```
// Morris Huang, Hudson Wong
// 06/05/2025
// EE 271
// Lab 6 DE1_SoC

// This module simulates a game of frogger, where the player attempts to cross a
// road full of crossing cars. The game is simulated on an LED matrix, and if the player
// runs into the car, they lose. If they reach the end, they win, and the cars spawn and
move faster.
// There are eight difficulties in total, and reset will reset the difficulty to 0.
module DE1_SoC (CLOCK_50,HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, GPIO_1, SW);

   input logic CLOCK_50;
   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   output logic [35:0] GPIO_1;
   output logic [9:0] LEDR;
   input logic [3:0] KEY;
   input logic [9:0] SW;

   logic [31:0] clk;

   // Initializes two clock parameters. The whichClock controls the player movements,
   // while crossyClock controls the car movements. This makes it so that the player can move
   // faster than the cars.
   parameter whichClock = 4;
   parameter crossyClock =25;


   // Initializes the clock divider and the LED matrix
   clock_divider cdiv (CLOCK_50, clk);
   logic [15:0][15:0] RedPixels; // 16x16 array of red LEDs
   logic [15:0][15:0] GrnPixels;

   // Cycles through the LED matrix to light up each pixel
   LEDDriver(.GPIO_1(GPIO_1), .RedPixels(RedPixels), .GrnPixels(GrnPixels), .EnableCount(
1'b1), .CLK(clk[5]), .RST(SW[0]));

   logic reset_playfield, reset, L, R, U, D;  // configure reset, the player moves, and the
gameover logic
   logic [35:0] gameover; // Initializes gameover as a 36 bit array to take in the inputs
from all of the LEDs in the 6x6 block
   logic [2:0] difficulty;
   assign reset = SW[9]; // Reset when SW[9] is toggled


   // Sends the input to a user input module to make sure that a long button
   // press only registers as one input. Also implements a pair of flip flops
   // for metastability.
   // Key 0 corresponds to the right movement, key 1 to up, key 2 to down, and key 3 to
left.
   user_input right(.clk(clk[whichClock]), .reset(reset), .button(~KEY[0]), .out(R));
   user_input left(.clk(clk[whichClock]), .reset(reset), .button(~KEY[3]), .out(L));
   user_input down(.clk(clk[whichClock]), .reset(reset), .button(~KEY[2]), .out(D));
   user_input up(.clk(clk[whichClock]), .reset(reset), .button(~KEY[1]), .out(U));

   // Initializes a 6x6 playing field where the player can move freely.
   // Whenever the player moves right after reaching the rightmost edge, transfers
   // them to the leftmost edge, and same for the left side. They cannot move down
   // after reaching the bottom, but the frogger disappears after reaching the top and
pressing up
   frog_LED f1010(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[10][11]), .
NR(GrnPixels[10][15]), .ND(GrnPixels[11][10]), .NU(1'b0), .L(L), .R(R), .D(D), .U(U),  .
green_LED(GrnPixels[10][10]), .red_LED(RedPixels[10][10]), .gameover(gameover[0]));
   frog_LED f1011(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[10][12]), .
NR(GrnPixels[10][10]), .ND(GrnPixels[11][11]), .NU(1'b0), .L(L), .R(R), .D(D), .U(U),  .
green_LED(GrnPixels[10][11]), .red_LED(RedPixels[10][11]), .gameover(gameover[1]));
   frog_LED f1012(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[10][13]), .
NR(GrnPixels[10][11]), .ND(GrnPixels[11][12]), .NU(1'b0), .L(L), .R(R), .D(D), .U(U),  .
green_LED(GrnPixels[10][12]), .red_LED(RedPixels[10][12]), .gameover(gameover[2]));
   frog_LED f1013(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[10][14]), .
NR(GrnPixels[10][12]), .ND(GrnPixels[11][13]), .NU(1'b0), .L(L), .R(R), .D(D), .U(U),  .
green_LED(GrnPixels[10][13]), .red_LED(RedPixels[10][13]), .gameover(gameover[3]));
   frog_LED f1014(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[10][15]), .
NR(GrnPixels[10][13]), .ND(GrnPixels[11][14]), .NU(1'b0), .L(L), .R(R), .D(D), .U(U),  .
green_LED(GrnPixels[10][14]), .red_LED(RedPixels[10][14]), .gameover(gameover[4]));
   frog_LED f1015(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[10][10]), .
NR(GrnPixels[10][14]), .ND(GrnPixels[11][15]), .NU(1'b0), .L(L), .R(R), .D(D), .U(U),  .green_LED(GrnPixels[10][15]), .red_LED(RedPixels[10][15]), .gameover(gameover[5]));

    frog_LED f1110(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[11][11]), .NR(GrnPixels[11][15]), .ND(GrnPixels[12][10]), .NU(GrnPixels[10][10]), .L(L), .R(R), .D(D), .U(U),  .green_LED(GrnPixels[11][10]), .red_LED(RedPixels[11][10]), .gameover(gameover[6]));
    frog_LED f1111(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[11][12]), .NR(GrnPixels[11][10]), .ND(GrnPixels[12][11]), .NU(GrnPixels[10][11]), .L(L), .R(R), .D(D), .U(U),  .green_LED(GrnPixels[11][11]), .red_LED(RedPixels[11][11]), .gameover(gameover[7]));
    frog_LED f1112(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[11][13]), .NR(GrnPixels[11][11]), .ND(GrnPixels[12][12]), .NU(GrnPixels[10][12]), .L(L), .R(R), .D(D), .U(U),  .green_LED(GrnPixels[11][12]), .red_LED(RedPixels[11][12]), .gameover(gameover[8]));
    frog_LED f1113(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[11][14]), .NR(GrnPixels[11][12]), .ND(GrnPixels[12][13]), .NU(GrnPixels[10][13]), .L(L), .R(R), .D(D), .U(U),  .green_LED(GrnPixels[11][13]), .red_LED(RedPixels[11][13]), .gameover(gameover[9]));
    frog_LED f1114(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[11][15]), .NR(GrnPixels[11][13]), .ND(GrnPixels[12][14]), .NU(GrnPixels[10][14]), .L(L), .R(R), .D(D), .U(U),  .green_LED(GrnPixels[11][14]), .red_LED(RedPixels[11][14]), .gameover(gameover[10]));
    frog_LED f1115(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[11][10]), .NR(GrnPixels[11][14]), .ND(GrnPixels[12][15]), .NU(GrnPixels[10][15]), .L(L), .R(R), .D(D), .U(U),  .green_LED(GrnPixels[11][15]), .red_LED(RedPixels[11][15]), .gameover(gameover[11]));

    frog_LED_start f1515(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[15][10]), .NR(GrnPixels[15][14]), .ND(1'b0), .NU(GrnPixels[14][15] || GrnPixels[15][15]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[15][15]), .red_LED(RedPixels[15][15]), .gameover(gameover[12]));
    frog_LED f1514(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[15][15]), .NR(GrnPixels[15][13]), .ND(1'b0), .NU(GrnPixels[14][14] || GrnPixels[15][14]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[15][14]), .red_LED(RedPixels[15][14]), .gameover(gameover[13]));
    frog_LED f1513(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[15][14]), .NR(GrnPixels[15][12]), .ND(1'b0), .NU(GrnPixels[14][13] || GrnPixels[15][13]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[15][13]), .red_LED(RedPixels[15][13]), .gameover(gameover[14]));
    frog_LED f1512(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[15][13]), .NR(GrnPixels[15][11]), .ND(1'b0), .NU(GrnPixels[14][12] || GrnPixels[15][12]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[15][12]), .red_LED(RedPixels[15][12]), .gameover(gameover[15]));
    frog_LED f1511(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[15][12]), .NR(GrnPixels[15][10]), .ND(1'b0), .NU(GrnPixels[14][11] || GrnPixels[15][11]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[15][11]), .red_LED(RedPixels[15][11]), .gameover(gameover[16]));
    frog_LED f1510(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[15][11]), .NR(GrnPixels[15][15]), .ND(1'b0), .NU(GrnPixels[14][10] || GrnPixels[15][10]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[15][10]), .red_LED(RedPixels[15][10]), .gameover(gameover[17]));

    frog_LED f1415(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[14][10]), .NR(GrnPixels[14][14]), .ND(GrnPixels[15][15]), .NU(GrnPixels[13][15]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[14][15]), .red_LED(RedPixels[14][15]), .gameover(gameover[18]));
    frog_LED f1414(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[14][15]), .NR(GrnPixels[14][13]), .ND(GrnPixels[15][14]), .NU(GrnPixels[13][14]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[14][14]), .red_LED(RedPixels[14][14]), .gameover(gameover[19]));
    frog_LED f1413(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[14][14]), .NR(GrnPixels[14][12]), .ND(GrnPixels[15][13]), .NU(GrnPixels[13][13]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[14][13]), .red_LED(RedPixels[14][13]), .gameover(gameover[20]));
    frog_LED f1412(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[14][13]), .NR(GrnPixels[14][11]), .ND(GrnPixels[15][12]), .NU(GrnPixels[13][12]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[14][12]), .red_LED(RedPixels[14][12]), .gameover(gameover[21]));
    frog_LED f1411(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[14][12]), .NR(GrnPixels[14][10]), .ND(GrnPixels[15][11]), .NU(GrnPixels[13][11]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[14][11]), .red_LED(RedPixels[14][11]), .gameover(gameover[22]));
    frog_LED f1410(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[14][11]), .NR(GrnPixels[14][15]), .ND(GrnPixels[15][10]), .NU(GrnPixels[13][10]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[14][10]), .red_LED(RedPixels[14][10]), .gameover(gameover[23]));

