

```

1  // Morris Huang, Hudson Wong
2  // 06/05/2025
3  // EE 271
4  // Lab 6 DE1_SoC
5
6  // This module simulates a game of frogger, where the player attempts to cross a
7  // road full of crossing cars. The game is simulated on an LED matrix, and if the player
8  // runs into the car, they lose. If they reach the end, they win, and the cars spawn and
9  // move faster.
10 // There are eight difficulties in total, and reset will reset the difficulty to 0.
11 module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, GPIO_1, SW);
12     input logic CLOCK_50;
13     output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
14     output logic [35:0] GPIO_1;
15     output logic [9:0] LEDR;
16     input logic [3:0] KEY;
17     input logic [9:0] SW;
18
19     logic [31:0] clk;
20
21     // Initializes two clock parameters. The whichClock controls the player movements,
22     // while crossyClock controls the car movements. This makes it so that the player can move
23     // faster than the cars.
24     parameter whichClock = 4;
25     parameter crossyClock = 25;
26
27     // Initializes the clock divider and the LED matrix
28     clock_divider cdiv (CLOCK_50, clk);
29     logic [15:0][15:0] RedPixels; // 16x16 array of red LEDs
30     logic [15:0][15:0] GrnPixels;
31
32     // Cycles through the LED matrix to light up each pixel
33     LEDDriver(.GPIO_1(GPIO_1), .RedPixels(RedPixels), .GrnPixels(GrnPixels), .EnableCount(
34 1'b1), .CLK(clk[5]), .RST(SW[0]));
35
36     logic reset_playfield, reset, L, R, U, D; // configure reset, the player moves, and the
37     gameover logic
38     logic [35:0] gameover; // Initializes gameover as a 36 bit array to take in the inputs
39     from all of the LEDs in the 6x6 block
40     logic [2:0] difficulty;
41     assign reset = SW[9]; // Reset when SW[9] is toggled
42
43     // Sends the input to a user input module to make sure that a long button
44     // press only registers as one input. Also implements a pair of flip flops
45     // for metastability.
46     // Key 0 corresponds to the right movement, key 1 to up, key 2 to down, and key 3 to
47     left.
48     user_input right(.clk(clk[whichClock]), .reset(reset), .button(~KEY[0]), .out(R));
49     user_input left(.clk(clk[whichClock]), .reset(reset), .button(~KEY[3]), .out(L));
50     user_input down(.clk(clk[whichClock]), .reset(reset), .button(~KEY[2]), .out(D));
51     user_input up(.clk(clk[whichClock]), .reset(reset), .button(~KEY[1]), .out(U));
52
53     // Initializes a 6x6 playing field where the player can move freely.
54     // Whenever the player moves right after reaching the rightmost edge, transfers
55     // them to the leftmost edge, and same for the left side. They cannot move down
56     // after reaching the bottom, but the frogger disappears after reaching the top and
57     pressing up
58     frog_LED f1010(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[10][11]), .
59 NR(GrnPixels[10][15]), .ND(GrnPixels[11][10]), .NU(1'b0), .L(L), .R(R), .D(D), .U(U), .
60 green_LED(GrnPixels[10][10]), .red_LED(RedPixels[10][10]), .gameover(gameover[0]));
61     frog_LED f1011(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[10][12]), .
62 NR(GrnPixels[10][10]), .ND(GrnPixels[11][11]), .NU(1'b0), .L(L), .R(R), .D(D), .U(U), .
63 green_LED(GrnPixels[10][11]), .red_LED(RedPixels[10][11]), .gameover(gameover[1]));
64     frog_LED f1012(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[10][13]), .
65 NR(GrnPixels[10][11]), .ND(GrnPixels[11][12]), .NU(1'b0), .L(L), .R(R), .D(D), .U(U), .
66 green_LED(GrnPixels[10][12]), .red_LED(RedPixels[10][12]), .gameover(gameover[2]));
67     frog_LED f1013(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[10][14]), .
68 NR(GrnPixels[10][12]), .ND(GrnPixels[11][13]), .NU(1'b0), .L(L), .R(R), .D(D), .U(U), .
69 green_LED(GrnPixels[10][13]), .red_LED(RedPixels[10][13]), .gameover(gameover[3]));
70     frog_LED f1014(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[10][15]), .
71 NR(GrnPixels[10][13]), .ND(GrnPixels[11][14]), .NU(1'b0), .L(L), .R(R), .D(D), .U(U), .
72 green_LED(GrnPixels[10][14]), .red_LED(RedPixels[10][14]), .gameover(gameover[4]));
73     frog_LED f1015(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[10][10]), .

