Doruk Arisoy 1630389
EE271 A

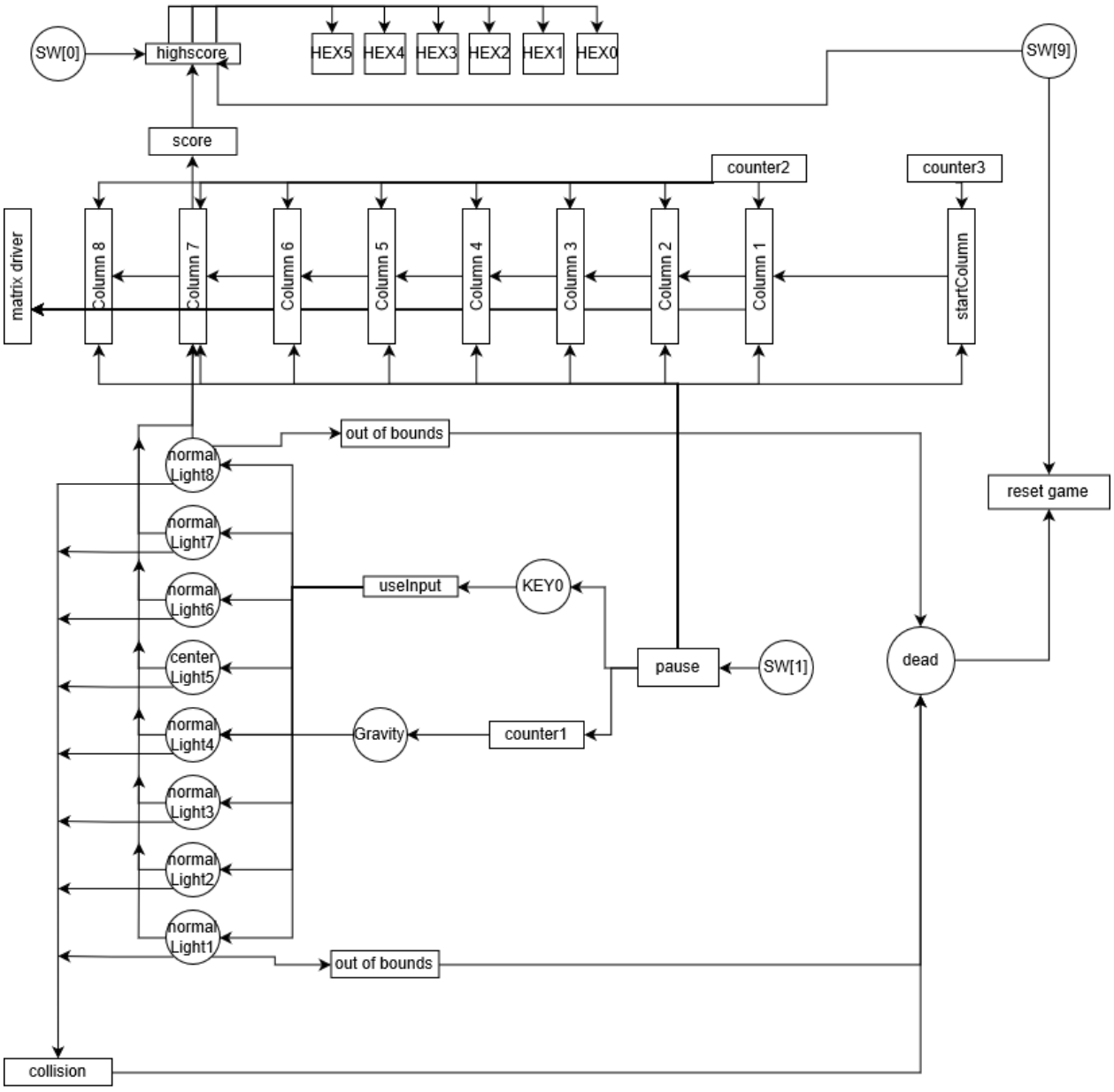
Lab8: Flappy Birds

a) How the game operates:

8x8 LED display is the playing field where the red dot represents the bird and the green dots are the barriers. The goal is to move the bird through as many barriers as possible. Game is over when the bird goes out of bounds (either from the top or bottom of the playing field) or the bird runs into a green barrier. The game automatically restarts when the bird dies. Every barrier crossed alive is one point the points will be displayed on the HEX displays. Here are the controls to play the game:

KEY[0]: To make the bird fly (jump up LEDs), press KEY[0]. Every key press is one jump. Holding the key down will only make the bird jump once.
SW[0]: When SW[0] is turned off, the HEX displays will show the score of the current game. When the switch is turned on, the HEX displays will show the high score (the highest score reached out of all of the games played so far).
SW[1]: Switching the SW[1] on will pause the current game. An orange pause sign will appear on the LED display. The pause sign will disappear once the switch is turned off and the game will continue from where it was left.
SW[9]: The game resets once SW[9] is turned on. High score will be set to 0.

b) Block diagram of the entire design:



c) Verilog code for all elements of the design:

```
// Top-level module defining I/Os for the DE1_SoC board
module  DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, GPIO_0);
   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_0;
   input  logic  [3:0]  KEY;
   input  logic  [9:0]  SW;

   // Create the local variables used in the module
   logic key0, key0_1, sw0, sw0_0, pause, pause_0, reset, input1, input2, enable2, enable3, out_of_bounds,
   dead, init8, init7, init6, init5, init4, init3, init2, init1, out0, out1, out2, out3;
   logic [7:0][7:0] green_array, red_array, green_array1, red_array1;
   logic [3:0] hex0, hex1, hex2, hex3;

   // Default values for unused outputs
   assign LEDR[9:0] = 10'b0000000000;
   assign GPIO_0[11:0] = 12'b000000000000;
   assign red_array[0][7]   = 1'b0;
   assign red_array[0][5:0] = 6'b000000;
   assign red_array[1][7]   = 1'b0;
   assign red_array[1][5:0] = 6'b000000;
   assign red_array[2][7]   = 1'b0;
   assign red_array[2][5:0] = 6'b000000;
   assign red_array[3][7]   = 1'b0;
   assign red_array[3][5:0] = 6'b000000;
   assign red_array[4][7]   = 1'b0;
   assign red_array[4][5:0] = 6'b000000;
   assign red_array[5][7]   = 1'b0;
   assign red_array[5][5:0] = 6'b000000;
   assign red_array[6][7]   = 1'b0;
   assign red_array[6][5:0] = 6'b000000;
   assign red_array[7][7]   = 1'b0;
   assign red_array[7][5:0] = 6'b000000;

   // Generate clk off of CLOCK_50, whichClock picks rate.
