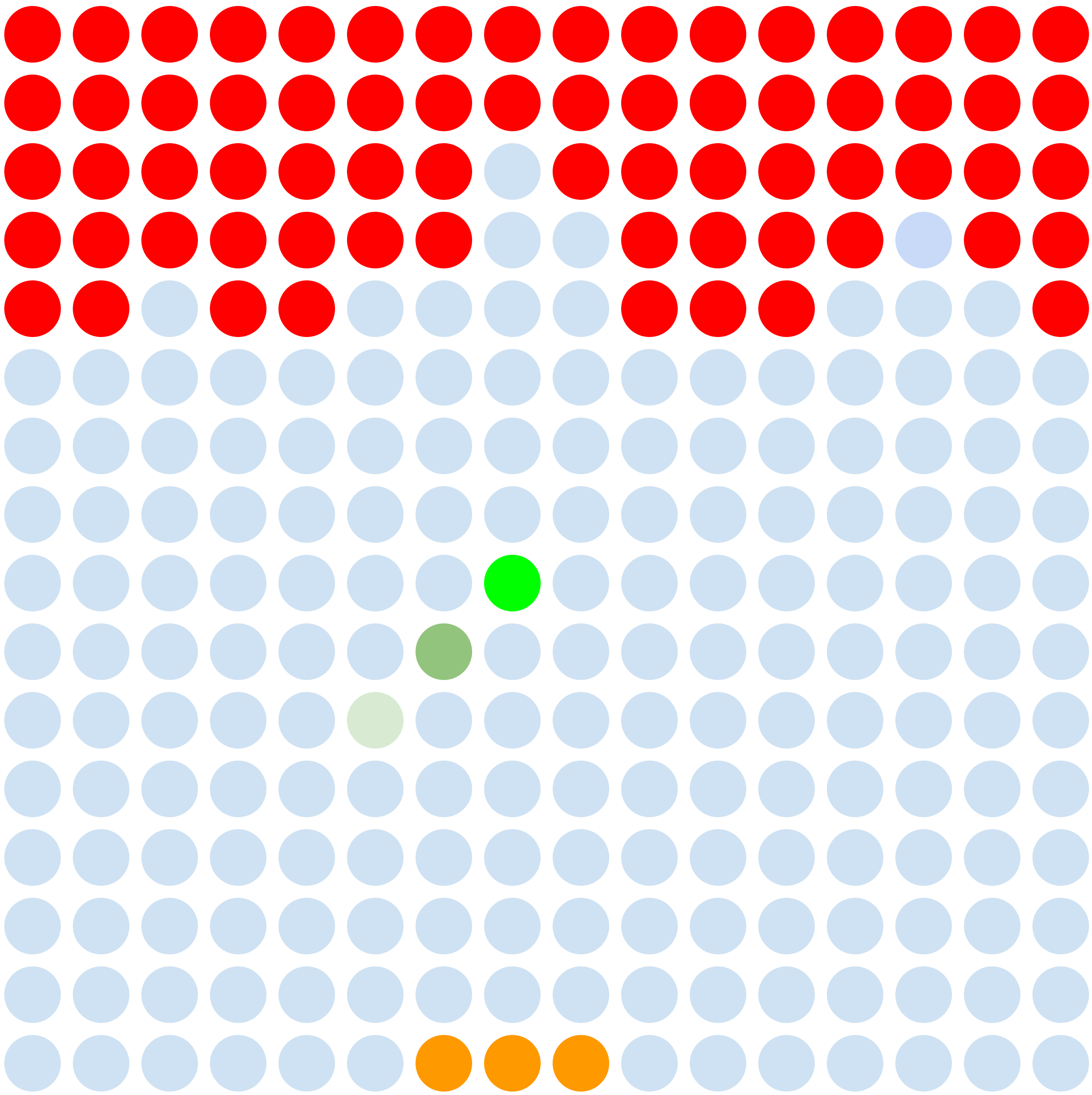
Nikolas Faulkner

JAILBREAK



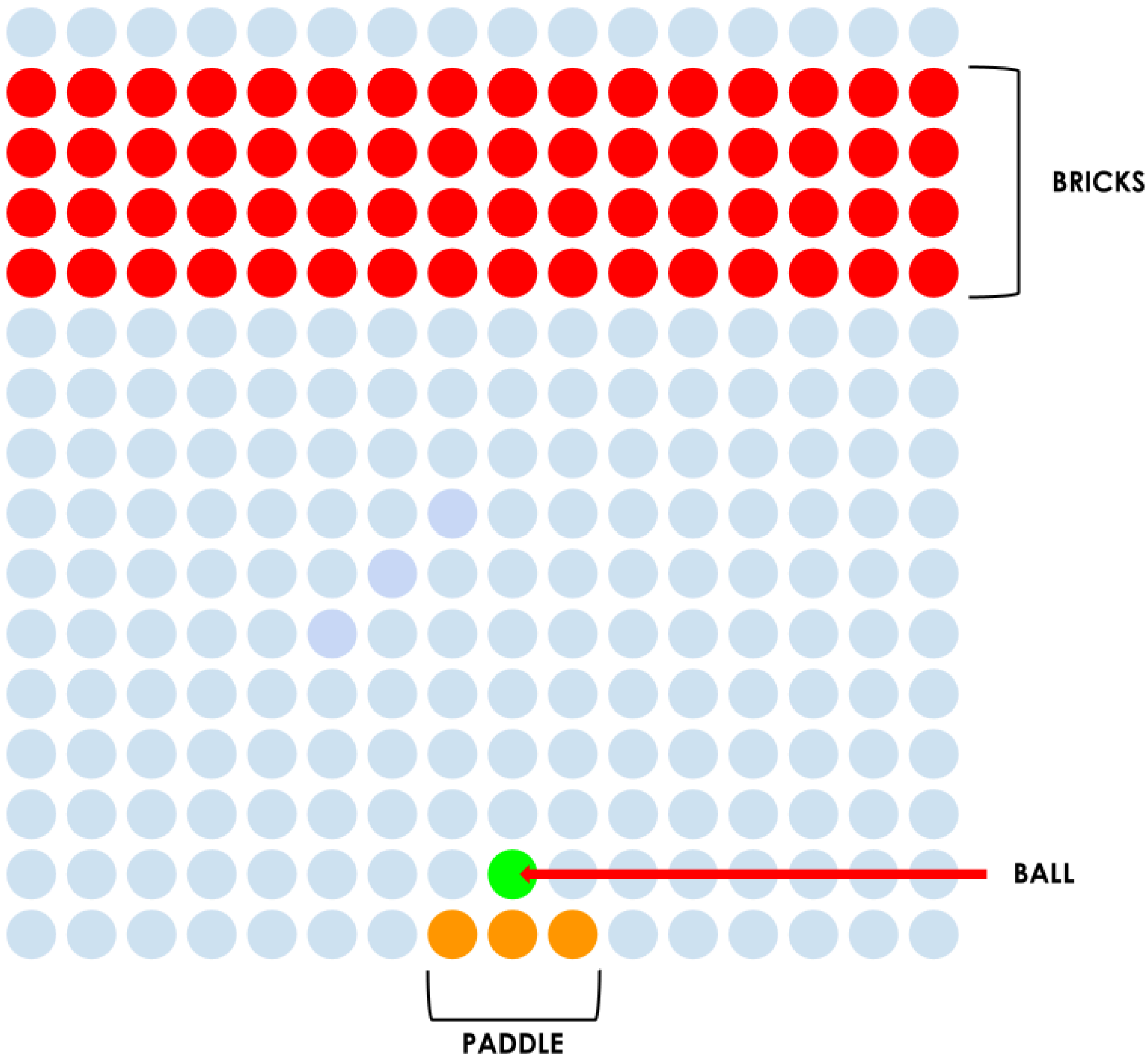
ELECTRONIC VIDEO GAME

EE271 | INSTRUCTION MANUAL

HOW TO PLAY:

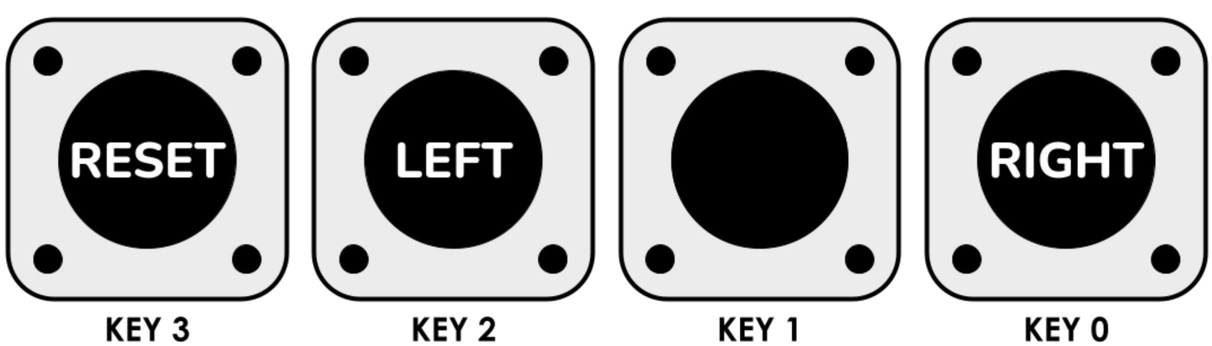
After turning on the game, press the RESET key (KEY 3) to start a new game.

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.

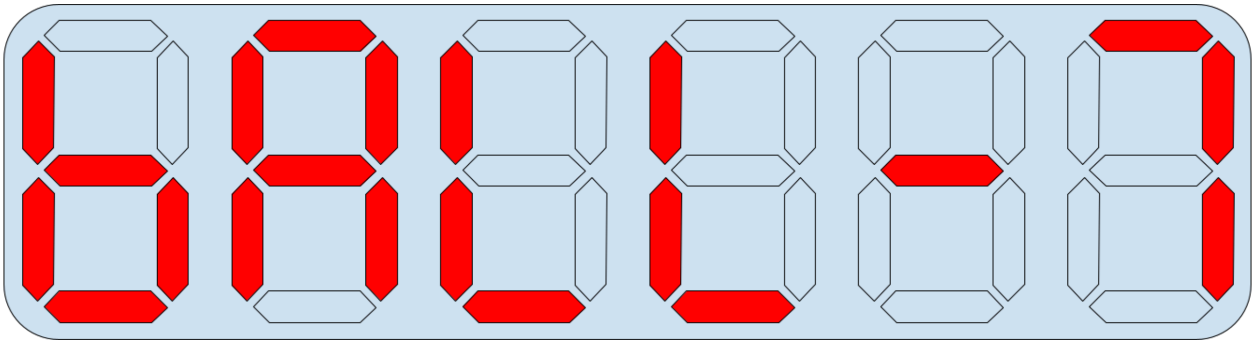
KEYS:



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.

EE271 LAB 8 Market and Usability Analysis   Nikolas Faulkner

