EE 271 Design Project Report: Flappy Bird

Diego Cervantes

Demo TA: Ryan Wang

June 6, 2022 @2:30 pm

Department of Computer & Electrical Engineering, University of Washington

EE 271: Digital Circuits and Systems

Prof. Denise Wilson

June 6, 2022

Design Problem:

The popular mobile phenomenon, Flappy Bird, was created using modules from previous labs and newly created ones. Flappy bird allows the user to play as a bird called Olly Oriole and try to achieve the highest possible score by "flapping" through the sky, avoiding the pipes. Olly is a single red LEDR, and the pipes floating from right to left are green. The user controls Olly by pressing key zero (KEY [0]). If the user does not press the key, then Olly will begin to descend downwards. Descending all the way to the "floor" will not cause Olly to die (end game). To begin playing, the user must use switch 9 (SW [9]); and this switch will be used as the "restart game" input. Three Hex displays are used to keep track of the gaps (points) cleared with a max score of 999. If a user surpasses this figure, the game will continue to be playable, but the score will no longer be trackable. Fifteen different pipe configurations have been created and will be outputted pseudo-randomly. The design of this lab uses many of the relevant concepts found in a digital and systems curriculum. Meta-stability, combinational logic, flip-flop implementation, and state diagrams are all concepts that are useful to complete a project like this.

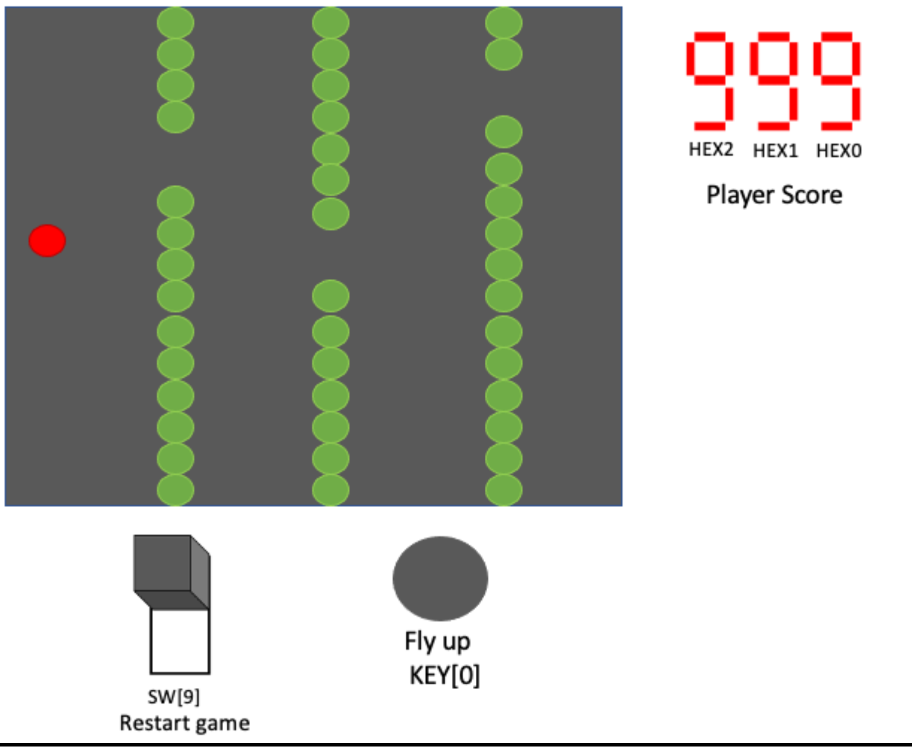


Figure 1. User-Level Block Diagram

Methods and Procedures:

Many of the modules from previous labs were used and modified to create the logic flappy bird required. For example, the modules from lab 5 controlling the center and normal lights met most of the logic required to control Olly. Quick modifications to these modules were made to account for "gravity," and slight changes to the leftmost and rightmost lights were made. The modules accounting for meta-stability and the user input from lab 7 were also recycled for the same reasons as mentioned above. A new module to generate the green pipes is used, while another newly created module serves to move the pipes across the LED array. To avoid the pipes from appearing in the same order all the time, a new LFSR module is implemented with a waiting gap of 4 columns between pipes to make the game playable. A module serving as a collision detector would then call the game over module to end the game. Whenever Olly surpassed a gap, a scoring module was called to track the player's score. The DE1_SoC was modified to contain a wide range of clocks, including one that controlled the entire system and another clock to simulate gravity.

