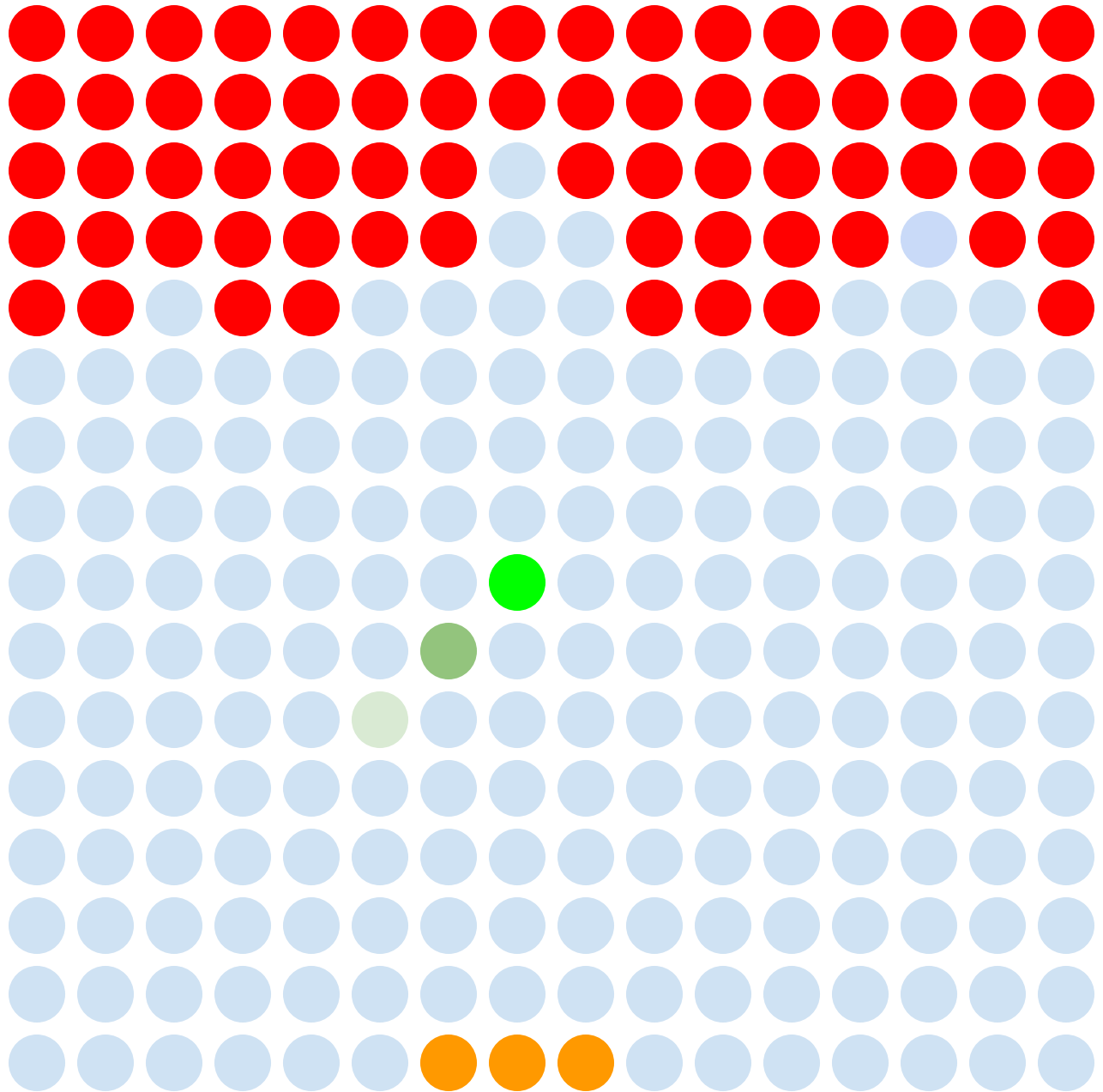


J A I L B R E A K



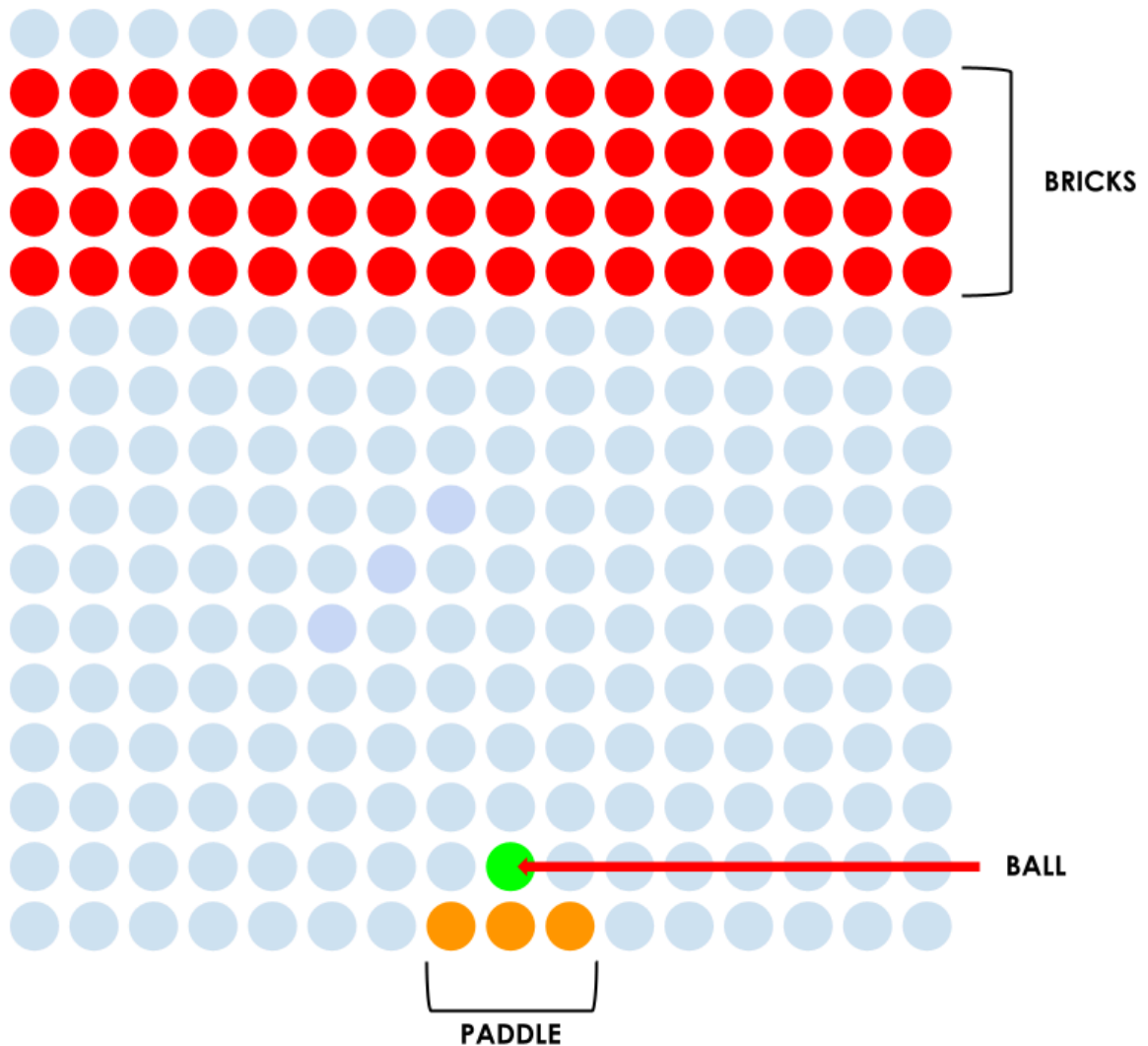
E L E C T R O N I C V I D E O G A M E

EE271 | INSTRUCTION
MANUAL

HOW TO PLAY:

After turning on the game, press the RESET key (KEY 3) to start a new game.

