CSC258H1: Pre-lab Exercise for Lab 6

Wang, Weiqing

Due: March 3rd, 2018 before 12 a.m.

1 Part I

1. Answer the following questions in your prelab: given the starter code, is the `resetn` signal is an synchronous or asynchronous reset? Is it active high, or active low? Given this, what do you have to do in simulation to reset the FSM to the starting state?

Solution.

The `resetn` is synchronous. It's active low. The clock can be set high throughout the simulation process.

2. Complete the state table showing how the present state and input value determine the next state and the output value. Fill in all the missing parts of the template code to implement the FSM based on the state table you derived. Include completed code in your prelab.

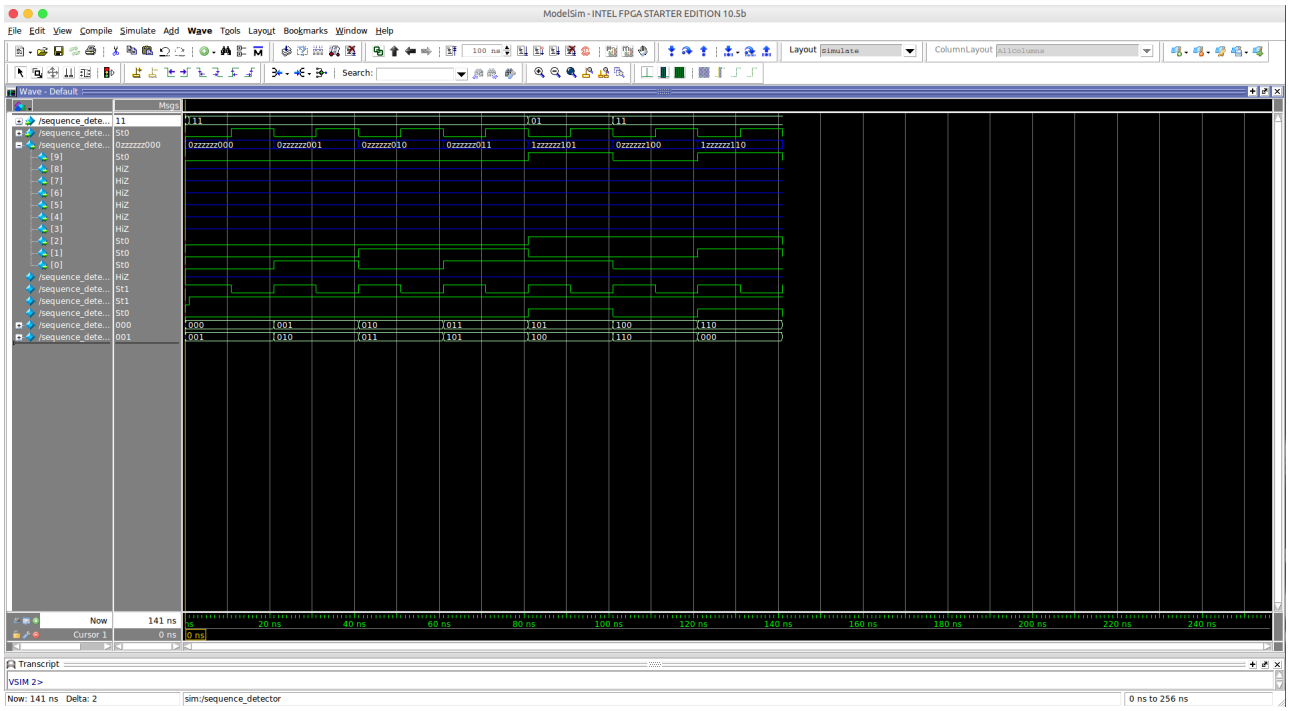
Solution.

```
module sequence_detector(SW, KEY, LEDR);
    input [1:0] SW;     //SW[0]: reset signal; SW[1]: input signal (w).
    input [3:0] KEY;    // KEY[0]: KEY[0]
    output [9:0] LEDR; // LEDR[2:0]: Current State; LEDR[9]: Output.
    reg [2:0] y_Q, Y_D;

    localparam A = 3'b000,
               B = 3'b001,
               C = 3'b010,
               D = 3'b011,
               E = 3'b100,
               F = 3'b101,
               G = 3'b110;
    assign LEDR[2:0] = y_Q;

    always @(*)
    begin
        case (y_Q)
            A: Y_D = SW[1] ? B : A;
            B: Y_D = SW[1] ? C : A;
            C: Y_D = SW[1] ? D : E;
            D: Y_D = SW[1] ? F : E;
            E: Y_D = SW[1] ? G : A;
            F: Y_D = SW[1] ? F : E;
            G: Y_D = SW[1] ? C : A;
            default: Y_D = A;
       endcase
    end

    always @(negedge KEY[0], negedge SW[0])
   begin
        if (SW[0] == 1'b0)
            y_Q <= A;
        else
            y_Q <= Y_D;
    end
```

```
37      assign LEDR[9] = ((y_Q == F) || (y_Q == G));
38  endmodule
```

