begin
            case (pos_ld)
                2'b00: pos <= 5'b00000;
                2'b01: pos <= start;
                2'b10: begin
                    if (dir == 1'b0) begin // Clockwise loop
                        if (pos >= 5'd29)
                            pos <= 5'd0;
                        else
                            pos <= pos + 1'b1;
                    end
                    else begin // Counter-clockwise loop
                        if (pos == 5'd0)
                            pos <= 5'd29;
                        else
                            pos <= pos - 1'b1;
                    end
                end
                2'b11: pos <= pos;
            endcase
        end
    end
endmodule
```

Requirements for the testbench:

- Generate a 100 MHz clock and proper reset
- Test initial reset
- Test loading a random start station
- Verify wraparound in both directions
- Test pos stays stable
- Make it self checking pretty please :)
- Don’t forget reset >:(

2.1.3. Letter decoder testbench prompt

Can you now write a Verilog testbench for the following `ascii_to_7seg` decoder module? The module takes in an 8-bit ascii character and outputs active-low 7-segment display values.

Verilog

```
`timescale 1ns / 1ps

module ascii_to_7seg (
    input wire [7:0] char,
    output reg [6:0] seg
);
    always @(*) begin
        case (char)
            "0": seg = 7'b0000001;
            "1": seg = 7'b1001111;
            "2": seg = 7'b0010010;
            "3": seg = 7'b0000110;
            "4": seg = 7'b1001100;
            "5": seg = 7'b0100100;
            "6": seg = 7'b0100000;
            "7": seg = 7'b0001111;
            "8": seg = 7'b0000000;
            "9": seg = 7'b0000100;
            "A": seg = 7'b0001000;
            "B": seg = 7'b1100000;
            "C": seg = 7'b0110001;
            "D": seg = 7'b1000010;
            "E": seg = 7'b0110000;
            "F": seg = 7'b0111000;
            "G": seg = 7'b0100001;
            "H": seg = 7'b1001000;
            "I": seg = 7'b1001111;
            "J": seg = 7'b1000011;
            "K": seg = 7'b1001000;
            "L": seg = 7'b1110001;
            "M": seg = 7'b0001000;
            "N": seg = 7'b1101010;
            "O": seg = 7'b0000001;
            "P": seg = 7'b0011000;
            "Q": seg = 7'b0001100;
            "R": seg = 7'b1111010;
            "S": seg = 7'b0100100;
            "T": seg = 7'b1110000;
            "U": seg = 7'b1000001;
            "V": seg = 7'b1000001;
            "W": seg = 7'b1000001;
            "X": seg = 7'b1001000;
            "Y": seg = 7'b1000100;
            "Z": seg = 7'b0010010;
            "!": seg = 7'b0111111;
            default: seg = 7'b1111111;
        endcase
    end
endmodule
```

Requirements for the testbench:

- Iterate through a representative set of ASCII characters
- Test an unmapped/invalid character
- Make it self-checking if possible

2.1.4. Integration testbench prompt

Referencing the specification provided in the earlier Lab 4a document, as well as the instructions for Lab 4b that I have attached to this prompt, can you generate a Verilog testbench for my top-level module (`ad_sign_top`)?

Here is the full Verilog code for the top-level module:

Verilog

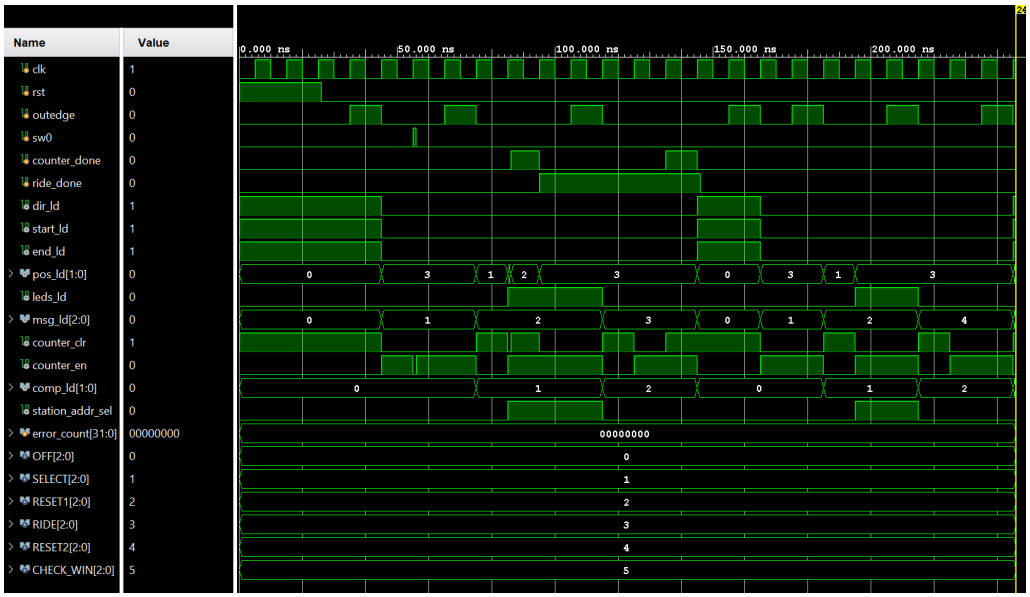
```verilog
`timescale 1ns / 1ps