   logic [31:0] clk;
   parameter whichClock = 14;
   clock_divider cdiv (CLOCK_50, clk);

   // Define the reset switch
   assign reset = SW[9];

   // Put the user inputs through 2 DFFs.
   always_ff @(posedge CLOCK_50) begin
      if (reset) begin
         key0 <= 1'b1;
         sw0  <= 1'b0;
         pause <= 1'b0;
      end
      else begin
         key0_1 <= KEY[0];
         key0   <= key0_1;
         sw0_0  <= SW[0];
         sw0    <= sw0_0;
         pause_0 <= SW[1];
         pause  <= pause_0;
      end
   end

   // User input only true for 1 clock cycle when a key is pressed and generate computer input
   useInput player     (.clk(clk[whichClock]), .reset, .keyin(~key0), .keyout(input1));

   // Slow down circuit clock
   counter1 countone   (.clk(clk[whichClock]), .reset, .pause, .enable(input2));
   counter2 counttwo   (.clk(clk[whichClock]), .reset, .pause, .enable(enable2));
   counter3 countthree (.clk(clk[whichClock]), .reset, .pause, .enable(enable3));
```

```
// Generate the location of the bird on the LED display
normalLight bird8 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .key(input1),
.gravity(input2), .top(1'b0),            .bottom(red_array[6][6]), .light(red_array[7][6]));
normalLight bird7 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .key(input1),
.gravity(input2), .top(red_array[7][6]), .bottom(red_array[5][6]), .light(red_array[6][6]));
normalLight bird6 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .key(input1),
.gravity(input2), .top(red_array[6][6]), .bottom(red_array[4][6]), .light(red_array[5][6]));
centerLight bird5 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .key(input1),
.gravity(input2), .top(red_array[5][6]), .bottom(red_array[3][6]), .light(red_array[4][6]));
normalLight bird4 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .key(input1),
.gravity(input2), .top(red_array[4][6]), .bottom(red_array[2][6]), .light(red_array[3][6]));
normalLight bird3 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .key(input1),
.gravity(input2), .top(red_array[3][6]), .bottom(red_array[1][6]), .light(red_array[2][6]));
normalLight bird2 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .key(input1),
.gravity(input2), .top(red_array[2][6]), .bottom(red_array[0][6]), .light(red_array[1][6]));
normalLight bird1 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .key(input1),
.gravity(input2), .top(red_array[1][6]), .bottom(1'b0),           .light(red_array[0][6]));

// Check if the bird has gone out of bounds
bounds dead1 (.clk(clk[whichClock]), .reset, .key(input1), .gravity(input2), .top(red_array[7][6]),
.bottom(red_array[0][6]), .out_of_bounds);

// Generae barriers and move them accross the LED display
startColumn initialcol  (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .enable3, .light8(init8),
.light7(init7), .light6(init6), .light5(init5), .light4(init4), .light3(init3), .light2(init2), .light1(init1));
normalColumn col1 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .enable2, .ol8(init8),
.ol7(init7), .ol6(init6), .ol5(init5), .ol4(init4), .ol3(init3), .ol2(init2), .ol1(init1),
                   .light8(green_array[7][0]), .light7(green_array[6][0]), .light6(green_array[5][0]),
                   .light5(green_array[4][0]), .light4(green_array[3][0]), .light3(green_array[2][0]),
                   .light2(green_array[1][0]), .light1(green_array[0][0]));
normalColumn col2 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .enable2,
.ol8(green_array[7][0]), .ol7(green_array[6][0]), .ol6(green_array[5][0]), .ol5(green_array[4][0]),
.ol4(green_array[3][0]), .ol3(green_array[2][0]), .ol2(green_array[1][0]), .ol1(green_array[0][0]),
                   .light8(green_array[7][1]), .light7(green_array[6][1]), .light6(green_array[5][1]),
                   .light5(green_array[4][1]), .light4(green_array[3][1]), .light3(green_array[2][1]),
                   .light2(green_array[1][1]), .light1(green_array[0][1]));
normalColumn col3 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .enable2,
.ol8(green_array[7][1]), .ol7(green_array[6][1]), .ol6(green_array[5][1]), .ol5(green_array[4][1]),
.ol4(green_array[3][1]), .ol3(green_array[2][1]), .ol2(green_array[1][1]), .ol1(green_array[0][1]),
                   .light8(green_array[7][2]), .light7(green_array[6][2]), .light6(green_array[5][2]),
                   .light5(green_array[4][2]), .light4(green_array[3][2]), .light3(green_array[2][2]),
                   .light2(green_array[1][2]), .light1(green_array[0][2]));
normalColumn col4 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .enable2,
.ol8(green_array[7][2]), .ol7(green_array[6][2]), .ol6(green_array[5][2]), .ol5(green_array[4][2]),
.ol4(green_array[3][2]), .ol3(green_array[2][2]), .ol2(green_array[1][2]), .ol1(green_array[0][2]),
                   .light8(green_array[7][3]), .light7(green_array[6][3]), .light6(green_array[5][3]),
                   .light5(green_array[4][3]), .light4(green_array[3][3]), .light3(green_array[2][3]),