3. Simulate your circuit using ModelSim for a variety of input settings, ensuring the output waveforms are correct. Include a few screenshots that shows the simulation output.

Solution.

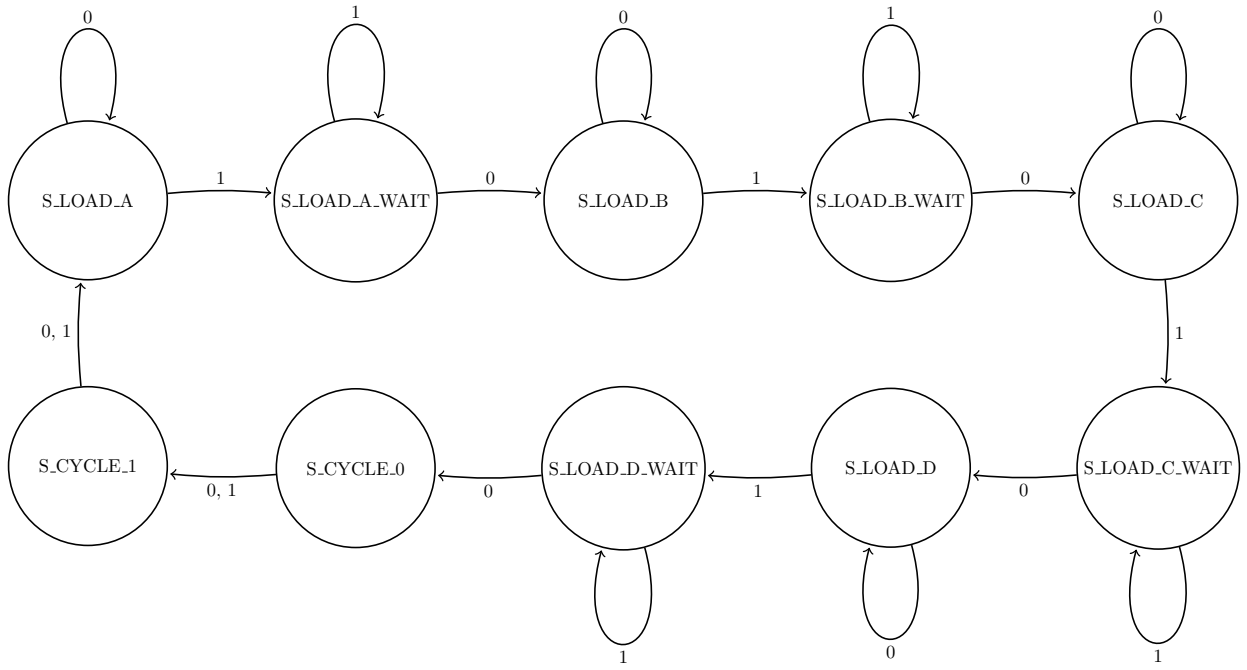


The above simulation guided the Finite State Machine to states $A \to B \to C \to D \to F \to E \to G \to A$.

2 Part II

1. Draw a state diagram for your controller starting with the register load states provided in the example FSM. Include the state diagram in your prelab.

Solution.



2. Modify the provided FSM code to implement your controller and synthesize it. You should only modify the control module. Include your modified code in the prelab.