    frog_LED f1315(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[13][10]), .NR(GrnPixels[13][14]), .ND(GrnPixels[14][15]), .NU(GrnPixels[12][15]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[13][15]), .red_LED(RedPixels[13][15]), .gameover(gameover[24]));
    frog_LED f1314(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[13][15]), .NR(GrnPixels[13][13]), .ND(GrnPixels[14][14]), .NU(GrnPixels[12][14]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[13][14]), .red_LED(RedPixels[13][14]), .gameover(gameover[25]));
    frog_LED f1313(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[13][14]), .NR(GrnPixels[13][12]), .ND(GrnPixels[14][13]), .NU(GrnPixels[12][13]), .L(L), .R(R), .D(D), .U(U), .green_LED(GrnPixels[13][13]), .red_LED(RedPixels[13][13]), .gameover(gameover[26]));
    frog_LED f1312(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[13][13]), .
NR(GrnPixels[13][11]), .ND(GrnPixels[14][12]), .NU(GrnPixels[12][12]), .L(L), .R(R), .D(D), 
.U(U), .green_LED(GrnPixels[13][12]), .red_LED(RedPixels[13][12]), .gameover(gameover[27]));
   frog_LED f1311(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[13][12]), .
NR(GrnPixels[13][10]), .ND(GrnPixels[14][11]), .NU(GrnPixels[12][11]), .L(L), .R(R), .D(D), 
.U(U), .green_LED(GrnPixels[13][11]), .red_LED(RedPixels[13][11]), .gameover(gameover[28]));
   frog_LED f1310(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[13][11]), .
NR(GrnPixels[13][15]), .ND(GrnPixels[14][10]), .NU(GrnPixels[12][10]), .L(L), .R(R), .D(D), 
.U(U), .green_LED(GrnPixels[13][10]), .red_LED(RedPixels[13][10]), .gameover(gameover[29]));

   frog_LED f1215(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[12][10]), .
NR(GrnPixels[12][14]), .ND(GrnPixels[13][15]), .NU(GrnPixels[11][15]), .L(L), .R(R), .D(D), 
.U(U), .green_LED(GrnPixels[12][15]), .red_LED(RedPixels[12][15]), .gameover(gameover[30]));
   frog_LED f1214(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[12][15]), .
NR(GrnPixels[12][13]), .ND(GrnPixels[13][14]), .NU(GrnPixels[11][14]), .L(L), .R(R), .D(D), 
.U(U), .green_LED(GrnPixels[12][14]), .red_LED(RedPixels[12][14]), .gameover(gameover[31]));
   frog_LED f1213(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[12][14]), .
NR(GrnPixels[12][12]), .ND(GrnPixels[13][13]), .NU(GrnPixels[11][13]), .L(L), .R(R), .D(D), 
.U(U), .green_LED(GrnPixels[12][13]), .red_LED(RedPixels[12][13]), .gameover(gameover[32]));
   frog_LED f1212(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[12][13]), .
NR(GrnPixels[12][11]), .ND(GrnPixels[13][12]), .NU(GrnPixels[11][12]), .L(L), .R(R), .D(D), 
.U(U), .green_LED(GrnPixels[12][12]), .red_LED(RedPixels[12][12]), .gameover(gameover[33]));
   frog_LED f1211(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[12][12]), .
NR(GrnPixels[12][10]), .ND(GrnPixels[13][11]), .NU(GrnPixels[11][11]), .L(L), .R(R), .D(D), 
.U(U), .green_LED(GrnPixels[12][11]), .red_LED(RedPixels[12][11]), .gameover(gameover[34]));
   frog_LED f1210(.clk(clk[whichClock]), .reset(reset_playfield), .NL(GrnPixels[12][11]), .
NR(GrnPixels[12][15]), .ND(GrnPixels[13][10]), .NU(GrnPixels[11][10]), .L(L), .R(R), .D(D), 
.U(U), .green_LED(GrnPixels[12][10]), .red_LED(RedPixels[12][10]), .gameover(gameover[35]));

   // Initializes the logic to randomize the car movement
   logic random1, random2, random3, frogger_car1, frogger_car2, frogger_car3;
   logic [9:0] random_cross1, random_cross2, random_cross3;

   // Uses a seeded_LFSR to ensure that the cars spawn randomly
   seeded_LFSR random_move1(.clk(clk[crossyClock - difficulty]), .reset(reset), .out(
random_cross1), .seed(10'b0000000000));
   seeded_LFSR random_move2(.clk(clk[crossyClock - difficulty]), .reset(reset), .out(
random_cross2), .seed(10'b0000011111));
   seeded_LFSR random_move3(.clk(clk[crossyClock - difficulty]), .reset(reset), .out(
random_cross3), .seed(10'b0101010101));

   // Uses a comparator to turn the 10 bit number from the LFSR into a singular output
   comparator rand1(.inputA(random_cross1), .inputB(10'b0111111111), .A_greater_B(random1), 
.clk(clk[crossyClock - difficulty]), .reset(reset));
   comparator rand2(.inputA(random_cross2), .inputB(10'b0111111111), .A_greater_B(random2), 
.clk(clk[crossyClock - difficulty]), .reset(reset));
   comparator rand3(.inputA(random_cross3), .inputB(10'b0111111111), .A_greater_B(random3), 
.clk(clk[crossyClock - difficulty]), .reset(reset));

   // Ensures that the car only spawns once if the output from comparator is true for 
multiple clock cycles
   user_input car_move1(.clk(clk[crossyClock - difficulty]), .reset(reset), .button(random1
), .out(frogger_car1));
   user_input car_move2(.clk(clk[crossyClock - difficulty]), .reset(reset), .button(random2
), .out(frogger_car2));
   user_input car_move3(.clk(clk[crossyClock - difficulty]), .reset(reset), .button(random3
), .out(frogger_car3));

   // Initializes the logic for the cars. Each car moves to the right every clock cycle until
   // it reaches the edge and disappears. Uses the LFSR to randomly determine when the
   // cars spawn. The cars are represented by red LEDs. They spawn on the 2nd, 4th, and 5th 
row
   crossy_road c1415(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
frogger_car1), .red_LED(RedPixels[14][15]));
   crossy_road c1414(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[14][15]), .red_LED(RedPixels[14][14]));
   crossy_road c1413(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[14][14]), .red_LED(RedPixels[14][13]));
   crossy_road c1412(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[14][13]), .red_LED(RedPixels[14][12]));
   crossy_road c1411(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[14][12]), .red_LED(RedPixels[14][11]));
   crossy_road c1410(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
RedPixels[14][11]), .red_LED(RedPixels[14][10]));


   crossy_road c1215(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(
frogger_car2), .red_LED(RedPixels[12][15]));
   crossy_road c1214(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(RedPixels[12][15]), .red_LED(RedPixels[12][14]));
   crossy_road c1213(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(RedPixels[12][14]), .red_LED(RedPixels[12][13]));
   crossy_road c1212(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(RedPixels[12][13]), .red_LED(RedPixels[12][12]));
   crossy_road c1211(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(RedPixels[12][12]), .red_LED(RedPixels[12][11]));
   crossy_road c1210(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(RedPixels[12][11]), .red_LED(RedPixels[12][10]));


   crossy_road c1015(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(frogger_car3), .red_LED(RedPixels[10][15]));
   crossy_road c1014(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(RedPixels[10][15]), .red_LED(RedPixels[10][14]));
   crossy_road c1013(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(RedPixels[10][14]), .red_LED(RedPixels[10][13]));
   crossy_road c1012(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(RedPixels[10][13]), .red_LED(RedPixels[10][12]));
   crossy_road c1011(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(RedPixels[10][12]), .red_LED(RedPixels[10][11]));
   crossy_road c1010(.clk(clk[crossyClock - difficulty]), .reset(reset_playfield), .NL(RedPixels[10][11]), .red_LED(RedPixels[10][10]));



   victory v0(.clk(clk[whichClock]), .reset(reset),.TopLED(GrnPixels[10][10] || GrnPixels[10][11] || GrnPixels[10][12] || GrnPixels[10][13] || GrnPixels[10][14] || GrnPixels[10][15]), .UpKey(U), .DownKey(D), .gameover(|gameover), .win(win), .lose(lose), .reset_playfield(reset_playfield), .difficulty(difficulty));

endmodule

// This testbench verifies the behavior of the DE1_SoC module, ensuring that the
// module instantiates a board that can run a game of Frogger, randomize the car movement, and so on.
// The parameters whichClock and crossyClock must be set to zero for this testbench to function, and the LED
// Driver must not be initialized.
module DE1_SoC_testbench();
   logic CLOCK_50;
   logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   logic [35:0] GPIO_1;
   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, GPIO_1, SW);


   parameter clock_period = 100;

    // toggle CLOCK_50 every half cycle
   initial begin
      CLOCK_50 <= 0;
      forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
   end


   initial begin
      KEY[0] <= 0; KEY[1] <= 0; KEY[2] <= 0; KEY[3] <= 0; SW[9] <= 0; @(posedge CLOCK_50); // Sets all relevant variables to 0.

      SW[9] <= 1; @(posedge CLOCK_50);
      SW[9] <= 0; @(posedge CLOCK_50); //Toggle reset

      repeat(20) @(posedge CLOCK_50); //Tests the car functionality
      KEY[2] <= 1; KEY[1] <= 1; repeat(10) @(posedge CLOCK_50); // Tests that opposite keys do not move the frog
      KEY[2] <= 0; KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);
      KEY[0] <= 1; KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
      KEY[0] <= 0; KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);