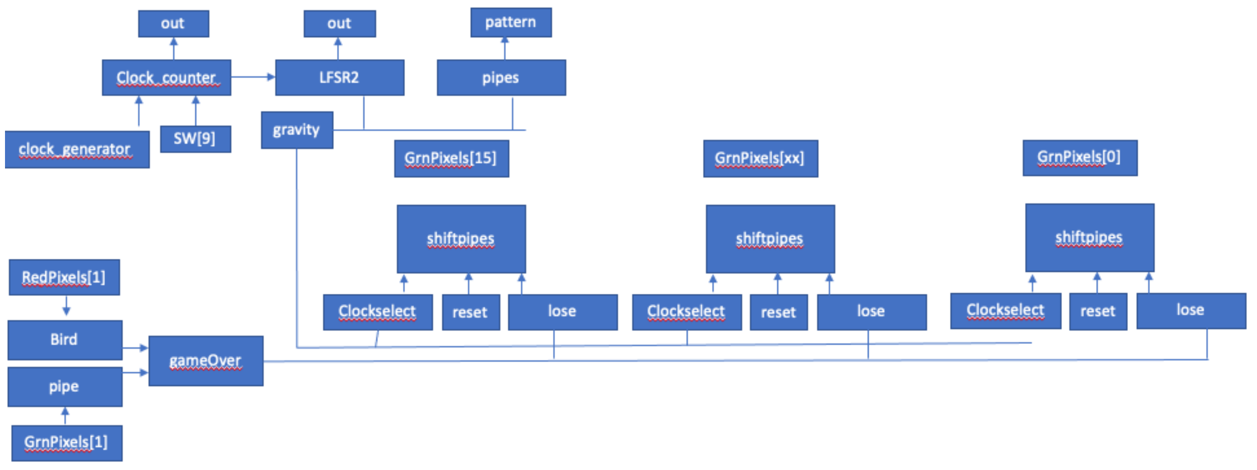


Figure 2. Designer-Level Block Diagram

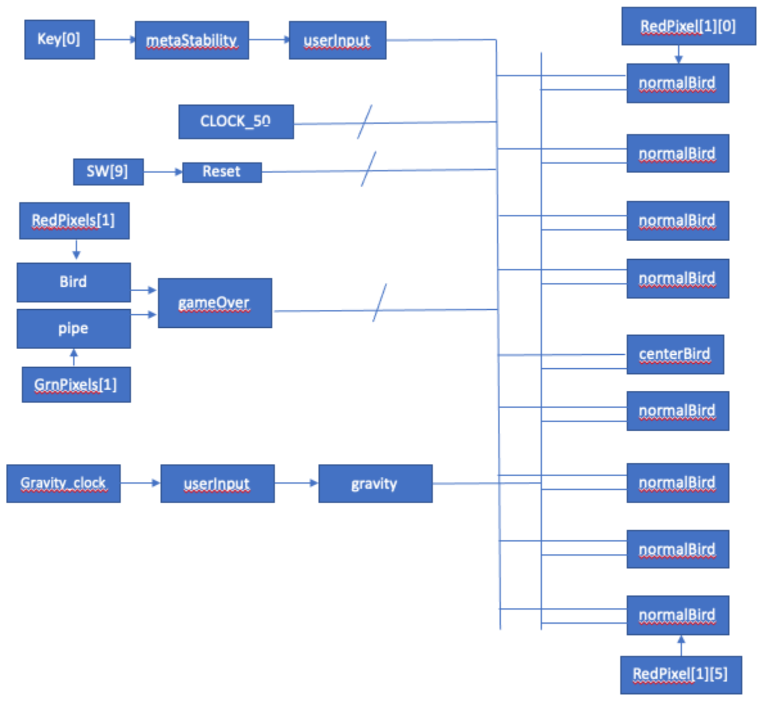


Figure 3. Designer-Level Block Diagram continued

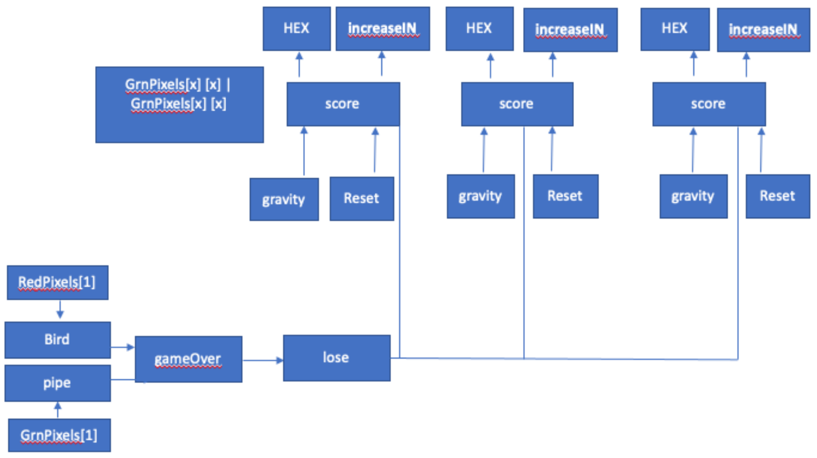
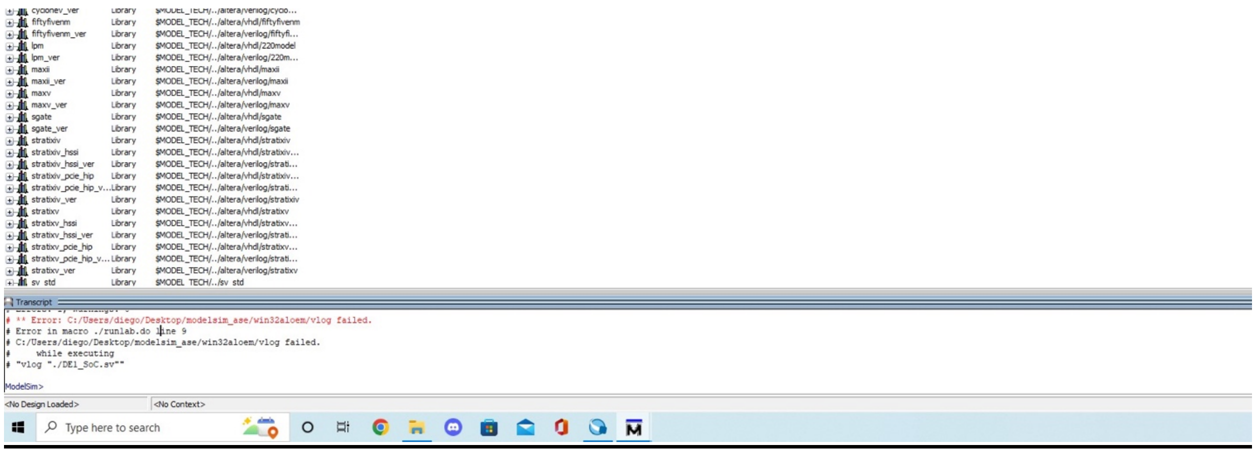