Solution.

```
module poly_function(SW, KEY, CLOCK_50, LEDR, HEX0, HEX1);
    input [9:0] SW;
    input [3:0] KEY;
    input CLOCK_50;
    output [9:0] LEDR;
    output [6:0] HEX0, HEX1;

    wire resetn;
    wire go;

    wire [7:0] data_result;
    assign go = ~KEY[1];
    assign resetn = KEY[0];

    part2 u0(
        .clk(CLOCK_50),
        .resetn(resetn),
        .go(go),
        .data_in(SW[7:0]),
        .data_result(data_result)
    );

```

```
23      assign LEDR[9:0] = {2'b00, data_result};
24
25      hex_decoder H0(
26          .hex_digit(data_result[3:0]),
27          .segments(HEX0)
28          );
29
30      hex_decoder H1(
31          .hex_digit(data_result[7:4]),
32          .segments(HEX1)
33          );
34
35  endmodule
36
37  module part2(
38      input clk,
39      input resetn,
40      input go,
41      input [7:0] data_in,
42      output [7:0] data_result
43      );
44
45      // lots of wires to connect our datapath and control
46      wire ld_a, ld_b, ld_c, ld_x, ld_r;
47      wire ld_alu_out;
48      wire [1:0]  alu_select_a, alu_select_b;
49      wire alu_op;
50
51      control C0(
52          .clk(clk),
53          .resetn(resetn),
54
55          .go(go),
56
57          .ld_alu_out(ld_alu_out),
58          .ld_x(ld_x),
59          .ld_a(ld_a),
60          .ld_b(ld_b),
61          .ld_c(ld_c),
62          .ld_r(ld_r),
63
64          .alu_select_a(alu_select_a),
65          .alu_select_b(alu_select_b),
66          .alu_op(alu_op)
67      );
68
69      datapath D0(
70          .clk(clk),
71          .resetn(resetn),
72
73          .ld_alu_out(ld_alu_out),
74          .ld_x(ld_x),
75          .ld_a(ld_a),
76          .ld_b(ld_b),
77          .ld_c(ld_c),
```

```
        .ld_r(ld_r),

        .alu_select_a(alu_select_a),
        .alu_select_b(alu_select_b),
        .alu_op(alu_op),

        .data_in(data_in),
        .data_result(data_result)
    );

endmodule


module control(
    input clk,
    input resetn,
    input go,

    output reg  ld_a, ld_b, ld_c, ld_x, ld_r,
    output reg  ld_alu_out,
    output reg [1:0]  alu_select_a, alu_select_b,
    output reg alu_op
    );

    reg [3:0] current_state, next_state;

    localparam  S_LOAD_A        = 4'd0,
                S_LOAD_A_WAIT   = 4'd1,
                S_LOAD_B        = 4'd2,
                S_LOAD_B_WAIT   = 4'd3,
                S_LOAD_C        = 4'd4,
                S_LOAD_C_WAIT   = 4'd5,
                S_LOAD_X        = 4'd6,
                S_LOAD_X_WAIT   = 4'd7,
                S_CYCLE_0       = 4'd8,
                S_CYCLE_1       = 4'd9,
                S_CYCLE_2       = 4'd10,
                S_CYCLE_3       = 4'd11,
                S_CYCLE_4       = 4'd12;

    // Next state logic aka our state table
    always@(*)
    begin: state_table
            case (current_state)
                S_LOAD_A: next_state = go ? S_LOAD_A_WAIT : S_LOAD_A;
                S_LOAD_A_WAIT: next_state = go ? S_LOAD_A_WAIT : S_LOAD_B;
                S_LOAD_B: next_state = go ? S_LOAD_B_WAIT : S_LOAD_B;
                S_LOAD_B_WAIT: next_state = go ? S_LOAD_B_WAIT : S_LOAD_C;
                S_LOAD_C: next_state = go ? S_LOAD_C_WAIT : S_LOAD_C;
                S_LOAD_C_WAIT: next_state = go ? S_LOAD_C_WAIT : S_LOAD_X;
                S_LOAD_X: next_state = go ? S_LOAD_X_WAIT : S_LOAD_X;
                S_LOAD_X_WAIT: next_state = go ? S_LOAD_X_WAIT : S_CYCLE_0;
                S_CYCLE_0: next_state = S_CYCLE_1;
                S_CYCLE_1: next_state = S_CYCLE_2;
                S_CYCLE_2: next_state = S_CYCLE_3;
```

```