module ad_sign_top (
    input wire clk,
    input wire rst,
    input wire BTNU,
    input wire [0:0] sw0,
    output wire [6:0] seg,
    output wire [3:0] an,
    output wire [4:0] leds,
    output wire outedge_led
);
    wire outedge;
    // assign outedge_led = outedge; // map led to outedge (debug)
    rising_edge_detector u_red (
        .clk(clk),
        .rst(rst),
        .BTNU(BTNU),
        .sw0(sw0),
        .outedge(outedge)
    );

    // controller + datapath signals
    wire dir_ld;
    wire start_ld;
    wire end_ld;
    wire leds_ld;
    wire counter_clr;
    wire counter_en;
    wire station_addr_sel;
    wire [1:0] pos_ld, comp_ld;
    wire [2:0] msg_ld;
    wire counter_done, ride_done;

    controller u_ctrl (
        .clk(clk),
        .rst(rst),
        .outedge(outedge),
        .sw0(sw0),
        .counter_done(counter_done),
        .ride_done(ride_done),
        .dir_ld(dir_ld),
        .start_ld(start_ld),
        .end_ld(end_ld),
        .pos_ld(pos_ld),
        .leds_ld(leds_ld),
        .msg_ld(msg_ld),
        .counter_clr(counter_clr),
        .counter_en(counter_en),
        .comp_ld(comp_ld),
        .station_addr_sel(station_addr_sel)
    );