Figure 4. Designer-Level Block Diagram continued

<u>Results:</u>

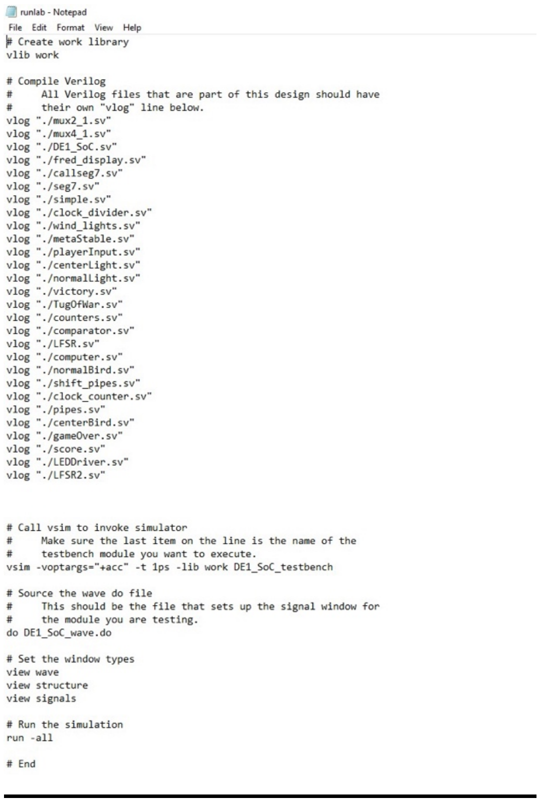
Unfortunately, a simulation of the top-level module was unsuccessful. Reasons were unknown but the actual project on the board works as it should.



```
cyclonev_ver          Library   $MODEL_TECH/../altera/verilog/cyclo...
fiftyfivenm           Library   $MODEL_TECH/../altera/vhdl/fiftyfivenm
fiftyfivenm_ver       Library   $MODEL_TECH/../altera/verilog/fiftyfi...
lpm                   Library   $MODEL_TECH/../altera/vhdl/220model
lpm_ver               Library   $MODEL_TECH/../altera/verilog/220m...
maxii                 Library   $MODEL_TECH/../altera/vhdl/maxii
maxii_ver             Library   $MODEL_TECH/../altera/verilog/maxii
maxv                  Library   $MODEL_TECH/../altera/vhdl/maxv
maxv_ver              Library   $MODEL_TECH/../altera/verilog/maxv
sgate                 Library   $MODEL_TECH/../altera/vhdl/sgate
sgate_ver             Library   $MODEL_TECH/../altera/verilog/sgate
stratixiv             Library   $MODEL_TECH/../altera/vhdl/stratixiv
stratixiv_hssi        Library   $MODEL_TECH/../altera/vhdl/stratixiv...
stratixiv_hssi_ver    Library   $MODEL_TECH/../altera/verilog/strati...
stratixiv_pcie_hip    Library   $MODEL_TECH/../altera/vhdl/stratixiv...
stratixiv_pcie_hip_v...Library  $MODEL_TECH/../altera/verilog/strati...
stratixiv_ver         Library   $MODEL_TECH/../altera/verilog/stratixiv
stratixv              Library   $MODEL_TECH/../altera/vhdl/stratixv
stratixv_hssi         Library   $MODEL_TECH/../altera/vhdl/stratixv...
stratixv_hssi_ver     Library   $MODEL_TECH/../altera/verilog/strati...
stratixv_pcie_hip     Library   $MODEL_TECH/../altera/vhdl/stratixv...
stratixv_pcie_hip_v...Library   $MODEL_TECH/../altera/verilog/strati...
stratixv_ver          Library   $MODEL_TECH/../altera/verilog/stratixv
sv_std                Library   $MODEL_TECH/../sv_std
```

```
# ** Error: C:/Users/diego/Desktop/modelsim_ase/win32aloem/vlog failed.
# Error in macro ./runlab.do line 9
# C:/Users/diego/Desktop/modelsim_ase/win32aloem/vlog failed.
#     while executing
# "vlog "./DE1_SoC.sv""

ModelSim>
```