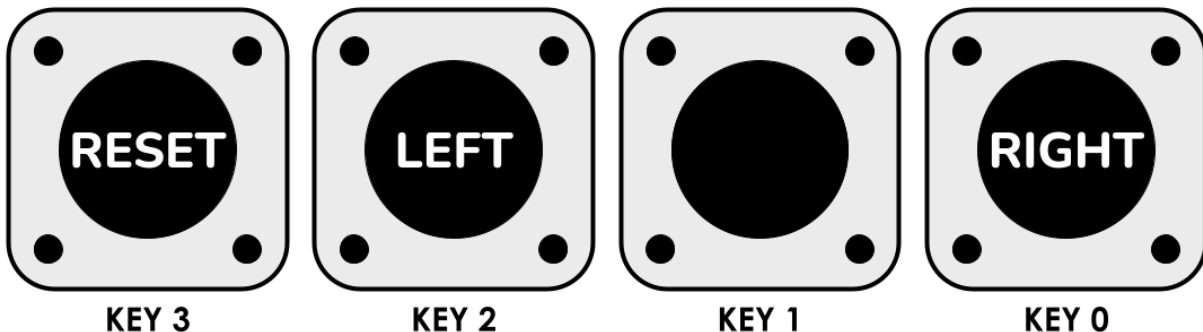
GAME BOARD:



The game board will display the bricks on the top of the screen in red, the paddle on the bottom of the screen in orange, and the ball in green.

The paddle will start in the bottom center of the screen, and can be moved left and right by pressing the LEFT (KEY 2) and RIGHT (KEY 0) keys.

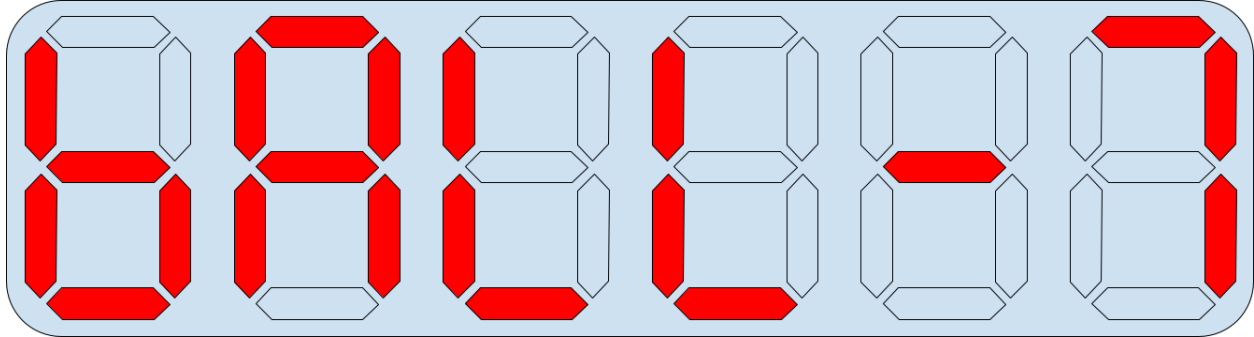
KEYS:



**The ball will start on top of the paddle, and after a short delay, will begin to move up and right.
If it hits a wall or a brick the ball will bounce off of it, breaking any bricks it hits.**

If the ball hits the bottom wall, it will be lost. The goal of the game is to break all of the bricks while minimizing the number of balls lost.

REMAINING BALL COUNTER:



At the start of the game, there are 7 balls remaining. Every time a ball is lost, the number of balls remaining decreases. If all 7 balls are lost before all bricks are broken, you have lost.

To start a new game after losing or breaking all of the bricks, press the RESET key (KEY 3) again.

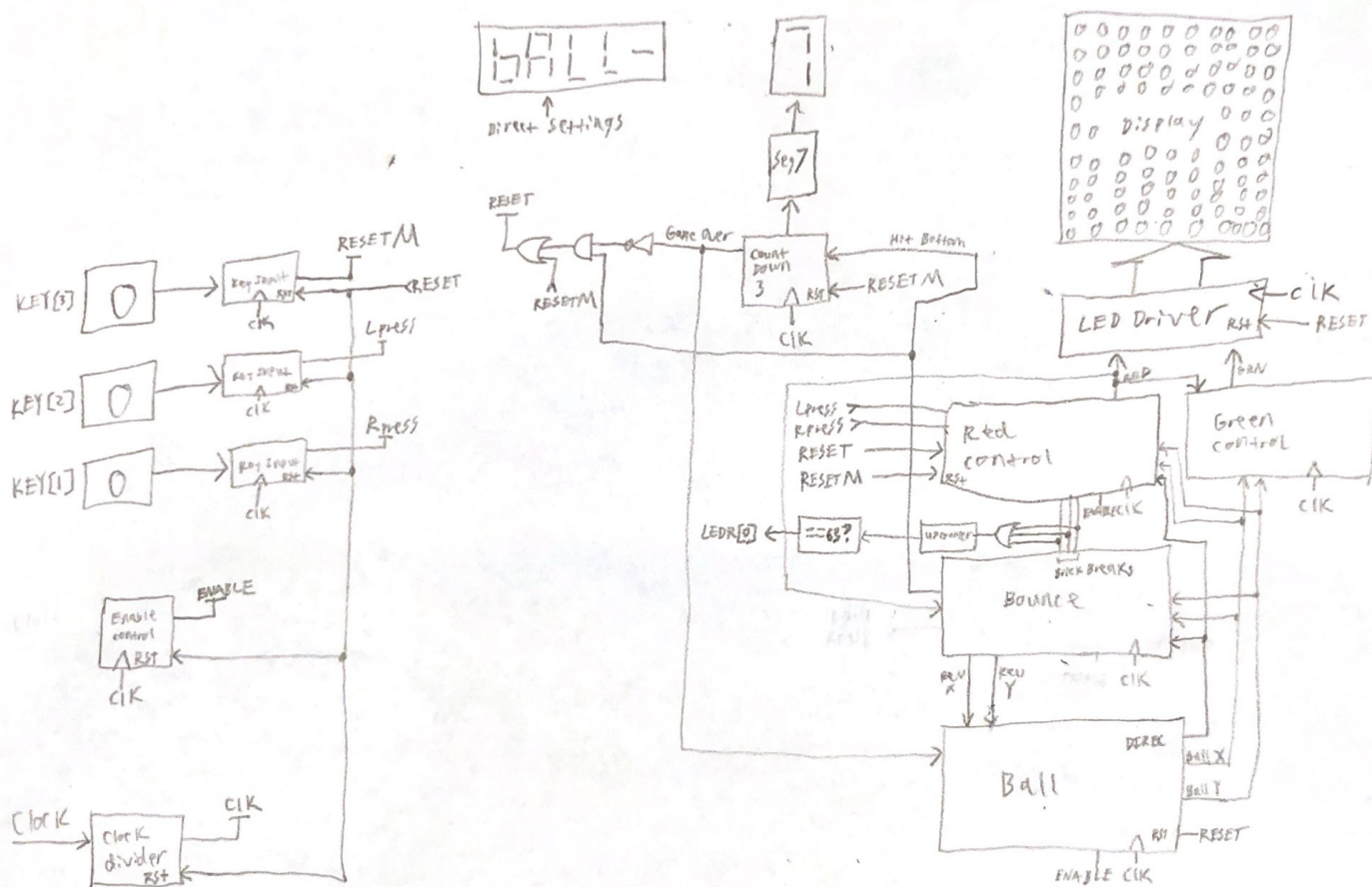
Useability - The system is very simple to use, with an intuitive graphical user interface and only 3 buttons for input. Color coding on the display makes it easy to differentiate between different objects on the screen, and the ball counter is clearly labeled to remove any confusion as to what the number means.

Suitability for the goals - The game is fun to play, because it is easy to understand and start playing, while also being fast enough to be challenging. This means that one can gradually build their skill over several playthroughs of the game, meaning the game stays entertaining for a longer period of time.

Cost - The system was designed to try to minimize the amount of logic used, eliminating extraneous logic as much as possible to create a streamlined design. The design utilizes 749 Combinational ALUTs and 406 Dedicated Logic Registers.