```

```
61 NR(GrnPixels[10][14]), .ND(GrnPixels[11][15]), .NU(1'b0), .L(L), .R(R), .D(D), .U(U),  
62 green_LED(GrnPixels[10][15]), .red_LED(RedPixels[10][15]), .gameover(gameover[5]));  
63 frog_LED f1110(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[11][11]),  
64 NR(GrnPixels[11][15]), .ND(GrnPixels[12][10]), .NU(GrnPixels[10][10]), .L(L), .R(R), .D(D),  
65 .U(U), .green_LED(GrnPixels[11][10]), .red_LED(RedPixels[11][10]), .gameover(gameover[6]));  
66 frog_LED f1111(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[11][12]),  
67 NR(GrnPixels[11][10]), .ND(GrnPixels[12][11]), .NU(GrnPixels[10][11]), .L(L), .R(R), .D(D),  
68 .U(U), .green_LED(GrnPixels[11][11]), .red_LED(RedPixels[11][11]), .gameover(gameover[7]));  
69 frog_LED f1112(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[11][13]),  
70 NR(GrnPixels[11][11]), .ND(GrnPixels[12][12]), .NU(GrnPixels[10][12]), .L(L), .R(R), .D(D),  
71 .U(U), .green_LED(GrnPixels[11][12]), .red_LED(RedPixels[11][12]), .gameover(gameover[8]));  
72 frog_LED f1113(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[11][14]),  
73 NR(GrnPixels[11][12]), .ND(GrnPixels[12][13]), .NU(GrnPixels[10][13]), .L(L), .R(R), .D(D),  
74 .U(U), .green_LED(GrnPixels[11][13]), .red_LED(RedPixels[11][13]), .gameover(gameover[9]));  
75 frog_LED f1114(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[11][15]),  
76 NR(GrnPixels[11][13]), .ND(GrnPixels[12][14]), .NU(GrnPixels[10][14]), .L(L), .R(R), .D(D),  
77 .U(U), .green_LED(GrnPixels[11][14]), .red_LED(RedPixels[11][14]), .gameover(gameover[10]));  
78 frog_LED f1115(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[11][10]),  
79 NR(GrnPixels[11][14]), .ND(GrnPixels[12][15]), .NU(GrnPixels[10][15]), .L(L), .R(R), .D(D),  
80 .U(U), .green_LED(GrnPixels[11][15]), .red_LED(RedPixels[11][15]), .gameover(gameover[11]));  
81 frog_LED_start f1515(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[15][10]  
82 ), .NR(GrnPixels[15][14]), .ND(1'b0), .NU(GrnPixels[14][15] || GrnPixels[15][15]), .L(L),  
83 .R(R), .D(D), .U(U), .green_LED(GrnPixels[15][15]), .red_LED(RedPixels[15][15]), .gameover(  
84 gameover[12]));  
85 frog_LED f1514(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[15][15]),  
86 NR(GrnPixels[15][13]), .ND(1'b0), .NU(GrnPixels[14][14] || GrnPixels[15][14]), .L(L), .R(R),  
87 .D(D), .U(U), .green_LED(GrnPixels[15][14]), .red_LED(RedPixels[15][14]), .gameover(  
88 gameover[13]));  
89 frog_LED f1513(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[15][14]),  
90 NR(GrnPixels[15][12]), .ND(1'b0), .NU(GrnPixels[14][13] || GrnPixels[15][13]), .L(L), .R(R),  
91 .D(D), .U(U), .green_LED(GrnPixels[15][13]), .red_LED(RedPixels[15][13]), .gameover(  
92 gameover[14]));  
93 frog_LED f1512(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[15][13]),  
94 NR(GrnPixels[15][11]), .ND(1'b0), .NU(GrnPixels[14][12] || GrnPixels[15][12]), .L(L), .R(R),  
95 .D(D), .U(U), .green_LED(GrnPixels[15][12]), .red_LED(RedPixels[15][12]), .gameover(  
96 gameover[15]));  
97 frog_LED f1511(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[15][12]),  
98 NR(GrnPixels[15][10]), .ND(1'b0), .NU(GrnPixels[14][11] || GrnPixels[15][11]), .L(L), .R(R),  
99 .D(D), .U(U), .green_LED(GrnPixels[15][11]), .red_LED(RedPixels[15][11]), .gameover(  
100 gameover[16]));  
101 frog_LED f1510(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[15][11]),  
102 NR(GrnPixels[15][15]), .ND(1'b0), .NU(GrnPixels[14][10] || GrnPixels[15][10]), .L(L), .R(R),  
103 .D(D), .U(U), .green_LED(GrnPixels[15][10]), .red_LED(RedPixels[15][10]), .gameover(  
104 gameover[17]));  
105 frog_LED f1415(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[14][10]),  
106 NR(GrnPixels[14][14]), .ND(GrnPixels[15][15]), .NU(GrnPixels[13][15]), .L(L), .R(R), .D(D),  
107 .U(U), .green_LED(GrnPixels[14][15]), .red_LED(RedPixels[14][15]), .gameover(gameover[18]));  
108 frog_LED f1414(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[14][15]),  
109 NR(GrnPixels[14][13]), .ND(GrnPixels[15][14]), .NU(GrnPixels[13][14]), .L(L), .R(R), .D(D),  
110 .U(U), .green_LED(GrnPixels[14][14]), .red_LED(RedPixels[14][14]), .gameover(gameover[19]));  
111 frog_LED f1413(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[14][14]),  
112 NR(GrnPixels[14][12]), .ND(GrnPixels[15][13]), .NU(GrnPixels[13][13]), .L(L), .R(R), .D(D),  
113 .U(U), .green_LED(GrnPixels[14][13]), .red_LED(RedPixels[14][13]), .gameover(gameover[20]));  
114 frog_LED f1412(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[14][13]),  
115 NR(GrnPixels[14][11]), .ND(GrnPixels[15][12]), .NU(GrnPixels[13][12]), .L(L), .R(R), .D(D),  
116 .U(U), .green_LED(GrnPixels[14][12]), .red_LED(RedPixels[14][12]), .gameover(gameover[21]));  
117 frog_LED f1411(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[14][12]),  
118 NR(GrnPixels[14][10]), .ND(GrnPixels[15][11]), .NU(GrnPixels[13][11]), .L(L), .R(R), .D(D),  
119 .U(U), .green_LED(GrnPixels[14][11]), .red_LED(RedPixels[14][11]), .gameover(gameover[22]));  
120 frog_LED f1410(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[14][11]),  
121 NR(GrnPixels[14][15]), .ND(GrnPixels[15][10]), .NU(GrnPixels[13][10]), .L(L), .R(R), .D(D),  
122 .U(U), .green_LED(GrnPixels[14][10]), .red_LED(RedPixels[14][10]), .gameover(gameover[23]));  
123 frog_LED f1315(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[13][10]),  
124 NR(GrnPixels[13][14]), .ND(GrnPixels[14][15]), .NU(GrnPixels[12][15]), .L(L), .R(R), .D(D),  
125 .U(U), .green_LED(GrnPixels[13][15]), .red_LED(RedPixels[13][15]), .gameover(gameover[24]));  
126 frog_LED f1314(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[13][15]),  
127 NR(GrnPixels[13][13]), .ND(GrnPixels[14][14]), .NU(GrnPixels[12][14]), .L(L), .R(R), .D(D),  
128 .U(U), .green_LED(GrnPixels[13][14]), .red_LED(RedPixels[13][14]), .gameover(gameover[25]));  
129 frog_LED f1313(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[13][14]),  
130 NR(GrnPixels[13][12]), .ND(GrnPixels[14][13]), .NU(GrnPixels[12][13]), .L(L), .R(R), .D(D),  
131 .U(U), .green_LED(GrnPixels[13][13]), .red_LED(RedPixels[13][13]), .gameover(gameover[26]));  
132 frog_LED f1312(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[13][13]), .
```

```

87 NR(GrnPixels[13][11]), .ND(GrnPixels[14][12]), .NU(GrnPixels[12][12]), .L(L), .R(R), .D(D),
   .U(U), .green_LED(GrnPixels[13][12]), .red_LED(RedPixels[13][12]), .gameover(gameover[27]));
88 frog_LED f1311(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[13][12]), .
   NR(GrnPixels[13][10]), .ND(GrnPixels[14][11]), .NU(GrnPixels[12][11]), .L(L), .R(R), .D(D),
   .U(U), .green_LED(GrnPixels[13][11]), .red_LED(RedPixels[13][11]), .gameover(gameover[28]));
89 frog_LED f1310(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[13][11]), .
   NR(GrnPixels[13][15]), .ND(GrnPixels[14][10]), .NU(GrnPixels[12][10]), .L(L), .R(R), .D(D),
   .U(U), .green_LED(GrnPixels[13][10]), .red_LED(RedPixels[13][10]), .gameover(gameover[29]));
90 frog_LED f1215(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[12][10]), .
   NR(GrnPixels[12][14]), .ND(GrnPixels[13][15]), .NU(GrnPixels[11][15]), .L(L), .R(R), .D(D),
   .U(U), .green_LED(GrnPixels[12][15]), .red_LED(RedPixels[12][15]), .gameover(gameover[30]));
91 frog_LED f1214(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[12][15]), .
   NR(GrnPixels[12][13]), .ND(GrnPixels[13][14]), .NU(GrnPixels[11][14]), .L(L), .R(R), .D(D),
   .U(U), .green_LED(GrnPixels[12][14]), .red_LED(RedPixels[12][14]), .gameover(gameover[31]));
92 frog_LED f1213(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[12][14]), .
   NR(GrnPixels[12][12]), .ND(GrnPixels[13][13]), .NU(GrnPixels[11][13]), .L(L), .R(R), .D(D),
   .U(U), .green_LED(GrnPixels[12][13]), .red_LED(RedPixels[12][13]), .gameover(gameover[32]));
93 frog_LED f1212(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[12][13]), .
   NR(GrnPixels[12][11]), .ND(GrnPixels[13][12]), .NU(GrnPixels[11][12]), .L(L), .R(R), .D(D),
   .U(U), .green_LED(GrnPixels[12][12]), .red_LED(RedPixels[12][12]), .gameover(gameover[33]));
94 frog_LED f1211(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[12][12]), .
   NR(GrnPixels[12][10]), .ND(GrnPixels[13][11]), .NU(GrnPixels[11][11]), .L(L), .R(R), .D(D),
   .U(U), .green_LED(GrnPixels[12][11]), .red_LED(RedPixels[12][11]), .gameover(gameover[34]));
95 frog_LED f1210(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[12][11]), .
   NR(GrnPixels[12][15]), .ND(GrnPixels[13][10]), .NU(GrnPixels[11][10]), .L(L), .R(R), .D(D),
   .U(U), .green_LED(GrnPixels[12][10]), .red_LED(RedPixels[12][10]), .gameover(gameover[35]));
96
97 // Initializes the logic to randomize the car movement
98 logic random1, random2, random3, frogger_car1, frogger_car2, frogger_car3;
99 logic [9:0] random_cross1, random_cross2, random_cross3;
100
101 // Uses a seeded_LFSR to ensure that the cars spawn randomly
102 seeded_LFSR random_move1(.clk(clk[crossyClock - difficulty]), .reset(reset), .out(
   random_cross1), .seed(10'b0000000000));
103 seeded_LFSR random_move2(.clk(clk[crossyClock - difficulty]), .reset(reset), .out(
   random_cross2), .seed(10'b0000011111));
104 seeded_LFSR random_move3(.clk(clk[crossyClock - difficulty]), .reset(reset), .out(
   random_cross3), .seed(10'b0101010101));
105
106 // Uses a comparator to turn the 10 bit number from the LFSR into a singular output
107 comparator rand1(.inputA(random_cross1), .inputB(10'b0111111111), .A_greater_B(random1),
   .clk(clk[crossyClock - difficulty]), .reset(reset));
108 comparator rand2(.inputA(random_cross2), .inputB(10'b0111111111), .A_greater_B(random2),
   .clk(clk[crossyClock - difficulty]), .reset(reset));
109 comparator rand3(.inputA(random_cross3), .inputB(10'b0111111111), .A_greater_B(random3),
   .clk(clk[crossyClock - difficulty]), .reset(reset));
110
111 // Ensures that the car only spawns once if the output from comparator is true for
   multiple clock cycles
112 user_input car_move1(.clk(clk[crossyClock - difficulty]), .reset(reset), .button(random1
   ), .out(frogger_car1));
113 user_input car_move2(.clk(clk[crossyClock - difficulty]), .reset(reset), .button(random2
   ), .out(frogger_car2));
114 user_input car_move3(.clk(clk[crossyClock - difficulty]), .reset(reset), .button(random3
   ), .out(frogger_car3));
115
116 // Initializes the logic for the cars. Each car moves to the right every clock cycle until
117 // it reaches the edge and disappears. Uses the LFSR to randomly determine when the
118 // cars spawn. The cars are represented by red LEDs. They spawn on the 2nd, 4th, and 5th
   row
119 crossy_road c1415(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
   frogger_car1), .red_LED(RedPixels[14][15]));
120 crossy_road c1414(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
   RedPixels[14][15]), .red_LED(RedPixels[14][14]));
121 crossy_road c1413(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
   RedPixels[14][14]), .red_LED(RedPixels[14][13]));
122 crossy_road c1412(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
   RedPixels[14][13]), .red_LED(RedPixels[14][12]));
123 crossy_road c1411(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
   RedPixels[14][12]), .red_LED(RedPixels[14][11]));
124 crossy_road c1410(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
   RedPixels[14][11]), .red_LED(RedPixels[14][10]));
125
126 crossy_road c1215(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(

