

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.

```

1 module sequence_detector(SW, KEY, LEDR);
2     input [1:0] SW;    //SW[0]: reset signal; SW[1]: input signal (w).
3     input [3:0] KEY;   // KEY[0]: KEY[0]
4     output [9:0] LEDR; // LEDR[2:0]: Current State; LEDR[9]: Output.
5     reg [2:0] y_Q, Y_D;
6
7     localparam A = 3'b000,
8                 B = 3'b001,
9                 C = 3'b010,
10                D = 3'b011,
11                E = 3'b100,
12                F = 3'b101,
13                G = 3'b110;
14     assign LEDR[2:0] = y_Q;
15
16     always @(*)
17     begin
18         case (y_Q)
19             A: Y_D = SW[1] ? B : A;
20             B: Y_D = SW[1] ? C : A;
21             C: Y_D = SW[1] ? D : E;
22             D: Y_D = SW[1] ? F : E;
23             E: Y_D = SW[1] ? G : A;
24             F: Y_D = SW[1] ? F : E;
25             G: Y_D = SW[1] ? C : A;
26             default: Y_D = A;
27         endcase
28     end
29
30     always @(negedge KEY[0], negedge SW[0])
31     begin
32         if (SW[0] == 1'b0)
33             y_Q <= A;
34         else
35             y_Q <= Y_D;
36     end

```

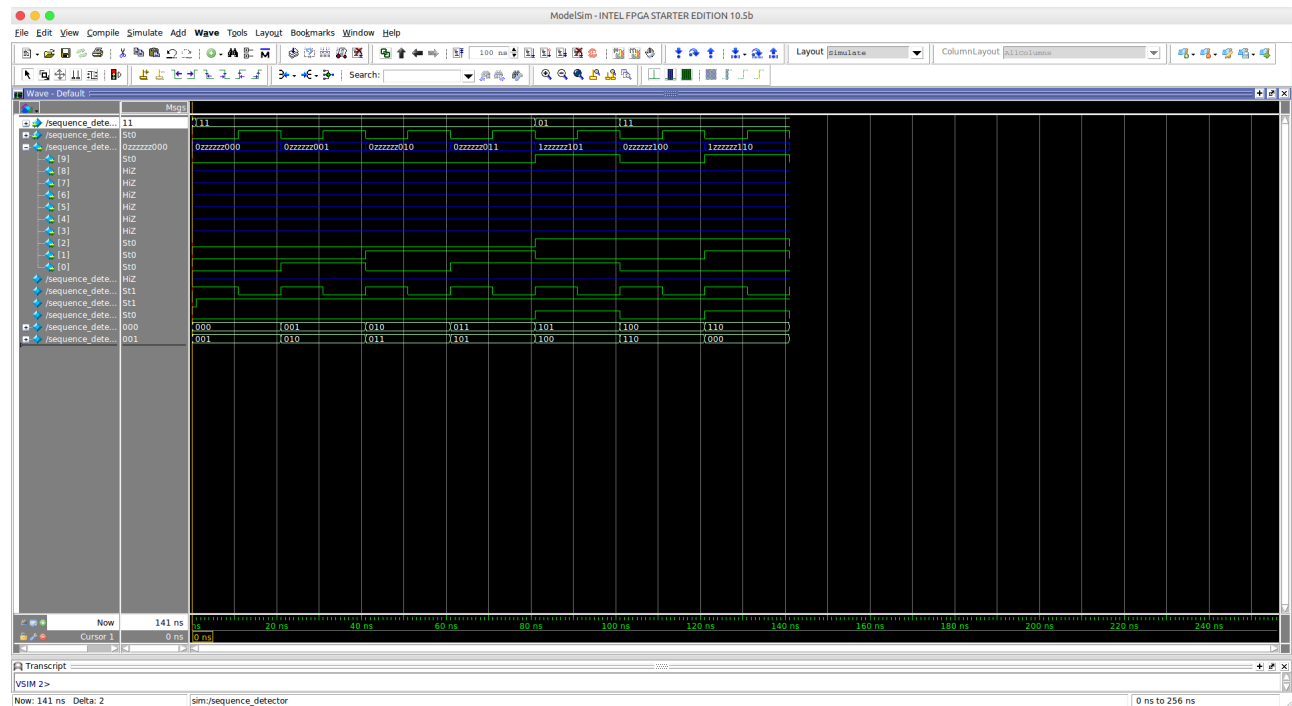
```

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.

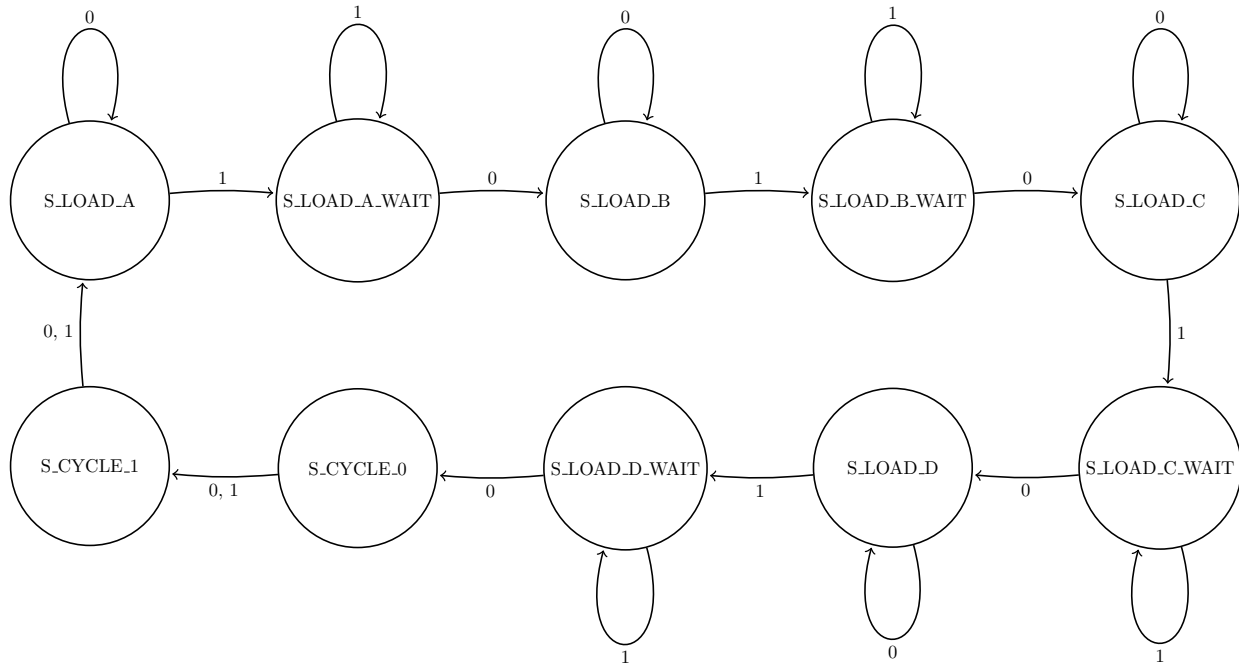


The above simulation guided the Finite State Machine to states $A \rightarrow B \rightarrow C \rightarrow D \rightarrow F \rightarrow E \rightarrow G \rightarrow A$.

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.