Environmental Factors - Building the design on an FPGA as opposed to discrete logic ICs would reduce environmental costs related to shipping and manufacturing of the game.

EE 271 Lab 8 Block Diagram / Nikolas Faulkner



	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins	Full Hierarchy Name	Entity Name	Library Name
1	DE1_SoC	749 (1)	406 (0)	0	0	103	0	DE1_SoC	DE1_SoC	work
1	Ball:BALL	15 (15)	10 (10)	0	0	0	0	DE1_SoC Ball:BALL	Ball	work
2	Bounce:BOUNCE	81 (81)	6 (6)	0	0	0	0	DE1_SoC Bounce:BOUNCE	Bounce	work
3	GrnControl:GREENLIGHTS	256 (256)	256 (256)	0	0	0	0	DE1_SoC GrnControl:GREENLIGHTS	GrnControl	work
4	LEDDriver:Driver	118 (118)	4 (4)	0	0	0	0	DE1_SoC LEDDriver:Driver	LEDDriver	work
5	RedControl:REDLIGHTS	215 (215)	84 (84)	0	0	0	0	DE1_SoC RedControl:REDLIGHTS	RedControl	work
6	clock_divider:cdiv	15 (15)	15 (15)	0	0	0	0	DE1_SoC clock_divider:cdiv	clock_divide	work
7	countDown3:lives	4 (4)	5 (5)	0	0	0	0	DE1_SoC countDown3:lives	countDown3	work
8	enableControl:en	30 (30)	20 (20)	0	0	0	0	DE1_SoC enableControl:en	enableContr	work
9	keyInput:Lk	2 (2)	2 (2)	0	0	0	0	DE1_SoC keyInput:Lk	keyInput	work
10	keyInput:Rk	2 (2)	2 (2)	0	0	0	0	DE1_SoC keyInput:Rk	keyInput	work
11	keyInput:rstk	2 (2)	2 (2)	0	0	0	0	DE1_SoC keyInput:rstk	keyInput	work
12	seg7:ballCountDisp	8 (8)	0 (0)	0	0	0	0	DE1_SoC seg7:ballCountDisp	seg7	work

```

1  module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, GPIO_1);
2      input logic CLOCK_50; // 50MHz clock.
3      output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
4      output logic [9:0] LEDR;
5      output logic [35:0] GPIO_1;
6      input logic [3:0] KEY; // True when not pressed, False when pressed
7      input logic [9:0] SW;
8
9
10     // Internal logic
11     logic L, Lpress, Lkey, R, Rpress, Rkey, hitBottom, win;
12     logic reset; //Soft reset, from main reset or life lost
13     logic resetm; //Main reset, from KEY[3]
14     logic gameOver;
15     logic [31:0] div_clk;
16     logic [2:0] ballCount;
17     assign reset = (resetm | ((hitBottom) & ~gameOver));
18
19     //Clock and Enable
20     // Generate clk off of CLOCK_50, whichClock picks rate.
21     parameter whichClock = 15; // 0.75 Hz clock
22     clock_divider cdiv (.clock(CLOCK_50), .reset, .divided_clocks(div_clk));
23     logic clkSelect, enable;
24     // Clock selection; allows for easy switching between simulation and board clocks
25     // Set up system base clock to 1526 Hz (50 MHz / 2**(14+1))
26     // If you notice flickering, set SYSTEM_CLOCK faster, however this may reduce the
    brightness of the LED board.
27
28     // Uncomment ONE of the following two lines depending on intention:
29
30     //assign clkSelect = CLOCK_50; // for simulation
31     assign clkSelect = div_clk[14]; // 1526 Hz clock signal for display
32     //assign clkSelect = div_clk[whichClock]; // for board
33     enableControl en (.clock(clkSelect), .reset, .enable); //Sets enable signal
34
35     //Main reset control
36     keyInput rstk (.clk(clkSelect), .reset, .keyIn(~KEY[3]), .keyOut(resetm));
37
38     //Ball Count Display
39     assign HEX5 = 7'b0000011; //b
40     assign HEX4 = 7'b0001000; //A
41     assign HEX3 = 7'b1000111; //L
42     assign HEX2 = 7'b1000111; //L
43     assign HEX1 = 7'b0111111; //-
44     countdown3 lives (.out(ballCount), .incr(hitBottom), .reset(resetm), .clk(clkSelect), .
    gameOver); //Decrement life count from 7 upon death
45     seg7 ballCountDisp (.bcd(ballCount), .leds(HEX0)); //Display ball count to HEX 0
46
47
48     // Set up LED board driver
49     logic [15:0][15:0] RedPixels; // 16 x 16 array representing red LEDs
50     logic [15:0][15:0] GrnPixels; // 16 x 16 array representing green LEDs
51
52     /* Standard LED Driver instantiation - set once and 'forget it'.
53     See LEDDriver.sv for more info. Do not modify unless you know what you are doing! */
54     LEDDriver Driver (.CLK(clkSelect), .RST(reset), .EnableCount(1'b1), .RedPixels, .
    GrnPixels, .GPIO_1);
55
56     //LED_test test (.RST(~KEY[0]), .RedPixels);
57
58     //Barmover (.clk(clkSelect), .reset, .w, .GrnPixels);
59
60
61     // Set up FSM inputs and outputs.
62     logic keyinL, keyinR;
63     assign keyinL = ~KEY[2]; //Left key
64     assign keyinR = ~KEY[0]; //Right key
65
66     keyInput Lk (.clk(clkSelect), .reset, .keyIn(keyinL), .keyOut(Lpress));
67     userInput Lu (.clk(clkSelect), .reset, .usIn(Lkey), .usOut(L), .enable);
68
69     keyInput Rk (.clk(clkSelect), .reset, .keyIn(keyinR), .keyOut(Rpress));
70     userInput Ru (.clk(clkSelect), .reset, .usIn(Rkey), .usOut(R), .enable);
71
72     logic revX, revY, killBrickCorner, killBrickY, killBrickX, enableEarly;

```

```

73     logic [3:0] ballX, ballY;
74     logic [2:0] hitcount;
75     logic [1:0] DIREC; //[Y direc, X direc], [00] = down left, [01] = down right, [10] = up
left, [11] = up right
76
77     // Modules
78     Ball BALL (.clk(clkSelect), .reset, .enable, .revX, .revY, .ballX, .ballY, .DIREC, .
gameover);
79     Bounce BOUNCE (.revX, .revY, .ballX, .ballY, .RedPixels, .killBrickCorner, .killBrickY, .
killBrickX, .hitBottom, .DIREC, .clock(clkSelect));
80     RedControl REDLIGHTS (.clk(clkSelect), .resetm, .reset, .ballX, .ballY, .DIREC, .
RedPixels, .killBrickCorner, .killBrickY, .killBrickX, .L(Lpress), .R(Rpress), .enable);
81     GrnControl GREENLIGHTS (.clk(clkSelect), .reset(resetm), .ballX, .ballY, .GrnPixels, .
RedPixels);
82     counter3b hitcounter (.out(hitcount), .incr(killBrickCorner | killBrickY | killBrickX), .
reset(resetm), .clk(clkSelect), .enable);
83
84     assign LEDR[0] = (hitcount == 63);
85     assign LEDR[1] = clkSelect;
86     assign LEDR[2] = enable;
87     assign LEDR[3] = killBrickX;
88     assign LEDR[4] = killBrickCorner;
89     assign LEDR[5] = DIREC[1];
90     assign LEDR[6] = DIREC[0];
91     assign LEDR[7] = reset;
92     assign LEDR[8] = gameover;
93     assign LEDR[9] = Lpress;
94
95     endmodule
96
97     module DE1_SoC_testbench();
98         logic CLOCK_50;
99         logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
100        logic [9:0] LEDR;
101        logic [3:0] KEY;
102        logic [9:0] SW;
103
104        DE1_SoC dut (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);
105
106        // Set up a simulated clock.
107        parameter CLOCK_PERIOD=100;
108        initial begin
109            CLOCK_50 <= 0;
110            forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
111        end
112
113        // Test the design.
114        initial begin
115            repeat(1) @(posedge CLOCK_50);
116            KEY[3] <= 0; repeat(1) @(posedge CLOCK_50); // Reset
117            KEY[3] <= 1; repeat(1) @(posedge CLOCK_50);
118            KEY[0] <= 0; repeat(2) @(posedge CLOCK_50); // Press right
119            KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
120            KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
121            KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
122            KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
123            KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
124            KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right, then wait for a while
125            KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
126            KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
127            KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
128            KEY[2] <= 0; repeat(1) @(posedge CLOCK_50); // Press left
129            KEY[2] <= 1; repeat(2) @(posedge CLOCK_50);
130            KEY[2] <= 0; repeat(1) @(posedge CLOCK_50); // Press left
131            KEY[2] <= 1; repeat(2) @(posedge CLOCK_50);
132            KEY[2] <= 0; repeat(1) @(posedge CLOCK_50); // Press left
133            KEY[2] <= 1; repeat(2) @(posedge CLOCK_50);
134            KEY[2] <= 0; repeat(1) @(posedge CLOCK_50); // Press left
135            KEY[2] <= 1; repeat(2) @(posedge CLOCK_50);
136            KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
137            KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
138            KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
139            KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
140            KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
141            KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);