                   .light2(green_array[1][3]), .light1(green_array[0][3]));
normalColumn col5 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .enable2,
.ol8(green_array[7][3]), .ol7(green_array[6][3]), .ol6(green_array[5][3]), .ol5(green_array[4][3]),
.ol4(green_array[3][3]), .ol3(green_array[2][3]), .ol2(green_array[1][3]), .ol1(green_array[0][3]),
                   .light8(green_array[7][4]), .light7(green_array[6][4]), .light6(green_array[5][4]),
                   .light5(green_array[4][4]), .light4(green_array[3][4]), .light3(green_array[2][4]),
                   .light2(green_array[1][4]), .light1(green_array[0][4]));
normalColumn col6 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .enable2,
.ol8(green_array[7][4]), .ol7(green_array[6][4]), .ol6(green_array[5][4]), .ol5(green_array[4][4]),
.ol4(green_array[3][4]), .ol3(green_array[2][4]), .ol2(green_array[1][4]), .ol1(green_array[0][4]),
                   .light8(green_array[7][5]), .light7(green_array[6][5]), .light6(green_array[5][5]),
                   .light5(green_array[4][5]), .light4(green_array[3][5]), .light3(green_array[2][5]),
                   .light2(green_array[1][5]), .light1(green_array[0][5]));
normalColumn col7 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .enable2,
.ol8(green_array[7][5]), .ol7(green_array[6][5]), .ol6(green_array[5][5]), .ol5(green_array[4][5]),
.ol4(green_array[3][5]), .ol3(green_array[2][5]), .ol2(green_array[1][5]), .ol1(green_array[0][5]),
                   .light8(green_array[7][6]), .light7(green_array[6][6]), .light6(green_array[5][6]),
                   .light5(green_array[4][6]), .light4(green_array[3][6]), .light3(green_array[2][6]),
                   .light2(green_array[1][6]), .light1(green_array[0][6]));
normalColumn col8 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .pause, .enable2,
.ol8(green_array[7][6]), .ol7(green_array[6][6]), .ol6(green_array[5][6]), .ol5(green_array[4][6]),
.ol4(green_array[3][6]), .ol3(green_array[2][6]), .ol2(green_array[1][6]), .ol1(green_array[0][6]),
                   .light8(green_array[7][7]), .light7(green_array[6][7]), .light6(green_array[5][7]),
                   .light5(green_array[4][7]), .light4(green_array[3][7]), .light3(green_array[2][7]),
                   .light2(green_array[1][7]), .light1(green_array[0][7]));

// Check if the bird has ran into a barrier
collision dead2 (.red_array, .green_array, .dead);

// Keep track of the current game's score on the HEX display
score     s0 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .green_array, .enable(enable2),
.hex(hex0), .out(out0));
increment s1 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .in(out0),    .hex(hex1), .out(out1));
increment s2 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .in(out1),    .hex(hex2), .out(out2));
increment s3 (.clk(clk[whichClock]), .reset(reset | out_of_bounds | dead), .in(out2),    .hex(hex3), .out(out3));

// Display the high score when SW0 is on
highscore hi (.reset, .on(sw0), .hex0, .hex1, .hex2, .hex3, .out0(HEX0), .out1(HEX1), .out2(HEX2),
.out3(HEX3), .out4(HEX4), .out5(HEX5));

// Display a pause symbol in the display
pauseGame stop (.pause, .red_array, .green_array, .red_array1, .green_array1);

// Output which LEDs are supposed to be on
matrix_driver out (.clk(clk[whichClock]), .red_array(red_array1), .green_array(green_array1),
.red_driver(GPIO_0[27:20]), .green_driver(GPIO_0[35:28]), .row_sink(GPIO_0[19:12]));
```

```
endmodule

// Test the DE1_SoC module
module DE1_SoC_testbench();
   logic          clk;
   logic  [6:0]  HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   logic  [9:0]  LEDR;
   logic  [3:0]  KEY;
   logic  [9:0]  SW;
   logic  [35:0] GPIO_0;

   DE1_SoC dut (.CLOCK_50(clk), .HEX0, .HEX1, .HEX2, .HEX3, .HEX4, .HEX5, .KEY, .LEDR, .SW, .GPIO_0);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   // Setup the inputs to the design
   initial begin
                                                          repeat(10)  @(posedge clk);
      SW[9] <= 1;                                         repeat(40)  @(posedge clk);
      SW[9] <= 0; KEY[0] <= 1; SW[0] <= 0; SW[1] <= 0;    repeat(160) @(posedge clk);
                  KEY[0] <= 0;                            repeat(10)  @(posedge clk);
                  KEY[0] <= 1;             SW[1] <= 1;    repeat(100) @(posedge clk);
                                           SW[1] <= 0;    repeat(10)  @(posedge clk);
                  KEY[0] <= 0;                            repeat(40)  @(posedge clk);
                  KEY[0] <= 1; SW[0] <= 1;                repeat(100) @(posedge clk);
                               SW[0] <= 0;                repeat(10)  @(posedge clk);
                                                          repeat(250) @(posedge clk);
      $stop;
   end

endmodule


// Slows down the actual clock used by the circuit
// Barriers move left in this speed.