```

```

frogger_car2), .red_LED(RedPixels[12][15]));
128 crossy_road c1214(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[12][15]), .red_LED(RedPixels[12][14]));
129 crossy_road c1213(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[12][14]), .red_LED(RedPixels[12][13]));
130 crossy_road c1212(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[12][13]), .red_LED(RedPixels[12][12]));
131 crossy_road c1211(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[12][12]), .red_LED(RedPixels[12][11]));
132 crossy_road c1210(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[12][11]), .red_LED(RedPixels[12][10]));
133
134
135 crossy_road c1015(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
frogger_car3), .red_LED(RedPixels[10][15]));
136 crossy_road c1014(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[10][15]), .red_LED(RedPixels[10][14]));
137 crossy_road c1013(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[10][14]), .red_LED(RedPixels[10][13]));
138 crossy_road c1012(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[10][13]), .red_LED(RedPixels[10][12]));
139 crossy_road c1011(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[10][12]), .red_LED(RedPixels[10][11]));
140 crossy_road c1010(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[10][11]), .red_LED(RedPixels[10][10]));
141
142
143
144 victory v0(.clk(clk[whichClock]), .reset(reset), .TopLED(GrnPixels[10][10] || GrnPixels[10]
[11] || GrnPixels[10][12] || GrnPixels[10][13] || GrnPixels[10][14] || GrnPixels[10][15]),
.UpKey(U), .DownKey(D), .gameover(!gameover), .win(win), .lose(lose), .reset_playfield(
reset_playfield), .difficulty(difficulty));
145
146 endmodule
147
148 // This testbench verifies the behavior of the DE1_SoC module, ensuring that the
149 // module instantiates a board that can run a game of Frogger, randomize the car movement,
and so on.
150 // The parameters whichClock and crossyClock must be set to zero for this testbench to
function, and the LED
151 // Driver must not be initialized.
152 module DE1_SoC_testbench ();
153     logic CLOCK_50;
154     logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
155     logic [35:0] GPIO_1;
156     logic [9:0] LEDR;
157     logic [3:0] KEY;
158     logic [9:0] SW;
159
160
161     DE1_SoC dut(CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, GPIO_1, SW);
162
163
164     parameter clock_period = 100;
165
166     // toggle CLOCK_50 every half cycle
167     initial begin
168         CLOCK_50 <= 0;
169         forever #(clock_period / 2) CLOCK_50 <= ~CLOCK_50;
170     end
171
172
173     initial begin
174         KEY[0] <= 0; KEY[1] <= 0; KEY[2] <= 0; KEY[3] <= 0; SW[9] <= 0; @(posedge CLOCK_50);
// Sets all relevant variables to 0.
175
176         SW[9] <= 1; @(posedge CLOCK_50);
177         SW[9] <= 0; @(posedge CLOCK_50); //Toggle reset
178
179         repeat(20) @(posedge CLOCK_50); //Tests the car functionality
180         KEY[2] <= 1; KEY[1] <= 1; repeat(10) @(posedge CLOCK_50); // Tests that opposite keys
do not move the frog
181         KEY[2] <= 0; KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);
182         KEY[0] <= 1; KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
183         KEY[0] <= 0; KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);
184