    datapath u_dp (
        .clk(clk),
        .rst(rst),
        .dir_ld(dir_ld),
        .start_ld(start_ld),
        .end_ld(end_ld),
        .pos_ld(pos_ld),
        .leds_ld(leds_ld),
        .msg_ld(msg_ld),
        .counter_clr(counter_clr),
        .counter_en(counter_en),
        .comp_ld(comp_ld),
        .station_addr_sel(station_addr_sel),
        .seg(seg),
        .an(an),
        .leds(leds),
        .counter_done(counter_done),
        .ride_done(ride_done)
    );
endmodule
```

The testbench should verify the following integration scenarios:

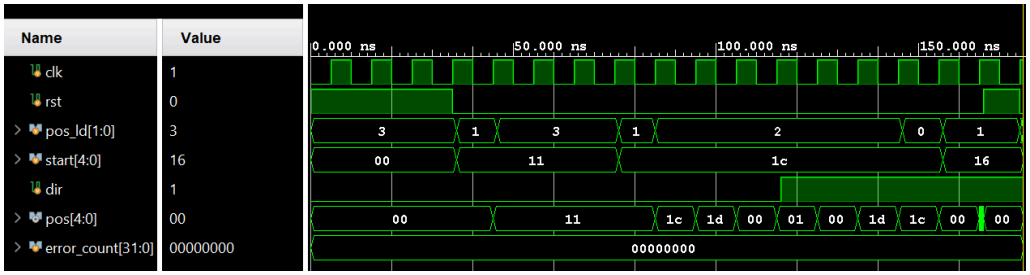
1. OFF –> SELECT: Press BTNU to wake the system from OFF. The sign should display the random target message “RIDE TO “
2. SELECT –> RESET1 –> RIDE: Press BTNU again to start the ride loop
3. RIDE –> Verify that the 7-segment display steps/scrolls through stations while updating the LED pattern to the station number in binary
4. Toggling the pause switch freezes `pos` and `clk_counter` (visible as a static `sseg` sequence)
5. Releasing pause resumes panning from the frozen position — the next position must be the one that would have followed the frozen one, not position 0.
6. WIN –> Press BTNU exactly when the train reaches the target destination station (`ride_done` = 1), verifying the 7 segment display pans “RIGHT STATION”.
7. LOSE –> In a separate test run, press BTNU at the wrong station (`ride_done` = 0), verifying the 7 segment display pans “WRONG STATION”.
8. Return to OFF: Press BTNU on the win/lose screen to return the system to OFF (where the 7-segment display prints “ OFF”). Also, after the win/loss message scrolls fully, the system should automatically return to OFF on its on.

2.2. Unit-test simulation waveforms for the controller, the position counter, and the letter decoder

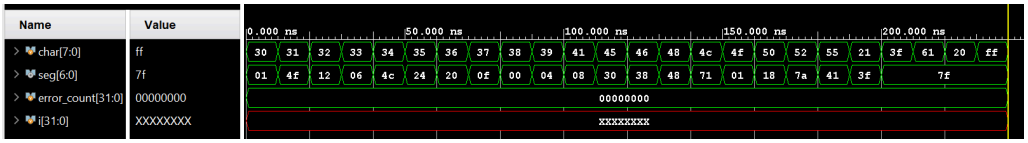
2.2.1. Controller waveform



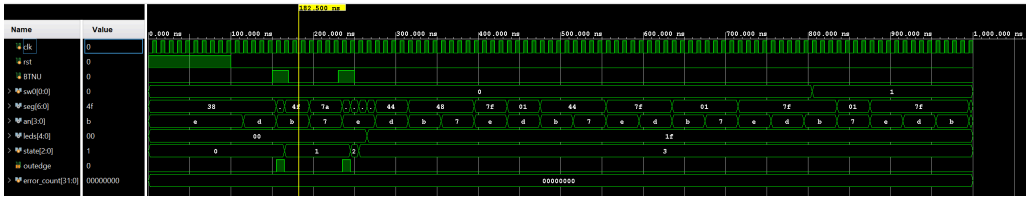
2.2.2. Position counter waveform



2.2.3. Letter decoder waveform



2.3. Integration-test simulation waveform showing all integration scenarios



3. Part 3 — Hardware Deployment

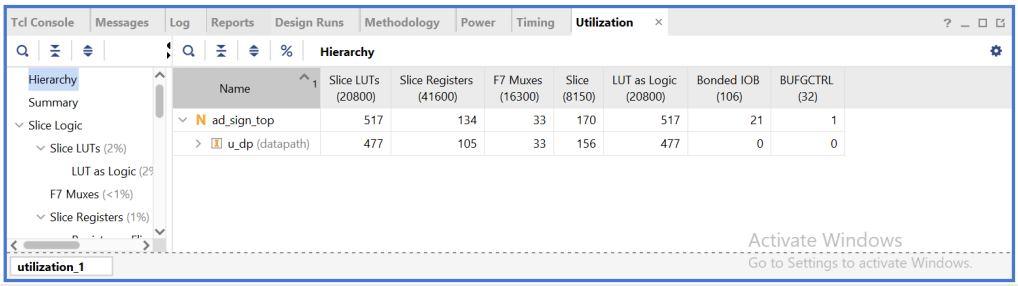
3.1. Final constraints mapping

Port	Pin	Basys3 Component
clk	W5	100 MHz internal clock
BTNU	T18	Button BTNU (mode advance)
sw0	V17	Switch SW0 (pause)
leds[0]	U16	LED LD0
leds[1]	E19	LED LD1
leds[2]	U19	LED LD2
leds[3]	V19	LED LD3
leds[4]	W18	LED LD4
outedge_led	L1	LED LD15 (rising edge debug LED)
an[0]	U2	Anode AN0
an[1]	U4	Anode AN1
an[2]	V4	Anode AN2

Port	Pin	Basys3 Component
an[3]	W4	Anode AN3
seg[6]	W7	Segment a (active low)
seg[5]	W6	Segment b (active low)
seg[4]	U8	Segment c (active low)
seg[3]	V8	Segment d (active low)
seg[2]	U5	Segment e (active low)
seg[1]	V5	Segment f (active low)
seg[0]	U7	Segment g (active low)

3.2. Synthesis and implementation reports

3.2.1. Utilization summary



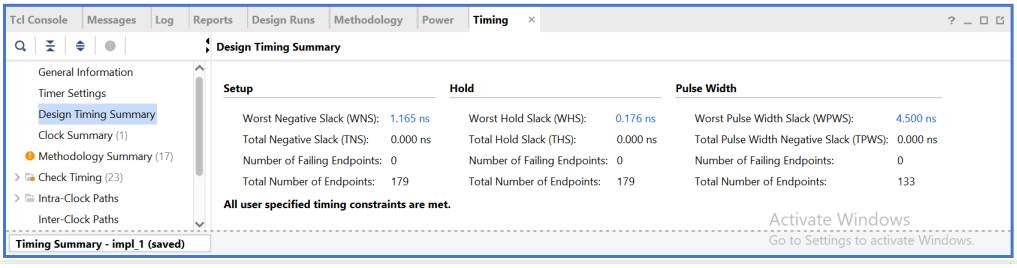