module counter2 (clk, reset, pause, enable);
   input  logic clk, reset, pause;
   output logic enable;
   logic [10:0] counter;

   always_ff @(posedge clk) begin
      if (reset)
         counter <= 11'b00000000000;
      else if (pause)
         counter <= counter;
      else
         counter <= counter + 11'b00000000001;
   end

   assign enable = (counter == 11'b00000000000);
endmodule

// Test the counter2 module
module counter2_testbench();
   logic clk, enable, reset, pause;

   counter2 dut (.clk, .reset, .pause, .enable);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   initial begin
        reset <= 1; pause <= 0;              @(posedge clk);
        reset <= 0;              repeat(5000) @(posedge clk);
                   pause <= 1;   repeat(5)    @(posedge clk);
                   pause <= 0;   repeat(5000) @(posedge clk);
      $stop;
   end

endmodule
```

```
// Checks if the bird goes out of the game's bounds
module bounds (clk, reset, key, gravity, top, bottom, out_of_bounds);
   input logic clk, reset, key, gravity, top, bottom;
   output logic out_of_bounds;

   // State variables
   enum { YES, NO } ps, ns;

   // Next state logic
   always_comb begin
      case (ps)
         YES:                                                          ns = NO;
         NO:  if ((bottom & ~key & gravity) | (top & key & ~gravity))  ns = YES;
              else                                                     ns = NO;
      endcase
   end

   // Output logic
   assign out_of_bounds = (ps == YES);

   // DFFs
   always_ff @(posedge clk) begin
      if (reset)
         ps <= NO;
      else
         ps <= ns;
   end

endmodule

// Test the bounds module
module bounds_testbench();
   logic clk, reset, key, gravity, top, bottom;
   logic out_of_bounds;

   bounds dut (.clk, .reset, .key, .gravity, .top, .bottom, .out_of_bounds);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   // Setup the inputs to the design
   initial begin
                                                                    @(posedge clk);
      reset <= 1;                                                   @(posedge clk);
                                                                    @(posedge clk);
      reset <= 0; key <= 0; gravity <= 0; top <= 0; bottom <= 0;    @(posedge clk);
                                                                    @(posedge clk);
                  key <= 1;               top <= 1;                 @(posedge clk);
                  key <= 0;                                         @(posedge clk);
                                                                    @(posedge clk);
                                          top <= 0;                 @(posedge clk);
                                                                    @(posedge clk);
                                                    bottom <= 1;    @(posedge clk);
                              gravity <= 1;                         @(posedge clk);
                              gravity <= 0;                         @(posedge clk);
                                                                    @(posedge clk);
      $stop;
   end

endmodule
```

```
// Display the 4-bit input as decimal on hex leds
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

// Test the 7 segment module
module seg7_testbench();
   logic [3:0] bcd;
   logic [6:0] leds;

   seg7 dut (.bcd, .leds);

   integer i;
   initial begin
      for(i = 0; i < 16; i++) begin
         bcd[3:0] = i; #10;
      end
   end
endmodule
```

```
// Module defining input to be true for only one clock cycle when the key is pressed
module useInput (clk, reset, keyin, keyout);
   input logic clk, reset, keyin;
   output logic keyout;

   // State variables
   enum { ZeroZero, ZeroOne } ps, ns;

   // Next state logic
   always_comb begin
      case (ps)
         ZeroZero: if (keyin)  ns = ZeroOne;
                   else        ns = ZeroZero;
         ZeroOne:  if (~keyin) ns = ZeroZero;
                   else        ns = ZeroOne;
      endcase
   end

   // Output logic
   assign keyout = (ps == ZeroZero) & (ns == ZeroOne);

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

endmodule

// Test the useInput module
module useInput_testbench();
   logic clk, reset, keyin;
   logic keyout;

   useInput dut (.clk, .reset, .keyin, .keyout);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   // Setup the inputs to the design
   initial begin
                              @(posedge clk);
      reset <= 1;             @(posedge clk);
                              @(posedge clk);
      reset <= 0; keyin <= 0; @(posedge clk);
                  keyin <= 1; @(posedge clk);
                  keyin <= 0; @(posedge clk);
                  keyin <= 1; @(posedge clk);
                              @(posedge clk);
                              @(posedge clk);
                  keyin <= 0; @(posedge clk);
                              @(posedge clk);
      $stop;
   end

endmodule
```

```
// Module defining if the middle LED should be on or off for the bird
module centerLight (clk, reset, pause, key, gravity, top, bottom, light);
   input logic clk, reset, pause, key, gravity, top, bottom;
   output logic light;

   // State variables
   enum { ON, OFF } ps, ns;

   // Next state logic
   always_comb begin
      case (ps)
         ON:  if ((key & ~gravity) | (~key & gravity))                 ns = OFF;
              else                                                     ns = ON;
         OFF: if ((bottom & key & ~gravity) | (top & ~key & gravity))  ns = ON;
              else                                                     ns = OFF;
      endcase
   end


   // Output logic
   assign light = (ps == ON);

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

endmodule

// Test the centerLight module
module centerLight_testbench();
   logic clk, reset, key, gravity, top, bottom;
   logic light;

   centerLight dut (.clk, .reset, .key, .gravity, .top, .bottom, .light);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   // Setup the inputs to the design
   initial begin
                                                                  @(posedge clk);
      reset <= 1;                                                 @(posedge clk);
                                                                  @(posedge clk);
      reset <= 0; key <= 1; gravity <= 1; top <= 0; bottom <= 0;  @(posedge clk);
                                                                  @(posedge clk);
                  key <= 0; gravity <= 0;                         @(posedge clk);
                                                                  @(posedge clk);
                  key <= 1;                                       @(posedge clk);
                                                                  @(posedge clk);
                  key <= 0; gravity <= 1;                         @(posedge clk);