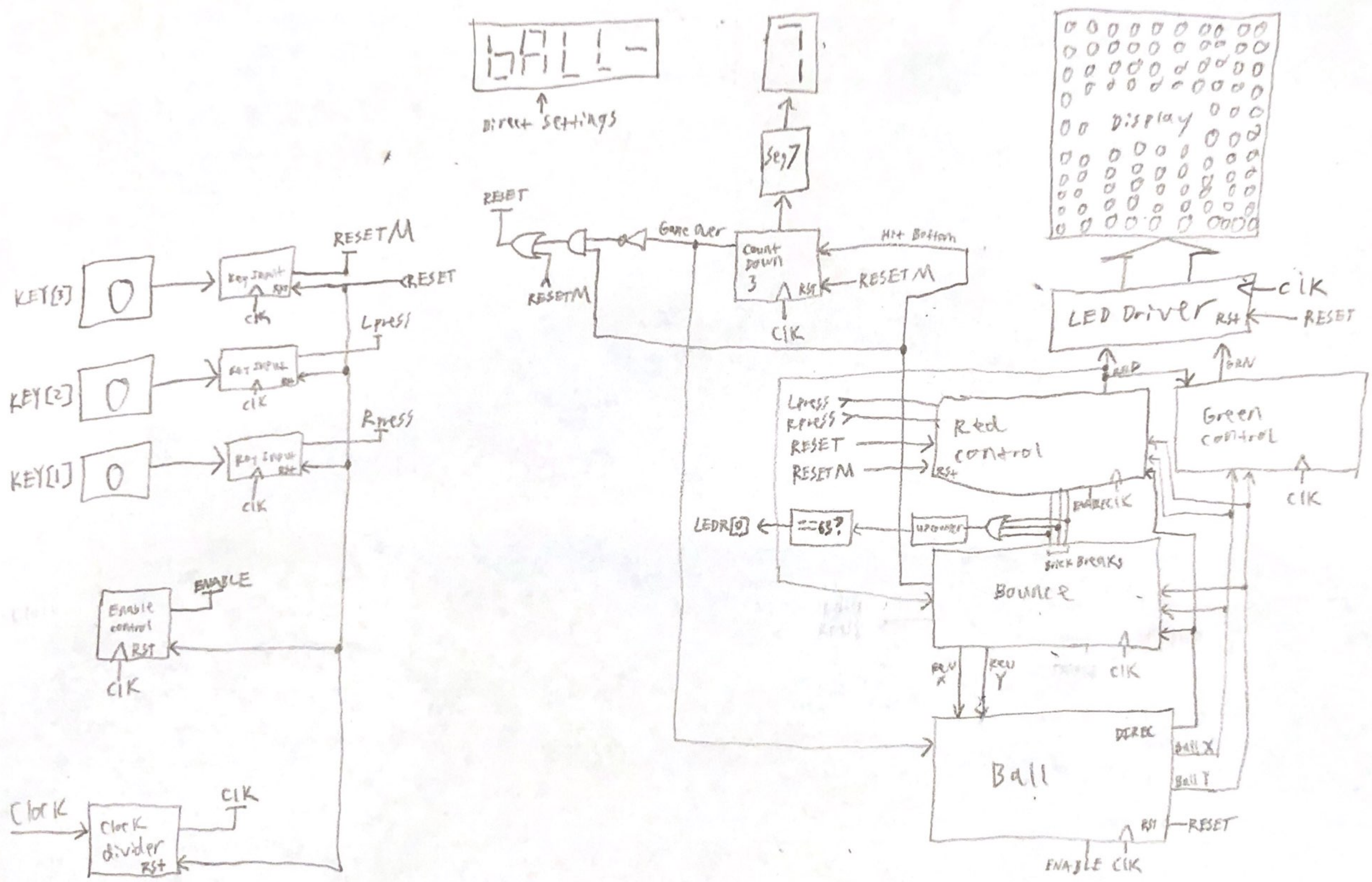
Useability - The system is very simple to use, with an intuitive graphical user interface and only 3 buttons for input. Color coding on the display makes it easy to differentiate between different objects on the screen, and the ball counter is clearly labeled to remove any confusion as to what the number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

```
module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, GPIO_1);
   input logic CLOCK_50; // 50MHz clock.