```



```
142     KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
143     KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
144     KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
145     KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
146     KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
147     KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
148     KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
149     KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
150     KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
151     KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
152     $stop; // End the simulation.
153     end
154 endmodule
155
```

```

1  module Ball (clk, reset, enable, revX, revY, ballX, ballY, DIREC, gameover);
2      input logic clk, reset, enable, revX, revY, gameover;
3      output logic [1:0] DIREC; // [Y direc, X direc], [00] = down left, [01] = down right, [10]
= up left, [11] = up right
4      output logic [3:0] ballX, ballY;
5      // State variables
6      logic [3:0] px, nx, py, ny;
7      logic [1:0] nDIREC;
8
9      // Bounce around the room, if revX or revY occurs, reverse that direction
10     // Output the direction, ball position, and display data for the ball.
11
12     // Next State logic
13     always_comb begin
14
15         if (revY) begin // If y bounce, don't change y, swap y direction
16             ny = py;
17             nDIREC[1] = ~DIREC[1];
18         end
19         else if (DIREC[1]) begin // If going up and not y bounce, go up
20             ny = py-1;
21             nDIREC[1] = DIREC[1];
22         end
23         else begin // If going down and not y bounce, go down
24             ny = py+1;
25             nDIREC[1] = DIREC[1];
26         end
27
28         if (revX) begin // If x bounce, don't change x, swap x direction
29             nx = px;
30             nDIREC[0] = ~DIREC[0];
31         end
32         else if (DIREC[0]) begin // If going up and not x bounce, go up
33             nx = px-1;
34             nDIREC[0] = DIREC[0];
35         end
36         else begin // If going down and not x bounce, go down
37             nx = px+1;
38             nDIREC[0] = DIREC[0];
39         end
40     end
41
42     assign ballX = px;
43     assign ballY = py;
44
45     // DFFs
46     always_ff @(posedge clk) begin
47         if (reset) begin // Ball start position
48             px <= 4'b0111;
49             py <= 4'b1110;
50             DIREC <= 2'b11; // Start going up right
51         end
52         else if (enable && (~gameover)) begin
53             px <= nx;
54             py <= ny;
55             DIREC = nDIREC;
56         end
57     end
58 end
59
60 endmodule
61
62
63
64 module Ball_testbench();
65     logic clock, reset, enable, revX, revY, gameover;
66     logic [1:0] DIREC; // [Y direc, X direc], [00] = down left, [01] = down right, [10] = up
left, [11] = up right
67     logic [3:0] ballX, ballY;
68
69     Ball dut (.clk(clock), .reset, .enable, .revX, .revY, .ballX, .ballY, .DIREC, .gameover);
70
71     // Set up a simulated clock.
72     parameter CLOCK_PERIOD=100;
73     initial begin

```

```
74     clock <= 0;
75     forever #(CLOCK_PERIOD/2) clock <= ~clock; // Forever toggle the clock
76     end
77
78     // Set up the inputs to the design. Each line is a clock cycle.
79     initial begin
80         @(posedge clock);
81         gameover <= 0; revX <= 0; revY <= 0;
82         enable <= 1; @(posedge clock);
83         reset <= 1; @(posedge clock);
84         reset <= 0; @(posedge clock);
85         @(posedge clock); @(posedge clock); @(posedge clock); @(posedge clock);
86         revX <= 1; @(posedge clock);
87         revX <= 0; @(posedge clock);
88         @(posedge clock); @(posedge clock); @(posedge clock);
89         revY <= 1; @(posedge clock);
90         revY <= 0; @(posedge clock);
91         @(posedge clock); @(posedge clock); @(posedge clock);
92         revX <= 1; @(posedge clock);
93         revX <= 0; @(posedge clock);
94         @(posedge clock); @(posedge clock); @(posedge clock);
95         revY <= 1; @(posedge clock);
96         revY <= 0; @(posedge clock);
97         @(posedge clock); @(posedge clock); @(posedge clock);
98         $stop; // End the simulation.
99     end
100 endmodule
```

```
1  module userInput (clk, reset, usIn, usOut, enable);
2      input logic clk, reset, usIn, enable;
3      output logic usOut;
4      // State variables
5      enum { low, high } ns, ps;
6
7      // Next State logic
8      always_comb begin
9          if (usIn) ns = high;
10         else ns = low;
11     end
12
13     // Output logic - could also be another always_comb block.
14     assign usOut = ((ns == high) & (ps == low));
15
16     // DFFs
17     always_ff @(posedge clk) begin
18         if (reset) begin
19             ps <= low;
20         end else if (enable) begin
21             ps <= ns;
22         end
23     end
24
25     //always_ff @(posedge clk) begin
26     //if (reset)
27     //out <= low;
28     //else
29     //out <= mid;
30     //end
31
32     endmodule
33
34     module userInput_testbench();
35         logic clk, reset, usIn, usOut;
36         logic out;
37
38         userInput dut (clk, reset, usIn, usOut);
39
40         // Set up a simulated clock.
41         parameter CLOCK_PERIOD=100;
42         initial begin
43             clk <= 0;
44             forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
45         end
46
47         // Set up the inputs to the design. Each line is a clock cycle.
48         initial begin
49             @(posedge clk);
50             reset <= 1; @(posedge clk); // Always reset FSMs at start
51             reset <= 0; usIn <= 0; @(posedge clk);
52             @(posedge clk);
53             @(posedge clk);
54             @(posedge clk);
55             usIn <= 1; @(posedge clk);
56             @(posedge clk);
57             @(posedge clk);
58             usIn <= 0; @(posedge clk);
59             @(posedge clk);
60             @(posedge clk);
61             usIn <= 1; @(posedge clk);
62             @(posedge clk);
63             @(posedge clk);
64             @(posedge clk);
65             @(posedge clk);
66             @(posedge clk);
67             usIn <= 0; @(posedge clk);
68             @(posedge clk);
69             @(posedge clk);
70             @(posedge clk);
71             usIn <= 1; @(posedge clk);
72             usIn <= 0; @(posedge clk);
73             @(posedge clk);
74             @(posedge clk);
75             @(posedge clk);
```

```
76    @(posedge clk);  
77    $stop; // End the simulation.  
78    end  
79    endmodule
```

```

1  module enableControl (clock, reset, enable);
2  input logic reset, clock;
3  output logic enable;
4  logic [9:0] counter1, counter2;
5  //logic pause;
6
7  always_ff @(posedge clock) begin
8      if(reset) begin
9          //pause <= 1;
10         counter1 <= 10'b0000000000;
11         counter2 <= 10'b0000000000;
12         //divided_clocks <= 32'b00000000000000000000000000000000;
13         end
14
15         else if (counter2 == (10'b1000010000)) begin //Regular time counter
16             if (counter1 > 150) begin //Sets ball speed
17                 counter1 <= 10'b0000000000;
18             end
19             else begin
20                 counter1 <= counter1+1;
21             end
22             end
23
24             else begin //Startup delay after soft reset
25                 counter2 <= counter2+1;
26             end
27         end
28     end
29
30     assign enable = (counter1 == 100);
31 endmodule
32
33
34
35 module enableControl_testbench();
36 logic clock, reset;
37 logic enable;
38 logic [9:0] counter1, counter2;
39
40 enableControl dut (clock, reset, enable, counter1, counter2);
41
42 // Set up a simulated clock.
43 parameter CLOCK_PERIOD=100;
44 initial begin
45     clock <= 0;
46     forever #(CLOCK_PERIOD/2) clock <= ~clock; // Forever toggle the clock
47     end
48
49 // Set up the inputs to the design. Each line is a clock cycle.
50 initial begin
51     @(posedge clock);
52     reset <= 1; @(posedge clock); // Always reset FSMs at start
53     reset <= 0; @(posedge clock);
54     @(posedge clock);
55     @(posedge clock);
56     @(posedge clock);
57     @(posedge clock);
58     @(posedge clock);
59     @(posedge clock);
60     @(posedge clock);
61     @(posedge clock);
62     @(posedge clock);
63     @(posedge clock);
64     @(posedge clock);
65     @(posedge clock);
66     @(posedge clock);
67     @(posedge clock);
68     @(posedge clock);
69     @(posedge clock);
70     @(posedge clock);
71     @(posedge clock);
72     @(posedge clock);
73     @(posedge clock);
74     @(posedge clock);
75     @(posedge clock);