                                                                  @(posedge clk);
                                    top <= 1;                     @(posedge clk);
                  key <= 1; gravity <= 0;                         @(posedge clk);
                                                                  @(posedge clk);
                  key <= 1;                       bottom <= 1;    @(posedge clk);
                                                                  @(posedge clk);
      $stop;
   end

endmodule
```

```
// Module defining if the LEDs, except for the middle, should be on or off
module normalLight (clk, reset, pause, key, gravity, top, bottom, light);
   input logic clk, reset, pause, key, gravity, top, bottom;
   output logic light;

   // State variables
   enum { ON, OFF } ps, ns;

   // Next state logic
   always_comb begin
      case (ps)
         ON:  if ((key & ~gravity) | (~key & gravity))                 ns = OFF;
              else                                                     ns = ON;
         OFF: if ((bottom & key & ~gravity) | (top & ~key & gravity))  ns = ON;
              else                                                     ns = OFF;
      endcase
   end

   // Output logic
   assign light = (ps == ON);

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

endmodule

// Test the normalLight module
module normalLight_testbench();
   logic clk, reset, pause, key, gravity, top, bottom;
   logic light;

   normalLight dut (.clk, .reset, .pause, .key, .gravity, .top, .bottom, .light);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   // Setup the inputs to the design
   initial begin
                                                                    @(posedge clk);
      reset <= 1;                                                   @(posedge clk);
                                                                    @(posedge clk);
      reset <= 0; key <= 0; gravity <= 0; top <= 0; bottom <= 0;    @(posedge clk);
                                                                    @(posedge clk);
                            gravity <= 1; top <= 1;                 @(posedge clk);
                            gravity <= 0;                           @(posedge clk);
                                                                    @(posedge clk);
                  key <= 1;                                         @(posedge clk);
                                                                    @(posedge clk);
                                                bottom <= 1;        @(posedge clk);
                            gravity <= 1;                           @(posedge clk);
                            gravity <= 0;                           @(posedge clk);
                                                                    @(posedge clk);
      $stop;
   end

endmodule
```

```
// Generate a random 4-bit number every clock cycle
module random (clk, reset, number);
   input logic clk, reset;
   output logic [3:0] number;

   // DFFs
   always @(posedge clk) begin
      if(reset)
         number <= 4'b0000;
      else  begin
         number[2:0] <= number[3:1];
         number[3]   <= (number[0] & number[1]) | (~number[0] & ~number[1]);
         end
      end

endmodule

// Test the random module
module random_testbench();
   logic clk, reset;
   logic [3:0] number;

   random dut (.clk, .reset, .number);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   // Setup the inputs to the design
   initial begin
                                 @(posedge clk);
      reset <= 1;                @(posedge clk);
      reset <= 0;    repeat(1025) @(posedge clk);
      $stop;
   end

endmodule

// Compare two 4-bit inputs and output true if the first one is greater than the second one
module compare (reset, hex0, hex1, hex2, hex3, hi0, hi1, hi2, hi3, out0, out1, out2, out3);
   input logic reset;
   input logic [3:0] hex0, hex1, hex2, hex3, hi0, hi1, hi2, hi3;
   output logic [3:0] out0, out1, out2, out3;

   always_comb begin
      if (reset) begin
         out3 = 1'b0000;
         out2 = 1'b0000;
         out1 = 1'b0000;
         out0 = 1'b0000;
      end
      else begin
         if((hex3 > hi3) |
           ((hex3 == hi3) & (hex2 > hi2)) |
           ((hex3 == hi3) & (hex2 == hi2) & (hex1 > hi1)) |
           ((hex3 == hi3) & (hex2 == hi2) & (hex1 == hi1) & (hex0 > hi0))) begin
            out3 = hex3;
            out2 = hex2;
            out1 = hex1;
            out0 = hex0;
         end
         else begin
            out3 = hi3;
            out2 = hi2;
            out1 = hi1;
            out0 = hi0;
         end
      end
   end

endmodule

// Test the compare module.
module compare_testbench();
   logic [3:0] hex0, hex1, hex2, hex3, hi0, hi1, hi2, hi3, out0, out1, out2, out3;
   logic reset;

   compare dut (.reset, .hex0, .hex1, .hex2, .hex3, .hi0, .hi1, .hi2, .hi3, .out0, .out1, .out2, .out3);

   initial begin
      hex0 = 4'b1001; hi0 = 4'b0101;
      hex1 = 4'b1001; hi1 = 4'b0000;
      hex2 = 4'b0000; hi2 = 4'b0000;
      hex3 = 4'b1111; hi3 = 4'b0000; #10;
      hex0 = 4'b1001; hi0 = 4'b1001;
      hex1 = 4'b1001; hi1 = 4'b0000;
      hex2 = 4'b0000; hi2 = 4'b0000;
      hex3 = 4'b1111; hi3 = 4'b0000; #10;
   end
endmodule
```

```
// Module displaying the LEDs on the 8x8 board row by row
module matrix_driver (clk, red_array, green_array, red_driver, green_driver, row_sink);
   input logic             clk;
   input logic  [7:0][7:0] red_array, green_array;
   output logic [7:0]      red_driver, green_driver, row_sink;
   logic        [2:0]      count;

   always_ff @(posedge clk)
      count <= count + 3'b001;

   // Display 1 row at a time
   always_comb begin
      case (count)
         3'b000: row_sink = 8'b11111110;
         3'b001: row_sink = 8'b11111101;
         3'b010: row_sink = 8'b11111011;
         3'b011: row_sink = 8'b11110111;
         3'b100: row_sink = 8'b11101111;
         3'b101: row_sink = 8'b11011111;
         3'b110: row_sink = 8'b10111111;
         3'b111: row_sink = 8'b01111111;
      endcase
   end

   assign red_driver = red_array[count];
   assign green_driver = green_array[count];
endmodule

// Test the matrix_driver module
module matrix_driver_testbench();
   logic             clk;
   logic [7:0][7:0] red_array, green_array;
   logic [7:0]      red_driver, green_driver, row_sink;

   matrix_driver dut (.clk, .red_array, .green_array, .red_driver, .green_driver, .row_sink);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   // Setup the inputs to the design
   initial begin

      red_array   <= { 8'b00000001, 8'b00000010, 8'b00000100, 8'b00001000, 8'b00010000, 8'b00010000,
      8'b00100000, 8'b01000000 };
      green_array <= { 8'b00000001, 8'b00000010, 8'b00000100, 8'b00001000, 8'b00010000, 8'b00010000,
      8'b00100000, 8'b01000000 }; repeat(100) @(posedge clk);
      $stop;
   end

endmodule
```

```
// Slows down the actual clock used by the circuit
// Simultes gravity that pulls down the bird.