```

```
        KEY[0] <= 1; repeat(10) @(posedge CLOCK_50); // Tests that each of the movement keys works
        KEY[0] <= 0; repeat(10)@(posedge CLOCK_50);
        KEY[1] <= 1; repeat(10)@(posedge CLOCK_50);
        KEY[1] <= 0; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
        KEY[3] <= 1; repeat(10)@(posedge CLOCK_50);
        KEY[3] <= 0; repeat(10)@(posedge CLOCK_50);

        KEY[3] <= 1; repeat(10)@(posedge CLOCK_50); // Frog moves and wins
        KEY[3] <= 0; repeat(10)@(posedge CLOCK_50);
        KEY[3] <= 1; repeat(10)@(posedge CLOCK_50);
        KEY[3] <= 0; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 1; repeat(10)@(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);

        // Random moves to ensure everything works
        KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[1] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[1] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[0] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[0] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[2] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[2] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);

        KEY[3] <= 1; repeat(10) @(posedge CLOCK_50);
        KEY[3] <= 0; repeat(10) @(posedge CLOCK_50);

        SW[9] <= 1; repeat(10)@(posedge CLOCK_50); // Toggle reset
        SW[9] <= 0; @(posedge CLOCK_50);

        KEY[2] <= 1; repeat(10)@(posedge CLOCK_50); // Frog runs into a car and resets
        KEY[2] <= 0; repeat(10)@(posedge CLOCK_50);
        repeat(20)    @(posedge CLOCK_50);

      $stop;
    end
endmodule
```

```
// Morris Huang, Hudson Wong
// 06/04/2025
// EE 271
// Victory Module

// This module determines when the player wins by reaching the top of the field or
// loses when it collides with a car. In either case, it resets the playfield.
module victory (clk, reset, UpKey,DownKey, win, lose, reset_playfield, gameover, TopLED,
difficulty);
    input logic clk, reset, UpKey, DownKey, gameover, TopLED;
    output logic win, lose;
    output logic reset_playfield;
    output logic [2:0] difficulty;

    assign win = (TopLED & UpKey & ~DownKey);
    assign lose = gameover;
    three_bit_counter counter(.clk(clk), .reset(reset), .in(win), .out(difficulty));

   // Manage reset_playfield signal. If the player wins or loses or if the reset switch is
toggled,
   // it resets the field to its default state.
    always_ff @(posedge clk) begin
        if (reset)
            reset_playfield <= 1;
        else if (win || lose)
            reset_playfield <= 1;
        else
            reset_playfield <= 0;
    end
endmodule

//   This testbench verifies the logic of the victory module
//   It provides a clock signal and the inputs and tests whether it
// correctly displays the win and lose conditions as well as increment the difficulty.
module victory_testbench();

    logic clk, reset, UpKey, DownKey, gameover, TopLED, win, lose, reset_playfield;
    logic [2:0] difficulty;

    // Instantuate Device under test
    victory dut(clk, reset, UpKey,DownKey, win, lose, reset_playfield, gameover, TopLED,
difficulty);

     parameter clock_period = 100;

      // Runs every 100/2 picoseconds for each clock cycle
      initial begin
         clk <= 0;
         forever #(clock_period /2) clk <= ~clk;

      end

      initial begin

      reset <= 1; UpKey <= 0; DownKey <= 0; TopLED <= 0;gameover <= 0; @(posedge clk);
      reset <= 0; @(posedge clk);

      // Test winning conditions and difficulty logic
      UpKey <= 1; DownKey <= 0; TopLED <= 1; @(posedge clk);
                                 TopLED <= 0;            @(posedge clk);

      UpKey <= 1; DownKey <= 0; TopLED <= 1; @(posedge clk);
                                 TopLED <= 0;            @(posedge clk);
      UpKey <= 1; DownKey <= 0; TopLED <= 1; @(posedge clk);
                                 TopLED <= 0;            @(posedge clk);
                                 @(posedge clk);


      // Test losing conditions
      UpKey <= 1; DownKey <= 0; gameover<=1; @(posedge clk);


      // Test regular in game condition
      UpKey <= 0; DownKey <= 0; TopLED <= 1; gameover<=0; @(posedge clk);
      UpKey <= 0; DownKey <= 0; TopLED <= 0; gameover<=0;  @(posedge clk);