```

```
76    @(posedge clock);
77    @(posedge clock);
78    @(posedge clock);
79    @(posedge clock);
80    @(posedge clock);
81    @(posedge clock);
82    @(posedge clock);
83    @(posedge clock);
84    @(posedge clock);
85    @(posedge clock);
86    @(posedge clock);
87    @(posedge clock);
88    @(posedge clock);
89    @(posedge clock);
90    @(posedge clock);
91    @(posedge clock);
92    @(posedge clock);
93    $stop; // End the simulation.
94    end
95    endmodule
```

```

1  module countDown3 #(parameter WIDTH=3) (out, incr, reset, clk, gameOver);
2  input logic incr, reset, clk;
3  output logic [WIDTH-1:0] out;
4  output logic gameOver;
5  logic lastIncr;
6
7  always_ff @(posedge clk) begin
8      if(reset) begin
9          out<=7;
10         lastIncr<=0;
11     end else if(incr & ~lastIncr) begin
12         out<=out-1;
13         lastIncr<=1;
14     end else if(~incr & lastIncr) begin
15         out<=out;
16         lastIncr<=0;
17     end else
18         out<=out; //This is not strictly necessary, it wastes characters but makes it clear
to the reader. Can be a comment instead.
19
20     if(reset)
21         gameOver <= 0;
22     else if((out == 0) & incr & (gameOver == 0))
23         gameOver <= 1;
24     else
25         gameOver <= gameOver;
26 end
27 endmodule
28
29 module countDown3_testbench();
30 logic incr, reset, clk, gameOver;
31 logic out;
32
33 counter3 dut (out, incr, reset, clk, gameOver);
34
35 // Set up a simulated clock.
36 parameter CLOCK_PERIOD=100;
37 initial begin
38     clk <= 0;
39     forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
40 end
41
42 // Set up the inputs to the design. Each line is a clock cycle.
43 initial begin
44     @(posedge clk);
45     reset <= 1; @(posedge clk); // Always reset FSMs at start
46     reset <= 0; incr <= 0; @(posedge clk);
47     @(posedge clk);
48     @(posedge clk);
49     @(posedge clk);
50     incr <= 1; @(posedge clk);
51     incr <= 0; @(posedge clk);
52     @(posedge clk);
53     @(posedge clk);
54     incr <= 1; @(posedge clk);
55     incr <= 0; @(posedge clk);
56     @(posedge clk);
57     @(posedge clk);
58     incr <= 1; @(posedge clk);
59     incr <= 0; @(posedge clk);
60     incr <= 1; @(posedge clk);
61     incr <= 0; @(posedge clk);
62     incr <= 1; @(posedge clk);
63     incr <= 0; @(posedge clk);
64     incr <= 1; @(posedge clk);
65     incr <= 0; @(posedge clk);
66     incr <= 1; @(posedge clk);
67     incr <= 0; @(posedge clk);
68     @(posedge clk);
69     @(posedge clk);
70     incr <= 1; @(posedge clk);
71     incr <= 0; @(posedge clk);
72     @(posedge clk);
73     $stop; // End the simulation.
74 end

```



```
75     endmodule
```

```

1 module GrnControl (clk, reset, ballX, ballY, GrnPixels, RedPixels);
2   input logic clk, reset;
3   input logic [3:0] ballX, ballY;
4   input logic [15:0][15:0] RedPixels;
5   output logic [15:0][15:0] GrnPixels;
6   // State variables
7   //logic [15:0][15:0] grnMem, ngrnMem; //FF Memory for red pixels (bricks only)
8
9
10  // DFFs
11  always_ff @(posedge clk) begin
12    //Set all green to zero, except bottom row, which has paddle
13    //Set lowest row to the same as the red pixels
14    GrnPixels[00] <= 16'b0000000000000000;
15    GrnPixels[01] <= 16'b0000000000000000;
16    GrnPixels[02] <= 16'b0000000000000000;
17    GrnPixels[03] <= 16'b0000000000000000;
18    GrnPixels[04] <= 16'b0000000000000000;
19    GrnPixels[05] <= 16'b0000000000000000;
20    GrnPixels[06] <= 16'b0000000000000000;
21    GrnPixels[07] <= 16'b0000000000000000;
22    GrnPixels[08] <= 16'b0000000000000000;
23    GrnPixels[09] <= 16'b0000000000000000;
24    GrnPixels[10] <= 16'b0000000000000000;
25    GrnPixels[11] <= 16'b0000000000000000;
26    GrnPixels[12] <= 16'b0000000000000000;
27    GrnPixels[13] <= 16'b0000000000000000;
28    GrnPixels[14] <= 16'b0000000000000000;
29    GrnPixels[15] <= RedPixels[15];
30
31    //Set position of ball to 1
32    GrnPixels[ballY][ballX] <= 1'b1;
33  end
34
35 endmodule
36
37
38
39
40 module GrnControl_testbench();
41   logic clock, reset;
42   logic [3:0] ballX, ballY;
43   logic [15:0][15:0] RedPixels;
44   logic [15:0][15:0] GrnPixels;
45
46   GrnControl dut (.clk(clock), .reset, .ballX, .ballY, .GrnPixels, .RedPixels);
47
48   // Set up a simulated clock.
49   parameter CLOCK_PERIOD=100;
50   initial begin
51     clock <= 0;
52     forever #(CLOCK_PERIOD/2) clock <= ~clock; // Forever toggle the clock
53   end
54
55   // Set up the inputs to the design. Each line is a clock cycle.
56   initial begin
57     @(posedge clock);
58     ballX <= 5; @(posedge clock);
59     ballY <= 5; @(posedge clock);
60     RedPixels[00] <= 16'b0000000000000000;
61     RedPixels[01] <= 16'b1111111111111111;
62     RedPixels[02] <= 16'b1111111111111111;
63     RedPixels[03] <= 16'b1111111111111111;
64     RedPixels[04] <= 16'b0000000000000000;
65     RedPixels[05] <= 16'b0000000000000000;
66     RedPixels[06] <= 16'b0000000000000000;
67     RedPixels[07] <= 16'b0000000000000000;
68     RedPixels[08] <= 16'b0000000000000000;
69     RedPixels[09] <= 16'b0000000000000000;
70     RedPixels[10] <= 16'b0000000000000000;
71     RedPixels[11] <= 16'b0000000000000000;
72     RedPixels[12] <= 16'b0000000000000000;
73     RedPixels[13] <= 16'b0000000000000000;
74     RedPixels[14] <= 16'b0000000000000000;
75     RedPixels[15] <= 16'b0000000111000000; @(posedge clock);