                S_CYCLE_3: next_state = S_CYCLE_4;
                S_CYCLE_4: next_state = S_LOAD_A;
                default:   next_state = S_LOAD_A;
        endcase
    end // state_table

    // Output logic aka all of our datapath control signals
    always @(*)
    begin: enable_signals
        // By default make all our signals 0
        ld_alu_out = 1'b0;
        ld_a = 1'b0;
        ld_b = 1'b0;
        ld_c = 1'b0;
        ld_x = 1'b0;
        ld_r = 1'b0;
        alu_select_a = 2'b00;
        alu_select_b = 2'b00;
        alu_op       = 1'b0;

        case (current_state)
            S_LOAD_A: begin
                ld_a = 1'b1;
                end
            S_LOAD_B: begin
                ld_b = 1'b1;
                end
            S_LOAD_C: begin
                ld_c = 1'b1;
                end
            S_LOAD_X: begin
                ld_x = 1'b1;
                end
            S_CYCLE_0: begin // RB <- BX
                ld_alu_out = 1'b1; ld_b = 1'b1; // store result back into RB
                alu_select_a = 2'b01; // Select register B
                alu_select_b = 2'b11; // Select register x
                alu_op = 1'b1; // Multiply operation     Bx
            end
            S_CYCLE_1: begin // RB <- Bx + A
                ld_alu_out = 1'b1; ld_b = 1'b1; // store result back into RB
                alu_select_a = 2'b01; // Select register B
                alu_select_b = 2'b00; // Select register A
                alu_op = 1'b0; // Multiply operation         Bx + A
            end
            S_CYCLE_2: begin // RA <- CX
                ld_alu_out = 1'b1; ld_a = 1'b1;// store result back into RA
                alu_select_a = 2'b10; // Select register C
                alu_select_b = 2'b11; // Select register X
                alu_op = 1'b1; // Multiply operation      CX
            end
            S_CYCLE_3: begin // RA <- CX^2
                ld_alu_out = 1'b1; ld_a = 1'b1; // store result back into RB
                alu_select_a = 2'b00; // Select register A       CX
                alu_select_b = 2'b11; // Select register X     x
```

```
                alu_op = 1'b1; // Addition operation              CX^2
            end

            S_CYCLE_4: begin    //r <- Cx^2 + Bx + A
                ld_r = 1'b1; // store result in result register
                alu_select_a = 2'b01; // Select register B
                alu_select_b = 2'b00; // Select register A
                alu_op = 1'b0; //Addition              Cx^2 + Bx + A
            end
        endcase
    end // enable_signals

    // current_state registers
    always@(posedge clk)
    begin: state_FFs
        if(!resetn)
            current_state <= S_LOAD_A;
        else
            current_state <= next_state;
    end // state_FFS
endmodule

module datapath(
    input clk,
    input resetn,
    input [7:0] data_in,
    input ld_alu_out,
    input ld_x, ld_a, ld_b, ld_c,
    input ld_r,
    input alu_op,
    input [1:0] alu_select_a, alu_select_b,
    output reg [7:0] data_result
    );

    // input registers
    reg [7:0] a, b, c, x;

    // output of the alu
    reg [7:0] alu_out;
    // alu input muxes
    reg [7:0] alu_a, alu_b;

    // Registers a, b, c, x with respective input logic
    always @ (posedge clk) begin
        if (!resetn) begin
            a <= 8'd0;
            b <= 8'd0;
            c <= 8'd0;
            x <= 8'd0;
        end
        else begin
            if (ld_a)
                a <= ld_alu_out ? alu_out : data_in;
                // load alu_out if load_alu_out signal is high, otherwise load from dat
            if (ld_b)
```

