Lab Code [10 points]
Filename: ChipInterface.sv

```
 1 `default_nettype none
 2
 3 module ChipInterface
 4
 5   (input logic [17:0] SW,
 6    output logic [17:0] LEDR,
 7    output logic [7:0] LEDG,
 8    output logic [6:0] HEX9, HEX8, HEX7, HEX6, HEX5,
 9                       HEX4, HEX3, HEX2, HEX1, HEX0);
10
11   logic [12:0] inCode;
12   logic [12:0] outCode;
13   logic is1BitErr;
14   logic is2BitErr;
15   logic [3:0] syndrome;
16   logic [3:0] inData0, inData1, outData0, outData1;
17
18   assign inCode = SW[12:0];
19   assign LEDR[12:0] = outCode; //red LED
20   assign LEDG[7:4] = syndrome; //green LED
21   assign LEDG[0] = is1BitErr;  //green LED
22   assign LEDG[1] = is2BitErr;  //red LED
23
24   // divide each data bits into two four bits so we can represent in hex
25
26   assign inData1 = {inCode[12], inCode[11], inCode[10],
27                     inCode[9]};
28
29   assign inData0 = {inCode[7], inCode[6],
30                     inCode[5], inCode[3]};
31
32   // do the same for outdata
33   assign outData1 = {outCode[12], outCode[11], outCode[10],
34                      outCode[9]};
35
36   assign outData0 = {outCode[7], outCode[6],
37                      outCode[5], outCode[3]};
38
39   SECDEDdecoder dec(.inCode, .syndrome,
40                     .is1BitErr, .is2BitErr, .outCode);
41
42   SevenSegmentControl ssc(.HEX9, .HEX8, .HEX7, .HEX6, .HEX5,
43                           .HEX4, .HEX3, .HEX2, .HEX1, .HEX0,
44                           .BCH9(inData1), .BCH8(inData0), .BCH7(outData1),
45                           .BCH6(outData0), .BCH5(4'b0000), .BCH4(4'b0000),
46                           .BCH3(outCode[12]), .BCH2(outCode[11:8]),
47                           .BCH1(outCode[7:4]), .BCH0(outCode[3:0]));
48
49                           //hard coded for BCH5 and BCH4
50
51 endmodule: ChipInterface
```

Lab Code [10 points]
Filename: lab3.sv

```
 1 `default_nettype none
 2
 3 module SECDEDdecoder (input  logic [12:0] inCode,
 4                       output logic [3:0] syndrome,
 5                       output logic is1BitErr, is2BitErr,
 6                       output logic [12:0] outCode);
 7
 8   // instantiate all modules
 9   makeSyndrome dut1 (.codeWord(inCode), .*);
10
11   makeCorrect dut2 ( .codeWord(inCode), .syndrome(syndrome),
12                     .correctCodeWord(outCode));
13
14   errorCheck dut3 (.*);
15
16
17 endmodule:  SECDEDdecoder
18
19
20 module makeSyndrome (input  logic [12:0] codeWord,
21                      output logic [3:0] syndrome);
22
23
24   // use xor of different data and parity bits to obtain 4 bit syndrome
25   assign syndrome[0] = codeWord[1] ^ codeWord[3] ^ codeWord[5] ^ codeWord[7] ^
26                        codeWord[9] ^ codeWord[11];
27
28   assign syndrome[1] = codeWord[2] ^ codeWord[3] ^ codeWord[6] ^ codeWord[7] ^
29                        codeWord[10] ^ codeWord[11];
30
31   assign syndrome[2] = codeWord[4] ^ codeWord[5] ^ codeWord[6] ^ codeWord[7] ^
32                        codeWord[12];
33
34   assign syndrome[3] = codeWord[8] ^ codeWord[9] ^ codeWord[10] ^ codeWord[11] ^
35                        codeWord[12];
36
37
38 endmodule: makeSyndrome
39
40
41 module makeCorrect (input  logic [12:0] codeWord,
42                     input  logic [3:0] syndrome,
43                     output logic [12:0] correctCodeWord);
44
45   logic PGfail;
46
47   // check for even or odd parity
48   assign PGfail = ^codeWord;
49
50   always_comb begin
51
52   correctCodeWord = codeWord;
53   // change correct code based on cases
54   if (PGfail === 1 && syndrome !== 4'b0000)
55     correctCodeWord[syndrome] = ~codeWord[syndrome];
56
57   else if (PGfail === 1 && syndrome === 4'b0000)