```

```
        end

    endmodule
```

```
// Morris Huang, Hudson Wong
// 05/22/2025
// EE 271
// Lab 6 10-bit LFSR

// The LFSR module implements a 10-bit Linear Feedback Shift Register (LFSR)
//   using XNOR feedback logic. It shifts right on every clock cycle and
//   inserts a new bit into the MSB (bit 9) calculated as the XNOR of
//   bit 0 and bit 3. When reset, it goes to a predetermined seed that is passed in
module seeded_LFSR(clk, reset, out, seed);
   input logic clk, reset;
   input logic [9:0] seed;
   output logic [9:0] out;

    // LFSR logic triggered on the rising edge of the clock
   always_ff @(posedge clk)
      begin
         if(reset)
         out<=seed;
         else
            begin
            // Perform right shift on the entire register
            out <= out>>1;
            // Insert feedback bit at MSB (bit 9) using XNOR of bits 0 and 3 or
            // bits 7 and 10.
            out[9] <= ~(out[0] ^ out[3]);
         end
      end

   endmodule

//   This testbench verifies the shift and xnor logic of the LFSR.
//   It provides a clock signal and applies a synchronous reset, then
//   allows the LFSR to run for 1024 clock cycles to observe its output.
module seeded_LFSR_testbench();
   logic CLOCK_50;
   logic reset;
   logic [9:0] out, seed;

   // Instantiate the LFSR module DUT
   seeded_LFSR dut(CLOCK_50, reset, out, seed);

   parameter clock_period = 100;

    // toggle CLOCK_50 every half cycle
   initial begin
         CLOCK_50 <= 0;
         forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;

      end

   // apply reset and then run the LFSR
   initial begin
      seed <= 10'b0000000000;
       // Apply reset for one clock cycle
      reset <= 1; @(posedge CLOCK_50);
      reset <= 0; @(posedge CLOCK_50);
       // Let the LFSR run for 1024 clock cycles
      repeat(1024) @(posedge CLOCK_50);

      seed <= 10'b0100011100; // Test different seeds
       // Apply reset for one clock cycle
      reset <= 1; @(posedge CLOCK_50);
      reset <= 0; @(posedge CLOCK_50);
       // Let the LFSR run for 1024 clock cycles
      repeat(20) @(posedge CLOCK_50);

      seed <= 10'b0110011101; // Test different seeds
       // Apply reset for one clock cycle
      reset <= 1; @(posedge CLOCK_50);
      reset <= 0; @(posedge CLOCK_50);
       // Let the LFSR run for 1024 clock cycles
      repeat(20) @(posedge CLOCK_50);

      $stop;
      end
```

```

endmodule

