```
1   `timescale 1ns / 1ns // `timescale time_unit/time_precision
2
3
4   /* top-level entity for ALU */
5   module lab3Part3 (input [7:0] SW, input [2:0] KEY, output [9:0] LEDR,
6                     output [6:0] HEX0, HEX2, HEX4, HEX5);
7
8               assign LEDR[9:8] = 2'b00; // LEDR[9] and LEDR[8] are off
9
10              wire [7:0] ALUout;
11
12              /*
13               * SW[3:0] is the input B bus
14               * SW[7:4] is the input A bus
15               * KEY[2:0] is the operation selection bus
16               * LEDR[7:0] is the ALU output bus
17               * HEX0, HEX2, HEX4, HEX5 are for display of inputs and output
18               * HEX1 and HEX3 stay off (un-initialized)
19               */
20
21              bcdDecoder disp_A (.C(SW[7:4]), .HEX(HEX2)); // display input A
22              bcdDecoder disp_B (.C(SW[3:0]), .HEX(HEX0)); // display input B
23
24              ALU M1 (.A(SW[7:4]), .B(SW[3:0]),
25                    .opSelect(KEY[2:0]), .ALUout(ALUout)); // instantiating ALU
26
27              assign LEDR[7:0] = ALUout; // display ALUout bus
28              bcdDecoder disp_ALUout_1 (.C(ALUout[3:0]), .HEX(HEX4)); // display on HEX4
29              bcdDecoder disp_ALUout_2 (.C(ALUout[7:4]), .HEX(HEX5)); // display on HEX5
30
31  endmodule // lab3Part2
32
33
34  /* Arithmetic Logic Unit with 7 possible operations */
35  module ALU (input [3:0] A, B, input [2:0] opSelect, output [7:0] ALUout);
36
37              wire [7:0] opOut_0, opOut_1, opOut_2, opOut_3, opOut_4,
38                         opOut_5, opOut_6; // 7 8-bit possible outputs of the ALU
39
40              // case 0
41              fourBitAdder FA_4bit (.A(A), .B(B), .c_in(1'b0), .S(opOut_0[3:0]),
42                           .c_out(opOut_0[4])); // modular addition
43              assign opOut_0[7:5] = 3'b000; // pad the rest bits to zero
44
45              // case 1
46              assign opOut_1[4:0] = (A + B); // bitwise addition operator
47              assign opOut_1[7:5] = 3'b000; // pad the rest bits to zero
48
49              // case 2
50              assign opOut_2[3:0] = ~(A & B); // bitwise nand
51              assign opOut_2[7:4] = ~(A | B); // bitwise nor
52
53              // case 3
54              assign opOut_3 = ((A > 0) | (B > 0)) ? (8'b11000000):(8'b00000000);
55
56              // case 4
57              assign opOut_4 = (((~^A) & (A != 0) & (A != 15)) & // returns 1 if A has 2 ones
58                               ((^B) & (B >= 7) & (B != 8))) // returns 1 if B has 3 ones
59                              ? (8'b00111111):(8'b00000000);
60
61              // case 5
62              assign opOut_5[7:4] = B;
63              assign opOut_5[3:0] = ~A;
64
65              // case 6
66              assign opOut_6[3:0] = ~(A ^ B); // bitwise xnor
67              assign opOut_6[7:4] = (A ^ B); // bitwise xor
68
69              // selecting which operation to output using a 7 to 1 multiplexer
70              mux7to1 opSelecter_1 (.out0(opOut_0), .out1(opOut_1), .out2(opOut_2),
71                                    .out3(opOut_3), .out4(opOut_4), .out5(opOut_5),
72                                    .out6(opOut_6), .MuxSelect(opSelect), .muxOut(ALUout));
73
74  endmodule //ALU
75
76
```

```
77  /* 7 to 1 multiplexer module */
78  module mux7to1 (input [7:0] out0, out1, out2, out3, out4, out5, out6,
79                  input [2:0] MuxSelect, output reg [7:0] muxOut);
80
81              /*
82               * always statement implementing the
83               * the combinational logic for selecting signals
84               */
85