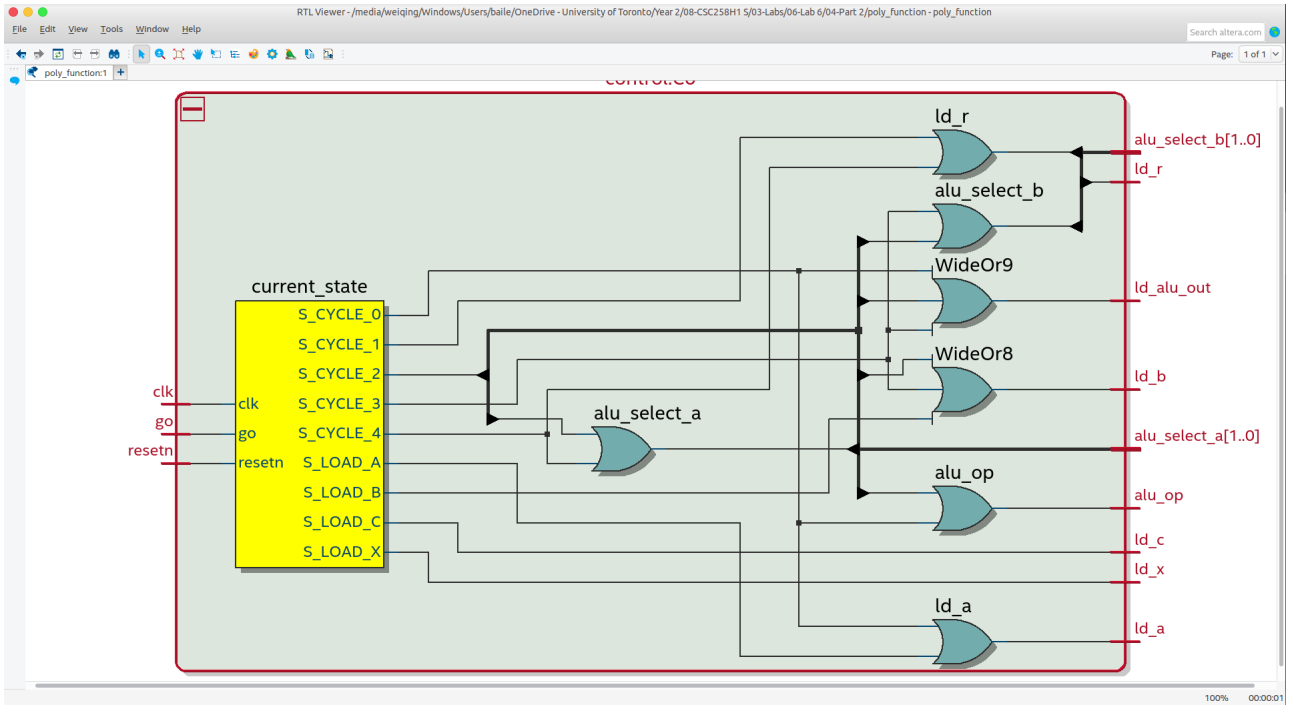
```
                b <= ld_alu_out ? alu_out : data_in;
                // load alu_out if load_alu_out signal is high, otherwise load from dat
            if (ld_x)
                x <= data_in;
            if (ld_c)
                c <= data_in;
        end
    end

    // Output result register
    always @ (posedge clk) begin
        if (!resetn)
            begin
                data_result <= 8'd0;
            end
        else if (ld_r)
            data_result <= alu_out;
    end

    // The ALU input multiplexers
    always @(*)
    begin
        case (alu_select_a)
            2'd0: alu_a = a;
            2'd1: alu_a = b;
            2'd2: alu_a = c;
            2'd3: alu_a = x;
            default: alu_a = 8'd0;
        endcase

        case (alu_select_b)
            2'd0: alu_b = a;
            2'd1: alu_b = b;
            2'd2: alu_b = c;
            2'd3: alu_b = x;
            default: alu_b = 8'd0;
        endcase
    end

    // The ALU
    always @(*)
    begin : ALU
        case (alu_op)
            0: begin
                    alu_out = alu_a + alu_b; //performs addition
               end
            1: begin
                    alu_out = alu_a * alu_b; //performs multiplication
               end
            default: alu_out = 8'd0;
        endcase
    end

endmodule