   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   output logic [9:0] LEDR;
   output logic [35:0] GPIO_1;
   input logic [3:0] KEY; // True when not pressed, False when pressed
   input logic [9:0] SW;


   // Internal logic
   logic L, Lpress, Lkey, R, Rpress, Rkey, hitBottom, win;
   logic reset; //Soft reset, from main reset or life lost
   logic resetm; //Main reset, from KEY[3]
   logic gameover;
   logic [31:0] div_clk;
   logic [2:0] ballCount;
   assign reset = (resetm | ((hitBottom) & ~gameover));

   //Clock and Enable
   // Generate clk off of CLOCK_50, whichClock picks rate.
   parameter whichClock = 15; // 0.75 Hz clock
   clock_divider cdiv (.clock(CLOCK_50), .reset, .divided_clocks(div_clk));
   logic clkSelect, enable;
   // Clock selection; allows for easy switching between simulation and board clocks
   // Set up system base clock to 1526 Hz (50 MHz / 2**(14+1))
   // If you notice flickering, set SYSTEM_CLOCK faster, however this may reduce the brightness of the LED board.

   // Uncomment ONE of the following two lines depending on intention:

   //assign clkSelect = CLOCK_50; // for simulation
   assign clkSelect = div_clk[14]; // 1526 Hz clock signal for display
   //assign clkSelect = div_clk[whichClock]; // for board
   enableControl en (.clock(clkSelect), .reset, .enable); //Sets enable signal

   //Main reset control
   keyInput rstk (.clk(clkSelect), .reset, .keyIn(~KEY[3]), .keyOut(resetm));

   //Ball Count Display
   assign HEX5 = 7'b0000011; //b
   assign HEX4 = 7'b0001000; //A
   assign HEX3 = 7'b1000111; //L
   assign HEX2 = 7'b1000111; //L
   assign HEX1 = 7'b0111111; //-
   countDown3 lives (.out(ballCount), .incr(hitBottom), .reset(resetm), .clk(clkSelect), .gameover); //Decrement life count from 7 upon death
   seg7 ballCountDisp (.bcd(ballCount), .leds(HEX0)); //Display ball count to HEX 0


   // Set up LED board driver
   logic [15:0][15:0]RedPixels; // 16 x 16 array representing red LEDs
   logic [15:0][15:0]GrnPixels; // 16 x 16 array representing green LEDs

   /* Standard LED Driver instantiation - set once and 'forget it'.
      See LEDDriver.sv for more info. Do not modify unless you know what you are doing! */
   LEDDriver Driver (.CLK(clkSelect), .RST(reset), .EnableCount(1'b1), .RedPixels, .GrnPixels, .GPIO_1);

   //LED_test test (.RST(~KEY[0]), .RedPixels);

   //Barmover (.clk(clkSelect), .reset, .w, .GrnPixels);


   // Set up FSM inputs and outputs.
   logic keyinL, keyinR;
   assign keyinL = ~KEY[2]; //Left key
   assign keyinR = ~KEY[0]; //Right key

   keyInput Lk (.clk(clkSelect), .reset, .keyIn(keyinL), .keyOut(Lpress));
   userInput Lu (.clk(clkSelect), .reset, .usIn(Lkey), .usOut(L), .enable);

   keyInput Rk (.clk(clkSelect), .reset, .keyIn(keyinR), .keyOut(Rpress));
   userInput Ru (.clk(clkSelect), .reset, .usIn(Rkey), .usOut(R), .enable);

   logic revX, revY, killBrickCorner, killBrickY, killBrickX, enableEarly;
```

```
    logic [3:0] ballX, ballY;
    logic [2:0] hitcount;
    logic [1:0] DIREC; //[Y direc, X direc], [00] = down left, [01] = down right, [10] = up left, [11] = up right

    // Modules
    Ball BALL (.clk(clkSelect), .reset, .enable, .revX, .revY, .ballX, .ballY, .DIREC, .gameover);
    Bounce BOUNCE (.revX, .revY, .ballX, .ballY, .RedPixels, .killBrickCorner, .killBrickY, .killBrickX, .hitBottom, .DIREC, .clock(clkSelect));
    RedControl REDLIGHTS (.clk(clkSelect), .resetm, .reset, .ballX, .ballY, .DIREC, .RedPixels, .killBrickCorner, .killBrickY, .killBrickX, .L(Lpress), .R(Rpress), .enable);
    GrnControl GREENLIGHTS (.clk(clkSelect), .reset(resetm), .ballX, .ballY, .GrnPixels, .RedPixels);
    counter3b hitcounter (.out(hitcount), .incr(killBrickCorner | killBrickY | killBrickX), .reset(resetm), .clk(clkSelect), .enable);

    assign LEDR[0] = (hitcount == 63);
    assign LEDR[1] = clkSelect;
    assign LEDR[2] = enable;
    assign LEDR[3] = killBrickX;
    assign LEDR[4] = killBrickCorner;
    assign LEDR[5] = DIREC[1];
    assign LEDR[6] = DIREC[0];
    assign LEDR[7] = reset;
    assign LEDR[8] = gameover;
    assign LEDR[9] = Lpress;

endmodule

module DE1_SoC_testbench();
    logic CLOCK_50;
    logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    logic [9:0] LEDR;
    logic [3:0] KEY;
    logic [9:0] SW;

    DE1_SoC dut (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW);

    // Set up a simulated clock.
    parameter CLOCK_PERIOD=100;
    initial begin
        CLOCK_50 <= 0;
        forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50; // Forever toggle the clock
    end

    // Test the design.