58     correctCodeWord[0] = ~codeWord[0];
59
60   end
61
62
63 endmodule:makeCorrect
64
65
66 module errorCheck
67
68   (input logic [12:0] inCode,
69    input logic [3:0] syndrome,
70    output logic is1BitErr,
```

```
 71    output logic is2BitErr);
 72
 73   logic isOdd, GPfail;
 74
 75   assign isOdd = ^inCode;  //if it is odd PG is correct
 76   assign GPfail = isOdd;  //if it is even, PG fails
 77
 78   always_comb begin
 79
 80   is1BitErr = 0;
 81   is2BitErr = 0;
 82
 83     if (~GPfail && syndrome === 4'b0000)
 84       begin
 85         is1BitErr = 0;
 86         is2BitErr = 0;
 87       end
 88     else if (GPfail) // 1 bit error, either in the GP or normal bit
 89       begin
 90         is1BitErr = 1;
 91         is2BitErr = 0;
 92       end
 93     else if (~GPfail && syndrome !== 4'b0000)  // 2bits errors
 94       begin
 95         is1BitErr = 0;
 96         is2BitErr = 1;
 97     end
 98   end
 99
100 endmodule : errorCheck
101
102
103 module BCHtoSevenSegment (input logic [3:0] BCH,
104                           output logic [6:0] segment);
105
106
107   // do all hex conversions with 4 bits
108   always_comb
109     unique case (BCH[3:0])
110       4'b0000: segment = 7'b100_0000;
111       4'b0001: segment = 7'b111_1001;
112       4'b0010: segment = 7'b010_0100;
113       4'b0011: segment = 7'b011_0000;
114       4'b0100: segment = 7'b001_1001;
115       4'b0101: segment = 7'b001_0010;
116       4'b0110: segment = 7'b000_0010;
117       4'b0111: segment = 7'b111_1000;
118       4'b1000: segment = 7'b000_0000;
119       4'b1001: segment = 7'b001_0000;
120       4'b1010: segment = 7'b000_1000;
121       4'b1011: segment = 7'b000_0011;
122       4'b1100: segment = 7'b100_0110;
123       4'b1101: segment = 7'b010_0001;
124       4'b1110: segment = 7'b000_0110;
125       4'b1111: segment = 7'b000_1110;
126     endcase
127 endmodule: BCHtoSevenSegment
128
129 module SevenSegmentDigit (input logic [3:0] BCH,
130                           output logic [6:0] segment,
131                           input logic blank);
132   logic [6:0] decoded;
133
134   BCHtoSevenSegment b2ss(.BCH(BCH), .segment(decoded));
135   // obtain for a specific digit
136   always_comb begin
137     if (blank)
138       segment = 7'b111_1111; // want blanks and 1 is off
139     else
140       segment = decoded;
141   end
```

```
142
143 endmodule: SevenSegmentDigit
144
145 module SevenSegmentControl
146   (output logic [6:0] HEX9, HEX8, HEX7, HEX6, HEX5,
147    output logic [6:0] HEX4, HEX3, HEX2, HEX1, HEX0,
148    input logic [3:0] BCH9, BCH8, BCH7, BCH6, BCH5,
149    input logic [3:0] BCH4, BCH3, BCH2, BCH1, BCH0);
150
151
152   logic turn_on;
153   assign turn_on = 0;
154   // instantiate for 10 digits
155   SevenSegmentDigit SSD0 (.BCH(BCH0), .segment(HEX0), .blank(turn_on));
156   SevenSegmentDigit SSD1 (.BCH(BCH1), .segment(HEX1), .blank(turn_on));
157   SevenSegmentDigit SSD2 (.BCH(BCH2), .segment(HEX2), .blank(turn_on));
158   SevenSegmentDigit SSD3 (.BCH(BCH3), .segment(HEX3), .blank(turn_on));
159   SevenSegmentDigit SSD4 (.BCH(BCH4), .segment(HEX4), .blank(~turn_on)); //no...
Line length of 83 (max is 80)
160   SevenSegmentDigit SSD5 (.BCH(BCH5), .segment(HEX5), .blank(~turn_on)); //no...
Line length of 83 (max is 80)
161   SevenSegmentDigit SSD6 (.BCH(BCH6), .segment(HEX6), .blank(turn_on));
162   SevenSegmentDigit SSD7 (.BCH(BCH7), .segment(HEX7), .blank(turn_on));
163   SevenSegmentDigit SSD8 (.BCH(BCH8), .segment(HEX8), .blank(turn_on));
164   SevenSegmentDigit SSD9 (.BCH(BCH9), .segment(HEX9), .blank(turn_on));
165
166 endmodule: SevenSegmentControl
167
168
169
170
171
172
173
174
175
176
177
178
179
180
```