```

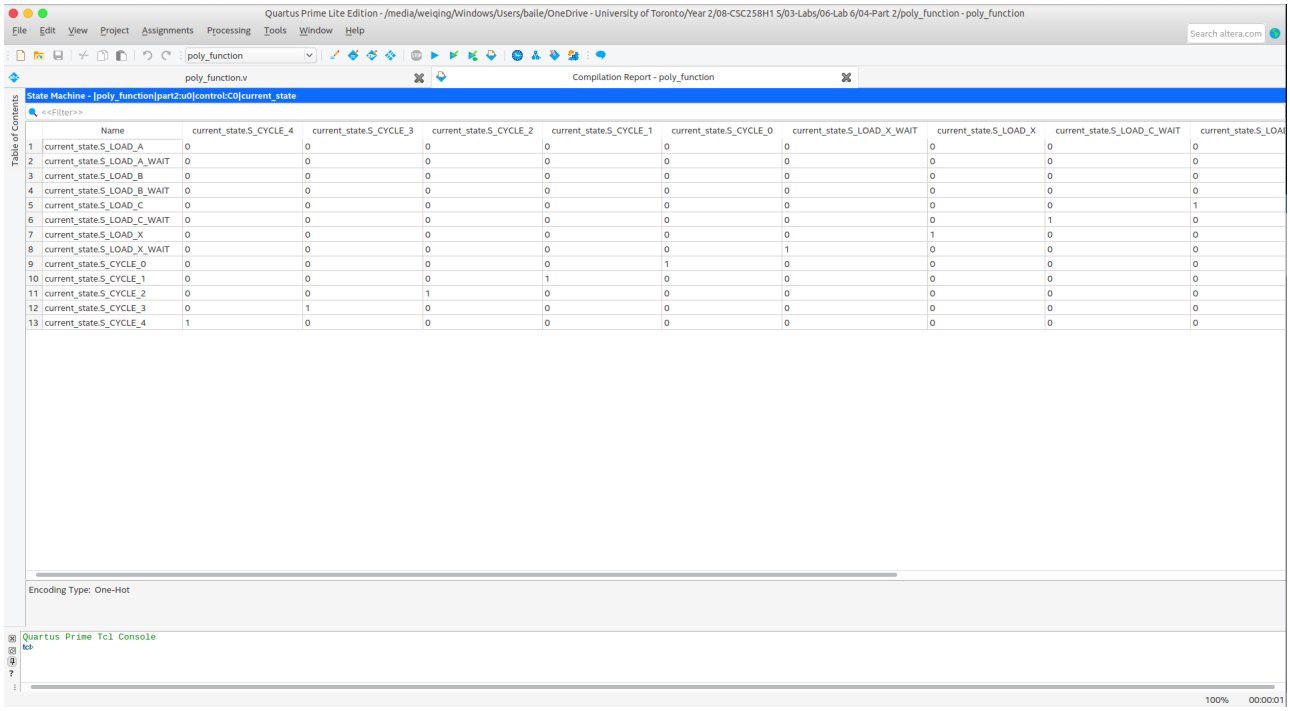
```
298 module hex_decoder(hex_digit, segments);
299     input [3:0] hex_digit;
300     output reg [6:0] segments;
301 
302     always @(*)
303         case (hex_digit)
304             4'h0: segments = 7'b100_0000;
305             4'h1: segments = 7'b111_1001;
306             4'h2: segments = 7'b010_0100;
307             4'h3: segments = 7'b011_0000;
308             4'h4: segments = 7'b001_1001;
309             4'h5: segments = 7'b001_0010;
310             4'h6: segments = 7'b000_0010;
311             4'h7: segments = 7'b111_1000;
312             4'h8: segments = 7'b000_0000;
313             4'h9: segments = 7'b001_1000;
314             4'hA: segments = 7'b000_1000;
315             4'hB: segments = 7'b000_0011;
316             4'hC: segments = 7'b100_0110;
317             4'hD: segments = 7'b010_0001;
318             4'hE: segments = 7'b000_0110;
319             4'hF: segments = 7'b000_1110;
320             default: segments = 7'h7f;
321         endcase
322 endmodule
```