module counter1 (clk, reset, pause, enable);
   input  logic clk, reset, pause;
   output logic enable;
   logic [9:0] counter;

   always_ff @(posedge clk) begin
      if (reset)
         counter <= 10'b0000000000;
      else if (pause)
         counter <= counter;
      else
         counter <= counter + 10'b0000000001;
   end

   assign enable = (counter == 10'b0000000000);
endmodule

// Test the counter1 module
module counter1_testbench();
   logic clk, enable, reset, pause;

   counter1 dut (.clk, .reset, .pause, .enable);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   initial begin
        reset <= 1; pause <= 0;               @(posedge clk);
        reset <= 0;                repeat(5000) @(posedge clk);
                    pause <= 1;    repeat(5)    @(posedge clk);
                    pause <= 0;    repeat(5000) @(posedge clk);
      $stop;
   end

endmodule
```

```
// Module defining the new barriers for the game
module startColumn (clk, reset, enable3, light8, light7, light6, light5, light4, light3, light2, light1);
   input logic clk, reset, enable3;
   output logic light8, light7, light6, light5, light4, light3, light2, light1;

   // Generate different barrier designs
   logic [15:0][7:0] barrier;
   assign barrier[0] [7:0] = 8'b00000000;
   assign barrier[1] [7:0] = 8'b00011111;
   assign barrier[2] [7:0] = 8'b10001111;
   assign barrier[3] [7:0] = 8'b11000111;
   assign barrier[4] [7:0] = 8'b11100011;
   assign barrier[5] [7:0] = 8'b11110001;
   assign barrier[6] [7:0] = 8'b11111000;
   assign barrier[7] [7:0] = 8'b11100111;
   assign barrier[8] [7:0] = 8'b11110011;
   assign barrier[9] [7:0] = 8'b11001111;
   assign barrier[10][7:0] = 8'b10001111;
   assign barrier[11][7:0] = 8'b11000111;
   assign barrier[12][7:0] = 8'b11100011;
   assign barrier[13][7:0] = 8'b11110001;
   assign barrier[14][7:0] = 8'b00001111;
   assign barrier[15][7:0] = 8'b00000111;

   // Pick a random number between 0 and 15
   logic [3:0] any;
   random num    (.clk, .reset, .number(any));

   logic local_enab;
   useInput enab (.clk, .reset, .keyin(enable3), .keyout(local_enab));

   // Pick a random barrier and send it through
   always_comb begin
      if (local_enab) begin
         light8 = barrier[any][7];
         light7 = barrier[any][6];
         light6 = barrier[any][5];
         light5 = barrier[any][4];
         light4 = barrier[any][3];
         light3 = barrier[any][2];
         light2 = barrier[any][1];
         light1 = barrier[any][0];
      end
      else begin
         light8 = barrier[0][7];
         light7 = barrier[0][6];
         light6 = barrier[0][5];
         light5 = barrier[0][4];
         light4 = barrier[0][3];
         light3 = barrier[0][2];
         light2 = barrier[0][1];
         light1 = barrier[0][0];
      end
   end

endmodule


// Test the startColumn module
module startColumn_testbench();
   logic clk, reset, enable3;
   logic light8, light7, light6, light5, light4, light3, light2, light1;

   startColumn dut (.clk, .reset, .enable3, .light8, .light7, .light6, .light5, .light4, .light3, .light2, .light1);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   // Setup the inputs to the design
   initial begin
                                    @(posedge clk);
      reset <= 1;                   @(posedge clk);
      reset <= 0; enable3 <= 0; repeat(5) @(posedge clk);
                  enable3 <= 1; repeat(5) @(posedge clk);
                  enable3 <= 0; repeat(5) @(posedge clk);
                  enable3 <= 1; repeat(5) @(posedge clk);
                  enable3 <= 0; repeat(5) @(posedge clk);
                  enable3 <= 1; repeat(5) @(posedge clk);
                  enable3 <= 0; repeat(5) @(posedge clk);
                  enable3 <= 1; repeat(5) @(posedge clk);
      $stop;
   end

endmodule
```

```
// Slows down the actual clock used by the circuit
// New barriers are generated in this frequency
module counter3 (clk, reset, pause, enable);
   input  logic clk, reset, pause;
   output logic enable;
   logic [12:0] counter;

   always_ff @(posedge clk) begin
      if (reset)
         counter <= 13'b0000000000000;
      else if (pause)
         counter <= counter;
      else
         counter <= counter + 13'b0000000000001;
   end

   assign enable = (counter == 13'b0000000000000);
endmodule

// Test the counter3 module
module counter3_testbench();
   logic clk, enable, reset, pause;

   counter3 dut (.clk, .reset, .pause, .enable);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   initial begin
        reset <= 1; pause <= 0;                @(posedge clk);
        reset <= 0;               repeat(5000) @(posedge clk);
                    pause <= 1;   repeat(5)    @(posedge clk);
                    pause <= 0;   repeat(5000) @(posedge clk);
      $stop;
   end

endmodule
```

```
// Module defining the led columns and moving barriers across the game
module normalcolumn (clk, reset, pause, enable2, ol8, ol7, ol6, ol5, ol4, ol3, ol2, ol1, light8, light7,
 light6, light5, light4, light3, light2, light1);
   input logic clk, reset, pause, enable2, ol8, ol7, ol6, ol5, ol4, ol3, ol2, ol1;
   output logic light8, light7, light6, light5, light4, light3, light2, light1;

   // DFFs
   always_ff @(posedge clk) begin
      if (reset) begin
         light8 <= 1'b0;
         light7 <= 1'b0;
         light6 <= 1'b0;
         light5 <= 1'b0;
         light4 <= 1'b0;
         light3 <= 1'b0;
         light2 <= 1'b0;
         light1 <= 1'b0;
      end
      else if (enable2 & ~pause) begin
         light8 <= ol8;
         light7 <= ol7;
         light6 <= ol6;
         light5 <= ol5;
         light4 <= ol4;
         light3 <= ol3;
         light2 <= ol2;
         light1 <= ol1;
      end
      else begin
         light8 <= light8;
         light7 <= light7;
         light6 <= light6;
         light5 <= light5;
         light4 <= light4;
         light3 <= light3;
         light2 <= light2;
         light1 <= light1;
      end
   end

endmodule

// Test the normalColumn module
module normalColumn_testbench();
   logic clk, reset, pause, enable2, ol8, ol7, ol6, ol5, ol4, ol3, ol2, ol1;