Figure 7. Simulation for top level module does not simulate successfully



```
# Create work library
vlib work

# Compile Verilog
#     All Verilog files that are part of this design should have
#     their own "vlog" line below.
vlog "./mux2_1.sv"
vlog "./mux4_1.sv"
vlog "./DE1_SoC.sv"
vlog "./fred_display.sv"
vlog "./callseg7.sv"
vlog "./seg7.sv"
vlog "./simple.sv"
vlog "./clock_divider.sv"
vlog "./wind_lights.sv"
vlog "./metaStable.sv"
vlog "./playerInput.sv"
vlog "./centerLight.sv"
vlog "./normalLight.sv"
vlog "./victory.sv"
vlog "./TugOfWar.sv"
vlog "./counters.sv"
vlog "./comparator.sv"
vlog "./LFSR.sv"
vlog "./computer.sv"
vlog "./normalBird.sv"
vlog "./shift_pipes.sv"
vlog "./clock_counter.sv"
vlog "./pipes.sv"
vlog "./centerBird.sv"
vlog "./gameOver.sv"
vlog "./score.sv"
vlog "./LEDDriver.sv"
vlog "./LFSR2.sv"


# Call vsim to invoke simulator
#     Make sure the last item on the line is the name of the
#     testbench module you want to execute.
vsim -voptargs="+acc" -t 1ps -lib work DE1_SoC_testbench

# Source the wave do file
#     This should be the file that sets up the signal window for
#     the module you are testing.
do DE1_SoC_wave.do

# Set the window types
view wave
view structure
view signals

# Run the simulation
run -all

# End
```

Figure 8. Runlab file

Reflection:

During the whole EE271 course labs, there were many skills learned that can be taken away from them and used for future classes in embedded systems. The knowledge that is likely to be retained from labs is the importance of state machines, combinational and sequential circuits, logic design, and the different techniques for modeling systems with Verilog. The state machines are used to map out every state of a process and look at the behavior of a machine. For combinational circuits, there is no feedback among the inputs and outputs. Therefore, combinational circuits are mainly used to do arithmetic (boolean algebra). On the other hand, sequential circuits are used to store important data. However, many systems consist of a mix of both combinational and sequential circuits which I could see being helpful for future embedded systems classes. Logic design in general is useful for building complex electronic components that are then used for digital systems. Creating the testbenches and analyzing the results on Modelsim helps to verify whether the machine is behaving the way it should. This will be very useful in future embedded systems classes, especially to analyze where something may be going wrong.

Throughout the quarter, I have learned many new topics about digital circuits and systems. Some of those included, logic design, boolean algebra, kmaps, flip flops, state machines, muxes and decoders. I was able to understand all of these topics pretty well and are important to know as fundamental knowledge for higher level embedded systems classes. Although there were still things I struggled with, I was able to apply each one of these topics into my labs.

Throughout the progress of the labs, there were things I became good at but also things I could continue to work on to improve the quality of each lab results. As for my strengths in terms of laboratory skills, I became good at debugging my code, loading and testing my code on the FPGA board, and implementing my code based off of the state machines I created. Whenever I ran the analysis on my code and encountered issues, I was able to find the errors fairly quickly and saved me some time. Uploading and testing my code was also a great strength of mine as I was able to execute different test cases on the board. In terms of the actual Verilog language, I was able to learn it quickly as I had never worked with it before. In terms of my weaknesses, I struggled with Modelsim and creating enough test cases for the modules. Sometimes I would not be able to get my Modelsim results to show up correctly so I couldn’t analyze the results on Modelsim very well. I would mainly analyze the results on the FPGA board. Another one of my weaknesses was creating the state machines for the code that I needed to implement. In general, I struggled visualizing what each state needed to look like.

Creating complex designs that involve more than one finite state machine or combinational logic block were time consuming to create and require careful analysis of the expected behavior of each state machine or combinational logic block. In the final project, there were many state machines and combinational logic blocks. The lab took a long time to create because it was important to have the correct states implemented in the code. It was important to be precise with the inputs and outputs of a machine throughout all the labs. The flappy bird project required a counter for the score, random patterns for the pipe, and different states for the bird. There was a close relationship between each state machine for the flappy bird project which shows how careful I needed to be with this complex design that involved more than one finite state machine/combinational logic block. All of this knowledge will help moving forward as I continue to create more complex designs.

