```
1   `timescale 1ns / 1ns // `timescale time_unit/time_precision
2
3
4   /* top-level entity for ALU */
5   module lab4Part2 (input [9:0] SW, input [3:0] KEY,
6                     output [9:0] LEDR, output [6:0] HEX0, HEX4, HEX5);
7
8               assign LEDR[9:8] = 2'b00; // LEDR[9] and LEDR[8] are off
9               wire [7:0] ALUout, regState;
10
11              /*
12               * SW[3:0] is the input A bus
13               * SW[9] is the reset
14               * KEY[0] is the clock signal
15               * KEY[3:1] is the operation selection bus
16               * LEDR[7:0] is the ALU output bus
17               * HEX0, HEX4, HEX5 are for display of inputs and output
18               * HEX1, HEX2, and HEX3 stay off (un-initialized)
19               * SW[8:4] are non-inputs for any function
20               */
21
22              bcdDecoder disp_A (.C(SW[3:0]), .HEX(HEX0)); // display input A
23
24              ALU A1 (.A(SW[3:0]), .B(regState[3:0]), .regState(regState),
25                    .opSelect(KEY[3:1]), .ALUout(ALUout)); // instantiating ALU
26              regALU R1 (.regInput(ALUout), .reset(SW[9]), .clock(~KEY[0]),
27                    .regState(regState)); // instantiating register
28
29              assign LEDR[7:0] = regState; // display register state bus
30              bcdDecoder disp_regState_1 (.C(regState[3:0]), .HEX(HEX4)); // display on HEX4
31              bcdDecoder disp_regState_2 (.C(regState[7:4]), .HEX(HEX5)); // display on HEX5
32
33  endmodule // lab4Part2
34
35
36  /* register to store output from the ALU */
37  module regALU (input [7:0] regInput, input reset, clock,
38                 output reg [7:0] regState);
39
40     always @(posedge clock) // triggered on rising edge of the clock signal
41     begin
42
43        if (reset == 1'b0) // active-low, clock synchronous reset to 0
44           regState <= 8'b00000000;
45        else
46           regState <= regInput; // if reset is not 0, register stores input
47
48     end
49
50  endmodule // regALU
51
52  /* Arithmetic Logic Unit with 8 possible operations */
53  module ALU (input [3:0] A, B, input [2:0] opSelect, input [7:0] regState,
54              output [7:0] ALUout);
55
56              wire [7:0] opOut_0, opOut_1, opOut_2, opOut_3, opOut_4,
57                         opOut_5, opOut_6, opOut_7; // 7 8-bit possible outputs of the ALU
58
59              // case 0
60              fourBitAdder FA_4bit (.A(A), .B(B), .c_in(1'b0), .S(opOut_0[3:0]),
61                           .c_out(opOut_0[4])); // modular addition
62              assign opOut_0[7:5] = 3'b000; // pad the rest bits to zero
63
64              // case 1
65              assign opOut_1[4:0] = (A + B); // bitwise addition operator
66              assign opOut_1[7:5] = 3'b000; // pad the rest bits to zero
67
68              // case 2
69              assign opOut_2[3:0] = ~(A & B); // bitwise nand
70              assign opOut_2[7:4] = ~(A | B); // bitwise nor
71
72              // case 3
73              assign opOut_3 = ((A > 0) | (B > 0)) ? (8'b11000000):(8'b00000000);
74
75              // case 4
76              assign opOut_4 = (((~^A) & (A != 0) & (A != 15)) & // returns 1 if A has 2 ones
```

```
77                                  ((^B) & (B >= 7) & (B != 8))) // returns 1 if B has 3 ones
78                                  ? (8'b00111111):(8'b00000000);
79
80                  // case 5
81                  assign opOut_5[7:4] = B; // B as MSB
82                  assign opOut_5[3:0] = ~A; // complement of A as LSB
83
84                  // case 6
85                  assign opOut_6[3:0] = ~(A ^ B); // bitwise xnor
86                  assign opOut_6[7:4] = (A ^ B); // bitwise xor
87
88                  // case 7
89                  assign opOut_7 = regState; // register values remain unchanged
90
91                  // selecting which operation to output using a 7 to 1 multiplexer
92                  mux7to1 opSelecter_1 (.out0(opOut_0), .out1(opOut_1), .out2(opOut_2),
93                                        .out3(opOut_3), .out4(opOut_4), .out5(opOut_5),
94                                        .out6(opOut_6), .out7(opOut_7),
95                                        .MuxSelect(opSelect), .muxOut(ALUout));