```

```
76    ballX <= 4; @(posedge clock);
77    ballY <= 4; @(posedge clock);
78    ballX <= 3; @(posedge clock);
79    ballY <= 3; @(posedge clock);
80    ballX <= 2; @(posedge clock);
81    ballY <= 2; @(posedge clock);
82    ballX <= 1; @(posedge clock);
83    ballY <= 1; @(posedge clock);
84    RedPixels[15] <= 16'b0001011011010100; @(posedge clock);
85    RedPixels[15] <= 16'b1001000101010010; @(posedge clock);
86    RedPixels[15] <= 16'b0100101101010101; @(posedge clock);
87    RedPixels[15] <= 16'b1111100001111111; @(posedge clock);
88    RedPixels[15] <= 16'b0000001110000000; @(posedge clock);
89    $stop; // End the simulation.
90    end
91 endmodule
```

```

1  module RedControl (clk, resetm, reset, ballX, ballY, DIREC, RedPixels, killBrickCorner,
killBrickX, killBrickY, L, R, enable);
2  input logic clk, resetm, reset, killBrickCorner, killBrickX, killBrickY, L, R, enable;
3  input logic [3:0] ballX, ballY;
4  input logic [1:0] DIREC;
5  output logic [15:0][15:0] RedPixels;
6  // State variables
7  logic [15:0][15:0] nredMem; //FF Memory for red pixels (bricks only)
8  logic [3:0] paddlePos, npaddlePos; //FF Paddle position memory
9
10
11 //Control paddle and brick positions.
12 //Delete bricks based on collisions.
13
14
15 // Next State logic
16 always_comb begin
17
18     nredMem[00] <= RedPixels[00];
19     nredMem[01] <= RedPixels[01];
20     nredMem[02] <= RedPixels[02];
21     nredMem[03] <= RedPixels[03];
22     nredMem[04] <= RedPixels[04];
23     nredMem[05] <= RedPixels[05];
24     nredMem[06] <= RedPixels[06];
25     nredMem[07] <= RedPixels[07];
26     nredMem[08] <= RedPixels[08];
27     nredMem[09] <= RedPixels[09];
28     nredMem[10] <= RedPixels[10];
29     nredMem[11] <= RedPixels[11];
30     nredMem[12] <= RedPixels[12];
31     nredMem[13] <= RedPixels[13];
32     nredMem[14] <= RedPixels[14];
33     nredMem[15] <= RedPixels[15];
34
35     if (killBrickX) begin
36         nredMem[ballY][ballX+1-(DIREC[0]+DIREC[0])] <= 0;
37     end
38
39     if (killBrickY) begin
40         nredMem[ballY+1-(DIREC[1]+DIREC[1])][ballX] <= 0;
41     end
42
43     if (killBrickCorner) begin
44         nredMem[ballY+1-(DIREC[1]+DIREC[1])][ballX+1-(DIREC[0]+DIREC[0])] <= 0;
45     end
46
47     if (~reset & L & (paddlePos<14)) begin
48         npaddlePos <= paddlePos+1;
49     end else if (~reset & R & (paddlePos>1)) begin
50         npaddlePos <= paddlePos-1;
51     end else begin
52         npaddlePos <= paddlePos;
53     end
54
55 end
56
57 // DFFs
58 always_ff @(posedge clk) begin
59     if (resetm) begin //Ball start position
60         RedPixels[00] <= 16'b0000000000000000;
61         RedPixels[01] <= 16'b1111111111111111;
62         RedPixels[02] <= 16'b1111111111111111;
63         RedPixels[03] <= 16'b1111111111111111;
64         RedPixels[04] <= 16'b1111111111111111;
65         RedPixels[05] <= 16'b0000000000000000;
66         RedPixels[06] <= 16'b0000000000000000;
67         RedPixels[07] <= 16'b0000000000000000;
68         RedPixels[08] <= 16'b0000000000000000;
69         RedPixels[09] <= 16'b0000000000000000;
70         RedPixels[10] <= 16'b0000000000000000;
71         RedPixels[11] <= 16'b0000000000000000;
72         RedPixels[12] <= 16'b0000000000000000;
73         RedPixels[13] <= 16'b0000000000000000;
74         RedPixels[14] <= 16'b0000000000000000;

```

```

75     RedPixels[15] <= 16'b0000000111000000;
76     paddlePos <= 4'b0111;
77 end
78 else if (enable) begin
79     RedPixels[00] <= nredMem[00];
80     RedPixels[01] <= nredMem[01];
81     RedPixels[02] <= nredMem[02];
82     RedPixels[03] <= nredMem[03];
83     RedPixels[04] <= nredMem[04];
84     RedPixels[05] <= nredMem[05];
85     RedPixels[06] <= nredMem[06];
86     RedPixels[07] <= nredMem[07];
87     RedPixels[08] <= nredMem[08];
88     RedPixels[09] <= nredMem[09];
89     RedPixels[10] <= nredMem[10];
90     RedPixels[11] <= nredMem[11];
91     RedPixels[12] <= nredMem[12];
92     RedPixels[13] <= nredMem[13];
93     RedPixels[14] <= nredMem[14];
94     if (reset) begin
95         paddlePos <= 4'b0111;
96     end
97     else begin
98         paddlePos <= npaddlePos;
99
100         if(npaddlePos == 1) begin
101             RedPixels[15] <= 16'b0000000000000111;
102         end else if(npaddlePos == 2) begin
103             RedPixels[15] <= 16'b0000000000001110;
104         end else if(npaddlePos == 3) begin
105             RedPixels[15] <= 16'b0000000000011100;
106         end else if(npaddlePos == 4) begin
107             RedPixels[15] <= 16'b0000000001110000;
108         end else if(npaddlePos == 5) begin
109             RedPixels[15] <= 16'b0000000011100000;
110         end else if(npaddlePos == 6) begin
111             RedPixels[15] <= 16'b0000000111000000;
112         end else if(npaddlePos == 7) begin
113             RedPixels[15] <= 16'b0000001110000000;
114         end else if(npaddlePos == 8) begin
115             RedPixels[15] <= 16'b0000011100000000;
116         end else if(npaddlePos == 9) begin
117             RedPixels[15] <= 16'b0000111000000000;
118         end else if(npaddlePos == 10) begin
119             RedPixels[15] <= 16'b0001110000000000;
120         end else if(npaddlePos == 11) begin
121             RedPixels[15] <= 16'b0011100000000000;
122         end else if(npaddlePos == 12) begin
123             RedPixels[15] <= 16'b0111000000000000;
124         end else if(npaddlePos == 13) begin
125             RedPixels[15] <= 16'b1110000000000000;
126         end else if(npaddlePos == 14) begin
127             RedPixels[15] <= 16'b1110000000000000;
128         end else begin
129             RedPixels[15] <= nredMem[00];
130         end
131     end
132 end
133 end
134
135 endmodule
136
137
138
139 module RedControl_testbench();
140     logic clock, resetm, reset, killBrickCorner, killBrickX, killBrickY, L, R, enable;
141     logic [3:0] ballX, ballY;
142     logic [1:0] DIREC;
143     logic [15:0][15:0] RedPixels;
144
145     RedControl dut (.clk(clock), .resetm, .reset, .ballX, .ballY, .DIREC, .RedPixels, .
killBrickCorner, .killBrickX, .killBrickY, .L, .R, .enable);
146
147     // Set up a simulated clock.
148     parameter CLOCK_PERIOD=100;