```

```
// Morris Huang, Hudson Wong
// 06/05/2025
// EE 271
// Lab 6 crossy_road

// This module simulates the logic for the cars running across the LED matrix using an FSM.
// It moves each car (LED) to the right every clock cycle.
// By passing in a random "output" from the LFSR, it can also randomly spawn cars
module crossy_road (clk, reset, NL, red_LED);

   // Initializes the logic. NL represents the LED to the left,
   // or in the case of the starting LED, the LFSR output
   input logic clk, reset, NL;
   output logic red_LED;

   // Starts the FSM logic
   enum {S0, S1} ps, ns;

   // If the left LED is on, turns the current LED to red.
   // Otherwise, turns off. This effectively shifts the car right every clock cycle
   always_comb begin
      case (ps)
         S0: if (NL) ns = S1;
                     else ns = S0;
         S1: ns = S0;

      endcase
   end

   assign red_LED = (ps == S1);

   // Toggles reset such that the light does not turn on.
   always_ff @(posedge clk) begin
      if (reset)
         ps <= S0;
      else
         ps <= ns;
   end

endmodule

// This module functions as a test bench for the crossy_road module.
// Verifies that the crossy_road LED turns on for exactly one clock cycle
// when the LED to the left of it is turned on.
module crossy_road_testbench();
    logic CLOCK_50, reset, NL, red_LED; // Initializes the logic

   crossy_road dut(CLOCK_50, reset, NL, red_LED); // Instantiates the module

   parameter clock_period = 100;

    // toggle CLOCK_50 every half cycle
   initial begin
      CLOCK_50 <= 0;
      forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
   end

   initial begin
      reset <= 1; @(posedge CLOCK_50); // Tests the reset
      reset <= 0; @(posedge CLOCK_50);

      NL <= 1; @(posedge CLOCK_50); // Tests that the LED lights up
      NL <= 0;
      @(posedge CLOCK_50); // Tests that the LED turns off after one clock cycle.
      @(posedge CLOCK_50);
      repeat(10) @(posedge CLOCK_50); // Tests that no random behavior happens.


      $stop;
   end

endmodule

