```
//Sw[7:0] data_in

//KEY[0] synchronous reset when pressed
//KEY[1] go signal

//LEDR displays result
//HEX0 & HEX1 also displays result
`timescale 1ns / 1ns // `timescale time_unit/time_precision

module polyFunction(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)
    );

    assign LEDR[7:0] = data_result;

    hex_decoder H0(
        .hex_digit(data_result[3:0]),
        .segments(HEX0)
        );

    hex_decoder H1(
        .hex_digit(data_result[7:4]),
        .segments(HEX1)
        );

endmodule

module part2(
    input clk,
    input resetn,
    input go,
    input [7:0] data_in,
    output [7:0] data_result
    );

    // lots of wires to connect our datapath and control
    wire ld_a, ld_b, ld_c, ld_x, ld_r;
    wire ld_alu_out;
    wire [1:0]  alu_select_a, alu_select_b;
    wire alu_op;

    control C0(
        .clk(clk),
        .resetn(resetn),

        .go(go),

        .ld_alu_out(ld_alu_out),
        .ld_x(ld_x),
        .ld_a(ld_a),
        .ld_b(ld_b),
```

```
70          .ld_c(ld_c),
71          .ld_r(ld_r),
72
73          .alu_select_a(alu_select_a),
74          .alu_select_b(alu_select_b),
75          .alu_op(alu_op)
76      );
77
78      datapath D0(
79          .clk(clk),
80          .resetn(resetn),
81
82          .ld_alu_out(ld_alu_out),
83          .ld_x(ld_x),
84          .ld_a(ld_a),
85          .ld_b(ld_b),
86          .ld_c(ld_c),
87          .ld_r(ld_r),
88
89          .alu_select_a(alu_select_a),
90          .alu_select_b(alu_select_b),
91          .alu_op(alu_op),
92
93          .data_in(data_in),
94          .data_result(data_result)
95      );
96
97  endmodule
98
99
100 module control(
101     input clk,
102     input resetn,
103     input go,
104
105     output reg  ld_a, ld_b, ld_c, ld_x, ld_r,
106     output reg  ld_alu_out,
107     output reg [1:0]  alu_select_a, alu_select_b,
108     output reg alu_op
109     );
110
111     reg [5:0] current_state, next_state;
112
113     localparam  S_LOAD_A        = 5'd0,
114                 S_LOAD_A_WAIT   = 5'd1,
115                 S_LOAD_B        = 5'd2,
116                 S_LOAD_B_WAIT   = 5'd3,
117                 S_LOAD_C        = 5'd4,
118                 S_LOAD_C_WAIT   = 5'd5,
119                 S_LOAD_X        = 5'd6,
120                 S_LOAD_X_WAIT   = 5'd7,
121                 S_CYCLE_0       = 5'd8,
122                 S_CYCLE_1       = 5'd9,
123                 S_CYCLE_2       = 5'd10,
124                 S_CYCLE_3       = 5'd11,
125                 S_CYCLE_4       = 5'd12;
126
127     // Next state logic aka our state table
128     always@(*)
129     begin: state_table
130             case (current_state)
131                 S_LOAD_A: next_state = go ? S_LOAD_A_WAIT : S_LOAD_A; // Loop in
                    current state until value is input
132                 S_LOAD_A_WAIT: next_state = go ? S_LOAD_A_WAIT : S_LOAD_B; // Loop in
                    current state until go signal goes low
133                 S_LOAD_B: next_state = go ? S_LOAD_B_WAIT : S_LOAD_B; // Loop in
                    current state until value is input
134                 S_LOAD_B_WAIT: next_state = go ? S_LOAD_B_WAIT : S_LOAD_C; // Loop in
                    current state until go signal goes low
```

```
135                 S_LOAD_C: next_state = go ? S_LOAD_C_WAIT : S_LOAD_C; // Loop in
                    current state until value is input
136                 S_LOAD_C_WAIT: next_state = go ? S_LOAD_C_WAIT : S_LOAD_X; // Loop in
                    current state until go signal goes low
137                 S_LOAD_X: next_state = go ? S_LOAD_X_WAIT : S_LOAD_X; // Loop in
                    current state until value is input
138                 S_LOAD_X_WAIT: next_state = go ? S_LOAD_X_WAIT : S_CYCLE_0; // Loop in
                    current state until go signal goes low
139                 S_CYCLE_0: next_state = S_CYCLE_1; // A=Ax
140                      S_CYCLE_1: next_state = S_CYCLE_2; // B=Bx
141                 S_CYCLE_2: next_state = S_CYCLE_3; // B=Bx
142                      S_CYCLE_3: next_state = S_CYCLE_4; // A=A+B
143                 S_CYCLE_4: next_state = S_LOAD_A; // R=A+C, we will be done our two
                    operations, start over after
144             default:     next_state = S_LOAD_A;
145         endcase
146     end // state_table
147
148
149     // Output logic aka all of our datapath control signals
150     always @(*)
151     begin: enable_signals
152         // By default make all our signals 0 to avoid latches.
153         // This is a different style from using a default statement.
154         // It makes the code easier to read.  If you add other out
155         // signals be sure to assign a default value for them here.