```



```
185
186 works KEY[0] <= 1; repeat(10) @(posedge CLOCK_50); // Tests that each of the movement keys
187 KEY[0] <= 0; repeat(10)@(posedge CLOCK_50);
188 KEY[1] <= 1; repeat(10)@(posedge CLOCK_50);
189 KEY[1] <= 0; repeat(10)@(posedge CLOCK_50);
190 KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
191 KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
192 KEY[3] <= 1; repeat(10)@(posedge CLOCK_50);
193 KEY[3] <= 0; repeat(10)@(posedge CLOCK_50);
194
195 KEY[3] <= 1; repeat(10)@(posedge CLOCK_50); // Frog moves and wins
196 KEY[3] <= 0; repeat(10)@(posedge CLOCK_50);
197 KEY[3] <= 1; repeat(10)@(posedge CLOCK_50);
198 KEY[3] <= 0; repeat(10)@(posedge CLOCK_50);
199 KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
200 KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
201 KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
202 KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
203 KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
204 KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
205 KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
206 KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
207 KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
208 KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
209 KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
210 KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
211
212 // Random moves to ensure everything works
213 KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
214 KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);
215
216 KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
217 KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);
218
219 KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
220 KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);
221
222 KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
223 KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);
224
225 KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
226 KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);
227
228 KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
229 KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);
230
231 KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
232 KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);
233
234 KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
235 KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);
236
237 KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
238 KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);
239
240 KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
241 KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);
242
243 KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
244 KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);
245
246 KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
247 KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);
248
249 KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
250 KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);
251
252 KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
253 KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);
254
255 KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
256 KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);
257
258 KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
259 KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);
```

```
260
261 KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
262 KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);
263
264 KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
265 KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);
266
267 KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
268 KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);
269
270 KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
271 KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);
272
273 KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
274 KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);
275
276 KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
277 KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);
278
279 KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
280 KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);
281
282 KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
283 KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);
284
285 KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
286 KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);
287
288 KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
289 KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);
290
291 KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
292 KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);
293
294 KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
295 KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);
296
297 KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
298 KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);
299
300 KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
301 KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);
302
303 KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
304 KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);
305
306 KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
307 KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);
308
309 KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
310 KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);
311
312 KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
313 KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);
314
315 KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
316 KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);
317
318 KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
319 KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);
320
321 KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
322 KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);
323
324 KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
325 KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);
326
327 KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
328 KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);
329
330 KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
331 KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);
332
333 KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
334 KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);
335
```

```
336         SW[9] <= 1; repeat(10)@(posedge CLOCK_50); // Toggle reset
337         SW[9] <= 0; @(posedge CLOCK_50);
338
339         KEY[2] <= 1; repeat(10)@(posedge CLOCK_50); // Frog runs into a car and resets
340         KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
341         repeat(20)    @(posedge CLOCK_50);
342
343         $stop;
344     end
345 endmodule
346
```

```

1  // Morris Huang, Hudson Wong
2  // 06/04/2025
3  // EE 271
4  // Victory Module
5
6  // This module determines when the player wins by reaching the top of the field or
7  // loses when it collides with a car. In either case, it resets the playfield.
8  module victory (clk, reset, UpKey, DownKey, win, lose, reset_playfield, gameover, TopLED,
9  difficulty);
10     input logic clk, reset, UpKey, DownKey, gameover, TopLED;
11     output logic win, lose;
12     output logic reset_playfield;
13     output logic [2:0] difficulty;
14
15     assign win = (TopLED & UpKey & ~DownKey);
16     assign lose = gameover;
17     three_bit_counter counter(.clk(clk), .reset(reset), .in(win), .out(difficulty));
18
19     // Manage reset_playfield signal. If the player wins or loses or if the reset switch is
20     // toggled,
21     // it resets the field to its default state.
22     always_ff @(posedge clk) begin
23         if (reset)
24             reset_playfield <= 1;
25         else if (win || lose)
26             reset_playfield <= 1;
27         else
28             reset_playfield <= 0;
29     end
30 endmodule
31
32 // This testbench verifies the logic of the victory module
33 // It provides a clock signal and the inputs and tests whether it
34 // correctly displays the win and lose conditions as well as increment the difficulty.
35 module victory_testbench();
36     logic clk, reset, UpKey, DownKey, gameover, TopLED, win, lose, reset_playfield;
37     logic [2:0] difficulty;
38
39     // Instantiate Device under test
40     victory dut(clk, reset, UpKey, DownKey, win, lose, reset_playfield, gameover, TopLED,
41 difficulty);
42
43     parameter clock_period = 100;
44
45     // Runs every 100/2 picoseconds for each clock cycle
46     initial begin
47         clk <= 0;
48         forever #(clock_period /2) clk <= ~clk;
49     end
50
51     initial begin
52         reset <= 1; UpKey <= 0; DownKey <= 0; TopLED <= 0; gameover <= 0; @(posedge clk);
53         reset <= 0; @(posedge clk);
54
55         // Test winning conditions and difficulty logic
56         UpKey <= 1; DownKey <= 0; TopLED <= 1; @(posedge clk);
57         TopLED <= 0; @(posedge clk);
58
59         UpKey <= 1; DownKey <= 0; TopLED <= 1; @(posedge clk);
60         TopLED <= 0; @(posedge clk);
61         UpKey <= 1; DownKey <= 0; TopLED <= 1; @(posedge clk);
62         TopLED <= 0; @(posedge clk);
63         @(posedge clk);
64
65         // Test losing conditions
66         UpKey <= 1; DownKey <= 0; gameover <= 1; @(posedge clk);
67
68         // Test regular in game condition
69         UpKey <= 0; DownKey <= 0; TopLED <= 1; gameover <= 0; @(posedge clk);
70         UpKey <= 0; DownKey <= 0; TopLED <= 0; gameover <= 0; @(posedge clk);
71
72
73

```



```
74         end
75
76     endmodule
77
78
79
80
81
82
```

```

1 // Morris Huang, Hudson Wong
2 // 05/22/2025
3 // EE 271
4 // Lab 6 10-bit LFSR
5
6 // The LFSR module implements a 10-bit Linear Feedback Shift Register (LFSR)
7 // using XNOR feedback logic. It shifts right on every clock cycle and
8 // inserts a new bit into the MSB (bit 9) calculated as the XNOR of
9 // bit 0 and bit 3. When reset, it goes to a predetermined seed that is passed in
10 module seeded_LFSR(clk, reset, out, seed);
11     input logic clk, reset;
12     input logic [9:0] seed;
13     output logic [9:0] out;
14
15     // LFSR logic triggered on the rising edge of the clock
16     always_ff @(posedge clk)
17     begin
18         if(reset)
19             out<=seed;
20         else
21             begin
22                 // Perform right shift on the entire register
23                 out <= out>>1;
24                 // Insert feedback bit at MSB (bit 9) using XNOR of bits 0 and 3 or
25                 // bits 7 and 10.
26                 out[9] <= ~(out[0] ^ out[3]);
27             end
28     end
29
30 endmodule
31
32 // This testbench verifies the shift and xnor logic of the LFSR.
33 // It provides a clock signal and applies a synchronous reset, then
34 // allows the LFSR to run for 1024 clock cycles to observe its output.
35 module seeded_LFSR_testbench();
36     logic CLOCK_50;
37     logic reset;
38     logic [9:0] out, seed;
39
40     // Instantiate the LFSR module DUT
41     seeded_LFSR dut(CLOCK_50, reset, out, seed);
42
43     parameter clock_period = 100;
44
45     // toggle CLOCK_50 every half cycle
46     initial begin
47         CLOCK_50 <= 0;
48         forever #(clock_period / 2) CLOCK_50 <= ~CLOCK_50;
49     end
50
51     // apply reset and then run the LFSR
52     initial begin
53         seed <= 10'b0000000000;
54         // Apply reset for one clock cycle
55         reset <= 1; @(posedge CLOCK_50);
56         reset <= 0; @(posedge CLOCK_50);
57         // Let the LFSR run for 1024 clock cycles
58         repeat(1024) @(posedge CLOCK_50);
59
60         seed <= 10'b0100011100; // Test different seeds
61         // Apply reset for one clock cycle
62         reset <= 1; @(posedge CLOCK_50);
63         reset <= 0; @(posedge CLOCK_50);
64         // Let the LFSR run for 1024 clock cycles
65         repeat(20) @(posedge CLOCK_50);
66
67         seed <= 10'b0110011101; // Test different seeds
68         // Apply reset for one clock cycle
69         reset <= 1; @(posedge CLOCK_50);
70         reset <= 0; @(posedge CLOCK_50);
71         // Let the LFSR run for 1024 clock cycles
72         repeat(20) @(posedge CLOCK_50);
73
74         $stop;
75     end
76