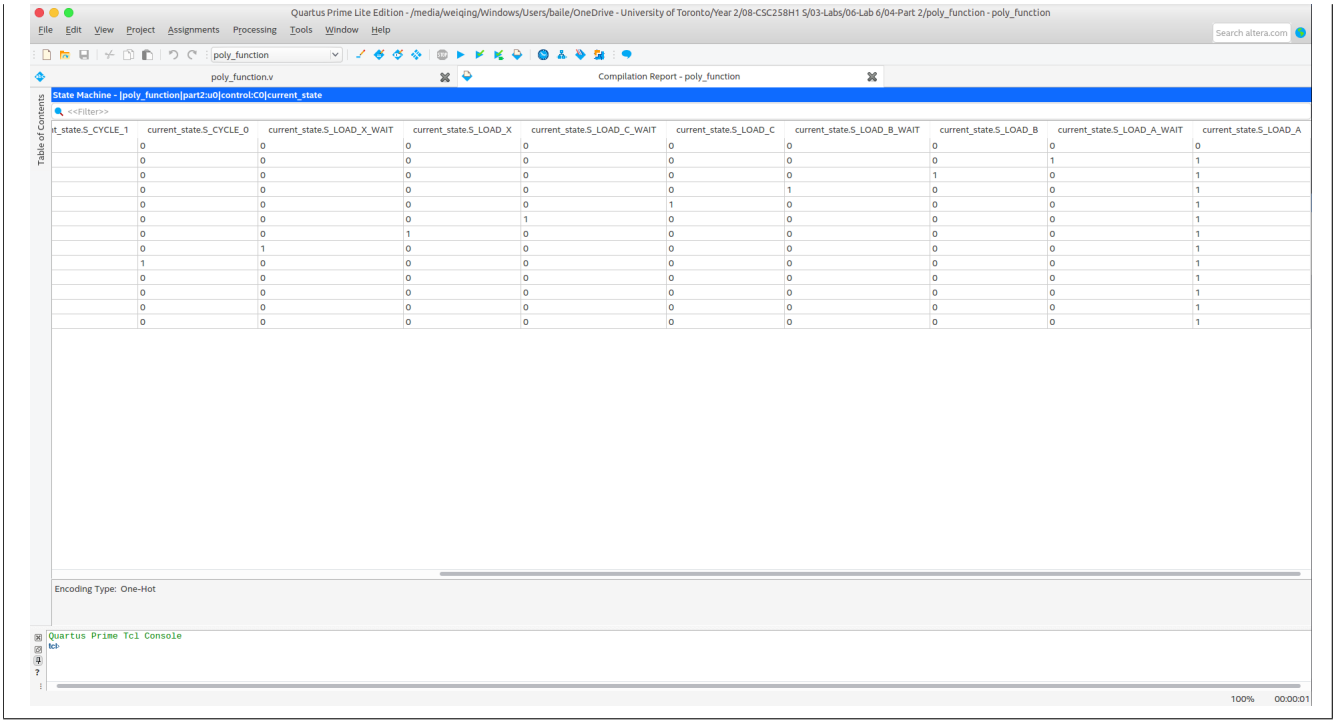
3. To examine the circuit produced by Quartus Prime open the RTL Viewer tool (Tools > Netlist Viewers > RTL Viewer). Find (on the left panel) and double-click on the box shown in the circuit that represents the finite state machine, and determine whether the state diagram that it shows properly corresponds to the one you have drawn. To see the state codes used for your FSM, open the Compilation Report, select the `Analysis and Synthesis` section of the report, and click on State Machines. Include a screenshot of the generated FSM in your prelab.

Solution.

See the screenshots below







4. Simulate your circuit with ModelSim for a variety of input settings, ensuring the output waveforms are correct. It is recommended that you start by simulating the datapath and controller modules separately. Only when you are satisfied that they are working individually should you combine them into the full design. Why is this approach better? (Hint: Consider the case when your design has 20 different modules.) Include few screenshots of simulation output in your prelab.

Solution.

See the screenshot below