   logic light8, light7, light6, light5, light4, light3, light2, light1;

   normalColumn dut (.clk, .reset, .pause, .enable2, .ol8, .ol7, .ol6, .ol5, .ol4, .ol3, .ol2, .ol1,
   .light8, .light7, .light6, .light5, .light4, .light3, .light2, .light1);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   // Setup the inputs to the design
   initial begin

      reset <= 1;
      reset <= 0; enable2 <= 0; ol8 <= 0; ol8 <= 0; ol7 <= 0; ol6 <= 0; ol5 <= 0; ol4 <= 0; ol3 <= 0;
      ol2 <= 0; ol1 <= 0; repeat(5) @(posedge clk);
                  enable2 <= 1;
                  repeat(5) @(posedge clk);
                  enable2 <= 0; ol8 <= 0; ol8 <= 1; ol7 <= 1; ol6 <= 0; ol5 <= 1; ol4 <= 1; ol3 <= 0;
                  ol2 <= 1; ol1 <= 1; repeat(5) @(posedge clk);
                  enable2 <= 1;
                  repeat(5) @(posedge clk);
                  enable2 <= 0; ol8 <= 0; ol8 <= 0; ol7 <= 0; ol6 <= 0; ol5 <= 0; ol4 <= 0; ol3 <= 0;
                  ol2 <= 0; ol1 <= 0; repeat(5) @(posedge clk);
                  enable2 <= 1;
                  repeat(5) @(posedge clk);
                  enable2 <= 0; ol8 <= 0; ol8 <= 1; ol7 <= 1; ol6 <= 1; ol5 <= 1; ol4 <= 1; ol3 <= 1;
                  ol2 <= 1; ol1 <= 1; repeat(5) @(posedge clk);
                  enable2 <= 1;
                  repeat(5) @(posedge clk);
      $stop;
   end

endmodule
```

```
// Checks if the bird flys into a barrier
module collision (red_array, green_array, dead);
   input logic [7:0][7:0] red_array, green_array;
   output logic dead;

   assign dead = ((red_array[7][6] & green_array[7][6]) | (red_array[6][6] & green_array[6][6])
                | (red_array[5][6] & green_array[5][6]) | (red_array[4][6] & green_array[4][6])
                | (red_array[3][6] & green_array[3][6]) | (red_array[2][6] & green_array[2][6])
                | (red_array[1][6] & green_array[1][6]) | (red_array[0][6] & green_array[0][6]));

endmodule

// Test the collision module
module collision_testbench();
   logic [7:0][7:0] red_array, green_array;
   logic dead;

   collision dut (.red_array, .green_array, .dead);

   initial begin
      red_array[7][6] = 0; green_array[7][6] = 1; #10;
      red_array[7][6] = 1;                        #10;
      red_array[6][6] = 0; green_array[6][6] = 1; #10;
      red_array[6][6] = 1;                        #10;
      red_array[5][6] = 0; green_array[5][6] = 1; #10;
      red_array[5][6] = 1;                        #10;
      red_array[4][6] = 0; green_array[4][6] = 1; #10;
      red_array[4][6] = 1;                        #10;
      red_array[3][6] = 0; green_array[3][6] = 1; #10;
      red_array[3][6] = 1;                        #10;
      red_array[2][6] = 0; green_array[2][6] = 1; #10;
      red_array[2][6] = 1;                        #10;
      red_array[1][6] = 0; green_array[1][6] = 1; #10;
      red_array[1][6] = 1;                        #10;
      red_array[0][6] = 0; green_array[0][6] = 1; #10;
      red_array[0][6] = 1;                        #10;
   end

endmodule
```

```
// Module controlling first HEX digit which displays the score
module score (clk, reset, green_array, enable, hex, out);
   input logic clk, reset, enable;
   input logic [7:0][7:0] green_array;
   output logic out;
   output logic [3:0] hex;
   logic [3:0] counter;

   // Output logic
   assign hex = counter;
   assign out = (enable & (counter == 4'b1001) &   (green_array[7][6] | green_array[6][6] |
                                                    green_array[5][6] | green_array[4][6] |
                                                    green_array[3][6] | green_array[2][6] |
                                                    green_array[1][6] | green_array[0][6]));

   // DFFs
   always_ff @(posedge clk) begin
      if (reset | (enable & (counter == 4'b1001) & (green_array[7][6] | green_array[6][6] |
                                                    green_array[5][6] | green_array[4][6] |
                                                    green_array[3][6] | green_array[2][6] |
                                                    green_array[1][6] | green_array[0][6])))
         counter <= 4'b0000;
      else if (enable &                            (green_array[7][6] | green_array[6][6] |
                                                    green_array[5][6] | green_array[4][6] |
                                                    green_array[3][6] | green_array[2][6] |
                                                    green_array[1][6] | green_array[0][6]))
         counter <= counter + 4'b0001;
      else
         counter <= counter;
   end

endmodule

// Test the score module
module score_testbench();
   logic clk, reset, enable;
   logic [7:0][7:0] green_array;
   logic out;
   logic [3:0] hex;

   score dut (.clk, .reset, .green_array, .enable, .hex, .out);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   // Setup the inputs to the design
   initial begin

      reset <= 1;
      reset <= 0; enable <= 0;
      green_array <= { 8'b00000000, 8'b00000000, 8'b00000000, 8'b00000000, 8'b00000000, 8'b00000000,
      8'b00000000, 8'b00000000 }; @(posedge clk);
                  enable <= 1; green_array[6][6] <= 1;
                  repeat(20) @(posedge clk);

      $stop;
   end

endmodule
```

```
// Module controlling one digit HEX which displays the score
module increment (clk, reset, in, hex, out);
   input logic clk, reset, in;
   output logic out;
   output logic [3:0] hex;
   logic [3:0] counter;

   // Output logic
   assign hex = counter;
   assign out = ((counter == 4'b1001) & in);

   logic local_in;
   useInput loc_in (.clk, .reset, .keyin(in), .keyout(local_in));

   // DFFs
   always_ff @(posedge clk) begin
      if (reset | ((counter == 4'b1001) & in ))
         counter <= 4'b0000;
      else if (local_in)
         counter <= counter + 4'b0001;
      else
         counter <= counter;
   end

endmodule

// Test the increment module
module increment_testbench();
   logic clk, reset, in;
   logic out;
   logic [3:0] hex;

   increment dut (.clk, .reset, .in, .hex, .out);

