```
1   `timescale 1ns / 1ns // `timescale time_unit/time_precision
2
3   /* top-level entity for polynomial calculator */
4   module fpga_top (input [9:0] SW, input [3:0] KEY, input CLOCK_50,
5                    output [9:0] LEDR, output [6:0] HEX0, HEX1);
6
7               wire resetn;
8               wire go;
9               wire [7:0] data_result;
10
11              /*
12               * Sw[7:0] is data_in
13               * KEY[0] synchronous reset when pressed
14               * KEY[1] is go signal
15               * LEDR displays result
16               * HEX0 and HEX1 also display result
17              */
18
19              assign go = ~KEY[1];
20              assign resetn = KEY[0];
21              assign LEDR[7:0] = data_result;
22
23              poly_calc u0 (.clk(CLOCK_50), .resetn(resetn), .go(go), .data_in(SW[7:0]),
24                            .data_result(data_result));
25              hex_decoder H0 (.hex_digit(data_result[3:0]), .segments(HEX0));
26              hex_decoder H1 (.hex_digit(data_result[7:4]), .segments(HEX1));
27
28  endmodule // fpga_top
29
30  /* polynomial calculator */
31  module poly_calc (input clk, resetn, go, input [7:0] data_in,
32                    output [7:0] data_result);
33
34              // wires to connect datapath and control
35              wire ld_a, ld_b, ld_c, ld_x, ld_r;
36              wire ld_alu_out;
37              wire [1:0]  alu_select_a, alu_select_b;
38              wire alu_op;
39
40              control C0 (.clk(clk), .resetn(resetn), .go(go), .ld_alu_out(ld_alu_out),
41                          .ld_x(ld_x), .ld_a(ld_a), .ld_b(ld_b), .ld_c(ld_c),
42                          .ld_r(ld_r), .alu_select_a(alu_select_a),
43                          .alu_select_b(alu_select_b), .alu_op(alu_op));
44
45              datapath D0 (.clk(clk), .resetn(resetn), .ld_alu_out(ld_alu_out),
46                          .ld_x(ld_x), .ld_a(ld_a), .ld_b(ld_b), .ld_c(ld_c),
47                          .ld_r(ld_r), .alu_select_a(alu_select_a),
48                          .alu_select_b(alu_select_b), .alu_op(alu_op),
49                          .data_in(data_in), .data_result(data_result));
50
51  endmodule // poly_calc
52
53  /* controls registers and state transition */
54  module control (input clk, resetn, go, output reg  ld_a, ld_b, ld_c, ld_x,
55                  ld_r, ld_alu_out, output reg [1:0]  alu_select_a, alu_select_b,
56                  output reg alu_op);
57
58              reg [5:0] current_state, next_state;
59              localparam  S_LOAD_A        = 5'd0,
60                          S_LOAD_A_WAIT   = 5'd1,
61                          S_LOAD_B        = 5'd2,
62                          S_LOAD_B_WAIT   = 5'd3,
63                          S_LOAD_C        = 5'd4,
64                          S_LOAD_C_WAIT   = 5'd5,
65                          S_LOAD_X        = 5'd6,
66                          S_LOAD_X_WAIT   = 5'd7,
67                          S_CYCLE_0       = 5'd8,
68                          S_CYCLE_1       = 5'd9,
69                          S_CYCLE_2       = 5'd10,
70                          S_CYCLE_3       = 5'd11;
71              // state table
72              always @(*)
73              begin
74
75              case (current_state)
76                  // loop in current state until value is input
```

```
77                S_LOAD_A: next_state = go ? S_LOAD_A_WAIT : S_LOAD_A;
78                // loop in current state until go signal goes low
79                S_LOAD_A_WAIT: next_state = go ? S_LOAD_A_WAIT : S_LOAD_B;
80                // loop in current state until value is input
81                S_LOAD_B: next_state = go ? S_LOAD_B_WAIT : S_LOAD_B;
82                // loop in current state until go signal goes low
83                S_LOAD_B_WAIT: next_state = go ? S_LOAD_B_WAIT : S_LOAD_C;
84                // loop in current state until value is input
85                S_LOAD_C: next_state = go ? S_LOAD_C_WAIT : S_LOAD_C;
86                // loop in current state until go signal goes low
87                S_LOAD_C_WAIT: next_state = go ? S_LOAD_C_WAIT : S_LOAD_X;
88                // loop in current state until value is input
89                S_LOAD_X: next_state = go ? S_LOAD_X_WAIT : S_LOAD_X;
90                // loop in current state until go signal goes low
91                S_LOAD_X_WAIT: next_state = go ? S_LOAD_X_WAIT : S_CYCLE_0;
92                S_CYCLE_0: next_state = S_CYCLE_1;
93                S_CYCLE_1: next_state = S_CYCLE_2;
94                S_CYCLE_2: next_state = S_CYCLE_3;
95                // start over after
96                S_CYCLE_3: next_state = S_LOAD_A;
97                default: next_state = S_LOAD_A;
98             endcase
99
100            end // state_table
101
102
103            // output logic for datapath control signals
104            always @(*)
105            begin
106            // all signals are 0 by default, to avoid latches.