```
1 Copyright 1986-2022 Xilinx, Inc. All Rights Reserved. Copyright 2022-2025 Advanced Micro Devices,
  Inc. All Rights Reserved.
2 ----------------------------------------------------------------------------------------------------
3 | Tool Version : Vivado v.2025.2 (win64) Build 6299465 Fri Nov 14 19:35:11 GMT 2025
4 | Date         : Sun Jul 26 00:10:53 2026
5 | Host         : DESKTOP-G9KGS0U running 64-bit major release  (build 9200)
6 | Command      : report_utilization -file ad_sign_top_utilization_synth.rpt -pb
  ad_sign_top_utilization_synth.pb
7 | Design       : ad_sign_top
8 | Device       : xc7a35tcpg236-1
9 | Speed File   : -1
10 | Design State : Synthesized
11 ----------------------------------------------------------------------------------------------------
12
13 Utilization Design Information
14
15 Table of Contents
16 -----------------
17 1. Slice Logic
18 1.1 Summary of Registers by Type
19 2. Memory
20 3. DSP
21 4. IO and GT Specific
22 5. Clocking
23 6. Specific Feature
24 7. Primitives
25 8. Black Boxes
26 9. Instantiated Netlists
27
28 1. Slice Logic
29 --------------
30
31 +-------------------------+------+-------+------------+-----------+-------+
```

```
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             |  521 |     0 |          0 |     20800 |  2.50 |
|   LUT as Logic          |  521 |     0 |          0 |     20800 |  2.50 |
|   LUT as Memory         |    0 |     0 |          0 |      9600 |  0.00 |
| Slice Registers         |  134 |     0 |          0 |     41600 |  0.32 |
|   Register as Flip Flop |  134 |     0 |          0 |     41600 |  0.32 |
|   Register as Latch     |    0 |     0 |          0 |     41600 |  0.00 |
| F7 Muxes                |   33 |     0 |          0 |     16300 |  0.20 |
| F8 Muxes                |    0 |     0 |          0 |      8150 |  0.00 |
| Unique Control Sets     |    7 |       |          0 |      8150 |  0.09 |
+-------------------------+------+-------+------------+-----------+-------+
* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a more realistic count.
Warning! LUT value is adjusted to account for LUT combining.
Warning! For any ECO changes, please run place_design if there are unplaced instances
** Note: Available Control Sets calculated as Slice * 1, Review the Control Sets Report for more information regarding control sets.