```

1 module poly_function(SW, KEY, CLOCK_50, LEDR, HEX0, HEX1);
2     input [9:0] SW;
3     input [3:0] KEY;
4     input CLOCK_50;
5     output [9:0] LEDR;
6     output [6:0] HEX0, HEX1;
7
8     wire resetn;
9     wire go;
10
11     wire [7:0] data_result;
12     assign go = ~KEY[1];
13     assign resetn = KEY[0];
14
15     part2 u0(
16         .clk(CLOCK_50),
17         .resetn(resetn),
18         .go(go),
19         .data_in(SW[7:0]),
20         .data_result(data_result)
21     );
22

```

```

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),

```

```

78     .ld_r(ld_r),
79
80     .alu_select_a(alu_select_a),
81     .alu_select_b(alu_select_b),
82     .alu_op(alu_op),
83
84     .data_in(data_in),
85     .data_result(data_result)
86 );
87
88 endmodule
89
90
91 module control(
92     input clk,
93     input resetn,
94     input go,
95
96     output reg ld_a, ld_b, ld_c, ld_x, ld_r,
97     output reg ld_alu_out,
98     output reg [1:0] alu_select_a, alu_select_b,
99     output reg alu_op
100 );
101
102 reg [3:0] current_state, next_state;
103
104 localparam S_LOAD_A          = 4'd0,
105            S_LOAD_A_WAIT     = 4'd1,
106            S_LOAD_B          = 4'd2,
107            S_LOAD_B_WAIT     = 4'd3,
108            S_LOAD_C          = 4'd4,
109            S_LOAD_C_WAIT     = 4'd5,
110            S_LOAD_X          = 4'd6,
111            S_LOAD_X_WAIT     = 4'd7,
112            S_CYCLE_0         = 4'd8,
113            S_CYCLE_1         = 4'd9,
114            S_CYCLE_2         = 4'd10,
115            S_CYCLE_3         = 4'd11,
116            S_CYCLE_4         = 4'd12;
117
118 // Next state logic aka our state table
119 always@(*)
120 begin: state_table
121     case (current_state)
122         S_LOAD_A: next_state = go ? S_LOAD_A_WAIT : S_LOAD_A;
123         S_LOAD_A_WAIT: next_state = go ? S_LOAD_A_WAIT : S_LOAD_B;
124         S_LOAD_B: next_state = go ? S_LOAD_B_WAIT : S_LOAD_B;
125         S_LOAD_B_WAIT: next_state = go ? S_LOAD_B_WAIT : S_LOAD_C;
126         S_LOAD_C: next_state = go ? S_LOAD_C_WAIT : S_LOAD_C;
127         S_LOAD_C_WAIT: next_state = go ? S_LOAD_C_WAIT : S_LOAD_X;
128         S_LOAD_X: next_state = go ? S_LOAD_X_WAIT : S_LOAD_X;
129         S_LOAD_X_WAIT: next_state = go ? S_LOAD_X_WAIT : S_CYCLE_0;
130         S_CYCLE_0: next_state = S_CYCLE_1;
131         S_CYCLE_1: next_state = S_CYCLE_2;
132         S_CYCLE_2: next_state = S_CYCLE_3;

```

```

133         S_CYCLE_3: next_state = S_CYCLE_4;
134         S_CYCLE_4: next_state = S_LOAD_A;
135         default:   next_state = S_LOAD_A;
136     endcase
137 end // state_table
138
139 // Output logic aka all of our datapath control signals
140 always @(*)
141 begin: enable_signals
142     // By default make all our signals 0
143     ld_alu_out = 1'b0;
144     ld_a = 1'b0;
145     ld_b = 1'b0;
146     ld_c = 1'b0;
147     ld_x = 1'b0;
148     ld_r = 1'b0;
149     alu_select_a = 2'b00;
150     alu_select_b = 2'b00;
151     alu_op       = 1'b0;
152
153     case (current_state)
154         S_LOAD_A: begin
155             ld_a = 1'b1;
156         end
157         S_LOAD_B: begin
158             ld_b = 1'b1;
159         end
160         S_LOAD_C: begin
161             ld_c = 1'b1;
162         end
163         S_LOAD_X: begin
164             ld_x = 1'b1;
165         end
166         S_CYCLE_0: begin // RB <- BX
167             ld_alu_out = 1'b1; ld_b = 1'b1; // store result back into RB
168             alu_select_a = 2'b01; // Select register B
169             alu_select_b = 2'b11; // Select register x
170             alu_op = 1'b1; // Multiply operation      Bx
171         end
172         S_CYCLE_1: begin // RB <- Bx + A
173             ld_alu_out = 1'b1; ld_b = 1'b1; // store result back into RB
174             alu_select_a = 2'b01; // Select register B
175             alu_select_b = 2'b00; // Select register A
176             alu_op = 1'b0; // Multiply operation      Bx + A
177         end
178         S_CYCLE_2: begin // RA <- CX
179             ld_alu_out = 1'b1; ld_a = 1'b1; // store result back into RA
180             alu_select_a = 2'b10; // Select register C
181             alu_select_b = 2'b11; // Select register X
182             alu_op = 1'b1; // Multiply operation      CX
183         end
184         S_CYCLE_3: begin // RA <- CX^2
185             ld_alu_out = 1'b1; ld_a = 1'b1; // store result back into RB
186             alu_select_a = 2'b00; // Select register A      CX
187             alu_select_b = 2'b11; // Select register X      x

```

```

188         alu_op = 1'b1; // Addition operation           CX^2
189     end
190
191     S_CYCLE_4: begin //r <- Cx^2 + Bx + A
192         ld_r = 1'b1; // store result in result register
193         alu_select_a = 2'b01; // Select register B
194         alu_select_b = 2'b00; // Select register A
195         alu_op = 1'b0; //Addition           Cx^2 + Bx + A
196     end
197 endcase
198 end // enable_signals
199
200 // current_state registers
201 always@(posedge clk)
202 begin: state_FFs
203     if(!resetn)
204         current_state <= S_LOAD_A;
205     else
206         current_state <= next_state;
207     end // state_FFS
208 endmodule
209
210 module datapath(
211     input clk,
212     input resetn,
213     input [7:0] data_in,
214     input ld_alu_out,
215     input ld_x, ld_a, ld_b, ld_c,
216     input ld_r,
217     input alu_op,
218     input [1:0] alu_select_a, alu_select_b,
219     output reg [7:0] data_result
220 );
221
222 // input registers
223 reg [7:0] a, b, c, x;
224
225 // output of the alu
226 reg [7:0] alu_out;
227 // alu input muxes
228 reg [7:0] alu_a, alu_b;
229
230 // Registers a, b, c, x with respective input logic
231 always @ (posedge clk) begin
232     if (!resetn) begin
233         a <= 8'd0;
234         b <= 8'd0;
235         c <= 8'd0;
236         x <= 8'd0;
237     end
238     else begin
239         if (ld_a)
240             a <= ld_alu_out ? alu_out : data_in;
241             // load alu_out if load_alu_out signal is high, otherwise load from data_in
242         if (ld_b)

```