```

```
77  
78     endmodule  
79
```

```
1 // Morris Huang, Hudson Wong
2 // 06/05/2025
3 // EE 271
4 // Lab 6 crossy_road
5
6 // This module simulates the logic for the cars running across the LED matrix using an FSM.
7 // It moves each car (LED) to the right every clock cycle.
8 // By passing in a random "output" from the LFSR, it can also randomly spawn cars
9 module crossy_road (clk, reset, NL, red_LED);
10
11 // Initializes the logic. NL represents the LED to the left,
12 // or in the case of the starting LED, the LFSR output
13 input logic clk, reset, NL;
14 output logic red_LED;
15
16 // Starts the FSM logic
17 enum {S0, S1} ps, ns;
18
19 // If the left LED is on, turns the current LED to red.
20 // Otherwise, turns off. This effectively shifts the car right every clock cycle
21 always_comb begin
22     case (ps)
23         S0: if (NL) ns = S1;
24             else ns = S0;
25         S1: ns = S0;
26     endcase
27 end
28
29 assign red_LED = (ps == S1);
30
31 // Toggles reset such that the light does not turn on.
32 always_ff @(posedge clk) begin
33     if (reset)
34         ps <= S0;
35     else
36         ps <= ns;
37 end
38
39 endmodule
40
41 // This module functions as a test bench for the crossy_road module.
42 // Verifies that the crossy_road LED turns on for exactly one clock cycle
43 // when the LED to the left of it is turned on.
44 module crossy_road_testbench();
45     logic CLOCK_50, reset, NL, red_LED; // Initializes the logic
46
47     crossy_road dut(CLOCK_50, reset, NL, red_LED); // Instantiates the module
48
49     parameter clock_period = 100;
50
51     // toggle CLOCK_50 every half cycle
52     initial begin
53         CLOCK_50 <= 0;
54         forever #(clock_period / 2) CLOCK_50 <= ~CLOCK_50;
55     end
56
57     initial begin
58         reset <= 1; @(posedge CLOCK_50); // Tests the reset
59         reset <= 0; @(posedge CLOCK_50);
60
61         NL <= 1; @(posedge CLOCK_50); // Tests that the LED lights up
62         NL <= 0;
63         @(posedge CLOCK_50); // Tests that the LED turns off after one clock cycle.
64         @(posedge CLOCK_50);
65         repeat(10) @(posedge CLOCK_50); // Tests that no random behavior happens.
66
67         $stop;
68     end
69
70 endmodule
71
72 endmodule
73
```

```

1 // Morris Huang, Hudson Wong
2 // 06/05/2025
3 // EE 271
4 // Lab 6 frog_LED_start
5
6 // This module represents the starting point for the frogger player using an FSM
7 // On reset, it automatically turns on, then it allows the player to move the LED around.
8 // Only one LED is lit at a time. If both the green LED and red LED are lit at the same time,
9 // it shows a game over.
10 module frog_LED_start (clk, reset, NL, NR, ND, NU, L, R, D, U, green_LED, red_LED, gameover);
11 // Initializes the logic. NL represents the LED to the left, NR is the LED to the right,
12 // ND is the bottom LED, and NU is the top LED.
13 // L, R, D, and U represent the player movements
14 input logic clk, reset, NL, NR, ND, NU, L, R, D, U, red_LED;
15 output logic gameover, green_LED;
16
17 // Initializes FSM logic.
18 enum {S0, S1} ps, ns;
19
20 // This is the logic for the FSM. If the adjacent LED is lit and the corresponding input
21 // is true, the LED turns green. If it is lit, having the corresponding input turn on
22 // will turn the LED off.
23 always_comb begin
24     case (ps)
25         S0: if ((NL&R&~L) || (NR&L&~R) || (NU&D&~U) || (ND&U&~D)) ns = S1;
26             else ns = S0;
27         S1: if ((R & ~L & ~NL) || (L & ~R & ~NR) || (U & ~D & ~ND) || (D & ~U & ~NU)) ns =
S0;
             else ns = S1;
28     endcase
29 end
30
31 // Assigns the state of the LED and whether the game is over.
32 assign green_LED = (ps == S1);
33 assign gameover = green_LED & red_LED;
34
35 // On reset, turns on the starting LED.
36 always_ff @(posedge clk) begin
37     if (reset)
38         ps <= S1;
39     else
40         ps <= ns;
41 end
42
43 endmodule
44
45 // This testbench verifies the logic of the frog_LED_start module.
46 // It provides a clock signal and applies a synchronous reset, then
47 // allows the frog_LED_start module to run under different conditions to test whether it
48 // lights up and turns off appropriately.
49 module frog_LED_start_testbench ();
50     logic CLOCK_50, reset, NL, NR, ND, NU, L, R, D, U, green_LED, red_LED, gameover; //
Initializes the logic
51
52     frog_LED_start dut(CLOCK_50, reset, NL, NR, ND, NU, L, R, D, U, green_LED, red_LED,
gameover); // Instantiates the module
53
54     parameter clock_period = 100;
55
56     // toggle CLOCK_50 every half cycle
57     initial begin
58         CLOCK_50 <= 0;
59         forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
60     end
61
62     initial begin
63         reset <= 1; @(posedge CLOCK_50); // Tests the reset turns on the green LED.
64         reset <= 0; @(posedge CLOCK_50);
65
66         R <= 1; @(posedge CLOCK_50); // Turns the LED off.
67         R <= 0; @(posedge CLOCK_50);
68
69         NL <= 0; NR <= 0; ND <= 0; NU <= 0; L <= 0; R <= 0; D <= 0; U <= 0; red_LED <= 0; //
Initializes all logic to 0.
70
71         ND <= 1; U <= 1; @(posedge CLOCK_50); // Bottom LED is lit, and the up button is

```


pressed.

72 ND <= 0; U <= 0; @(posedge CLOCK_50); // The green_LED should be lit currently.

73
74 U <= 1; D <= 1; @(posedge CLOCK_50); // Tests that LED will not turn off with
conflicting buttons

75 U <= 0; D <= 0; @(posedge CLOCK_50);

76 R <= 1; L <= 1; @(posedge CLOCK_50);

77 R <= 0; L <= 0; @(posedge CLOCK_50);

78 U <= 1; D <= 1; R <= 1; L <= 1; @(posedge CLOCK_50);

79 U <= 0; D <= 0; R <= 0; L <= 0; @(posedge CLOCK_50);

80
81 U <= 1; @(posedge CLOCK_50); // With one button press, the LED will turn off.

82 U <= 0; @(posedge CLOCK_50);

83 NR <= 1; L <= 1; @(posedge CLOCK_50); // turn LED back on

84 NR <= 0; L <= 0; @(posedge CLOCK_50);

85
86 R <= 1; @(posedge CLOCK_50); // should turn off

87 R <= 0; @(posedge CLOCK_50);

88 NL <= 1; R <= 1; @(posedge CLOCK_50); // turn LED back on

89 NL <= 0; R <= 0; @(posedge CLOCK_50);

90
91 L <= 1; @(posedge CLOCK_50); // should turn off

92 L <= 0; @(posedge CLOCK_50);

93 NU <= 1; D <= 1; @(posedge CLOCK_50); // turn LED back on

94 NU <= 0; D <= 0; @(posedge CLOCK_50);

95
96 D <= 1; @(posedge CLOCK_50); // should turn off

97 D <= 0; @(posedge CLOCK_50);

98 ND <= 1; U <= 1; @(posedge CLOCK_50); // turn LED back on

99 ND <= 0; U <= 0; @(posedge CLOCK_50);

100
101 red_LED <= 1; @(posedge CLOCK_50); // Test gameover condition102
103 \$stop;104 end
105 endmodule
106

```

1 // Morris Huang, Hudson Wong
2 // 06/05/2025
3 // EE 271
4 // Lab 6 frog_LED_start
5
6 // This module represents the normal LEDs for the frogger player using an FSM
7 // On reset, it automatically turns off, then it allows the player to move the LED around.
8 // Only one LED is lit at a time. If both the green LED and red LED are lit at the same time,
9 // it shows a game over.
10 module frog_LED (clk, reset, NL, NR, ND, NU, L, R, D, U, green_LED, red_LED, gameover);
11
12 // Initializes the logic. NL represents the LED to the left, NR is the LED to the right,
13 // ND is the bottom LED, and NU is the top LED.
14 // L, R, D, and U represent the player movements
15 input logic clk, reset, NL, NR, ND, NU, L, R, D, U, red_LED;
16 output logic green_LED, gameover;
17
18 // Initializes FSM logic.
19 enum {S0, S1} ps, ns;
20
21 // This is the logic for the FSM. If the adjacent LED is lit and the corresponding input
22 // is true, the LED turns green. If it is lit, having the corresponding input turn on
23 // will turn the LED off.
24 always_comb begin
25     case (ps)
26         S0: if ((NL&R&~L) || (NR&L&~R) || (NU&D&~U) || (ND&U&~D)) ns = S1;
27             else ns = S0;
28         S1: if ((R & ~L & ~NL) || (L & ~R & ~NR) || (U & ~D & ~ND) || (D & ~U & ~NU)) ns =
S0;
             else ns = S1;
29     endcase
30 end
31
32 // Assigns the state of the LED
33 assign green_LED = (ps == S1);
34 assign gameover = (ps == S1) & red_LED;
35
36 // On reset, turns off the LED.
37 always_ff @(posedge clk) begin
38     if (reset)
39         ps <= S0;
40     else
41         ps <= ns;
42 end
43
44 endmodule
45
46 // This testbench verifies the logic of the frog_LED module.
47 // It provides a clock signal and applies a synchronous reset, then
48 // allows the frog_LED module to run under different conditions to test whether it lights
49 // up and turns off appropriately.
50 module frog_LED_testbench();
51     logic CLOCK_50, reset, NL, NR, ND, NU, L, R, D, U, green_LED, red_LED, gameover; //
52     Initializes the logic
53
54     frog_LED dut(CLOCK_50, reset, NL, NR, ND, NU, L, R, D, U, green_LED, red_LED, gameover);
55 // Instantiates the module
56
57     parameter clock_period = 100;
58
59     // toggle CLOCK_50 every half cycle
60     initial begin
61         CLOCK_50 <= 0;
62         forever #(clock_period / 2) CLOCK_50 <= ~CLOCK_50;
63     end
64
65     initial begin
66         reset <= 1; @(posedge CLOCK_50); // Tests the reset
67         reset <= 0; @(posedge CLOCK_50);
68
69         NL <= 0; NR <= 0; ND <= 0; NU <= 0; L <= 0; R <= 0; D <= 0; U <= 0; red_LED <= 0; //
70         Initializes all logic to 0.
71
72         ND <= 1; U <= 1; @(posedge CLOCK_50); // Bottom LED is lit, and the up button is
73         pressed.