156         ld_alu_out = 1'b0;
157         ld_a = 1'b0;
158         ld_b = 1'b0;
159         ld_c = 1'b0;
160         ld_x = 1'b0;
161         ld_r = 1'b0;
162         alu_select_a = 2'b0;
163         alu_select_b = 2'b0;
164         alu_op       = 1'b0;
165
166         case (current_state)
167             S_LOAD_A: begin
168                 ld_a = 1'b1;
169                 end
170             S_LOAD_B: begin
171                 ld_b = 1'b1;
172                 end
173             S_LOAD_C: begin
174                 ld_c = 1'b1;
175                 end
176             S_LOAD_X: begin
177                 ld_x = 1'b1;
178                 end
179             S_CYCLE_0: begin // Do A <- A * x
180                 ld_alu_out = 1'b1; ld_a = 1'b1; // store result back into A
181                 alu_select_a = 2'b00; // Select register A
182                 alu_select_b = 2'b11; // Select register x
183                 alu_op = 1'b1; // Do multiply operation
184             end
185             S_CYCLE_1: begin // Do B <- B * x
186                 ld_alu_out = 1'b1; ld_b = 1'b1; // store result back into B
187                 alu_select_a = 2'b01; // Select register B
188                 alu_select_b = 2'b11; // Select register x
189                 alu_op = 1'b1; // Do multiply operation
190             end
191             S_CYCLE_2: begin // Do B <- B * x again (End result is B=B*x*x)
192                 ld_alu_out = 1'b1; ld_b = 1'b1; // store result back into B
193                 alu_select_a = 2'b01; // Select register B
194                 alu_select_b = 2'b11; // Select register x
195                 alu_op = 1'b1; // Do multiply operation
196             end
197             S_CYCLE_3: begin // Do A <- A + B
198                 ld_alu_out = 1'b1; ld_a = 1'b1; // store result back into A
```

```
199                 alu_select_a = 2'b00; // Select register A
200                 alu_select_b = 2'b01; // Select register B
201                 alu_op = 1'b0; // Do addition operation
202             end
203             S_CYCLE_4: begin // Do R <- A + C
204                 ld_r = 1'b1; // store result in result register
205                 alu_select_a = 2'b00; // Select register A
206                 alu_select_b = 2'b10; // Select register C
207                 alu_op = 1'b0; // Do Add operation
208             end
209         // default:    // don't need default since we already made sure all of our
            outputs were assigned a value at the start of the always block
210         endcase
211     end // enable_signals
212
213     // current_state registers
214     always@(posedge clk)
215     begin: state_FFs
216         if(!resetn)
217             current_state <= S_LOAD_A;
218         else
219             current_state <= next_state;
220     end // state_FFS
221 endmodule
222
223 module datapath(
224     input clk,
225     input resetn,
226     input [7:0] data_in,
227     input ld_alu_out,
228     input ld_x, ld_a, ld_b, ld_c,
229     input ld_r,
230     input alu_op,
231     input [1:0] alu_select_a, alu_select_b,
232     output reg [7:0] data_result
233     );
234
235     // input registers
236     reg [7:0] a, b, c, x;
237
238     // output of the alu
239     reg [7:0] alu_out;
240     // alu input muxes
241     reg [7:0] alu_a, alu_b;
242
243     // Registers a, b, c, x with respective input logic
244     always@(posedge clk) begin
245         if(!resetn) begin
246             a <= 8'b0;
247             b <= 8'b0;
248             c <= 8'b0;
249             x <= 8'b0;
250         end
251         else begin
252             if(ld_a)
253                 a <= ld_alu_out ? alu_out : data_in; // load alu_out if load_alu_out
                    signal is high, otherwise load from data_in
254             if(ld_b)
255                 b <= ld_alu_out ? alu_out : data_in; // load alu_out if load_alu_out
                    signal is high, otherwise load from data_in
256             if(ld_c)
257                 c <= data_in;
258                 if(ld_x)
259                 x <= data_in;
260         end
261     end
262
263     // Output result register
264     always@(posedge clk) begin
```

```
265         if(!resetn) begin
266             data_result <= 8'b0;
267         end
268         else
269             if(ld_r)
270                 data_result <= alu_out;
271     end
272 
273     // The ALU input multiplexers
274     always @(*)
275     begin
276         case (alu_select_a)
277             2'd0:
278                 alu_a = a;
279             2'd1:
280                 alu_a = b;
281             2'd2:
282                 alu_a = c;
283             2'd3:
284                 alu_a = x;
285             default: alu_a = 8'b0;
286         endcase
287 
288         case (alu_select_b)
289             2'd0:
290                 alu_b = a;
291             2'd1:
292                 alu_b = b;
293             2'd2:
294                 alu_b = c;
295             2'd3:
296                 alu_b = x;
297             default: alu_b = 8'b0;
298         endcase
299     end
300 
301     // The ALU
302     always @(*)
303     begin : ALU
304         // alu
305         case (alu_op)
306             0: begin
307                     alu_out = alu_a + alu_b; //performs addition
308                end
309             1: begin
310                     alu_out = alu_a * alu_b; //performs multiplication
311                end
312             default: alu_out = 8'b0;
313         endcase
314     end
315 
316 endmodule
317 
318 
319 module hex_decoder(hex_digit, segments);
320     input [3:0] hex_digit;
321     output reg [6:0] segments;
322 
323     always @(*)
324         case (hex_digit)
325             4'h0: segments = 7'b100_0000;
326             4'h1: segments = 7'b111_1001;
327             4'h2: segments = 7'b010_0100;
328             4'h3: segments = 7'b011_0000;
329             4'h4: segments = 7'b001_1001;
330             4'h5: segments = 7'b001_0010;
331             4'h6: segments = 7'b000_0010;
332             4'h7: segments = 7'b111_1000;
333             4'h8: segments = 7'b000_0000;
```

```
334                 4'h9: segments = 7'b001_1000;
335                 4'hA: segments = 7'b000_1000;
336                 4'hB: segments = 7'b000_0011;
337                 4'hC: segments = 7'b100_0110;
338                 4'hD: segments = 7'b010_0001;
339                 4'hE: segments = 7'b000_0110;
340                 4'hF: segments = 7'b000_1110;
341                 default: segments = 7'h7f;
342             endcase
343     endmodule
344
345
```