    initial begin
        repeat(1) @(posedge CLOCK_50);
        KEY[3] <= 0; repeat(1) @(posedge CLOCK_50); // Reset
        KEY[3] <= 1; repeat(1) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(2) @(posedge CLOCK_50); // Press right
        KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
        KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
        KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right, then wait for a while
        KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
        KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(1) @(posedge CLOCK_50); // Press left
        KEY[2] <= 1; repeat(2) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(1) @(posedge CLOCK_50); // Press left
        KEY[2] <= 1; repeat(2) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(1) @(posedge CLOCK_50); // Press left
        KEY[2] <= 1; repeat(2) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(1) @(posedge CLOCK_50); // Press left
        KEY[2] <= 1; repeat(2) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
        KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
        KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
        KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
```

```
            KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
            KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
            KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
            KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
            KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
            KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
            KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
            KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
            KEY[0] <= 0; repeat(1) @(posedge CLOCK_50); // Press right
            KEY[0] <= 1; repeat(2) @(posedge CLOCK_50);
            $stop; // End the simulation.
        end
    endmodule
```

```
module Ball (clk, reset, enable, revX, revY, ballX, ballY, DIREC, gameover);
 input logic clk, reset, enable, revX, revY, gameover;
 output logic [1:0] DIREC; //[Y direc, X direc], [00] = down left, [01] = down right, [10] = up left, [11] = up right
 output logic [3:0] ballX, ballY;
 // State variables
 logic [3:0] px, nx, py, ny;
 logic [1:0] nDIREC;

 //Bounce around the room, if revX or revY occurs, reverse that direction
 //Output the direction, ball position, and display data for the ball.

 // Next State logic
 always_comb begin

    if (revY) begin //If y bounce, don't change y, swap y direction
    ny = py;
    nDIREC[1] = ~DIREC[1];
    end
    else if (DIREC[1]) begin //If going up and not y bounce, go up
    ny = py-1;
    nDIREC[1] = DIREC[1];
    end
    else begin //If going down and not y bounce, go down
    ny = py+1;
    nDIREC[1] = DIREC[1];
    end

    if (revX) begin //If x bounce, don't change x, swap x direction
    nx = px;
    nDIREC[0] = ~DIREC[0];
    end
    else if (DIREC[0]) begin //If going up and not x bounce, go up
    nx = px-1;
    nDIREC[0] = DIREC[0];
    end
    else begin //If going down and not x bounce, go down
    nx = px+1;
    nDIREC[0] = DIREC[0];
    end

 end

 assign ballX = px;
 assign ballY = py;

 // DFFs
 always_ff @(posedge clk) begin
    if (reset) begin //Ball start position
       px <= 4'b0111;
       py <= 4'b1110;
       DIREC <= 2'b11; //Start going up right
    end
    else if (enable && (~gameover)) begin
       px <= nx;
       py <= ny;
       DIREC = nDIREC;
    end
 end

endmodule



module Ball_testbench();
 logic clock, reset, enable, revX, revY, gameover;
 logic [1:0] DIREC; //[Y direc, X direc], [00] = down left, [01] = down right, [10] = up left, [11] = up right
 logic [3:0] ballX, ballY;

 Ball dut (.clk(clock), .reset, .enable, .revX, .revY, .ballX, .ballY, .DIREC, .gameover);

 // Set up a simulated clock.
 parameter CLOCK_PERIOD=100;
 initial begin
```

```
    clock <= 0;
    forever #(CLOCK_PERIOD/2) clock <= ~clock; // Forever toggle the clock
    end

    // Set up the inputs to the design. Each line is a clock cycle.
    initial begin
    @(posedge clock);
    gameover <= 0; revX <= 0; revY <= 0;
    enable <= 1; @(posedge clock);
    reset <= 1; @(posedge clock);
    reset <= 0; @(posedge clock);
    @(posedge clock); @(posedge clock); @(posedge clock); @(posedge clock);
    revX <= 1; @(posedge clock);
    revX <= 0; @(posedge clock);
    @(posedge clock); @(posedge clock); @(posedge clock);
    revY <= 1; @(posedge clock);
    revY <= 0; @(posedge clock);
    @(posedge clock); @(posedge clock); @(posedge clock);
    revX <= 1; @(posedge clock);
    revX <= 0; @(posedge clock);
    @(posedge clock); @(posedge clock); @(posedge clock);
    revY <= 1; @(posedge clock);
    revY <= 0; @(posedge clock);
    @(posedge clock); @(posedge clock); @(posedge clock);
    $stop; // End the simulation.
    end
  endmodule
```

```
module userInput (clk, reset, usIn, usOut, enable);
 input logic clk, reset, usIn, enable;
 output logic usOut;
 // State variables
 enum { low, high } ns, ps;

 // Next State logic
 always_comb begin
  if (usIn) ns = high;
  else ns = low;
 end

 // Output logic - could also be another always_comb block.
 assign usOut = ((ns == high) & (ps == low));

// DFFs
 always_ff @(posedge clk) begin
 if (reset) begin
   ps <= low;
 end else if (enable) begin
   ps <= ns;
 end
 end

 //always_ff @(posedge clk) begin
 //if (reset)
 //out <= low;
 //else
 //out <= mid;
 //end

 endmodule

 module userInput_testbench();
 logic clk, reset, usIn, usOut;
 logic out;

 userInput dut (clk, reset, usIn, usOut);

 // Set up a simulated clock.
 parameter CLOCK_PERIOD=100;
 initial begin
 clk <= 0;
 forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
 end

 // Set up the inputs to the design. Each line is a clock cycle.
 initial begin
 @(posedge clk);
 reset <= 1; @(posedge clk); // Always reset FSMs at start
 reset <= 0; usIn <= 0; @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 usIn <= 1; @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 usIn <= 0; @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 usIn <= 1; @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 usIn <= 0; @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 usIn <= 1; @(posedge clk);
 usIn <= 0; @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 @(posedge clk);
```

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

```
module enableControl (clock, reset, enable);
 input logic reset, clock;
 output logic enable;
 logic [9:0] counter1, counter2;
 //logic pause;

 always_ff @(posedge clock) begin
    if(reset) begin
    //pause <= 1;
    counter1 <= 10'b0000000000;
    counter2 <= 10'b0000000000;
    //divided_clocks <= 32'b00000000000000000000000000000000;
    end

    else if (counter2 == (10'b1000010000)) begin //Regular time counter
    if (counter1 > 150) begin //Sets ball speed
    counter1 <= 10'b0000000000;
    end
    else begin
    counter1 <= counter1+1;
    end
    end

    else begin //Startup delay after soft reset
    counter2 <= counter2+1;
    end

 end

 assign enable = (counter1 == 100);

endmodule


module enableControl_testbench();
 logic clock, reset;
 logic enable;
 logic [9:0] counter1, counter2;

  enableControl dut (clock, reset, enable, counter1, counter2);

 // Set up a simulated clock.
 parameter CLOCK_PERIOD=100;
 initial begin
 clock <= 0;
 forever #(CLOCK_PERIOD/2) clock <= ~clock; // Forever toggle the clock
 end

 // Set up the inputs to the design. Each line is a clock cycle.
 initial begin
 @(posedge clock);
 reset <= 1; @(posedge clock); // Always reset FSMs at start
 reset <= 0; @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
 @(posedge clock);
```

```
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  @(posedge clock);
  $stop; // End the simulation.
  end
endmodule
```

```
module countDown3 #(parameter WIDTH=3) (out, incr, reset, clk, gameover);
input logic incr, reset, clk;
output logic [WIDTH-1:0] out;
output logic gameover;
logic lastIncr;

always_ff @(posedge clk) begin
   if(reset) begin
      out<=7;
      lastIncr<=0;
   end else if(incr & ~lastIncr) begin
      out<=out-1;
      lastIncr<=1;
   end else if(~incr & lastIncr) begin
      out<=out;
      lastIncr<=0;
   end else
      out<=out; //This is not strictly necessary, it wastes characters but makes it clear to the reader. Can be a comment instead.