1.1 Summary of Registers by Type
--------------------------------

+-------+--------------+-------------+--------------+
| Total | Clock Enable | Synchronous | Asynchronous |
+-------+--------------+-------------+--------------+
| 0     |            _ |           - |            - |
| 0     |            _ |           - |          Set |
| 0     |            _ |           - |        Reset |
| 0     |            _ |         Set |            - |
| 0     |            _ |       Reset |            - |
| 0     |          Yes |           - |            - |
| 1     |          Yes |           - |          Set |
| 88    |          Yes |           - |        Reset |
| 0     |          Yes |         Set |            - |
| 45    |          Yes |       Reset |            - |
+-------+--------------+-------------+--------------+


2. Memory
---------

+----------------+------+-------+------------+-----------+-------+
|    Site Type   | Used | Fixed | Prohibited | Available | Util% |
+----------------+------+-------+------------+-----------+-------+
| Block RAM Tile |    0 |     0 |          0 |        50 |  0.00 |
|   RAMB36/FIFO* |    0 |     0 |          0 |        50 |  0.00 |
|   RAMB18       |    0 |     0 |          0 |       100 |  0.00 |
+----------------+------+-------+------------+-----------+-------+
* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1


3. DSP
------

+-----------+------+-------+------------+-----------+-------+
| Site Type | Used | Fixed | Prohibited | Available | Util% |
+-----------+------+-------+------------+-----------+-------+
| DSPs      |    0 |     0 |          0 |        90 |  0.00 |
+-----------+------+-------+------------+-----------+-------+


4. IO and GT Specific
---------------------

+-----------------------------+------+-------+------------+-----------+-------+
|          Site Type          | Used | Fixed | Prohibited | Available | Util% |
+-----------------------------+------+-------+------------+-----------+-------+
| Bonded IOB                  |   21 |     0 |          0 |       106 | 19.81 |
| Bonded IPADs                |    0 |     0 |          0 |        10 |  0.00 |
| Bonded OPADs                |    0 |     0 |          0 |         4 |  0.00 |
| PHY_CONTROL                 |    0 |     0 |          0 |         5 |  0.00 |
| PHASER_REF                  |    0 |     0 |          0 |         5 |  0.00 |
```

```
| OUT_FIFO                    |    0 |     0 |          0 |        20 |  0.00 |
| IN_FIFO                     |    0 |     0 |          0 |        20 |  0.00 |
| IDELAYCTRL                  |    0 |     0 |          0 |         5 |  0.00 |
| IBUFDS                      |    0 |     0 |          0 |       104 |  0.00 |
| GTPE2_CHANNEL               |    0 |     0 |          0 |         2 |  0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |          0 |        20 |  0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |          0 |        20 |  0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |          0 |       250 |  0.00 |
| IBUFDS_GTE2                 |    0 |     0 |          0 |         2 |  0.00 |
| ILOGIC                      |    0 |     0 |          0 |       106 |  0.00 |
| OLOGIC                      |    0 |     0 |          0 |       106 |  0.00 |
+-----------------------------+------+-------+------------+-----------+-------+


5. Clocking
-----------

+------------+------+-------+------------+-----------+-------+
|  Site Type | Used | Fixed | Prohibited | Available | Util% |
+------------+------+-------+------------+-----------+-------+
| BUFGCTRL   |    1 |     0 |          0 |        32 |  3.13 |
| BUFIO      |    0 |     0 |          0 |        20 |  0.00 |
| MMCME2_ADV |    0 |     0 |          0 |         5 |  0.00 |
| PLLE2_ADV  |    0 |     0 |          0 |         5 |  0.00 |
| BUFMRCE    |    0 |     0 |          0 |        10 |  0.00 |
| BUFHCE     |    0 |     0 |          0 |        72 |  0.00 |
| BUFR       |    0 |     0 |          0 |        20 |  0.00 |
+------------+------+-------+------------+-----------+-------+