   // Setup the test clock
   parameter CLOCK_PERIOD=100;
   initial begin
      clk <= 0;
      forever #(CLOCK_PERIOD/2) clk <= ~clk;
   end

   // Setup the inputs to the design
   initial begin
                                                    @(posedge clk);
      reset <= 1;                                   @(posedge clk);
      reset <= 0; in <= 0;          repeat(5) @(posedge clk);
                  in <= 1;          repeat(5) @(posedge clk);
                  in <= 0;                          @(posedge clk);
                  in <= 1;                          @(posedge clk);
                  in <= 0;                          @(posedge clk);
                  in <= 1;                          @(posedge clk);
                  in <= 0;                          @(posedge clk);
                  in <= 1;                          @(posedge clk);
                  in <= 0;                          @(posedge clk);
                  in <= 1;                          @(posedge clk);
                  in <= 0;                          @(posedge clk);
                  in <= 1;                          @(posedge clk);
                  in <= 0;                          @(posedge clk);
                  in <= 1;                          @(posedge clk);
                  in <= 0;                          @(posedge clk);
                  in <= 1;                          @(posedge clk);
                  in <= 0;                          @(posedge clk);
                  in <= 1;                          @(posedge clk);
                  in <= 0;                          @(posedge clk);
                  in <= 1;                          @(posedge clk);
                  in <= 0;                          @(posedge clk);
                  in <= 1;                          @(posedge clk);
      $stop;
   end

endmodule
```

```
// Module displaying the high score user has achieved so far
module highscore (reset, on, hex0, hex1, hex2, hex3, out0, out1, out2, out3, out4, out5);
   input logic reset, on;
   input logic  [3:0] hex0, hex1, hex2, hex3;
   output logic [6:0] out0, out1, out2, out3, out4, out5;
   logic [3:0] hi0, hi1, hi2, hi3, pick0, pick1, pick2, pick3;

   compare com0 (.reset, .hex0, .hex1, .hex2, .hex3, .hi0, .hi1, .hi2, .hi3, .out0(hi0), .out1(hi1),
   .out2(hi2), .out3(hi3));

   always_comb begin
      if (on) begin
         pick0 = hi0;
         pick1 = hi1;
         pick2 = hi2;
         pick3 = hi3;
         out4 = 7'b1111001;
         out5 = 7'b0001001;
      end
      else begin
         pick0 = hex0;
         pick1 = hex1;
         pick2 = hex2;
         pick3 = hex3;
         out4  = 7'b1111111;
         out5  = 7'b1111111;
      end
   end

   // Output the score to the HEXs
   seg7 seg0 (.bcd(pick0), .leds(out0));
   seg7 seg1 (.bcd(pick1), .leds(out1));
   seg7 seg2 (.bcd(pick2), .leds(out2));
   seg7 seg3 (.bcd(pick3), .leds(out3));

endmodule

// Test the highscore module
module highscore_testbench();
   logic reset, on;
   logic [3:0] hex0, hex1, hex2, hex3;
   logic [6:0] out0, out1, out2, out3, out4, out5;

   highscore dut (.reset, .on, .hex0, .hex1, .hex2, .hex3, .out0, .out1, .out2, .out3, .out4, .out5);

   integer i;
   initial begin
      for(i = 0; i < 65536; i++) begin
         { hex0[3:0], hex1[3:0], hex2[3:0], hex3[3:0] } = i; on = 0; #10;
                                                             on = 1; #10;
      end
   end
endmodule
```

```
// Module displaying a pause symbol on the LED screen when the game is paused
module pauseGame (pause, red_array, green_array, red_array1, green_array1);
   input  logic pause;
   input  logic [7:0][7:0] red_array,  green_array;
   output logic [7:0][7:0] red_array1, green_array1;
   logic        [7:0][7:0] red_array2, green_array2;

   assign red_array2[0]   = 8'b00000000;
   assign red_array2[1]   = 8'b00011011;
   assign red_array2[2]   = 8'b00011011;
   assign red_array2[3]   = 8'b00011011;
   assign red_array2[4]   = 8'b00011011;
   assign red_array2[5]   = 8'b00011011;
   assign red_array2[6]   = 8'b00011011;
   assign red_array2[7]   = 8'b00000000;
   assign green_array2[0] = 8'b00000000;
   assign green_array2[1] = 8'b00011011;
   assign green_array2[2] = 8'b00011011;
   assign green_array2[3] = 8'b00011011;
   assign green_array2[4] = 8'b00011011;
   assign green_array2[5] = 8'b00011011;
   assign green_array2[6] = 8'b00011011;
   assign green_array2[7] = 8'b00000000;

   always_comb begin
      if (pause) begin
         red_array1   = red_array2;
         green_array1 = green_array2;
      end
      else begin
         red_array1   = red_array;
         green_array1 = green_array;
      end
   end

endmodule

// Test the pauseGame module
module pauseGame_testbench();
   logic pause;
   logic [7:0][7:0] red_array,  green_array;
   logic [7:0][7:0] red_array1, green_array1;

   pauseGame (.pause, .red_array, .green_array, .red_array1, .green_array1);

   integer row, col;
   initial begin
      for(row = 0; row < 8; row++) begin
         for (col = 0; col < 256; col++) begin
            red_array[row] = col; green_array[row] = col; #10;
         end
      end
   end

endmodule
```

d) Resource Utilization by Entity

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers
1	⌄ |DE1_SoC	252 (6)	156 (6)
1	|bounds:dead1|	1 (1)	1 (1)
2	|centerLight:bird5|	2 (2)	1 (1)
3	|clock_divider:cdiv|	15 (15)	15 (15)
4	|collision:dead2|	2 (2)	0 (0)
5	|counter1:countone|	13 (13)	10 (10)
6	|counter2:counttwo|	13 (13)	11 (11)
7	|counter3:countthree|	13 (13)	13 (13)
8	⌄ |highscore:hi|	77 (16)	0 (0)
1	|compare:com0|	33 (33)	0 (0)
2	|seg7:seg0|	7 (7)	0 (0)
3	|seg7:seg1|	7 (7)	0 (0)
4	|seg7:seg2|	7 (7)	0 (0)
5	|seg7:seg3|	7 (7)	0 (0)
9	⌄ |increment:s1|	8 (7)	5 (4)
1	|useInput:loc_in|	1 (1)	1 (1)
10	⌄ |increment:s2|	8 (7)	5 (4)
1	|useInput:loc_in|	1 (1)	1 (1)
11	⌄ |increment:s3|	7 (6)	5 (4)
1	|useInput:loc_in|	1 (1)	1 (1)
12	|matrix_driver:out|	40 (40)	3 (3)
13	|normalColumn:col1|	11 (11)	8 (8)
14	|normalColumn:col2|	0 (0)	8 (8)
15	|normalColumn:col3|	0 (0)	8 (8)
16	|normalColumn:col4|	0 (0)	8 (8)
17	|normalColumn:col5|	0 (0)	8 (8)