```

```
// Morris Huang, Hudson Wong
// 06/05/2025
// EE 271
// Lab 6 frog_LED_start

// This module represents the starting point for the frogger player using an FSM
// On reset, it automatically turns on, then it allows the player to move the LED around.
// Only one LED is lit at a time. If both the green LED and red LED are lit at the same time,
// it shows a game over.
module frog_LED_start (clk, reset, NL, NR, ND, NU, L, R, D, U, green_LED, red_LED, gameover);
   // Initializes the logic. NL represents the LED to the left, NR is the LED to the right,
   // ND is the bottom LED, and NU is the top LED.
   // L, R, D, and U represent the player movements
   input logic clk, reset, NL, NR, ND, NU, L, R, D, U, red_LED;
   output logic gameover, green_LED;

   // Initializes FSM logic.
   enum {S0, S1} ps, ns;

   // This is the logic for the FSM. If the adjacent LED is lit and the corresponding input
   // is true, the LED turns green. If it is lit, having the corresponding input turn on
   // will turn the LED off.
   always_comb begin
      case (ps)
         S0: if ((NL&R&~L) || (NR&L&~R) || (NU&D&~U) || (ND&U&~D)) ns = S1;
                                                             else ns = S0;
         S1: if ((R & ~L & ~NL) || (L & ~R & ~NR) || (U & ~D & ~ND) || (D & ~U & ~NU)) ns =
S0;
                                                          else ns = S1;
      endcase
   end

   // Assigns the state of the LED and whether the game is over.
   assign green_LED = (ps == S1);
   assign gameover = green_LED & red_LED;

   // On reset, turns on the starting LED.
   always_ff @(posedge clk) begin
      if (reset)
         ps <= S1;
      else
         ps <= ns;
   end

endmodule

//   This testbench verifies the logic of the frog_LED_start module.
//   It provides a clock signal and applies a synchronous reset, then
//   allows the frog_LED_start module to run under different conditions to test whether it
lights up and turns off appropriately.
module frog_LED__start_testbench();
   logic CLOCK_50, reset, NL, NR, ND, NU, L, R, D, U, green_LED, red_LED, gameover; //
Initializes the logic

   frog_LED_start dut(CLOCK_50, reset, NL, NR, ND, NU, L, R, D, U, green_LED, red_LED,
gameover); // Instantiates the module

   parameter clock_period = 100;

    // toggle CLOCK_50 every half cycle
   initial begin
      CLOCK_50 <= 0;
      forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
   end

   initial begin
      reset <= 1; @(posedge CLOCK_50); // Tests the reset turns on the green LED.
      reset <= 0; @(posedge CLOCK_50);

      R <= 1; @(posedge CLOCK_50); // Turns the LED off.
      R<= 0; @(posedge CLOCK_50);

      NL <= 0; NR <= 0; ND <= 0; NU <= 0; L <= 0; R <= 0; D <= 0; U <= 0;red_LED <= 0; //
Initializes all logic to 0.

      ND <= 1; U <= 1; @(posedge CLOCK_50); // Bottom LED is lit, and the up button is
```

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

        U <= 1; D <= 1; @(posedge CLOCK_50); // Tests that LED will not turn off with
conflicting buttons
        U <= 0; D <= 0; @(posedge CLOCK_50);
        R <= 1; L <= 1; @(posedge CLOCK_50);
        R <= 0; L <= 0; @(posedge CLOCK_50);
        U <= 1; D <= 1; R <= 1; L <= 1; @(posedge CLOCK_50);
        U <= 0; D <= 0; R <= 0; L <= 0; @(posedge CLOCK_50);

        U <= 1; @(posedge CLOCK_50); // With one button press, the LED will turn off.
        U <= 0; @(posedge CLOCK_50);
        NR <= 1; L <= 1; @(posedge CLOCK_50); // turn LED back on
        NR <= 0; L <= 0; @(posedge CLOCK_50);

        R <= 1; @(posedge CLOCK_50); // Should turn off
        R <= 0; @(posedge CLOCK_50);
        NL <= 1; R <= 1; @(posedge CLOCK_50); // turn LED back on
        NL <= 0; R <= 0; @(posedge CLOCK_50);

        L <= 1; @(posedge CLOCK_50); // Should turn off
        L <= 0; @(posedge CLOCK_50);
        NU <= 1; D <= 1; @(posedge CLOCK_50); // turn LED back on
        NU <= 0; D <= 0; @(posedge CLOCK_50);

        D <= 1; @(posedge CLOCK_50); // Should turn off
        D <= 0; @(posedge CLOCK_50);
        ND <= 1; U <= 1; @(posedge CLOCK_50); // turn LED back on
        ND <= 0; U <= 0; @(posedge CLOCK_50);

        red_LED <= 1; @(posedge CLOCK_50); // Test gameover condition

        $stop;
    end
endmodule

```

```
// Morris Huang, Hudson Wong
// 06/05/2025
// EE 271
// Lab 6 frog_LED_start

// This module represents the normal LEDs for the frogger player using an FSM
// On reset, it automatically turns off, then it allows the player to move the LED around.
// Only one LED is lit at a time. If both the green LED and red LED are lit at the same time,
// it shows a game over.
module frog_LED (clk, reset, NL, NR, ND, NU, L, R, D, U, green_LED, red_LED, gameover);

   // Initializes the logic. NL represents the LED to the left, NR is the LED to the right,
   // ND is the bottom LED, and NU is the top LED.
   // L, R, D, and U represent the player movements
   input logic clk, reset, NL, NR, ND, NU, L, R, D, U, red_LED;
   output logic green_LED, gameover;

   // Initializes FSM logic.
   enum {S0, S1} ps, ns;

   // This is the logic for the FSM. If the adjacent LED is lit and the corresponding input
   // is true, the LED turns green. If it is lit, having the corresponding input turn on
   // will turn the LED off.
   always_comb begin
      case (ps)
         S0: if ((NL&R&~L) || (NR&L&~R) || (NU&D&~U) || (ND&U&~D)) ns = S1;
                                                                  else ns = S0;
         S1: if ((R & ~L & ~NL) || (L & ~R & ~NR) || (U & ~D & ~ND) || (D & ~U & ~NU)) ns =
S0;
                                                               else ns = S1;
      endcase
   end

   // Assigns the state of the LED
   assign green_LED = (ps == S1);
   assign gameover = (ps == S1) & red_LED;


   // On reset, turns off the LED.
   always_ff @(posedge clk) begin
      if (reset)
         ps <= S0;
      else
         ps <= ns;
   end

endmodule

//   This testbench verifies the logic of the frog_LED module.
//   It provides a clock signal and applies a synchronous reset, then
//   allows the frog_LED module to run under different conditions to test whether it lights
up and turns off appropriately.
module frog_LED_testbench();
   logic CLOCK_50, reset, NL, NR, ND, NU, L, R, D, U, green_LED, red_LED, gameover; //
Initializes the logic

   frog_LED dut(CLOCK_50, reset, NL, NR, ND, NU, L, R, D, U, green_LED, red_LED, gameover);
// Instantiates the module

   parameter clock_period = 100;

    // toggle CLOCK_50 every half cycle
   initial begin
      CLOCK_50 <= 0;
      forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
   end

   initial begin
      reset <= 1; @(posedge CLOCK_50); // Tests the reset
      reset <= 0; @(posedge CLOCK_50);

      NL <= 0; NR <= 0; ND <= 0; NU <= 0; L <= 0; R <= 0; D <= 0; U <= 0;red_LED <= 0; //
Initializes all logic to 0.