Appendix

README

- DE1_SoC.sv is the top-level module and calls on most modules
- metaStable.sv prevents metastability from affecting the system
- playerInput.sv Prevents Olly (the bird) from flying by holding the key
- clock_devider.sv generates a clock that is easy to work with
- LEDDriver.sv is the file provided to control the 16x16 LED array
- centerBird.sv is the module responsible for when Olly is in the center
- normalBird.sv is the module that looks for the position of Olly besides the center light
- LFSR2.sv is a 4-bit linear feedback register that is used as a random generator every 4 clock cycles
- shift_pipes.sv controls the movement of pipes across all rows
- score.sv controls hex0-2 to display the users score
- clock_counter.sv is a clock that checks LFSR2 to see when it is time to create a new pattern and make sure they don't repeat
- gameOver.sv serves as a collision detector, ending the game once it hits
- pipes.sv creates the pattern for the pipes

```
    module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, LEDR, SW, KEY, GPIO_1);
       //The LED outputs and the 7segment display HEX outputs
       output logic [9:0] LEDR;
       output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
       output logic [35:0] GPIO_1;

       //CLOCK_50,SW, and KEY are inputs
       input logic CLOCK_50;
       input logic [9:0] SW;
       input logic [3:0] KEY;

       // Turn off unused HEX displays
       assign HEX3 = '1;
       assign HEX4 = '1;
       assign HEX5 = '1;
       //assigns sw9 as reset signal
       logic Reset;
       assign Reset = SW[9];

       logic [31:0] div_clk;

       parameter whichClock = 25; //768 Hz Clock
       parameter gravityClock = 23; //3 Hz Clock
       parameter systemClock = 14; // 1526 Hz clock

       clock_divider cdiv(.clock(CLOCK_50), .reset(Reset), .divided_clocks(div_clk));

       // Clock selection;
       // allows for easy switching between simulation and board clocks
       logic clkSelect;
       //Uncomment ONE of the following two lines depending on intention
       assign clkSelect = CLOCK_50;          // for simulation
       //assign clkSelect = div_clk[whichClock]; // for synthesis on DE1_SoC board

       logic gravitySelect;
       assign gravitySelect = div_clk[gravityClock];//simulates gravity
       logic systemSelect;
       assign systemSelect = div_clk[systemClock];

       /* If you notice flickering, set SYSTEM_CLOCK faster.
        However, this may reduce the brightness of the LED board. */


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

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

       /* LED board test submodule - paints the boards with a static pattern.
             Replace with your own code driving RedPixles and GrnPixels.

             KEY0      :Reset
          =====================================================================*/
       //button to controll Olly
       logic flap;
       assign flap = ~KEY[0];

       //prevents metastibility from happening by passing the input of the button into a dff
       logic input_player;
       metaStable met1(.Clock(CLOCK_50), .Reset, .player_in(flap), .player_out(input_player));
       //goes through metastable again
       logic stable_player;
       metaStable met2 (.Clock(CLOCK_50), .Reset, .player_in(input_player), .player_out(
 stable_player));

       logic flappy;
       playerInput player_1(.Clock(CLOCK_50), .Reset, .final_in(stable_player), .final_out(
 flappy));//player input

       logic gravityyyyy;
       playerInput player_2(.Clock(CLOCK_50), .Reset, .final_in(gravitySelect), .final_out(
 gravityyyyy));

       logic gmeover;
        gameOver over(.bird(RedPixels[1]), .pipe(GrnPixels[1]), .lose(gmeover));



       //logic [9:0] random_num;//random generator
       //logic restart_game;//reset game at center light after there is a winner


       //LFSR generator(.Clock(clkSelect), .Reset(restart_game), .out(random_num));

       //logic computer_push;//computer push input
       //computer cyberPlayer(.Clock(clkSelect), .Reset, .SW(SW[8:0]), .out(computer_push));

       // player1 and com1 holds result from the first DFF for both players
       //to prevent metastabilty
       //logic player1, com1;

       //metaStable met1 (.Clock(clkSelect), .Reset, .player_in(~KEY[0]),
 .player_out(player1));
       //metaStable met2 (.Clock(clkSelect), .Reset, .player_in(computer_push),
 .player_out(com1));

       // stable_p1 and stable_p2  holds result from the second DFF for both players
       //to prevent metastabilty
       //logic player1_2nd, com1_2nd;

       //Prevents holding of a key using the stable player inputs
       //playerInput player_1 (.Clock(clkSelect), .Reset, .final_in(player1),
 .final_out(player1_2nd));//player input
       //playerInput player_2 (.Clock(clkSelect), .Reset, .final_in(com1),
 .final_out(com1_2nd));//computer input


       //logic RESTART1, restart1,restart2;
       //assign RESTART1 = Reset | restart1 | restart2;//restart game condition
       //Temporary values sent to the normal and center light modules to get the output for
 the LEDs
       //logic hold1, hold2, hold3, hold4, hold5, hold6, hold7, hold8, hold9;

       logic lose;

       // set up the lights for the bird
       normalBird zero      (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL
 (RedPixels[1][1]),    .NR(1'b0),                .high(1'b1), .low(1'b0), .lose, .lightOn(
 RedPixels[1][0]));
       normalBird one       (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL
 (RedPixels[1][2]),    .NR(RedPixels[1][0]),    .high(1'b0), .low(1'b0), .lose, .lightOn(
 RedPixels[1][1]));
       normalBird two       (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL
 (RedPixels[1][3]),    .NR(RedPixels[1][1]),    .high(1'b0), .low(1'b0), .lose, .lightOn(
 RedPixels[1][2]));
       normalBird three     (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL
 (RedPixels[1][4]),    .NR(RedPixels[1][2]),    .high(1'b0), .low(1'b0), .lose, .lightOn(
 RedPixels[1][3]));
       normalBird four      (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL
 (RedPixels[1][5]),    .NR(RedPixels[1][3]),    .high(1'b0), .low(1'b0), .lose, .lightOn(
 RedPixels[1][4]));
       normalBird five      (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL
 (RedPixels[1][6]),    .NR(RedPixels[1][4]),    .high(1'b0), .low(1'b0), .lose, .lightOn(
 RedPixels[1][5]));
       normalBird six       (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL
 (RedPixels[1][7]),    .NR(RedPixels[1][5]),    .high(1'b0), .low(1'b0), .lose, .lightOn(
 RedPixels[1][6]));
       centerBird seven     (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL
 (RedPixels[1][8]),    .NR(RedPixels[1][6]),                                  .lose, .lightOn(
 RedPixels[1][7]));
       normalBird eight     (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL
 (RedPixels[1][9]),    .NR(RedPixels[1][7]),    .high(1'b0), .low(1'b0), .lose, .lightOn(
 RedPixels[1][8]));
       normalBird nine      (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL
 (RedPixels[1][10]),   .NR(RedPixels[1][8]),    .high(1'b0), .low(1'b0), .lose, .lightOn(
 RedPixels[1][9]));
       normalBird ten       (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL
(RedPixels[1][11]),   .NR(RedPixels[1][9]),   .high(1'b0), .low(1'b0), .lose, .lightOn(RedPixels[1][10]));
        normalBird eleven    (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL(RedPixels[1][12]),   .NR(RedPixels[1][10]),  .high(1'b0), .low(1'b0), .lose, .lightOn(RedPixels[1][11]));
        normalBird twelve    (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL(RedPixels[1][13]),   .NR(RedPixels[1][11]),  .high(1'b0), .low(1'b0), .lose, .lightOn(RedPixels[1][12]));