```

```
71      ND <= 0; U <= 0; @(posedge CLOCK_50); // The green_LED should be lit currently.
72
73      U <= 1; D <= 1; @(posedge CLOCK_50); // Tests that LED will not turn off with
conflicting buttons
74      U <= 0; D <= 0; @(posedge CLOCK_50);
75      R <= 1; L <= 1; @(posedge CLOCK_50);
76      R <= 0; L <= 0; @(posedge CLOCK_50);
77      U <= 1; D <= 1; R <= 1; L <= 1; @(posedge CLOCK_50);
78      U <= 0; D <= 0; R <= 0; L <= 0; @(posedge CLOCK_50);
79
80      U <= 1; @(posedge CLOCK_50); // With one button press, the LED will turn off.
81      U <= 0; @(posedge CLOCK_50);
82      NR <= 1; L <= 1; @(posedge CLOCK_50); // turn LED back on
83      NR <= 0; L <= 0; @(posedge CLOCK_50);
84
85      R <= 1; @(posedge CLOCK_50); // should turn off
86      R <= 0; @(posedge CLOCK_50);
87      NL <= 1; R <= 1; @(posedge CLOCK_50); // turn LED back on
88      NL <= 0; R <= 0; @(posedge CLOCK_50);
89
90      L <= 1; @(posedge CLOCK_50); // should turn off
91      L <= 0; @(posedge CLOCK_50);
92      NU <= 1; D <= 1; @(posedge CLOCK_50); // turn LED back on
93      NU <= 0; D <= 0; @(posedge CLOCK_50);
94
95      D <= 1; @(posedge CLOCK_50); // Should turn off
96      D <= 0; @(posedge CLOCK_50);
97      ND <= 1; U <= 1; @(posedge CLOCK_50); // turn LED back on
98      ND <= 0; U <= 0; @(posedge CLOCK_50);
99
100     red_LED <= 1; @(posedge CLOCK_50); // Test gameover condition
101
102     $stop;
103 end
104 endmodule
```

```

1  //Acknowledgement: This driver code was provided by Prof. Scott Hauck as part of his
2  // EE 271 course.
3
4  // A driver for the 16x16x2 LED display expansion board.
5  // Read below for an overview of the ports.
6  // IMPORTANT: You do not need to necessarily modify this file. But if you do, be sure you
   know what you are doing.
7
8  // FREQDIV: (Parameter) Sets the scanning speed (how often the display cycles through rows)
9  //           The CLK input divided by 2^(FREQDIV) is the interval at which the driver
   switches rows.
10 // GPIO_1: (Output) The 36-pin GPIO1 header, as on the DE1-SoC board.
11 // RedPixels: (Input) A 16x16 array of logic items corresponding to the red pixels you'd
   like to have lit on the display.
12 // GrnPixels: (Input) A 16x16 array of logic items corresponding to the green pixels you'd
   like to have lit on the display.
13 // EnableCount: (Input) Whether to continue moving through the rows.
14 // CLK: (Input) The system clock.
15 // RST: (Input) Resets the display driver. Required during startup before use.
16 module LEDDriver #(parameter FREQDIV = 8) (GPIO_1, RedPixels, GrnPixels, EnableCount, CLK,
   RST);
17     output logic [35:0] GPIO_1;
18     input logic [15:0][15:0] RedPixels ;
19     input logic [15:0][15:0] GrnPixels ;
20     input logic EnableCount, CLK, RST;
21
22     reg [(FREQDIV + 3):0] Counter;
23     logic [3:0] RowSelect;
24     assign RowSelect = Counter[(FREQDIV + 3):(FREQDIV + 0)];
25
26     always_ff @(posedge CLK)
27     begin
28         if(RST) Counter <= 'b0;
29         if(EnableCount) Counter <= Counter + 1'b1;
30     end
31
32     assign GPIO_1[35:32] = RowSelect;
33     assign GPIO_1[31:16] = { GrnPixels[RowSelect][0], GrnPixels[RowSelect][1], GrnPixels[
   RowSelect][2], GrnPixels[RowSelect][3], GrnPixels[RowSelect][4], GrnPixels[RowSelect][5],
   GrnPixels[RowSelect][6], GrnPixels[RowSelect][7], GrnPixels[RowSelect][8], GrnPixels[
   RowSelect][9], GrnPixels[RowSelect][10], GrnPixels[RowSelect][11], GrnPixels[RowSelect][12],
   GrnPixels[RowSelect][13], GrnPixels[RowSelect][14], GrnPixels[RowSelect][15] };
34     assign GPIO_1[15:0] = { RedPixels[RowSelect][0], RedPixels[RowSelect][1], RedPixels[
   RowSelect][2], RedPixels[RowSelect][3], RedPixels[RowSelect][4], RedPixels[RowSelect][5],
   RedPixels[RowSelect][6], RedPixels[RowSelect][7], RedPixels[RowSelect][8], RedPixels[
   RowSelect][9], RedPixels[RowSelect][10], RedPixels[RowSelect][11], RedPixels[RowSelect][12],
   RedPixels[RowSelect][13], RedPixels[RowSelect][14], RedPixels[RowSelect][15] };
35 endmodule
36
37 module LEDDriver_Test();
38     logic CLK, RST, EnableCount;
39     logic [15:0][15:0] RedPixels;
40     logic [15:0][15:0] GrnPixels;
41     logic [35:0] GPIO_1;
42
43     LEDDriver #(.FREQDIV(2)) Driver(.GPIO_1, .RedPixels, .GrnPixels, .EnableCount, .CLK, .
   RST);
44
45     initial
46     begin
47         CLK <= 1'b0;
48         forever #50 CLK <= ~CLK;
49     end
50
51     initial
52     begin
53         EnableCount <= 1'b0;
54         RedPixels <= '{default:0};
55         GrnPixels <= '{default:0};
56         @(posedge CLK);
57
58         RST <= 1; @(posedge CLK);
59         RST <= 0; @(posedge CLK);
60         @(posedge CLK); @(posedge CLK); @(posedge CLK);
61
62         GrnPixels[1][1] <= 1'b1; @(posedge CLK);