      ND <= 1; U <= 1; @(posedge CLOCK_50); // Bottom LED is lit, and the up button is
pressed.
```

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

        U <= 1; D <= 1; @(posedge CLOCK_50); // Tests that LED will not turn off with
conflicting buttons
        U <= 0; D <= 0; @(posedge CLOCK_50);
        R <= 1; L <= 1; @(posedge CLOCK_50);
        R <= 0; L <= 0; @(posedge CLOCK_50);
        U <= 1; D <= 1; R <= 1; L <= 1; @(posedge CLOCK_50);
        U <= 0; D <= 0; R <= 0; L <= 0; @(posedge CLOCK_50);

        U <= 1; @(posedge CLOCK_50); // With one button press, the LED will turn off.
        U <= 0; @(posedge CLOCK_50);
        NR <= 1; L <= 1; @(posedge CLOCK_50); // turn LED back on
        NR <= 0; L <= 0; @(posedge CLOCK_50);

        R <= 1; @(posedge CLOCK_50); // Should turn off
        R <= 0; @(posedge CLOCK_50);
        NL <= 1; R <= 1; @(posedge CLOCK_50); // turn LED back on
        NL <= 0; R <= 0; @(posedge CLOCK_50);

        L <= 1; @(posedge CLOCK_50); // Should turn off
        L <= 0; @(posedge CLOCK_50);
        NU <= 1; D <= 1; @(posedge CLOCK_50); // turn LED back on
        NU <= 0; D <= 0; @(posedge CLOCK_50);

        D <= 1; @(posedge CLOCK_50); // Should turn off
        D <= 0; @(posedge CLOCK_50);
        ND <= 1; U <= 1; @(posedge CLOCK_50); // turn LED back on
        ND <= 0; U <= 0; @(posedge CLOCK_50);

        red_LED <= 1; @(posedge CLOCK_50); // Test gameover condition

        $stop;
    end
endmodule
```

```
//Acknowledgement: This driver code was provided by Prof. Scott Hauck as part of his
// EE 271 course.

// A driver for the 16x16x2 LED display expansion board.
// Read below for an overview of the ports.
// IMPORTANT: You do not need to necessarily modify this file. But if you do, be sure you
know what you are doing.

// FREQDIV: (Parameter) Sets the scanning speed (how often the display cycles through rows)
//          The CLK input divided by 2^(FREQDIV) is the interval at which the driver
switches rows.
// GPIO_1: (Output) The 36-pin GPIO1 header, as on the DE1-SoC board.
// RedPixels: (Input) A 16x16 array of logic items corresponding to the red pixels you'd
like to have lit on the display.
// GrnPixels: (Input) A 16x16 array of logic items corresponding to the green pixels you'd
like to have lit on the display.
// EnableCount: (Input) Whether to continue moving through the rows.
// CLK: (Input) The system clock.
// RST: (Input) Resets the display driver. Required during startup before use.
module LEDDriver #(parameter FREQDIV = 8) (GPIO_1, RedPixels, GrnPixels, EnableCount, CLK,
RST);
    output logic [35:0] GPIO_1;
    input logic [15:0][15:0] RedPixels ;
    input logic [15:0][15:0] GrnPixels ;
    input logic EnableCount, CLK, RST;

    reg [(FREQDIV + 3):0] Counter;
    logic [3:0] RowSelect;
    assign RowSelect = Counter[(FREQDIV + 3):(FREQDIV + 0)];

    always_ff @(posedge CLK)
    begin
        if(RST) Counter <= 'b0;
        if(EnableCount) Counter <= Counter + 1'b1;
    end

    assign GPIO_1[35:32] = RowSelect;
    assign GPIO_1[31:16] = { GrnPixels[RowSelect][0], GrnPixels[RowSelect][1], GrnPixels[
RowSelect][2], GrnPixels[RowSelect][3], GrnPixels[RowSelect][4], GrnPixels[RowSelect][5],
GrnPixels[RowSelect][6], GrnPixels[RowSelect][7], GrnPixels[RowSelect][8], GrnPixels[
RowSelect][9], GrnPixels[RowSelect][10], GrnPixels[RowSelect][11], GrnPixels[RowSelect][12],
 GrnPixels[RowSelect][13], GrnPixels[RowSelect][14], GrnPixels[RowSelect][15] };
    assign GPIO_1[15:0] = { RedPixels[RowSelect][0], RedPixels[RowSelect][1], RedPixels[
RowSelect][2], RedPixels[RowSelect][3], RedPixels[RowSelect][4], RedPixels[RowSelect][5],
RedPixels[RowSelect][6], RedPixels[RowSelect][7], RedPixels[RowSelect][8], RedPixels[
RowSelect][9], RedPixels[RowSelect][10], RedPixels[RowSelect][11], RedPixels[RowSelect][12],
 RedPixels[RowSelect][13], RedPixels[RowSelect][14], RedPixels[RowSelect][15] };
endmodule

module LEDDriver_Test();
    logic CLK, RST, EnableCount;
    logic [15:0][15:0]RedPixels;
    logic [15:0][15:0]GrnPixels;
    logic [35:0] GPIO_1;

    LEDDriver #(.FREQDIV(2)) Driver(.GPIO_1, .RedPixels, .GrnPixels, .EnableCount, .CLK, .
RST);

    initial
    begin
        CLK <= 1'b0;
        forever #50 CLK <= ~CLK;
    end

    initial
    begin
        EnableCount <= 1'b0;
        RedPixels <= '{default:0};
        GrnPixels <= '{default:0};
        @(posedge CLK);

        RST <= 1; @(posedge CLK);
        RST <= 0; @(posedge CLK);
        @(posedge CLK); @(posedge CLK); @(posedge CLK);

        GrnPixels[1][1] <= 1'b1; @(posedge CLK);
```

```
            EnableCount <= 1'b1; @(posedge CLK); #1000;
            RedPixels[2][2] <= 1'b1;
            RedPixels[2][3] <= 1'b1;
            GrnPixels[2][3] <= 1'b1; @(posedge CLK); #1000;
            EnableCount <= 1'b0; @(posedge CLK); #1000;
            GrnPixels[1][1] <= 1'b0; @(posedge CLK);
            $stop;

        end
    endmodule

    module LEDDriver_TestPhysical (CLOCK_50, RST, Speed, GPIO_1);
        input logic CLOCK_50, RST;
        input logic [9:0] Speed;
        output logic [35:0] GPIO_1;
        logic [15:0][15:0]RedPixels;
        logic [15:0][15:0]GrnPixels;
        logic [31:0] Counter;
        logic EnableCount;

        LEDDriver #(.FREQDIV(15)) Driver (.CLK(CLOCK_50), .RST, .EnableCount, .RedPixels, .
    GrnPixels, .GPIO_1);

        //                          F E D C B A 9 8 7 6 5 4 3 2 1 0
        assign RedPixels[00] = '{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
        assign RedPixels[01] = '{1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1};
        assign RedPixels[02] = '{1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1};
        assign RedPixels[03] = '{1,0,1,1,0,0,0,0,0,0,0,0,1,1,0,1};
        assign RedPixels[04] = '{1,0,1,0,1,1,1,1,1,1,1,1,0,1,0,1};
        assign RedPixels[05] = '{1,0,1,0,1,1,0,0,0,0,1,1,0,1,0,1};
        assign RedPixels[06] = '{1,0,1,0,1,0,1,1,1,1,0,1,0,1,0,1};
        assign RedPixels[07] = '{1,0,1,0,1,0,1,0,1,1,0,1,0,1,0,1};
        assign RedPixels[08] = '{1,0,1,0,1,0,1,1,0,1,0,1,0,1,0,1};
        assign RedPixels[09] = '{1,0,1,0,1,0,1,1,1,1,0,1,0,1,0,1};
        assign RedPixels[10] = '{1,0,1,0,1,1,0,0,0,0,1,1,0,1,0,1};
        assign RedPixels[11] = '{1,0,1,0,1,1,1,1,1,1,1,1,0,1,0,1};
        assign RedPixels[12] = '{1,0,1,1,0,0,0,0,0,0,0,0,1,1,0,1};
        assign RedPixels[13] = '{1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1};
        assign RedPixels[14] = '{1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1};
        assign RedPixels[15] = '{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};

        assign GrnPixels[00] = '{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1};
        assign GrnPixels[01] = '{0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0};
        assign GrnPixels[02] = '{0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0};
        assign GrnPixels[03] = '{0,1,0,1,1,1,1,1,1,1,1,1,1,0,1,0};
        assign GrnPixels[04] = '{0,1,0,1,1,0,0,0,0,0,0,1,1,0,1,0};
        assign GrnPixels[05] = '{0,1,0,1,0,1,1,1,1,1,1,0,1,0,1,0};
        assign GrnPixels[06] = '{0,1,0,1,0,1,1,0,0,1,1,0,1,0,1,0};
        assign GrnPixels[07] = '{0,1,0,1,0,1,0,1,0,0,1,0,1,0,1,0};
        assign GrnPixels[08] = '{0,1,0,1,0,1,0,0,1,0,1,0,1,0,1,0};
        assign GrnPixels[09] = '{0,1,0,1,0,1,1,0,0,1,1,0,1,0,1,0};
        assign GrnPixels[10] = '{0,1,0,1,0,1,1,1,1,1,1,0,1,0,1,0};
        assign GrnPixels[11] = '{0,1,0,1,1,0,0,0,0,0,0,1,1,0,1,0};
        assign GrnPixels[12] = '{0,1,0,1,1,1,1,1,1,1,1,1,1,0,1,0};
        assign GrnPixels[13] = '{0,1,1,0,0,0,0,0,0,0,0,0,0,1,1,0};
        assign GrnPixels[14] = '{0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0};
        assign GrnPixels[15] = '{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1};

        always_ff @(posedge CLOCK_50)
        begin
            if(RST) Counter <= 'b0;
            else
            begin
                Counter <= Counter + 1'b1;
                if(Counter >= Speed)
                begin
                    EnableCount <= 1'b1;
                    Counter <= 'b0;
                end
                else EnableCount <= 1'b0;
            end
        end
    endmodule
```

```
// Morris Huang, Hudson Wong
// 06/07/2025
// EE 271
// Lab 6 clock_divider

// Takes in a clock signal, divides the clock cycle and outputs 32
// divided clock signals of varying frequency.
// divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ...
// [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ...
module clock_divider (clock, divided_clocks);

   input logic clock;
   output logic [31:0] divided_clocks = 32'b0;

   // Adds 1 to the logic divided_clocks at the
   // positive edge of each clock cycle.
   // Given the nature of 32 bit numbers,
   // the lowest bit toggles between 1 and 0 twice as fast
   // as the next bit, the next bit toggles twice as fast
   // as the next higher bit, and so on. This allows
   // the clock divider to halve the frequency up to 31 times.
   always_ff @(posedge clock) begin
      divided_clocks <= divided_clocks + 1;
   end
endmodule

// Tests whether the clock divider, given an initialized
// clock, properly outputs the divided frequency signals.
module clock_divider_testbench ();
   logic clock;
   logic [31:0] divided_clocks;

   clock_divider dut (.clock(clock), .divided_clocks(divided_clocks));

   // Sets up a clock that toggles every 10 nanoseconds
   initial begin
      clock <= 0;
      forever #10 clock <= ~clock;

   end //initial

      integer i;
      initial begin

      // Iterates through 2^32 clock cycles to test whether
      // each divided clock cycle is twice as long as the previous.
      // Due to processing limitations, does not test the maximum
      // clock cycles.
      for(i=0; i<2**32;i++) begin
       @(posedge clock);
      end // for loop

      $stop;
   end // initial
endmodule
```

```systemverilog
// Morris Huang, Hudson Wong
// 06/07/025
// EE 271
// Lab 6 comparator

// Comparator takes in two inputs inputA and inputB and returns whether A is greater than B.
// If it is less than or equal to B, returns 0, otherwise returns 1.
module comparator(inputA, inputB, A_greater_B, clk, reset);
   input logic [9:0] inputA, inputB;
   input logic clk, reset;
   output logic A_greater_B;

   // Sequential logic to set A_greater_B
   always_ff @(posedge clk) begin
      if (reset)
         A_greater_B <= 1'b0;
      else
         A_greater_B <= (inputA > inputB);
   end
endmodule

// This module tests the comparator by instantiating several test cases to make
// sure that it correctly outputs when A is greater than B.
module comparator_testbench();

   logic [9:0] inputA, inputB;
   logic CLOCK_50, reset, A_greater_B;

   // Calls comparator
   comparator dut(inputA, inputB, A_greater_B, CLOCK_50, reset);

   parameter clock_period = 100;

    // toggle CLOCK_50 every half cycle
   initial begin
      CLOCK_50 <= 0;
      forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;
   end

   initial begin
       // Apply reset for one clock cycle
      reset <= 1; @(posedge CLOCK_50);
      reset <= 0; @(posedge CLOCK_50);

      inputA <= 1000000000; inputB <= 0100000000; @(posedge CLOCK_50); // Test A > B
      @(posedge CLOCK_50);
      inputA <= 0100000000; inputB <= 0100000000; @(posedge CLOCK_50); // Test A = B
      @(posedge CLOCK_50);
      inputA <= 0100000000; inputB <= 1000000000; @(posedge CLOCK_50); // Test A < B
      inputA <= 0000000000; inputB <= 0000000000; @(posedge CLOCK_50); // Test edge cases
      inputA <= 1111111111; inputB <= 1111111111; @(posedge CLOCK_50);
      inputA <= 0000000000; inputB <= 1111111111; @(posedge CLOCK_50);
      inputA <= 1111111111; inputB <= 0000000000; @(posedge CLOCK_50); // Test A < B

      inputA <= 0100001111; inputB <= 1000001100; @(posedge CLOCK_50); // Test random values
      inputA <= 1100010100; inputB <= 100101000; @(posedge CLOCK_50);
      inputA <= 0100011110; inputB <= 1001111100; @(posedge CLOCK_50);
      inputA <= 0100101111; inputB <= 1010101100; @(posedge CLOCK_50);
      inputA <= 0101010110; inputB <= 0101010110; @(posedge CLOCK_50);
      inputA <= 0100101100; inputB <= 0010100000; @(posedge CLOCK_50);
      inputA <= 0100101100; inputB <= 0100001111; @(posedge CLOCK_50);
      inputA <= 1100011000; inputB <= 1000100110; @(posedge CLOCK_50);
      inputA <= 1111000000; inputB <= 1000000010; @(posedge CLOCK_50);
      inputA <= 0100000000; inputB <= 1010110010; @(posedge CLOCK_50);
      inputA <= 0101011100; inputB <= 0101100101; @(posedge CLOCK_50);
      inputA <= 1100111000; inputB <= 1111111100; @(posedge CLOCK_50);
      inputA <= 0100110000; inputB <= 1000110000; @(posedge CLOCK_50);


      $stop;
   end
endmodule