```

```
149 initial begin
150 clock <= 0;
151 forever #(CLOCK_PERIOD/2) clock <= ~clock; // Forever toggle the clock
152 end
153
154 // Set up the inputs to the design. Each line is a clock cycle.
155 initial begin
156 @(posedge clock);
157 killBrickCorner<=0; killBrickX<=0; killBrickY<=0;
158 enable <= 1; @(posedge clock);
159 reset <= 1; @(posedge clock);
160 resetm <= 1; @(posedge clock);
161 reset <= 0; @(posedge clock);
162 resetm <= 0; @(posedge clock);
163 ballX <= 5; @(posedge clock);
164 ballY <= 2; @(posedge clock);
165 L <= 1; @(posedge clock);
166 L <= 0; @(posedge clock);
167 R <= 1; @(posedge clock);
168 R <= 0; @(posedge clock);
169 DIREC <= 2'b11;
170 killBrickCorner <=1; @(posedge clock);
171 killBrickCorner <=0; @(posedge clock);
172 killBrickX <=1; @(posedge clock);
173 killBrickX <=0; @(posedge clock);
174 killBrickY <=1; @(posedge clock);
175 killBrickY <=0; @(posedge clock);
176 resetm <= 1; @(posedge clock); resetm <= 0; @(posedge clock);
177 DIREC <= 2'b00;
178 killBrickCorner <=1; @(posedge clock);
179 killBrickCorner <=0; @(posedge clock);
180 killBrickX <=1; @(posedge clock);
181 killBrickX <=0; @(posedge clock);
182 killBrickY <=1; @(posedge clock);
183 killBrickY <=0; @(posedge clock);
184 resetm <= 1; @(posedge clock); resetm <= 0; @(posedge clock);
185 DIREC <= 2'b10;
186 killBrickCorner <=1; @(posedge clock);
187 killBrickCorner <=0; @(posedge clock);
188 killBrickX <=1; @(posedge clock);
189 killBrickX <=0; @(posedge clock);
190 killBrickY <=1; @(posedge clock);
191 killBrickY <=0; @(posedge clock);
192 resetm <= 1; @(posedge clock); resetm <= 0; @(posedge clock);
193 DIREC <= 2'b01;
194 killBrickCorner <=1; @(posedge clock);
195 killBrickCorner <=0; @(posedge clock);
196 killBrickX <=1; @(posedge clock);
197 killBrickX <=0; @(posedge clock);
198 killBrickY <=1; @(posedge clock);
199 killBrickY <=0; @(posedge clock);
200 $stop; // End the simulation.
201 end
202 endmodule
```

```

1  module Bounce (revX, revY, ballX, ballY, RedPixels, killBrickCorner, killBricky, killBrickX,
2     hitBottom, DIREC, clock);
3     input logic clock;
4     input logic [3:0] ballX, ballY;
5     input logic [1:0] DIREC; // [Y direc, X direc], [00] = down left, [01] = down right, [10] =
up left, [11] = up right
6     input logic [15:0][15:0] RedPixels; // List of occupied spaces
7     output logic revX, revY, killBrickCorner, killBricky, killBrickX, hitBottom;
8     logic nrevX, nrevY, nkillBrickCorner, nkillBricky, nkillBrickX, nhitBottom;
9     // State variables
10    // Allow bouncing around the room; if collision imminent based on direction and red array,
send revX or revY
11
12    /* Ball position: [Y] [X]
13    Y\X  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 <-X
14    [00] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
15
16    [01] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
17
18    [02] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
19
20    [03] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
21
22    [04] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
23
24    [05] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
25
26    [06] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
27
28    [07] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
29
30    [08] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
31
32    [09] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
33
34    [10] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
35
36    [11] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
37
38    [12] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
39
40    [13] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
41
42    [14] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
43
44    [15] 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
45    */
46
47    // Next State logic
48    always_comb begin
49        // Check for Y bounce
50        // Going up:
51        if(DIREC[1]) begin
52            // Contact Flat
53            if(RedPixels[ballY-1][ballX]) begin
54                nkillBricky = 1;
55                nkillBrickCorner = 0;
56                nrevY = 1;
57            end
58            // Contact Corner
59            else if(RedPixels[ballY-1][ballX+1-(DIREC[0]+DIREC[0])]) begin // If the corner in the
direction the ball is going is red
60                nkillBricky = 0;
61                nkillBrickCorner = 1;
62                nrevY = 1;
63            end
64            // Contact wall
65            else if(ballY == 0) begin
66                nkillBricky = 0;
67                nkillBrickCorner = 0;
68                nrevY = 1;
69            end
70            // No Contact
71            else begin

```

```

72     nkillBrickY = 0;
73     nkillBrickCorner = 0;
74     nrevY = 0;
75 end
76 nhitBottom = 0;
77 end
78
79 //Going down:
80 else begin
81     //Contact Flat
82     if(RedPixels[bally+1][ballX]) begin
83         nkillBrickY = 1;
84         nkillBrickCorner = 0;
85         nrevY = 1;
86         nhitBottom = 0;
87     end
88     //Contact Corner
89     else if(RedPixels[bally+1][ballX+1-(DIREC[0]+DIREC[0])]) begin //If the corner in the
direction the ball is going is red
90         nkillBrickY = 0;
91         nkillBrickCorner = 1;
92         nrevY = 1;
93         nhitBottom = 0;
94     end
95     //Contact wall
96     else if(bally == 15) begin
97         nkillBrickY = 0;
98         nkillBrickCorner = 0;
99         nrevY = 0;
100        nhitBottom = 1;
101    end
102    //No Contact
103    else begin
104        nkillBrickY = 0;
105        nkillBrickCorner = 0;
106        nrevY = 0;
107        nhitBottom = 0;
108    end
109 end
110
111 //Check for x bounce
112 //Check for bounce against wall
113 if(((DIREC[0]) & (ballX == 0)) | ((~DIREC[0]) & (ballX == 15))) begin
114     nkillBrickX = 0;
115     nrevX = 1;
116 end
117 //Check for bounce against brick
118 else if(RedPixels[bally][ballX+1-(DIREC[0]+DIREC[0])]) begin
//RedPixels[bally][ballX+1]&(DIREC[0]) | (RedPixels[bally][ballX-1]&(~DIREC[0]))
119     nkillBrickX = 1;
120     nrevX = 1;
121 end
122 //No Contact
123 else begin
124     nkillBrickX = 0;
125     nrevX = 0;
126 end
127
128 end
129
130
131 always_ff @(posedge clock) begin
132     //if(enable) begin
133         revX = nrevX;
134         revY = nrevY;
135         killBrickCorner = nkillBrickCorner;
136         killBrickY = nkillBrickY;
137         killBrickX = nkillBrickX;
138         hitBottom = nhitBottom;
139     /*end else begin
140         revX = revX;
141         revY = revY;
142         killBrickCorner = killBrickCorner;
143         killBrickY = killBrickY;
144         killBrickX = killBrickX;