   if(reset)
      gameover <= 0;
   else if((out == 0) & incr & (gameover == 0))
      gameover <= 1;
   else
      gameover <= gameover;
end
endmodule

module countDown3_testbench();
 logic incr, reset, clk, gameover;
 logic out;

 counter3 dut (out, incr, reset, clk, gameover);

 // Set up a simulated clock.
 parameter CLOCK_PERIOD=100;
 initial begin
 clk <= 0;
 forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
 end

 // Set up the inputs to the design. Each line is a clock cycle.
 initial begin
 @(posedge clk);
 reset <= 1; @(posedge clk); // Always reset FSMs at start
 reset <= 0; incr <= 0; @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 incr <= 1; @(posedge clk);
 incr <= 0; @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 incr <= 1; @(posedge clk);
 incr <= 0; @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 incr <= 1; @(posedge clk);
 incr <= 0; @(posedge clk);
 incr <= 1; @(posedge clk);
 incr <= 0; @(posedge clk);
 incr <= 1; @(posedge clk);
 incr <= 0; @(posedge clk);
 incr <= 1; @(posedge clk);
 incr <= 0; @(posedge clk);
 incr <= 1; @(posedge clk);
 incr <= 0; @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 incr <= 1; @(posedge clk);
 incr <= 0; @(posedge clk);
 @(posedge clk);
 $stop; // End the simulation.
 end
```

```
endmodule
```

```
module GrnControl (clk, reset, ballX, ballY, GrnPixels, RedPixels);
 input logic clk, reset;
 input logic [3:0] ballX, ballY;
 input logic [15:0][15:0] RedPixels;
 output logic [15:0][15:0] GrnPixels;
 // State variables
 //logic [15:0][15:0] grnMem, ngrnMem; //FF Memory for red pixels (bricks only)


 // DFFs
 always_ff @(posedge clk) begin
      //Set all green to zero, except bottom row, which has paddle
      //Set lowest row to the same as the red pixels
      GrnPixels[00] <= 16'b0000000000000000;
      GrnPixels[01] <= 16'b0000000000000000;
      GrnPixels[02] <= 16'b0000000000000000;
      GrnPixels[03] <= 16'b0000000000000000;
      GrnPixels[04] <= 16'b0000000000000000;
      GrnPixels[05] <= 16'b0000000000000000;
      GrnPixels[06] <= 16'b0000000000000000;
      GrnPixels[07] <= 16'b0000000000000000;
      GrnPixels[08] <= 16'b0000000000000000;
      GrnPixels[09] <= 16'b0000000000000000;
      GrnPixels[10] <= 16'b0000000000000000;
      GrnPixels[11] <= 16'b0000000000000000;
      GrnPixels[12] <= 16'b0000000000000000;
      GrnPixels[13] <= 16'b0000000000000000;
      GrnPixels[14] <= 16'b0000000000000000;
      GrnPixels[15] <= RedPixels[15];

      //Set position of ball to 1
      GrnPixels[ballY][ballX] <= 1'b1;
 end

endmodule




module GrnControl_testbench();
 logic clock, reset;
 logic [3:0] ballX, ballY;
 logic [15:0][15:0] RedPixels;
 logic [15:0][15:0] GrnPixels;

 GrnControl dut (.clk(clock), .reset, .ballX, .ballY, .GrnPixels, .RedPixels);

 // Set up a simulated clock.
 parameter CLOCK_PERIOD=100;
 initial begin
 clock <= 0;
 forever #(CLOCK_PERIOD/2) clock <= ~clock; // Forever toggle the clock
 end

 // Set up the inputs to the design. Each line is a clock cycle.
 initial begin
 @(posedge clock);
 ballX <= 5; @(posedge clock);
 ballY <= 5; @(posedge clock);
 RedPixels[00] <= 16'b0000000000000000;
 RedPixels[01] <= 16'b1111111111111111;
 RedPixels[02] <= 16'b1111111111111111;
 RedPixels[03] <= 16'b1111111111111111;
 RedPixels[04] <= 16'b0000000000000000;
 RedPixels[05] <= 16'b0000000000000000;
 RedPixels[06] <= 16'b0000000000000000;
 RedPixels[07] <= 16'b0000000000000000;
 RedPixels[08] <= 16'b0000000000000000;
 RedPixels[09] <= 16'b0000000000000000;
 RedPixels[10] <= 16'b0000000000000000;
 RedPixels[11] <= 16'b0000000000000000;
 RedPixels[12] <= 16'b0000000000000000;
 RedPixels[13] <= 16'b0000000000000000;
 RedPixels[14] <= 16'b0000000000000000;
 RedPixels[15] <= 16'b0000000111000000; @(posedge clock);
```

```
     ballX <= 4; @(posedge clock);
     ballY <= 4; @(posedge clock);
     ballX <= 3; @(posedge clock);
     ballY <= 3; @(posedge clock);
     ballX <= 2; @(posedge clock);
     ballY <= 2; @(posedge clock);
     ballX <= 1; @(posedge clock);
     ballY <= 1; @(posedge clock);
     RedPixels[15] <= 16'b0001011011010100; @(posedge clock);
     RedPixels[15] <= 16'b1001000101010010; @(posedge clock);
     RedPixels[15] <= 16'b0100101101010101; @(posedge clock);
     RedPixels[15] <= 16'b1111100001111111; @(posedge clock);
     RedPixels[15] <= 16'b0000001110000000; @(posedge clock);
     $stop; // End the simulation.
     end
    endmodule
```

```
module RedControl (clk, resetm, reset, ballX, ballY, DIREC, RedPixels, killBrickCorner,
killBrickX, killBrickY, L, R, enable);
 input logic clk, resetm, reset, killBrickCorner, killBrickX, killBrickY, L, R, enable;
 input logic [3:0] ballX, ballY;
 input logic [1:0] DIREC;
 output logic [15:0][15:0] RedPixels;
 // State variables
 logic [15:0][15:0] nredMem; //FF Memory for red pixels (bricks only)
 logic [3:0] paddlePos, npaddlePos; //FF Paddle position memory


 //Control paddle and brick positions.