```

```

63     EnableCount <= 1'b1; @(posedge CLK); #1000;
64     RedPixels[2][2] <= 1'b1;
65     RedPixels[2][3] <= 1'b1;
66     GrnPixels[2][3] <= 1'b1; @(posedge CLK); #1000;
67     EnableCount <= 1'b0; @(posedge CLK); #1000;
68     GrnPixels[1][1] <= 1'b0; @(posedge CLK);
69     $stop;
70
71     end
72 endmodule
73
74 module LEDDriver_TestPhysical(CLOCK_50, RST, Speed, GPIO_1);
75     input logic CLOCK_50, RST;
76     input logic [9:0] Speed;
77     output logic [35:0] GPIO_1;
78     logic [15:0][15:0] RedPixels;
79     logic [15:0][15:0] GrnPixels;
80     logic [31:0] Counter;
81     logic EnableCount;
82
83     LEDDriver #(.FREQDIV(15)) Driver (.CLK(CLOCK_50), .RST, .EnableCount, .RedPixels, .
GrnPixels, .GPIO_1);
84
85     //
86     assign RedPixels[00] = '{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
87     assign RedPixels[01] = '{1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1};
88     assign RedPixels[02] = '{1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1};
89     assign RedPixels[03] = '{1,0,1,1,0,0,0,0,0,0,0,0,0,1,1,0};
90     assign RedPixels[04] = '{1,0,1,0,1,1,1,1,1,1,1,1,1,0,1,0};
91     assign RedPixels[05] = '{1,0,1,0,1,1,0,0,0,0,1,1,0,1,0,1};
92     assign RedPixels[06] = '{1,0,1,0,1,0,1,1,1,1,0,1,0,1,0,1};
93     assign RedPixels[07] = '{1,0,1,0,1,0,1,0,1,1,0,1,0,1,0,1};
94     assign RedPixels[08] = '{1,0,1,0,1,0,1,1,0,1,0,1,0,1,0,1};
95     assign RedPixels[09] = '{1,0,1,0,1,0,1,1,1,1,0,1,0,1,0,1};
96     assign RedPixels[10] = '{1,0,1,0,1,1,0,0,0,0,1,1,0,1,0,1};
97     assign RedPixels[11] = '{1,0,1,0,1,1,1,1,1,1,1,1,0,1,0,1};
98     assign RedPixels[12] = '{1,0,1,1,0,0,0,0,0,0,0,0,1,1,0,1};
99     assign RedPixels[13] = '{1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1};
100    assign RedPixels[14] = '{1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1};
101    assign RedPixels[15] = '{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
102
103    assign GrnPixels[00] = '{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
104    assign GrnPixels[01] = '{0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0};
105    assign GrnPixels[02] = '{0,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1};
106    assign GrnPixels[03] = '{0,1,0,1,1,1,1,1,1,1,1,1,1,1,0,1};
107    assign GrnPixels[04] = '{0,1,0,1,1,0,0,0,0,0,0,0,1,1,0,1};
108    assign GrnPixels[05] = '{0,1,0,1,0,1,1,1,1,1,1,0,1,0,1,0};
109    assign GrnPixels[06] = '{0,1,0,1,0,1,1,0,0,1,1,0,1,0,1,0};
110    assign GrnPixels[07] = '{0,1,0,1,0,1,0,1,0,0,1,0,1,0,1,0};
111    assign GrnPixels[08] = '{0,1,0,1,0,1,0,0,1,0,1,0,1,0,1,0};
112    assign GrnPixels[09] = '{0,1,0,1,0,1,1,0,0,1,1,0,1,0,1,0};
113    assign GrnPixels[10] = '{0,1,0,1,0,1,1,1,1,1,1,0,1,0,1,0};
114    assign GrnPixels[11] = '{0,1,0,1,1,0,0,0,0,0,0,1,1,0,1,0};
115    assign GrnPixels[12] = '{0,1,0,1,1,1,1,1,1,1,1,1,1,0,1,0};
116    assign GrnPixels[13] = '{0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0};
117    assign GrnPixels[14] = '{0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0};
118    assign GrnPixels[15] = '{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
119
120    always_ff @(posedge CLOCK_50)
121    begin
122        if(RST) Counter <= 'b0;
123        else
124        begin
125            Counter <= Counter + 1'b1;
126            if(Counter >= Speed)
127            begin
128                EnableCount <= 1'b1;
129                Counter <= 'b0;
130            end
131            else EnableCount <= 1'b0;
132        end
133    end
134 endmodule

```



```
1 // Morris Huang, Hudson Wong
2 // 06/07/2025
3 // EE 271
4 // Lab 6 clock_divider
5
6 // Takes in a clock signal, divides the clock cycle and outputs 32
7 // divided clock signals of varying frequency.
8 // divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ...
9 // [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ...
10 module clock_divider (clock, divided_clocks);
11
12     input logic clock;
13     output logic [31:0] divided_clocks = 32'b0;
14
15     // Adds 1 to the logic divided_clocks at the
16     // positive edge of each clock cycle.
17     // Given the nature of 32 bit numbers,
18     // the lowest bit toggles between 1 and 0 twice as fast
19     // as the next bit, the next bit toggles twice as fast
20     // as the next higher bit, and so on. This allows
21     // the clock divider to halve the frequency up to 31 times.
22     always_ff @(posedge clock) begin
23         divided_clocks <= divided_clocks + 1;
24     end
25 endmodule
26
27 // Tests whether the clock divider, given an initialized
28 // clock, properly outputs the divided frequency signals.
29 module clock_divider_testbench ();
30     logic clock;
31     logic [31:0] divided_clocks;
32
33     clock_divider dut (.clock(clock), .divided_clocks(divided_clocks));
34
35     // Sets up a clock that toggles every 10 nanoseconds
36     initial begin
37         clock <= 0;
38         forever #10 clock <= ~clock;
39     end //initial
40
41     integer i;
42     initial begin
43
44         // Iterates through 2^32 clock cycles to test whether
45         // each divided clock cycle is twice as long as the previous.
46         // Due to processing limitations, does not test the maximum
47         // clock cycles.
48         for(i=0; i<2**32;i++) begin
49             @(posedge clock);
50         end // for loop
51
52         $stop;
53     end // initial
54 endmodule
55
56
```

```

1 // Morris Huang, Hudson Wong
2 // 06/07/025
3 // EE 271
4 // Lab 6 comparator
5
6 // Comparator takes in two inputs inputA and inputB and returns whether A is greater than B.
7 // If it is less than or equal to B, returns 0, otherwise returns 1.
8 module comparator(inputA, inputB, A_greater_B, clk, reset);
9     input logic [9:0] inputA, inputB;
10    input logic clk, reset;
11    output logic A_greater_B;
12
13    // Sequential logic to set A_greater_B
14    always_ff @(posedge clk) begin
15        if (reset)
16            A_greater_B <= 1'b0;
17        else
18            A_greater_B <= (inputA > inputB);
19    end
20 endmodule
21
22 // This module tests the comparator by instantiating several test cases to make
23 // sure that it correctly outputs when A is greater than B.
24 module comparator_testbench();
25
26    logic [9:0] inputA, inputB;
27    logic CLOCK_50, reset, A_greater_B;
28
29    // Calls comparator
30    comparator dut(inputA, inputB, A_greater_B, CLOCK_50, reset);
31
32    parameter clock_period = 100;
33
34    // toggle CLOCK_50 every half cycle
35    initial begin
36        CLOCK_50 <= 0;
37        forever #(clock_period / 2) CLOCK_50 <= ~CLOCK_50;
38    end
39
40    initial begin
41        // Apply reset for one clock cycle
42        reset <= 1; @(posedge CLOCK_50);
43        reset <= 0; @(posedge CLOCK_50);
44
45        inputA <= 1000000000; inputB <= 0100000000; @(posedge CLOCK_50); // Test A > B
46        @(posedge CLOCK_50);
47        inputA <= 0100000000; inputB <= 0100000000; @(posedge CLOCK_50); // Test A = B
48        @(posedge CLOCK_50);
49        inputA <= 0100000000; inputB <= 1000000000; @(posedge CLOCK_50); // Test A < B
50        inputA <= 0000000000; inputB <= 0000000000; @(posedge CLOCK_50); // Test edge cases
51        inputA <= 1111111111; inputB <= 1111111111; @(posedge CLOCK_50);
52        inputA <= 0000000000; inputB <= 1111111111; @(posedge CLOCK_50);
53        inputA <= 1111111111; inputB <= 0000000000; @(posedge CLOCK_50); // Test A < B
54
55        inputA <= 0100001111; inputB <= 1000001100; @(posedge CLOCK_50); // Test random values
56        inputA <= 1100010100; inputB <= 100101000; @(posedge CLOCK_50);
57        inputA <= 0100011110; inputB <= 1001111100; @(posedge CLOCK_50);
58        inputA <= 0100101111; inputB <= 1010101100; @(posedge CLOCK_50);
59        inputA <= 0101010110; inputB <= 0101010110; @(posedge CLOCK_50);
60        inputA <= 0100101100; inputB <= 0010100000; @(posedge CLOCK_50);
61        inputA <= 0100101100; inputB <= 0100001111; @(posedge CLOCK_50);
62        inputA <= 1100011000; inputB <= 1000100110; @(posedge CLOCK_50);
63        inputA <= 1111000000; inputB <= 1000000010; @(posedge CLOCK_50);
64        inputA <= 0100000000; inputB <= 1010110010; @(posedge CLOCK_50);
65        inputA <= 0101011100; inputB <= 0101100101; @(posedge CLOCK_50);
66        inputA <= 1100111000; inputB <= 1111111100; @(posedge CLOCK_50);
67        inputA <= 0100110000; inputB <= 1000110000; @(posedge CLOCK_50);
68
69        $stop;
70    end
71 endmodule
72
73
74