        normalBird thirteen  (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL(RedPixels[1][14]),   .NR(RedPixels[1][12]),  .high(1'b0), .low(1'b0), .lose, .lightOn(RedPixels[1][13]));
        normalBird fourteen  (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL(RedPixels[1][15]),   .NR(RedPixels[1][13]),  .high(1'b0), .low(1'b0), .lose, .lightOn(RedPixels[1][14]));
        normalBird fifteen   (.Clock(CLOCK_50), .Reset, .gravity(gravityyyyy), .R(flappy), .NL(1'b0),               .NR(RedPixels[1][14]),  .high(1'b0), .low(1'b1), .lose, .lightOn(RedPixels[1][15]));

        shift_pipes rows15   (.Clock(clkSelect), .Reset, .newpattern(bar), .lose, .out(GrnPixels[15]));

        shift_pipes rows0    (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[1]), .lose, .out(GrnPixels[0]));
        shift_pipes rows1    (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[2]), .lose, .out(GrnPixels[1]));
        shift_pipes rows2    (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[3]), .lose, .out(GrnPixels[2]));
        shift_pipes rows3    (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[4]), .lose, .out(GrnPixels[3]));
        shift_pipes rows4    (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[5]), .lose, .out(GrnPixels[4]));
        shift_pipes rows5    (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[6]), .lose, .out(GrnPixels[5]));
        shift_pipes rows6    (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[7]), .lose, .out(GrnPixels[6]));
        shift_pipes rows7    (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[8]), .lose, .out(GrnPixels[7]));
        shift_pipes rows8    (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[9]), .lose, .out(GrnPixels[8]));
        shift_pipes rows9    (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[10]), .lose, .out(GrnPixels[9]));
        shift_pipes rows10   (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[11]), .lose, .out(GrnPixels[10]));
        shift_pipes rows11   (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[12]), .lose, .out(GrnPixels[11]));
        shift_pipes rows12   (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[13]), .lose, .out(GrnPixels[12]));
        shift_pipes rows13   (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[14]), .lose, .out(GrnPixels[13]));
        shift_pipes rows14   (.Clock(clkSelect), .Reset, .newpattern(GrnPixels[15]), .lose, .out(GrnPixels[14]));


        logic generate_clock;
        clock_counter count(.Clock(clkSelect), .Reset, .out(generate_clock));

        logic [3:0] random_num; // random generator
        LFSR2 random(.Clock(generate_clock), .Reset, .out(random_num));

        logic [15:0] bar;
        pipes pipe(.LFSR2_out(random_num), .pattern(bar), .generate_clock);

        //logic [15:0] actual_pattern;
        //random_pattern repetition (.Clock(clkSelect), .Reset, .pattern(bar), .actualpattern(actual_pattern), .clockgenerator(generate_clock));

        // set up pipes
        //shift_pipes rows15 (.Clock(clkSelect), .Reset, .newpattern(actual_pattern), .lose, .out(GrnPixels[15]));

        gameOver endgame(.bird(RedPixels[1]), .pipe(GrnPixels[1]), .lose);

        //passes the temp. signal into the score module
        logic [6:0] fakesignal1, fakesignal2, fakesignal3;
        logic up1, up2, up3;
        score first_digit(.Clock(clkSelect), .Reset, .increaseIN(GrnPixels[0][5] | GrnPixels[0
][1]), .lose, .HEX(fakesignal1), .increaseout(up1));
        score second_digit(.Clock(clkSelect), .Reset, .increaseIN(up1), .lose, .HEX(
fakesignal2), .increaseout(up2));
        score third_digit(.Clock(clkSelect), .Reset, .increaseIN(up2), .lose, .HEX(fakesignal3
), .increaseout(up3));

        //temporary signal to get information for hex displays
        assign HEX0 = ~fakesignal1;
        assign HEX1 = ~fakesignal2;
        assign HEX2 = ~fakesignal3;

        endmodule

        /*//Setting the temporary outputs from the center and normal light modules to the
actual LEDs
        assign LEDR[1] = hold1;
        assign LEDR[2] = hold2;
        assign LEDR[3] = hold3;
        assign LEDR[4] = hold4;
        assign LEDR[5] = hold5;
        assign LEDR[6] = hold6;
        assign LEDR[7] = hold7;
        assign LEDR[8] = hold8;
        assign LEDR[9] = hold9;*/

        //Temporary value to send to counters module to display the winner in the 7segment
HEX0 and HEX1
        //logic [6:0] hold00;
        //logic [6:0] hold11;

        //Assign the temporary value to HEX0 based on the result from the victory module
        //assign HEX0 = ~hold00;
        //assign HEX1 = ~hold11;

        //Turns off all (2-5) 7segment HEX displays
        /*assign HEX2 = 7'b1111111;
        assign HEX3 = 7'b1111111;
        assign HEX4 = 7'b1111111;
        assign HEX5 = 7'b1111111;*/


        //controls display of HEX0 and HEX1 for the scores of the player and computer
        //counters player_win(.Clock(clkSelect ), .Reset, .LR(player1_2nd & ~com1_2nd),
.LRM(hold1), .HEX(hold00), .restart(restart1));
        //counters com_win(.Clock(clkSelect ), .Reset, .LR(~player1_2nd & com1_2nd),
.LRM(hold9), .HEX(hold11), .restart(restart2));


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

        logic CLOCK_50; //50MHZ clock
        logic [9:0] SW;
        logic [3:0] KEY;
        logic [35:0] GPIO_1;

        DE1_SoC dut(CLOCK_50, LEDR, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, SW, KEY, GPIO_1);

        // Set up a simulated clock to toggle (from low to high or high to low)
        // every 50 time steps
        parameter  CLOCK_PERIOD=100;
        initial begin
           CLOCK_50 <= 0;
           forever #(CLOCK_PERIOD/2) CLOCK_50 <= ~CLOCK_50;//toggle the clock indefinitely
        end
    initial begin

                                                                                   @(posedge
CLOCK_50);
        SW[9] <= 1;                                                                @(posedge
CLOCK_50);//Always reset FSM
        SW[9] <= 0;         KEY[0] <= 1'b0;      repeat(2)   @(posedge CLOCK_50);//player wins
                            KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                            KEY[0] <= 1'b0;      repeat(2)   @(posedge CLOCK_50);
                            KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
```

```
                        KEY[0] <= 1'b0;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b0;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b0;      repeat(2)   @(posedge CLOCK_50);//reset after player wins
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b0;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b0;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b0;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b0;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b0;      repeat(2)   @(posedge CLOCK_50);//hold test
                        KEY[0] <= 1'b0;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b0;      repeat(2)   @(posedge CLOCK_50);
      SW[9] <= 1;                                        @(posedge CLOCK_50);//reset
      SW[9] <= 0;       KEY[0] <= 1'b0;      repeat(2)   @(posedge CLOCK_50);//moves right
      SW[9] <= 1;                                        @(posedge CLOCK_50);//reset
      SW[9] <= 0;       KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);//moves left
      SW[9] <= 1;                                        @(posedge CLOCK_50);//reset
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);//computer wins
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);
      SW[9] <= 1;                                        @(posedge CLOCK_50);//reset
                        KEY[0] <= 1'b1;      repeat(2)   @(posedge CLOCK_50);//both plerys push at the same time

         $stop;// End of simulation
      end
   endmodule