6. Specific Feature
-------------------

+-------------+------+-------+------------+-----------+-------+
|  Site Type  | Used | Fixed | Prohibited | Available | Util% |
+-------------+------+-------+------------+-----------+-------+
| BSCANE2     |    0 |     0 |          0 |         4 |  0.00 |
| CAPTUREE2   |    0 |     0 |          0 |         1 |  0.00 |
| DNA_PORT    |    0 |     0 |          0 |         1 |  0.00 |
| EFUSE_USR   |    0 |     0 |          0 |         1 |  0.00 |
| FRAME_ECCE2 |    0 |     0 |          0 |         1 |  0.00 |
| ICAPE2      |    0 |     0 |          0 |         2 |  0.00 |
| PCIE_2_1    |    0 |     0 |          0 |         1 |  0.00 |
| STARTUPE2   |    0 |     0 |          0 |         1 |  0.00 |
| XADC        |    0 |     0 |          0 |         1 |  0.00 |
+-------------+------+-------+------------+-----------+-------+


7. Primitives
-------------

+----------+------+---------------------+
| Ref Name | Used | Functional Category |
+----------+------+---------------------+
| LUT6     |  367 |                 LUT |
| LUT5     |   94 |                 LUT |
| FDCE     |   88 |        Flop & Latch |
| LUT4     |   61 |                 LUT |
| FDRE     |   45 |        Flop & Latch |
| LUT3     |   43 |                 LUT |
| MUXF7    |   33 |              MuxFx |
| CARRY4   |   22 |          CarryLogic |
| OBUF     |   16 |                  IO |
| LUT2     |   16 |                 LUT |
| LUT1     |    5 |                 LUT |
| IBUF     |    4 |                  IO |
| OBUFT    |    1 |                  IO |
| FDPE     |    1 |        Flop & Latch |
| BUFG     |    1 |               Clock |
+----------+------+---------------------+


8. Black Boxes
--------------

```

```
178 +----------+------+
179 | Ref Name | Used |
180 +----------+------+
181
182
183 9. Instantiated Netlists
184 -------------------------
185
186 +----------+------+
187 | Ref Name | Used |
188 +----------+------+
189
190
191 ----
```

3.2.2. Timing Summary



3.3. Which part of Lab 4a's plan changed during implementation, and why?

For the most part, Lab 4a's plan was followed fairly closely while implementing the system in Verilog. The top-level controller state machine, core datapath structure, and overall interfaces remained intact. Only a few small changes were made to simplify the development of the Verilog code. For example, instead of having a whole seperate magnitude comparator module (`mag_cmp`) and instantiating it in our datapath, a logical operator was used instead[1]. Similarily, I replaced the LFSRs with clock counters for randomness, as the clock cycles so fast that when the user presses at any given moment the current value of the start and end station (as well as direction) appear random[2]. This has the effect of simplifying things, as well as making the testbench deterministic. Other minor implementation changes include changing the addresses used to select the current message (collapsing everything into one mux instead of two) and modifying the port list to add `sw0` as an input to the `controller` so that pause functionality could be implemented.

There were also some minor timing issues that I noticed and proactively fixed. For example, a `rst` signal was added to the port list of many different modules (like the rising_edge_detector FSM) to ensure the inital state would be correct. Also, the wrapround check was moved to when `pos` was actively incremented or decremented to prevent it from being 31 for one clock cycle[3]. Moreover, a new `rising_edge_detector` instance was added to `clk_counter` so it wouldn't count for millions of clock cycles every time slow clock went high. In the same manner, the `slow_clk` count was reset at the same time as the `clk_counter` to prevent the system from being in the middle of a `slow_clk` cycle during a transition (which would cause the first character to scroll by very fast).

[1] As seen in lines like `assign counter_done = (clk_ctr == comp_target)`
[2] Prime numbers are used so the start doesn't equal the end
[3] Even using a modulo wouldn't work since −1 underflows to 31