```



```

145         hitBottom = hitBottom;
146     end */
147 end
148 endmodule
149
150
151
152 module Bounce_testbench();
153     logic clock;
154     logic [3:0] ballX, ballY;
155     logic [1:0] DIREC; //[Y direc, x direc], [00] = down left, [01] = down right, [10] = up
left, [11] = up right
156     logic [15:0][15:0] RedPixels; //List of occupied spaces
157     logic revX, revY, killBrickCorner, killBricky, killBrickX, hitBottom;
158
159     Bounce dut (.revX, .revY, .ballX, .ballY, .RedPixels, .killBrickCorner, .killBricky, .
killBrickX, .hitBottom, .DIREC, .clock);
160
161     // Set up a simulated clock.
162     parameter CLOCK_PERIOD=100;
163     initial begin
164         clock <= 0;
165         forever #(CLOCK_PERIOD/2) clock <= ~clock; // Forever toggle the clock
166     end
167
168     // Set up the inputs to the design. Each line is a clock cycle.
169     initial begin
170         @(posedge clock);
171         ballX <= 5; @(posedge clock);
172         ballY <= 5; @(posedge clock);
173         RedPixels[00] <= 16'b0000000000000000;
174         RedPixels[01] <= 16'b1111111111111111;
175         RedPixels[02] <= 16'b1111111111111111;
176         RedPixels[03] <= 16'b1111111111111111;
177         RedPixels[04] <= 16'b0000000000000000;
178         RedPixels[05] <= 16'b0000000000000000;
179         RedPixels[06] <= 16'b0000000000000000;
180         RedPixels[07] <= 16'b0000000000000000;
181         RedPixels[08] <= 16'b0000000000000000;
182         RedPixels[09] <= 16'b0000000000000000;
183         RedPixels[10] <= 16'b0000000000000000;
184         RedPixels[11] <= 16'b0000000000000000;
185         RedPixels[12] <= 16'b0000000000000000;
186         RedPixels[13] <= 16'b0000000000000000;
187         RedPixels[14] <= 16'b0000000000000000;
188         RedPixels[15] <= 16'b0000000111000000;
189         DIREC <= 2'b11;
190         RedPixels[5][5] <= 1; @(posedge clock); RedPixels[5][5] <= 1; @(posedge clock);
191         RedPixels[4][5] <= 1; @(posedge clock); RedPixels[4][5] <= 0; @(posedge clock);
192         RedPixels[5][4] <= 1; @(posedge clock); RedPixels[5][4] <= 0; @(posedge clock);
193         RedPixels[6][5] <= 1; @(posedge clock); RedPixels[6][5] <= 0; @(posedge clock);
194         RedPixels[5][6] <= 1; @(posedge clock); RedPixels[5][6] <= 0; @(posedge clock);
195         RedPixels[6][6] <= 1; @(posedge clock); RedPixels[6][6] <= 0; @(posedge clock);
196         RedPixels[4][4] <= 1; @(posedge clock); RedPixels[4][4] <= 0; @(posedge clock);
197         RedPixels[4][6] <= 1; @(posedge clock); RedPixels[4][6] <= 0; @(posedge clock);
198         RedPixels[6][4] <= 1; @(posedge clock); RedPixels[6][4] <= 0; @(posedge clock);
199         DIREC <= 2'b00;
200         RedPixels[5][5] <= 1; @(posedge clock); RedPixels[5][5] <= 1; @(posedge clock);
201         RedPixels[4][5] <= 1; @(posedge clock); RedPixels[4][5] <= 0; @(posedge clock);
202         RedPixels[5][4] <= 1; @(posedge clock); RedPixels[5][4] <= 0; @(posedge clock);
203         RedPixels[6][5] <= 1; @(posedge clock); RedPixels[6][5] <= 0; @(posedge clock);
204         RedPixels[5][6] <= 1; @(posedge clock); RedPixels[5][6] <= 0; @(posedge clock);
205         RedPixels[6][6] <= 1; @(posedge clock); RedPixels[6][6] <= 0; @(posedge clock);
206         RedPixels[4][4] <= 1; @(posedge clock); RedPixels[4][4] <= 0; @(posedge clock);
207         RedPixels[4][6] <= 1; @(posedge clock); RedPixels[4][6] <= 0; @(posedge clock);
208         RedPixels[6][4] <= 1; @(posedge clock); RedPixels[6][4] <= 0; @(posedge clock);
209         DIREC <= 2'b01;
210         RedPixels[5][5] <= 1; @(posedge clock); RedPixels[5][5] <= 1; @(posedge clock);
211         RedPixels[4][5] <= 1; @(posedge clock); RedPixels[4][5] <= 0; @(posedge clock);
212         RedPixels[5][4] <= 1; @(posedge clock); RedPixels[5][4] <= 0; @(posedge clock);
213         RedPixels[6][5] <= 1; @(posedge clock); RedPixels[6][5] <= 0; @(posedge clock);
214         RedPixels[5][6] <= 1; @(posedge clock); RedPixels[5][6] <= 0; @(posedge clock);
215         RedPixels[6][6] <= 1; @(posedge clock); RedPixels[6][6] <= 0; @(posedge clock);
216         RedPixels[4][4] <= 1; @(posedge clock); RedPixels[4][4] <= 0; @(posedge clock);
217         RedPixels[4][6] <= 1; @(posedge clock); RedPixels[4][6] <= 0; @(posedge clock);

```

```
218 RedPixels[6][4] <= 1; @(posedge clock); RedPixels[6][4] <= 0; @(posedge clock);
219 DIREC <= 2'b10;
220 RedPixels[5][5] <= 1; @(posedge clock); RedPixels[5][5] <= 1; @(posedge clock);
221 RedPixels[4][5] <= 1; @(posedge clock); RedPixels[4][5] <= 0; @(posedge clock);
222 RedPixels[5][4] <= 1; @(posedge clock); RedPixels[5][4] <= 0; @(posedge clock);
223 RedPixels[6][5] <= 1; @(posedge clock); RedPixels[6][5] <= 0; @(posedge clock);
224 RedPixels[5][6] <= 1; @(posedge clock); RedPixels[5][6] <= 0; @(posedge clock);
225 RedPixels[6][6] <= 1; @(posedge clock); RedPixels[6][6] <= 0; @(posedge clock);
226 RedPixels[4][4] <= 1; @(posedge clock); RedPixels[4][4] <= 0; @(posedge clock);
227 RedPixels[4][6] <= 1; @(posedge clock); RedPixels[4][6] <= 0; @(posedge clock);
228 RedPixels[6][4] <= 1; @(posedge clock); RedPixels[6][4] <= 0; @(posedge clock);
229 $stop; // End the simulation.
230 end
231 endmodule
```

```
1 module keyInput (clk, reset, keyIn, keyOut);
2   input logic clk, reset, keyIn;
3   output logic keyOut;
4   // State variables
5   enum { low, high } in, mid, out;
6
7   // Next State logic
8   always_comb begin
9     if (keyIn) in = high;
10    else in = low;
11  end
12
13  // Output logic - could also be another always_comb block.
14  assign keyOut = (out == high);
15
16  // DFFs
17  always_ff @(posedge clk) begin
18    if (reset) begin
19      mid <= low;
20      out <= low;
21    end else begin
22      out <= mid;
23      mid <= in;
24    end
25  end
26
27  //always_ff @(posedge clk) begin
28  //if (reset)
29  //out <= low;
30  //else
31  //out <= mid;
32  //end
33
34  endmodule
35
36
37
38
39
40 module keyInput_testbench();
41   logic clk, reset, keyIn, keyOut;
42   logic out;
43
44   keyInput dut (clk, reset, keyIn, keyOut);
45
46   // Set up a simulated clock.
47   parameter CLOCK_PERIOD=100;
48   initial begin
49     clk <= 0;
50     forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
51   end
52
53   // Set up the inputs to the design. Each line is a clock cycle.
54   initial begin
55     @(posedge clk);
56     reset <= 1; @(posedge clk); // Always reset FSMs at start
57     reset <= 0; keyIn <= 0; @(posedge clk);
58     @(posedge clk);
59     @(posedge clk);
60     @(posedge clk);
61     keyIn <= 1; @(posedge clk);
62     keyIn <= 0; @(posedge clk);
63     keyIn <= 1; @(posedge clk);
64     @(posedge clk);
65     @(posedge clk);
66     @(posedge clk);
67     @(posedge clk);
68     @(posedge clk);
69     keyIn <= 0; @(posedge clk);
70     @(posedge clk);
71     @(posedge clk);
72     @(posedge clk);
73     keyIn <= 1; @(posedge clk);
74     keyIn <= 0; @(posedge clk);
75     @(posedge clk);
```

```
76    @(posedge clk);  
77    @(posedge clk);  
78    @(posedge clk);  
79    $stop; // End the simulation.  
80    end  
81    endmodule
```

```
1  module seg7 (bcd, leds);
2  input logic [3:0] bcd;
3  output logic [6:0] leds;
4
5  always_comb begin
6  case (bcd)
7  // Light: 6543210
8  4'b0000: leds = 7'b1000000; // 0
9  4'b0001: leds = 7'b1111001; // 1
10 4'b0010: leds = 7'b0100100; // 2
11 4'b0011: leds = 7'b0110000; // 3
12 4'b0100: leds = 7'b0011001; // 4
13 4'b0101: leds = 7'b0010010; // 5
14 4'b0110: leds = 7'b0000010; // 6
15 4'b0111: leds = 7'b1111000; // 7
16 4'b1000: leds = 7'b0000000; // 8
17 4'b1001: leds = 7'b0010000; // 9
18 default: leds = 7'bx;
19 endcase
20 end
21 endmodule
22 //0123456
23 // 6
24 //1 5
25 // 0
26 //2 4
27 // 3
```