```

```
1 // Morris Huang, Hudson Wong
2 // 06/07/025
3 // EE 271
4 // Lab 6 three_bit_counter
5
6 // Three_bit_counter increments a counter by 1 every time an input is true.
7 // It starts at 0 and goes to 7 using binary numbers and a finite state machine.
8 // It takes in parameters clk, reset, in, and out.
9 // clk represents the clock cycles,
10 // reset sets the counter back to zero,
11 // in increments the number by one,
12 // and out represents the current count
13 module three_bit_counter (clk, reset, in, out);
14     input logic clk, reset, in;
15     output logic [2:0] out;
16
17     // Sets the eight states 0-7
18     enum {s0, s1, s2, s3, s4, s5, s6, s7} ps, ns; // Present state, next state
19
20     // Sets the finite state machine for the counter. If the input is true,
21     // goes to the next state. Otherwise, stays on the same state.
22     always_comb begin
23         case(ps)
24             s0: if (in) ns = s1;
25                 else ns = s0;
26             s1: if (in) ns = s2;
27                 else ns = s1;
28             s2: if (in) ns = s3;
29                 else ns = s2;
30             s3: if (in) ns = s4;
31                 else ns = s3;
32             s4: if (in) ns = s5;
33                 else ns = s4;
34             s5: if (in) ns = s6;
35                 else ns = s5;
36             s6: if (in) ns = s7;
37                 else ns = s6;
38             s7: ns = s7;
39         endcase
40     end
41
42     // Sets the output to whatever state it corresponds to from 0-7
43     always_comb begin
44         case (ps)
45             s0: out = 3'b000;
46             s1: out = 3'b001;
47             s2: out = 3'b010;
48             s3: out = 3'b011;
49             s4: out = 3'b100;
50             s5: out = 3'b101;
51             s6: out = 3'b110;
52             s7: out = 3'b111;
53         endcase
54     end
55
56     // Resets the present state to the first state, going back to 0
57     always_ff @(posedge clk) begin
58         if (reset)
59             ps <= s0;
60         else
61             ps <= ns;
62     end
63 endmodule
64
65 // This module tests the three bit counter, ensuring that it only increments
66 // when in is true, and that out represents the correct output
67 module three_bit_counter_testbench ();
68
69     // Instantiates the logic
70     logic CLOCK_50; // 50MHz clock
71     logic reset, in;
72     logic [2:0] out;
73
74     // Calls the module
75     three_bit_counter dut(CLOCK_50, reset, in, out);
76
```

```

77     parameter clock_period = 100;
78
79     // Runs every 100/2 picoseconds for each clock cycle
80     initial begin
81         CLOCK_50 <= 0;
82         forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
83     end //initial
84
85     initial begin
86         reset <= 1;           @(posedge CLOCK_50); // assert reset
87         reset <= 0;           @(posedge CLOCK_50); // release reset
88         in<=1;                 @(posedge CLOCK_50); // Ensures that the out increments by 1
89                               @(posedge CLOCK_50); // Also ensures that it maxes out at 7
90
91     without causing errors    @(posedge CLOCK_50);
92                               @(posedge CLOCK_50);
93                               @(posedge CLOCK_50);
94                               @(posedge CLOCK_50);
95                               @(posedge CLOCK_50);
96                               @(posedge CLOCK_50);
97                               @(posedge CLOCK_50);
98                               @(posedge CLOCK_50);
99                               @(posedge CLOCK_50);
100        in<=0;                 @(posedge CLOCK_50);
101        reset<=1;              @(posedge CLOCK_50);
102        reset<=0;              @(posedge CLOCK_50);
103                               @(posedge CLOCK_50);
104        in<=1;                  @(posedge CLOCK_50);
105        in<=0;                  @(posedge CLOCK_50); // Ensures that there are no false
106
107     increments                @(posedge CLOCK_50);
108                               @(posedge CLOCK_50);
109                               @(posedge CLOCK_50);
110                               @(posedge CLOCK_50);
111                               @(posedge CLOCK_50);
112                               @(posedge CLOCK_50);
113                               @(posedge CLOCK_50);
114                               @(posedge CLOCK_50);
115                               @(posedge CLOCK_50);
116                               @(posedge CLOCK_50);
117                               @(posedge CLOCK_50);
118
119        $stop;
120    end //initial
121 endmodule

```

```

1 // Morris Huang, Hudson Wong
2 // 06/07/2025
3 // EE 271
4 // Lab 6 user_input
5
6 // Takes a button press that, as long as it is pressed,
7 // only registers once even across multiple clock cycles.
8 // Has parameters clk which sets the clock cycles,
9 // reset which sets the output to a default state,
10 // button which is the key input, and out which
11 // registers once for as long as the button is pressed.
12 module user_input(clk, reset, button, out);
13     input logic clk, reset, button;
14     output logic out;
15     logic sync_button, next_button;
16
17     // A pair of flip flops for metastability. Delays the input by two cycles
18     always_ff @(posedge clk) begin
19         if (reset) begin
20             sync_button<=1'b0;
21             next_button<=1'b0;
22         end else begin
23             sync_button<=button;
24             next_button<=sync_button;
25         end
26     end
27
28     enum {S0, S1} ps, ns;
29
30     // If the button is held down, only registers once
31     always_comb begin
32         case (ps)
33             S0: if (next_button) ns = S1;
34                 else ns = S0;
35             S1: if (next_button) ns = S1;
36                 else ns = S0;
37         endcase
38     end
39
40     assign out = (ps==S0) & next_button;
41
42     // Resets the output to 0
43     always_ff @(posedge clk) begin
44         if (reset)
45             ps <= S0;
46         else
47             ps <= ns;
48     end
49 endmodule
50
51 // Tests the user input module using clock cycles and inputs
52 module user_input_testbench();
53     logic CLOCK_50; // 50MHz clock
54     logic reset, button, out;
55
56     user_input dut(CLOCK_50, reset, button, out);
57     parameter clock_period = 100;
58
59     initial begin
60         CLOCK_50 <= 0;
61         forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
62     end //initial
63
64     initial begin
65         reset <= 1; button<=0; @(posedge CLOCK_50);
66         reset <= 0;   @(posedge CLOCK_50);
67         button<=1;    @(posedge CLOCK_50);
68         button<=0;    @(posedge CLOCK_50);
69         button<=1;    @(posedge CLOCK_50);
70         button<=0;    @(posedge CLOCK_50);
71         button<=1;    @(posedge CLOCK_50);
72         button<=0;    @(posedge CLOCK_50);
73         button<=1;    @(posedge CLOCK_50); // Tests whether out only registers once
74         button<=0;    @(posedge CLOCK_50);
75         button<=1;    @(posedge CLOCK_50);
76         button<=0;    @(posedge CLOCK_50);

```



```
77         @(posedge CLOCK_50);
78         @(posedge CLOCK_50);
79         @(posedge CLOCK_50);
80         button<=1; @(posedge CLOCK_50); // Tests whether multiple buttons counts as one press
81         button<=1; @(posedge CLOCK_50);
82         button<=1; @(posedge CLOCK_50);
83         button<=1; @(posedge CLOCK_50);
84         button<=1; @(posedge CLOCK_50);
85         button<=0; @(posedge CLOCK_50);
86         button<=1; @(posedge CLOCK_50);
87         button<=1; @(posedge CLOCK_50);
88         button<=1; @(posedge CLOCK_50);
89         button<=1; @(posedge CLOCK_50);
90         button<=1; @(posedge CLOCK_50);
91         $stop;
92         end //initial
93     endmodule
94
95
```