 //Delete bricks based on collisions.


 // Next State logic
 always_comb begin

   nredMem[00] <= RedPixels[00];
   nredMem[01] <= RedPixels[01];
   nredMem[02] <= RedPixels[02];
   nredMem[03] <= RedPixels[03];
   nredMem[04] <= RedPixels[04];
   nredMem[05] <= RedPixels[05];
   nredMem[06] <= RedPixels[06];
   nredMem[07] <= RedPixels[07];
   nredMem[08] <= RedPixels[08];
   nredMem[09] <= RedPixels[09];
   nredMem[10] <= RedPixels[10];
   nredMem[11] <= RedPixels[11];
   nredMem[12] <= RedPixels[12];
   nredMem[13] <= RedPixels[13];
   nredMem[14] <= RedPixels[14];
   nredMem[15] <= RedPixels[15];

   if (killBrickX) begin
      nredMem[ballY][ballX+1-(DIREC[0]+DIREC[0])] <= 0;
   end

   if (killBrickY) begin
      nredMem[ballY+1-(DIREC[1]+DIREC[1])][ballX] <= 0;
   end

   if (killBrickCorner) begin
      nredMem[ballY+1-(DIREC[1]+DIREC[1])][ballX+1-(DIREC[0]+DIREC[0])] <= 0;
   end

   if (~reset & L & (paddlePos<14)) begin
      npaddlePos <= paddlePos+1;
   end else if (~reset & R & (paddlePos>1)) begin
      npaddlePos <= paddlePos-1;
   end else begin
      npaddlePos <= paddlePos;
   end

 end

 // DFFs
 always_ff @(posedge clk) begin
    if (resetm) begin //Ball start position
       RedPixels[00] <= 16'b0000000000000000;
       RedPixels[01] <= 16'b1111111111111111;
       RedPixels[02] <= 16'b1111111111111111;
       RedPixels[03] <= 16'b1111111111111111;
       RedPixels[04] <= 16'b1111111111111111;
       RedPixels[05] <= 16'b0000000000000000;
       RedPixels[06] <= 16'b0000000000000000;
       RedPixels[07] <= 16'b0000000000000000;
       RedPixels[08] <= 16'b0000000000000000;
       RedPixels[09] <= 16'b0000000000000000;
       RedPixels[10] <= 16'b0000000000000000;
       RedPixels[11] <= 16'b0000000000000000;
       RedPixels[12] <= 16'b0000000000000000;
       RedPixels[13] <= 16'b0000000000000000;
       RedPixels[14] <= 16'b0000000000000000;
```

```
       RedPixels[15] <= 16'b0000000111000000;
       paddlePos <= 4'b0111;
    end
    else if (enable) begin
       RedPixels[00] <= nredMem[00];
       RedPixels[01] <= nredMem[01];
       RedPixels[02] <= nredMem[02];
       RedPixels[03] <= nredMem[03];
       RedPixels[04] <= nredMem[04];
       RedPixels[05] <= nredMem[05];
       RedPixels[06] <= nredMem[06];
       RedPixels[07] <= nredMem[07];
       RedPixels[08] <= nredMem[08];
       RedPixels[09] <= nredMem[09];
       RedPixels[10] <= nredMem[10];
       RedPixels[11] <= nredMem[11];
       RedPixels[12] <= nredMem[12];
       RedPixels[13] <= nredMem[13];
       RedPixels[14] <= nredMem[14];
       if (reset) begin
         paddlePos <= 4'b0111;
       end
       else begin
          paddlePos <= npaddlePos;

          if(npaddlePos == 1) begin
            RedPixels[15] <= 16'b0000000000000111;
          end else if(npaddlePos == 2) begin
            RedPixels[15] <= 16'b0000000000001110;
          end else if(npaddlePos == 3) begin
            RedPixels[15] <= 16'b0000000000011100;
          end else if(npaddlePos == 4) begin
            RedPixels[15] <= 16'b0000000000111000;
          end else if(npaddlePos == 5) begin
            RedPixels[15] <= 16'b0000000001110000;
          end else if(npaddlePos == 6) begin
            RedPixels[15] <= 16'b0000000011100000;
          end else if(npaddlePos == 7) begin
            RedPixels[15] <= 16'b0000000111000000;
          end else if(npaddlePos == 8) begin
            RedPixels[15] <= 16'b0000001110000000;
          end else if(npaddlePos == 9) begin
            RedPixels[15] <= 16'b0000011100000000;
          end else if(npaddlePos == 10) begin
            RedPixels[15] <= 16'b0000111000000000;
          end else if(npaddlePos == 11) begin
            RedPixels[15] <= 16'b0001110000000000;
          end else if(npaddlePos == 12) begin
            RedPixels[15] <= 16'b0011100000000000;
          end else if(npaddlePos == 13) begin
            RedPixels[15] <= 16'b0111000000000000;
          end else if(npaddlePos == 14) begin
            RedPixels[15] <= 16'b1110000000000000;
          end else begin
            RedPixels[15] <= nredMem[00];
          end
       end
    end
 end

endmodule



module RedControl_testbench();
 logic clock, resetm, reset, killBrickCorner, killBrickX, killBrickY, L, R, enable;
 logic [3:0] ballX, ballY;
 logic [1:0] DIREC;
 logic [15:0][15:0] RedPixels;

 RedControl dut (.clk(clock), .resetm, .reset, .ballX, .ballY, .DIREC, .RedPixels, .killBrickCorner, .killBrickX, .killBrickY, .L, .R, .enable);

 // Set up a simulated clock.
 parameter CLOCK_PERIOD=100;
```

```
  initial begin
  clock <= 0;
  forever #(CLOCK_PERIOD/2) clock <= ~clock; // Forever toggle the clock
  end

  // Set up the inputs to the design. Each line is a clock cycle.
  initial begin
  @(posedge clock);
  killBrickCorner<=0; killBrickX<=0; killBrickY<=0;
  enable <= 1; @(posedge clock);
  reset <= 1; @(posedge clock);
  resetm <= 1; @(posedge clock);
  reset <= 0; @(posedge clock);
  resetm <= 0; @(posedge clock);
  ballX <= 5; @(posedge clock);
  ballY <= 2; @(posedge clock);
  L <= 1; @(posedge clock);
  L <= 0; @(posedge clock);
  R <= 1; @(posedge clock);
  R <= 0; @(posedge clock);
  DIREC <= 2'b11;
  killBrickCorner <=1; @(posedge clock);
  killBrickCorner <=0; @(posedge clock);
  killBrickX <=1; @(posedge clock);
  killBrickX <=0; @(posedge clock);
  killBrickY <=1; @(posedge clock);
  killBrickY <=0; @(posedge clock);
  resetm <= 1; @(posedge clock); resetm <= 0; @(posedge clock);
  DIREC <= 2'b00;
  killBrickCorner <=1; @(posedge clock);
  killBrickCorner <=0; @(posedge clock);
  killBrickX <=1; @(posedge clock);
  killBrickX <=0; @(posedge clock);
  killBrickY <=1; @(posedge clock);
  killBrickY <=0; @(posedge clock);
  resetm <= 1; @(posedge clock); resetm <= 0; @(posedge clock);
  DIREC <= 2'b10;
  killBrickCorner <=1; @(posedge clock);
  killBrickCorner <=0; @(posedge clock);
  killBrickX <=1; @(posedge clock);
  killBrickX <=0; @(posedge clock);
  killBrickY <=1; @(posedge clock);
  killBrickY <=0; @(posedge clock);
  resetm <= 1; @(posedge clock); resetm <= 0; @(posedge clock);
  DIREC <= 2'b01;
  killBrickCorner <=1; @(posedge clock);
  killBrickCorner <=0; @(posedge clock);
  killBrickX <=1; @(posedge clock);
  killBrickX <=0; @(posedge clock);
  killBrickY <=1; @(posedge clock);
  killBrickY <=0; @(posedge clock);
  $stop; // End the simulation.