```

```
// Morris Huang, Hudson Wong
// 06/07/025
// EE 271
// Lab 6 three_bit_counter

// Three_bit_counter increments a counter by 1 every time an input is true.
// It starts at 0 and goes to 7 using binary numbers and a finite state machine.
// It takes in parameters clk, reset, in, and out.
// clk represents the clock cycles,
// reset sets the counter back to zero,
// in increments the number by one,
// and out represents the current count
module three_bit_counter (clk, reset, in, out);
   input logic clk, reset, in;
   output logic [2:0] out;

   // Sets the eight states 0-7
   enum {S0, S1, S2, S3, S4, S5, S6, S7} ps, ns; // Present state, next state

   // Sets the finite state machine for the counter. If the input is true,
   // goes to the next state. Otherwise, stays on the same state.
   always_comb begin
      case(ps)
         S0: if (in) ns = S1;
             else ns = S0;
         S1: if (in) ns = S2;
             else ns = S1;
         S2: if (in) ns = S3;
             else ns = S2;
         S3: if (in) ns = S4;
             else ns = S3;
         S4: if (in) ns = S5;
             else ns = S4;
         S5: if (in) ns = S6;
             else ns = S5;
         S6: if (in) ns = S7;
             else ns = S6;
         S7: ns = S7;
      endcase
   end

   // Sets the output to whatever state it corresponds to from 0-7
   always_comb begin
        case (ps)
            S0: out = 3'b000;
            S1: out = 3'b001;
            S2: out = 3'b010;
            S3: out = 3'b011;
         S4: out = 3'b100;
         S5: out = 3'b101;
         S6: out = 3'b110;
         S7: out = 3'b111;
        endcase
    end

   // Resets the present state to the first state, going back to 0
   always_ff @(posedge clk) begin
      if (reset)
         ps <= S0;
      else
         ps <= ns;
   end
endmodule

// This module tests the three bit counter, ensuring that it only increments
// when in is true, and that out represents the correct output
module three_bit_counter_testbench ();

   // Instantiates the logic
    logic CLOCK_50; // 50MHz clock
    logic reset, in;
    logic [2:0] out;

   // Calls the module
   three_bit_counter dut(CLOCK_50, reset, in, out);

   parameter clock_period = 100;

   // Runs every 100/2 picoseconds for each clock cycle
   initial begin
      CLOCK_50 <= 0;
      forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;

   end //initial

   initial begin
         reset <= 1;             @(posedge CLOCK_50); // assert reset
         reset <= 0;             @(posedge CLOCK_50); // release reset
         in<=1;                  @(posedge CLOCK_50); // Ensures that the out increments by 1
                              @(posedge CLOCK_50); // Also ensures that it maxes out at 7 without causing errors
                              @(posedge CLOCK_50);
                              @(posedge CLOCK_50);
                              @(posedge CLOCK_50);
                              @(posedge CLOCK_50);
                              @(posedge CLOCK_50);
                              @(posedge CLOCK_50);
                              @(posedge CLOCK_50);
                              @(posedge CLOCK_50);
                              @(posedge CLOCK_50);
         in<=0;                  @(posedge CLOCK_50);
         reset<=1;               @(posedge CLOCK_50);
         reset<=0;               @(posedge CLOCK_50);
                              @(posedge CLOCK_50);
         in<=1;                  @(posedge CLOCK_50);
         in<=0;                  @(posedge CLOCK_50); // Ensures that there are no false increments
         in<=1;                  @(posedge CLOCK_50);
         in<=0;                  @(posedge CLOCK_50);
         in<=1;                  @(posedge CLOCK_50);
         in<=0;                  @(posedge CLOCK_50);
         in<=1;                  @(posedge CLOCK_50);
         in<=0;                  @(posedge CLOCK_50);
         in<=1;                  @(posedge CLOCK_50);
         in<=0;                  @(posedge CLOCK_50);
         in<=1;                  @(posedge CLOCK_50);
         in<=0;                  @(posedge CLOCK_50);
         in<=1;                  @(posedge CLOCK_50);
         in<=0;                  @(posedge CLOCK_50);

      $stop;
      end //initial
endmodule
```

```
// Morris Huang, Hudson Wong
// 06/07/2025
// EE 271
// Lab 6 user_input

// Takes a button press that, as long as it is pressed,
// only registers once even across multiple clock cycles.
// Has parameters clk which sets the clock cycles,
// reset which sets the output to a default state,
// button which is the key input, and out which
// registers once for as long as the button is pressed.
module user_input(clk, reset, button, out);
   input logic clk, reset, button;
   output logic out;
   logic sync_button, next_button;

   // A pair of flip flops for metastability. Delays the input by two cycles
   always_ff @(posedge clk) begin
      if (reset) begin
         sync_button <=1'b0;
         next_button <=1'b0;
      end else begin
         sync_button <=button;
         next_button <=sync_button;
      end
   end

   enum {S0, S1} ps, ns;

   // If the button is held down, only registers once
   always_comb begin
        case (ps)
            S0: if (next_button) ns = S1;
                            else ns = S0;
            S1: if (next_button) ns = S1;
                            else ns = S0;
        endcase
    end

    assign out = (ps==S0) & next_button;

   // Resets the output to 0
   always_ff @(posedge clk) begin
      if (reset)
         ps <= S0;
      else
         ps <= ns;
   end

endmodule

// Tests the user input module using clock cycles and inputs
module user_input_testbench();
   logic CLOCK_50; // 50MHz clock
   logic reset, button, out;

   user_input dut(CLOCK_50, reset, button, out);
   parameter clock_period = 100;


   initial begin
      CLOCK_50 <= 0;
      forever #(clock_period /2) CLOCK_50 <= ~CLOCK_50;

   end //initial

   initial begin
      reset <= 1; button<=0; @(posedge CLOCK_50);
      reset <= 0;  @(posedge CLOCK_50);
      button<=1;   @(posedge CLOCK_50);
                   @(posedge CLOCK_50);
                   @(posedge CLOCK_50);
                   @(posedge CLOCK_50);
      button<=0;   @(posedge CLOCK_50); // Tests whether out only registers once
      button<=1;   @(posedge CLOCK_50);
      button<=0;   @(posedge CLOCK_50);
```

```
                        @(posedge CLOCK_50);
                        @(posedge CLOCK_50);
                        @(posedge CLOCK_50);
            button<=1;  @(posedge CLOCK_50); // Tests whether multiple buttons counts as one press
            button<=1;  @(posedge CLOCK_50);
            button<=1;  @(posedge CLOCK_50);
            button<=1;  @(posedge CLOCK_50);
            button<=1;  @(posedge CLOCK_50);
            button<=0;  @(posedge CLOCK_50);
            button<=1;  @(posedge CLOCK_50);
            button<=1;  @(posedge CLOCK_50);
            button<=1;  @(posedge CLOCK_50);
            button<=1;  @(posedge CLOCK_50);
            button<=1;  @(posedge CLOCK_50);
            $stop;
            end //initial
      endmodule


```