86              always @(*)
87              begin
88                 case (MuxSelect[2:0])
89
90                    // inverted MuxSelect bits for input through KEY
91                    3'b111: muxOut = out0; // output of operation 0
92                    3'b110: muxOut = out1; // output of operation 1
93                    3'b101: muxOut = out2; // output of operation 2
94                    3'b100: muxOut = out3; // output of operation 3
95                    3'b011: muxOut = out4; // output of operation 4
96                    3'b010: muxOut = out5; // output of operation 5
97                    3'b001: muxOut = out6; // output of operation 6
98                    default: muxOut = 8'b00000000; // default case
99
100                endcase
101             end
102
103 endmodule // mux7to1
104
105
106 /* 4-bit adder module */
107 module fourBitAdder (input [3:0] A, B, input c_in,
108                      output [3:0] S, output c_out);
109
110             wire [2:0] C; // carry-pin vector
111
112             // check schematic for wiring
113             fullAdder bit0 (.c_in(c_in), .a(A[0]), .b(B[0]), .s(S[0]), .c_out(C[0]));
114             fullAdder bit1 (.c_in(C[0]), .a(A[1]), .b(B[1]), .s(S[1]), .c_out(C[1]));
115             fullAdder bit2 (.c_in(C[1]), .a(A[2]), .b(B[2]), .s(S[2]), .c_out(C[2]));
116             fullAdder bit3 (.c_in(C[2]), .a(A[3]), .b(B[3]), .s(S[3]), .c_out(c_out));
117
118 endmodule // fourBitAdder
119
120
121 /* full adder module */
122 module fullAdder (input c_in, a, b, output s, c_out);
123
124             assign s = (a ^ b ^ c_in); // odd function for sum bit
125             assign c_out = ((a & b) | (a & c_in) | (b & c_in)); // majority function for
    carry-out bit
126
127 endmodule // fullAdder
128
129
130 /* BCD to common-anode seven-segment display decoder */
131 module bcdDecoder (input [3:0] C, output [6:0] HEX);
132
133             // maxterms for every segment LEDs with common anode
134             assign HEX[0] = !((C[3]|C[2]|C[1]|!C[0]) & (C[3]|!C[2]|C[1]|C[0]) &
135                               (!C[3]|C[2]|!C[1]|!C[0]) & (!C[3]|!C[2]|C[1]|!C[0]));
136
137             assign HEX[1] = !((C[3]|!C[2]|C[1]|!C[0]) & (C[3]|!C[2]|!C[1]|C[0]) &
138                               (!C[3]|C[2]|!C[1]|!C[0]) & (!C[3]|!C[2]|C[1]|C[0]) &
139                               (!C[3]|!C[2]|!C[1]|C[0]) & (!C[3]|!C[2]|!C[1]|!C[0]));
140
141             assign HEX[2] = !((C[3]|C[2]|!C[1]|C[0]) & (!C[3]|!C[2]|C[1]|C[0]) &
142                               (!C[3]|!C[2]|!C[1]|C[0]) & (!C[3]|!C[2]|!C[1]|!C[0]));
143
144             assign HEX[3] = !((C[3]|C[2]|C[1]|!C[0]) & (C[3]|!C[2]|C[1]|C[0]) &
145                               (C[3]|!C[2]|!C[1]|!C[0]) & (!C[3]|C[2]|C[1]|!C[0]) &
146                               (!C[3]|C[2]|!C[1]|C[0]) & (!C[3]|!C[2]|!C[1]|!C[0]));
147
148             assign HEX[4] = !((C[3]|C[2]|C[1]|!C[0]) & (C[3]|C[2]|!C[1]|!C[0]) &
149                               (C[3]|!C[2]|C[1]|C[0]) & (C[3]|!C[2]|C[1]|C[0]) &
150                               (C[3]|!C[2]|C[1]|!C[0]) & (C[3]|!C[2]|!C[1]|!C[0]) &
151                               (!C[3]|C[2]|C[1]|!C[0]));
```

```
152
153             assign HEX[5] = !((C[3]|C[2]|C[1]||!C[0]) & (C[3]|C[2]||!C[1]|C[0]) &
154                         (C[3]|C[2]||!C[1]||!C[0]) & (C[3]||!C[2]||!C[1]||!C[0]) &
155                         (!C[3]||!C[2]|C[1]||!C[0]));
156
157             assign HEX[6] = !((C[3]|C[2]|C[1]|C[0]) & (C[3]|C[2]|C[1]||!C[0]) &
158                         (C[3]||!C[2]||!C[1]||!C[0]) & (!C[3]||!C[2]|C[1]|C[0]));
159
160 endmodule // bcdDecoder
```