  end
endmodule
```

```
module Bounce (revX, revY, ballX, ballY, RedPixels, killBrickCorner, killBrickY, killBrickX, hitBottom, DIREC, clock);
 input logic clock;
 input logic [3:0] ballX, ballY;
 input logic [1:0] DIREC; //[Y direc, X direc], [00] = down left, [01] = down right, [10] = up left, [11] = up right
 input logic [15:0][15:0] RedPixels; //List of occupied spaces
 output logic revX, revY, killBrickCorner, killBrickY, killBrickX, hitBottom;
 logic nrevX, nrevY, nkillBrickCorner, nkillBrickY, nkillBrickX, nhitBottom;
 // State variables

 //Allow bouncing around the room; if collision imminent based on direction and red array, send revX or revY

 /* Ball position: [Y] [X]
 Y\/  0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15 <-X
 [00] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [01] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [02] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [03] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [04] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [05] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [06] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [07] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [08] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [09] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [10] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [11] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [12] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [13] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [14] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O

 [15] O   O   O   O   O   O   O   O   O   O   O   O   O   O   O   O
 */

 // Next State logic
 always_comb begin
   //Check for Y bounce
   //Going up:
   if(DIREC[1]) begin
      //Contact Flat
      if(RedPixels[ballY-1][ballX]) begin
         nkillBrickY = 1;
         nkillBrickCorner = 0;
         nrevY = 1;
      end
      //Contact Corner
      else if(RedPixels[ballY-1][ballX+1-(DIREC[0]+DIREC[0])]) begin //If the corner in the direction the ball is going is red
         nkillBrickY = 0;
         nkillBrickCorner = 1;
         nrevY = 1;
      end
      //Contact Wall
      else if(ballY == 0) begin
         nkillBrickY = 0;
         nkillBrickCorner = 0;
         nrevY = 1;
      end
      //No Contact
      else begin
```

```
           nkillBrickY = 0;
           nkillBrickCorner = 0;
           nrevY = 0;
        end
        nhitBottom = 0;
     end

     //Going down:
     else begin
        //Contact Flat
        if(RedPixels[ballY+1][ballX]) begin
           nkillBrickY = 1;
           nkillBrickCorner = 0;
           nrevY = 1;
           nhitBottom = 0;
        end
        //Contact Corner
        else if(RedPixels[ballY+1][ballX+1-(DIREC[0]+DIREC[0])]) begin //If the corner in the direction the ball is going is red
           nkillBrickY = 0;
           nkillBrickCorner = 1;
           nrevY = 1;
           nhitBottom = 0;
        end
        //Contact Wall
        else if(ballY == 15) begin
           nkillBrickY = 0;
           nkillBrickCorner = 0;
           nrevY = 0;
           nhitBottom = 1;
        end
        //No Contact
        else begin
           nkillBrickY = 0;
           nkillBrickCorner = 0;
           nrevY = 0;
           nhitBottom = 0;
        end
     end

     //Check for X bounce
     //Check for bounce against wall
     if(((DIREC[0]) & (ballX == 0)) | ((~DIREC[0]) & (ballX == 15))) begin
        nkillBrickX = 0;
        nrevX = 1;
     end
     //Check for bounce against brick
     else if(RedPixels[ballY][ballX+1-(DIREC[0]+DIREC[0])]) begin //(RedPixels[ballY][ballX+1]&(DIREC[0])) | (RedPixels[ballY][ballX-1]&(~DIREC[0]))
        nkillBrickX = 1;
        nrevX = 1;
     end
     //No Contact
     else begin
        nkillBrickX = 0;
        nrevX = 0;
     end

   end


   always_ff @(posedge clock) begin
     //if(enable) begin
        revX = nrevX;
        revY = nrevY;
        killBrickCorner = nkillBrickCorner;
        killBrickY = nkillBrickY;
        killBrickX = nkillBrickX;
        hitBottom = nhitBottom;
     /*end else begin
        revX = revX;
        revY = revY;
        killBrickCorner = killBrickCorner;
        killBrickY = killBrickY;
        killBrickX = killBrickX;
```

```
            hitBottom = hitBottom;
        end */
    end
endmodule



module Bounce_testbench();
  logic clock;
  logic [3:0] ballX, ballY;
  logic [1:0] DIREC; //[Y direc, X direc], [00] = down left, [01] = down right, [10] = up left, [11] = up right
  logic [15:0][15:0] RedPixels; //List of occupied spaces
  logic revX, revY, killBrickCorner, killBrickY, killBrickX, hitBottom;

  Bounce dut (.revX, .revY, .ballX, .ballY, .RedPixels, .killBrickCorner, .killBrickY, .killBrickX, .hitBottom, .DIREC, .clock);

  // Set up a simulated clock.
  parameter CLOCK_PERIOD=100;
  initial begin
  clock <= 0;
  forever #(CLOCK_PERIOD/2) clock <= ~clock; // Forever toggle the clock
  end

  // Set up the inputs to the design. Each line is a clock cycle.