107
108               ld_alu_out = 1'b0;
109               ld_a = 1'b0;
110               ld_b = 1'b0;
111               ld_c = 1'b0;
112               ld_x = 1'b0;
113               ld_r = 1'b0;
114               alu_select_a = 2'b0;
115               alu_select_b = 2'b0;
116               alu_op       = 1'b0;
117
118               case (current_state)
119                  S_LOAD_A: begin
120                              ld_a = 1'b1;
121                            end
122                  S_LOAD_B: begin
123                              ld_b = 1'b1;
124                            end
125                  S_LOAD_C: begin
126                              ld_c = 1'b1;
127                            end
128                  S_LOAD_X: begin
129                              ld_x = 1'b1;
130                            end
131                  S_CYCLE_0: begin // do B <- B * x
132                              ld_alu_out = 1'b1; ld_b = 1'b1; // store result back into B
133                              alu_select_a = 2'd1; // select register B
134                              alu_select_b = 2'd3; // also select register x
135                              alu_op = 1'b1; // do multiply operation
136                             end
137                  S_CYCLE_1: begin // do A <- A + B
138                              ld_alu_out = 1'b1; ld_a = 1'b1; // store result back into A
139                              alu_select_a = 2'd0; // select register A
140                              alu_select_b = 2'd1; // also select register B
141                              alu_op = 1'b0; // do addition operation
142                             end
143                  S_CYCLE_2: begin // do A <- A * x
144                              ld_alu_out = 1'b1; ld_a = 1'b1; // store result back into A
145                              alu_select_a = 2'd0; // select register A
146                              alu_select_b = 2'd3; // also select register x
147                              alu_op = 1'b1; // do multiply operation
148                             end
149                  S_CYCLE_3: begin // do A + C
150                              ld_r = 1'b1; // store result in result register
151                              alu_select_a = 2'd0; // select register A
152                              alu_select_b = 2'd2; // select register C
```

```
153                                  alu_op = 1'b0; // do addition operation
154                                 end
155                     // no default needed; all of our outputs were assigned a value
156                  endcase
157
158               end // enable_signals
159
160               // current_state registers
161               always @(posedge clk)
162               begin
163                  if(!resetn)
164                     current_state <= S_LOAD_A;
165                  else
166                     current_state <= next_state;
167               end // state_FFS
168
169   endmodule // control
170
171   module datapath (input clk, resetn, ld_x, ld_a, ld_b, ld_c,
172                    ld_r, alu_op, ld_alu_out, input [7:0] data_in,
173                    input [1:0] alu_select_a, alu_select_b,
174                    output reg [7:0] data_result);
175
176               reg [7:0] a, b, c, x; // input registers
177               reg [7:0] alu_out; // output of the alu
178               reg [7:0] alu_a, alu_b; // alu input muxes
179
180               // registers a, b, c, x with respective input logic
181               always@(posedge clk) begin
182
183                 if(!resetn)
184                 begin
185                     a <= 8'b0;
186                     b <= 8'b0;
187                     c <= 8'b0;
188                     x <= 8'b0;
189                 end
190
191                 else
192                 begin
193                     // load alu_out if load_alu_out signal is high, otherwise load from data_in
194                     if(ld_a)
195                         a <= ld_alu_out ? alu_out : data_in;
196                     // load alu_out if load_alu_out signal is high, otherwise load from data_in
197                     if(ld_b)
198                         b <= ld_alu_out ? alu_out : data_in;
199                     if(ld_x)
200                         x <= data_in;
201                     if(ld_c)
202                         c <= data_in;
203                 end
204               end
205
206               // output result register
207               always@(posedge clk) begin
208                 if(!resetn) begin
209                     data_result <= 8'b0;
210                 end
211                 else
212                     if(ld_r)
213                         data_result <= alu_out;
214               end
215
216               // the ALU input multiplexers
217               always @(*)
218               begin
219                 case (alu_select_a)
220                     2'd0:
221                         alu_a = a;
222                     2'd1:
223                         alu_a = b;
224                     2'd2:
225                         alu_a = c;
226                     2'd3:
227                         alu_a = x;
228                     default: alu_a = 8'b0;
```

```
229                   endcase
230
231                   case (alu_select_b)
232                       2'd0:
233                           alu_b = a;
234                       2'd1:
235                           alu_b = b;
236                       2'd2:
237                           alu_b = c;
238                       2'd3:
239                           alu_b = x;
240                       default: alu_b = 8'b0;
241                   endcase
242               end
243
244               // ALU
245               always @(*)
246               begin : ALU
247                   case (alu_op)
248                       // performs addition
249                       0: begin
250                               alu_out = alu_a + alu_b;
251                          end
252                       // performs multiplication
253                       1: begin
254                               alu_out = alu_a * alu_b;
255                          end
256                       default: alu_out = 8'b0;
257                   endcase
258               end
259
260   endmodule // datapath
261
262   /* bcd to hex for seven-segment display */
263   module hex_decoder(input [3:0] hex_digit, output reg [6:0] segments);
264
265               always @(*)
266               begin
267
268                   case (hex_digit)
269                       4'h0: segments = 7'b100_0000;
270                       4'h1: segments = 7'b111_1001;
271                       4'h2: segments = 7'b010_0100;
272                       4'h3: segments = 7'b011_0000;
273                       4'h4: segments = 7'b001_1001;
274                       4'h5: segments = 7'b001_0010;
275                       4'h6: segments = 7'b000_0010;
276                       4'h7: segments = 7'b111_1000;
277                       4'h8: segments = 7'b000_0000;
278                       4'h9: segments = 7'b001_1000;
279                       4'hA: segments = 7'b000_1000;
280                       4'hB: segments = 7'b000_0011;
281                       4'hC: segments = 7'b100_0110;
282                       4'hD: segments = 7'b010_0001;
283                       4'hE: segments = 7'b000_0110;
284                       4'hF: segments = 7'b000_1110;
285                       default: segments = 7'h7f;
286                   endcase
287
288               end
289
290   endmodule // hex_decoder
```