```

```
module centerBird (Clock, Reset, gravity, R, NL, NR, lose, lightOn);
   input logic Clock, Reset;

   input logic gravity, R, NL, NR, lose;

   // when lightOn is true, the center light should be on.
   //if lightOn is true, center light is also true
   output logic lightOn;

   enum {on, off} ps, ns; // on and off states

   // logic for the on and off states for the center light
   always_comb begin
      case(ps)
         on:   if((gravity & ~R) | (~gravity & R)) begin
                  ns = off;
               end
               else begin
                  ns = on;
               end
         off:  if((gravity & NR & ~R) | (R & NL & ~gravity)) begin
                  ns = on;
               end
               else begin
                  ns = off;
               end
      endcase
   end

   always_comb begin
      case (ps)
         on : lightOn = 1'b1; //true if ON state
         off: lightOn = 1'b0; //false if OFF state
      endcase
   end

   always_ff @(posedge Clock) begin
      if(Reset) begin // centerlight stays on if on reset
         ps <= on;
      end
      else if(lose) begin
         ps <= ps;
      end
      else begin
         ps <= ns;
      end
   end

endmodule

module centerBird_testbench();
   logic Clock, Reset, gravity, R, NL, NR, lose;
   logic lightOn;

   centerBird dut (Clock, Reset, gravity, R, NL, NR, lose, lightOn);

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

   // Set up the inputs to the design.  Each line is a clock cycle.
   initial begin

   @(posedge Clock);
      Reset <= 1;                                                                       @(posedge
Clock);
      Reset <= 0; gravity <= 1'b0; R <= 1'b1; NL <= 1'b0; NR <= 1'b1; lose <= 1'b0; repeat(3
)   @(posedge Clock);
      Reset <= 0; gravity <= 1'b1; R <= 1'b0; NL <= 1'b0; NR <= 1'b1; lose <= 1'b0; repeat(3
)   @(posedge Clock);
      Reset <= 1;                                                                       @(posedge
Clock);
        Reset <= 0; gravity <= 1'b1; R <= 1'b1; NL <= 1'b0; NR <= 1'b1; lose <= 1'b1; repeat(3)   @(posedge Clock);
        Reset <= 0; gravity <= 1'b1; R <= 1'b0; NL <= 1'b0; NR <= 1'b1; lose <= 1'b0; repeat(3)   @(posedge Clock);

    @(posedge Clock);

    @(posedge Clock);
        Reset <= 1;                                                                   @(posedge Clock);
        Reset <= 0; gravity <= 1'b0; R <= 1'b1; NL <= 1'b1; NR <= 1'b0; lose <= 1'b0; repeat(3)   @(posedge Clock);
        Reset <= 0; gravity <= 1'b1; R <= 1'b0; NL <= 1'b0; NR <= 1'b1; lose <= 1'b1; repeat(3)   @(posedge Clock);

    @(posedge Clock);
        Reset <= 1;                                                                   @(posedge Clock);
        Reset <= 0; gravity <= 1'b1; R <= 1'b1; NL <= 1'b0; NR <= 1'b1; lose <= 1'b1; repeat(3)   @(posedge Clock);
        Reset <= 0; gravity <= 1'b1; R <= 1'b0; NL <= 1'b0; NR <= 1'b1; lose <= 1'b0; repeat(3)   @(posedge Clock);

    @(posedge Clock);
        Reset <= 1;                                                                   @(posedge Clock);
        Reset <= 0; gravity <= 1'b1; R <= 1'b1; NL <= 1'b1; NR <= 1'b0; lose <= 1'b0; repeat(3)   @(posedge Clock);
        Reset <= 0; gravity <= 1'b0; R <= 1'b0; NL <= 1'b0; NR <= 1'b1; lose <= 1'b1; repeat(3)   @(posedge Clock);

    @(posedge Clock);
        Reset <= 1;                                                                   @(posedge Clock);
        Reset <= 1; gravity <= 1'b0; R <= 1'b1; NL <= 1'b1; NR <= 1'b0; lose <= 1'b0; repeat(3)   @(posedge Clock);
        Reset <= 0; gravity <= 1'b1; R <= 1'b0; NL <= 1'b1; NR <= 1'b0; lose <= 1'b1; repeat(3)   @(posedge Clock);

    @(posedge Clock);

    @(posedge Clock);
        Reset <= 1;                                                                   repeat(3)   @(posedge Clock);

        $stop; // End the simulation.
    end
endmodule