```

243         b <= ld_alu_out ? alu_out : data_in;
244         // load alu_out if load_alu_out signal is high, otherwise load from data_in
245         if (ld_x)
246             x <= data_in;
247         if (ld_c)
248             c <= data_in;
249         end
250     end
251
252     // Output result register
253     always @ (posedge clk) begin
254         if (!resetn)
255             begin
256                 data_result <= 8'd0;
257             end
258         else if (ld_r)
259             data_result <= alu_out;
260     end
261
262     // The ALU input multiplexers
263     always @(*)
264     begin
265         case (alu_select_a)
266             2'd0: alu_a = a;
267             2'd1: alu_a = b;
268             2'd2: alu_a = c;
269             2'd3: alu_a = x;
270             default: alu_a = 8'd0;
271         endcase
272
273         case (alu_select_b)
274             2'd0: alu_b = a;
275             2'd1: alu_b = b;
276             2'd2: alu_b = c;
277             2'd3: alu_b = x;
278             default: alu_b = 8'd0;
279         endcase
280     end
281
282     // The ALU
283     always @(*)
284     begin : ALU
285         case (alu_op)
286             0: begin
287                 alu_out = alu_a + alu_b; //performs addition
288             end
289             1: begin
290                 alu_out = alu_a * alu_b; //performs multiplication
291             end
292             default: alu_out = 8'd0;
293         endcase
294     end
295
296 endmodule
297

```



```
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

The top screenshot shows the RTL Viewer of a circuit. A yellow box highlights the 'current_state' block, which contains registers for S_CYCLE_0 through S_CYCLE_4, S_LOAD_A, S_LOAD_B, S_LOAD_C, and S_LOAD_X. The circuit includes logic for selecting ALU operations and data based on these state signals. The bottom screenshot shows the State Machine table in the Compilation Report, listing 13 states and their corresponding bit patterns for the current_state signals.

Name	current_state_S_CYCLE_4	current_state_S_CYCLE_3	current_state_S_CYCLE_2	current_state_S_CYCLE_1	current_state_S_CYCLE_0	current_state_S_LOAD_X_WAIT	current_state_S_LOAD_X	current_state_S_LOAD_C_WAIT	current_state_S_LOAD_C	current_state_S_LOAD_B_WAIT	current_state_S_LOAD_B	current_state_S_LOAD_A_WAIT	current_state_S_LOAD_A
1 current_state_S_LOAD_A	0	0	0	0	0	0	0	0	0	0	0	0	0
2 current_state_S_LOAD_A_WAIT	0	0	0	0	0	0	0	0	0	0	0	0	0
3 current_state_S_LOAD_B	0	0	0	0	0	0	0	0	0	0	0	0	0
4 current_state_S_LOAD_B_WAIT	0	0	0	0	0	0	0	0	0	0	0	0	0
5 current_state_S_LOAD_C	0	0	0	0	0	0	0	0	0	0	0	0	0
6 current_state_S_LOAD_C_WAIT	0	0	0	0	0	0	0	0	0	0	0	0	0
7 current_state_S_LOAD_X	0	0	0	0	0	0	1	0	0	0	0	0	0
8 current_state_S_LOAD_X_WAIT	0	0	0	0	0	0	0	0	0	0	0	0	0
9 current_state_S_CYCLE_0	0	0	0	0	1	0	0	0	0	0	0	0	0
10 current_state_S_CYCLE_1	0	0	0	1	0	0	0	0	0	0	0	0	0
11 current_state_S_CYCLE_2	0	0	1	0	0	0	0	0	0	0	0	0	0
12 current_state_S_CYCLE_3	0	1	0	0	0	0	0	0	0	0	0	0	0
13 current_state_S_CYCLE_4	1	0	0	0	0	0	0	0	0	0	0	0	0

Encoding Type: One-Hot

Quartus Prime Tcl Console

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.

See the screenshot below