96
97      endmodule //ALU
98
99
100     /* 7 to 1 multiplexer module */
101     module mux7to1 (input [7:0] out0, out1, out2, out3, out4, out5, out6, out7,
102                     input [2:0] MuxSelect, output reg [7:0] muxOut);
103
104                 /*
105                  * always statement implementing the
106                  * the combinational logic for selecting signals
107                  */
108
109                 always @(*)
110                 begin
111                    case (~MuxSelect[2:0])
112
113                       // inverted MuxSelect bits for input through KEY
114                       3'b000: muxOut = out0; // output of operation 0
115                       3'b001: muxOut = out1; // output of operation 1
116                       3'b010: muxOut = out2; // output of operation 2
117                       3'b011: muxOut = out3; // output of operation 3
118                       3'b100: muxOut = out4; // output of operation 4
119                       3'b101: muxOut = out5; // output of operation 5
120                       3'b110: muxOut = out6; // output of operation 6
121                       3'b111: muxOut = out7; // output of operation 7
122                       default: muxOut = 8'b00000000; // default case
123
124                    endcase
125                 end
126
127     endmodule // mux7to1
128
129
130     /* 4-bit adder module */
131     module fourBitAdder (input [3:0] A, B, input c_in,
132                          output [3:0] S, output c_out);
133
134                 wire [2:0] C; // carry-pin vector
135
136                 // check schematic for wiring
137                 fullAdder bit0 (.c_in(c_in), .a(A[0]), .b(B[0]), .s(S[0]), .c_out(C[0]));
138                 fullAdder bit1 (.c_in(C[0]), .a(A[1]), .b(B[1]), .s(S[1]), .c_out(C[1]));
139                 fullAdder bit2 (.c_in(C[1]), .a(A[2]), .b(B[2]), .s(S[2]), .c_out(C[2]));
140                 fullAdder bit3 (.c_in(C[2]), .a(A[3]), .b(B[3]), .s(S[3]), .c_out(c_out));
141
142     endmodule // fourBitAdder
143
144
145     /* full adder module */
146     module fullAdder (input c_in, a, b, output s, c_out);
147
148                 assign s = (a ^ b ^ c_in); // odd function for sum bit
149                 assign c_out = ((a & b) | (a & c_in) | (b & c_in)); // majority function for
    carry-out bit
150
151     endmodule // fullAdder
```

```
152
153
154   /* BCD to common-anode seven-segment display decoder */
155   module bcdDecoder (input [3:0] C, output [6:0] HEX);
156
157               // maxterms for every segment LEDs with common anode
158               assign HEX[0] = !((C[3]|C[2]|C[1]||C[0]) & (C[3]||C[2]|C[1]|C[0]) &
159                           (!C[3]|C[2]||C[1]||C[0]) & (!C[3]||C[2]|C[1]||C[0]));
160
161               assign HEX[1] = !((C[3]||C[2]|C[1]||C[0]) & (C[3]||C[2]||C[1]|C[0]) &
162                           (!C[3]|C[2]||C[1]||C[0]) & (!C[3]||C[2]|C[1]|C[0]) &
163                           (!C[3]||C[2]||C[1]|C[0]) & (!C[3]||C[2]||C[1]||C[0]));
164
165               assign HEX[2] = !((C[3]|C[2]||C[1]|C[0]) & (!C[3]||C[2]|C[1]|C[0]) &
166                           (!C[3]||C[2]||C[1]|C[0]) & (!C[3]||C[2]||C[1]||C[0]));
167
168               assign HEX[3] = !((C[3]|C[2]|C[1]||C[0]) & (C[3]||C[2]|C[1]|C[0]) &
169                           (C[3]||C[2]||C[1]||C[0]) & (!C[3]|C[2]|C[1]||C[0]) &
170                           (!C[3]|C[2]||C[1]|C[0]) & (!C[3]||C[2]||C[1]||C[0]));
171
172               assign HEX[4] = !((C[3]|C[2]|C[1]||C[0]) & (C[3]|C[2]||C[1]||C[0]) &
173                           (C[3]||C[2]|C[1]|C[0]) & (C[3]||C[2]|C[1]|C[0]) &
174                           (C[3]||C[2]|C[1]||C[0]) & (C[3]||C[2]||C[1]||C[0]) &
175                           (!C[3]|C[2]|C[1]||C[0]));
176
177               assign HEX[5] = !((C[3]|C[2]|C[1]||C[0]) & (C[3]|C[2]||C[1]|C[0]) &
178                           (C[3]|C[2]||C[1]||C[0]) & (C[3]||C[2]||C[1]||C[0]) &
179                           (!C[3]||C[2]|C[1]||C[0]));
180
181               assign HEX[6] = !((C[3]|C[2]|C[1]|C[0]) & (C[3]|C[2]|C[1]||C[0]) &
182                           (C[3]||C[2]||C[1]||C[0]) & (!C[3]||C[2]|C[1]|C[0]));
183
184   endmodule // bcdDecoder
```