  initial begin
  @(posedge clock);
  ballX <= 5; @(posedge clock);
  ballY <= 5; @(posedge clock);
  RedPixels[00] <= 16'b0000000000000000;
  RedPixels[01] <= 16'b1111111111111111;
  RedPixels[02] <= 16'b1111111111111111;
  RedPixels[03] <= 16'b1111111111111111;
  RedPixels[04] <= 16'b0000000000000000;
  RedPixels[05] <= 16'b0000000000000000;
  RedPixels[06] <= 16'b0000000000000000;
  RedPixels[07] <= 16'b0000000000000000;
  RedPixels[08] <= 16'b0000000000000000;
  RedPixels[09] <= 16'b0000000000000000;
  RedPixels[10] <= 16'b0000000000000000;
  RedPixels[11] <= 16'b0000000000000000;
  RedPixels[12] <= 16'b0000000000000000;
  RedPixels[13] <= 16'b0000000000000000;
  RedPixels[14] <= 16'b0000000000000000;
  RedPixels[15] <= 16'b0000000111000000;
  DIREC <= 2'b11;
  RedPixels[5][5] <= 1; @(posedge clock); RedPixels[5][5] <= 1; @(posedge clock);
  RedPixels[4][5] <= 1; @(posedge clock); RedPixels[4][5] <= 0; @(posedge clock);
  RedPixels[5][4] <= 1; @(posedge clock); RedPixels[5][4] <= 0; @(posedge clock);
  RedPixels[6][5] <= 1; @(posedge clock); RedPixels[6][5] <= 0; @(posedge clock);
  RedPixels[5][6] <= 1; @(posedge clock); RedPixels[5][6] <= 0; @(posedge clock);
  RedPixels[6][6] <= 1; @(posedge clock); RedPixels[6][6] <= 0; @(posedge clock);
  RedPixels[4][4] <= 1; @(posedge clock); RedPixels[4][4] <= 0; @(posedge clock);
  RedPixels[4][6] <= 1; @(posedge clock); RedPixels[4][6] <= 0; @(posedge clock);
  RedPixels[6][4] <= 1; @(posedge clock); RedPixels[6][4] <= 0; @(posedge clock);
  DIREC <= 2'b00;
  RedPixels[5][5] <= 1; @(posedge clock); RedPixels[5][5] <= 1; @(posedge clock);
  RedPixels[4][5] <= 1; @(posedge clock); RedPixels[4][5] <= 0; @(posedge clock);
  RedPixels[5][4] <= 1; @(posedge clock); RedPixels[5][4] <= 0; @(posedge clock);
  RedPixels[6][5] <= 1; @(posedge clock); RedPixels[6][5] <= 0; @(posedge clock);
  RedPixels[5][6] <= 1; @(posedge clock); RedPixels[5][6] <= 0; @(posedge clock);
  RedPixels[6][6] <= 1; @(posedge clock); RedPixels[6][6] <= 0; @(posedge clock);
  RedPixels[4][4] <= 1; @(posedge clock); RedPixels[4][4] <= 0; @(posedge clock);
  RedPixels[4][6] <= 1; @(posedge clock); RedPixels[4][6] <= 0; @(posedge clock);
  RedPixels[6][4] <= 1; @(posedge clock); RedPixels[6][4] <= 0; @(posedge clock);
  DIREC <= 2'b01;
  RedPixels[5][5] <= 1; @(posedge clock); RedPixels[5][5] <= 1; @(posedge clock);
  RedPixels[4][5] <= 1; @(posedge clock); RedPixels[4][5] <= 0; @(posedge clock);
  RedPixels[5][4] <= 1; @(posedge clock); RedPixels[5][4] <= 0; @(posedge clock);
  RedPixels[6][5] <= 1; @(posedge clock); RedPixels[6][5] <= 0; @(posedge clock);
  RedPixels[5][6] <= 1; @(posedge clock); RedPixels[5][6] <= 0; @(posedge clock);
  RedPixels[6][6] <= 1; @(posedge clock); RedPixels[6][6] <= 0; @(posedge clock);
  RedPixels[4][4] <= 1; @(posedge clock); RedPixels[4][4] <= 0; @(posedge clock);
  RedPixels[4][6] <= 1; @(posedge clock); RedPixels[4][6] <= 0; @(posedge clock);
```

```
    RedPixels[6][4] <= 1; @(posedge clock); RedPixels[6][4] <= 0; @(posedge clock);
    DIREC <= 2'b10;
    RedPixels[5][5] <= 1; @(posedge clock); RedPixels[5][5] <= 1; @(posedge clock);
    RedPixels[4][5] <= 1; @(posedge clock); RedPixels[4][5] <= 0; @(posedge clock);
    RedPixels[5][4] <= 1; @(posedge clock); RedPixels[5][4] <= 0; @(posedge clock);
    RedPixels[6][5] <= 1; @(posedge clock); RedPixels[6][5] <= 0; @(posedge clock);
    RedPixels[5][6] <= 1; @(posedge clock); RedPixels[5][6] <= 0; @(posedge clock);
    RedPixels[6][6] <= 1; @(posedge clock); RedPixels[6][6] <= 0; @(posedge clock);
    RedPixels[4][4] <= 1; @(posedge clock); RedPixels[4][4] <= 0; @(posedge clock);
    RedPixels[4][6] <= 1; @(posedge clock); RedPixels[4][6] <= 0; @(posedge clock);
    RedPixels[6][4] <= 1; @(posedge clock); RedPixels[6][4] <= 0; @(posedge clock);
    $stop; // End the simulation.
    end
  endmodule
```

```
module keyInput (clk, reset, keyIn, keyOut);
 input logic clk, reset, keyIn;
 output logic keyOut;
 // State variables
 enum { low, high } in, mid, out;

 // Next State logic
 always_comb begin
  if (keyIn) in = high;
  else in = low;
 end

 // Output logic - could also be another always_comb block.
 assign keyOut = (out == high);

// DFFs
 always_ff @(posedge clk) begin
 if (reset) begin
   mid <= low;
   out <= low;
 end else begin
   out <= mid;
   mid <= in;
 end
 end

 //always_ff @(posedge clk) begin
 //if (reset)
 //out <= low;
 //else
 //out <= mid;
 //end

 endmodule




 module keyInput_testbench();
 logic clk, reset, keyIn, keyOut;
 logic out;

 keyInput dut (clk, reset, keyIn, keyOut);

 // Set up a simulated clock.
 parameter CLOCK_PERIOD=100;
 initial begin
 clk <= 0;
 forever #(CLOCK_PERIOD/2) clk <= ~clk; // Forever toggle the clock
 end

 // Set up the inputs to the design. Each line is a clock cycle.
 initial begin
 @(posedge clk);
 reset <= 1; @(posedge clk); // Always reset FSMs at start
 reset <= 0; keyIn <= 0; @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 keyIn <= 1; @(posedge clk);
 keyIn <= 0; @(posedge clk);
 keyIn <= 1; @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 keyIn <= 0; @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 @(posedge clk);
 keyIn <= 1; @(posedge clk);
 keyIn <= 0; @(posedge clk);
 @(posedge clk);
```

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

```
module seg7 (bcd, leds);
 input logic [3:0] bcd;
 output logic [6:0] leds;

 always_comb begin
 case (bcd)
 // Light: 6543210
 4'b0000: leds = 7'b1000000; // 0
 4'b0001: leds = 7'b1111001; // 1
 4'b0010: leds = 7'b0100100; // 2
 4'b0011: leds = 7'b0110000; // 3
 4'b0100: leds = 7'b0011001; // 4
 4'b0101: leds = 7'b0010010; // 5
 4'b0110: leds = 7'b0000010; // 6
 4'b0111: leds = 7'b1111000; // 7
 4'b1000: leds = 7'b0000000; // 8
 4'b1001: leds = 7'b0010000; // 9
 default: leds = 7'bX;
 endcase
 end
endmodule
//0123456
//  6
//1   5
//  0
//2   4
//  3
```