```

```
module normalBird (Clock, Reset, gravity, R, NL, NR, high, low, lose, lightOn);
   input logic Clock, Reset;

   //gravity is true when there is no input from key
   input logic gravity, R, NL, NR, high, low, lose;

   output logic lightOn;

   enum {on, off} ps, ns; // two possible state: on and off states

   //cominational logic for the ON and OFF states
   always_comb begin
      case(ps)
         on: if((~low & gravity & ~R) | (~gravity & R & ~high)) begin //on when not low
and the right key is pressed
                  ns = off;
                  end
                  else if(gravity & ~R & ~low) begin
                     ns = off;
                  end
               else begin
                  ns = on;
               end
         off:  if((gravity & NR & ~R) | (R & NL & ~gravity)) begin
                  ns = on;
               end
               else begin
                  ns = off;
               end
         endcase
      end

   always_comb begin
      case (ps)
         on : lightOn = 1'b1; //Outputs a light turned on if it's in the on state
         off: lightOn = 1'b0; //Outputs a light turned off if it's in the on state
      endcase
   end

   always_ff @(posedge Clock) begin
      if(Reset) begin // if reset, normal light turns off
         ps <= off;
      end
      else if(lose) begin
         ps <= ps;
      end
      else begin
         ps <= ns;
      end
   end
endmodule

module normalBird_testbench();
   logic Clock, Reset, gravity, R, NL, NR, high, low, lose;
   logic lightOn;

   normalBird dut (Clock, Reset, gravity, R, NL, NR, high, low, lightOn, lose);

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

   // Set up the inputs to the design.  Each line is a clock cycle.
   initial begin

                                    @(posedge Clock);
      Reset <= 1;

            @(posedge Clock);
      Reset <= 1;

            @(posedge Clock);
      Reset <= 1;
```

```
            @(posedge Clock);
      Reset <= 0; gravity <= 1'b0; R <= 1'b1; NL <= 1'b0; NR <=1'b1; high <= 1'b0; low <= 1'b1; lose <= 1'b0;  repeat(3)   @(posedge Clock);
      Reset <= 0; gravity <= 1'b1; R <= 1'b0; NL <= 1'b0; NR <=1'b1; high <= 1'b1; low <= 1'b0; lose <= 1'b0;  repeat(3)   @(posedge Clock);
      Reset <= 0; gravity <= 1'b0; R <= 1'b1; NL <= 1'b1; NR <=1'b0; high <= 1'b1; low <= 1'b0; lose <= 1'b0;  repeat(3)   @(posedge Clock);
                  gravity <= 1'b1; R <= 1'b0; NL <= 1'b1; NR <=1'b0; high <= 1'b1; low <= 1'b0; lose <= 1'b1;  repeat(3)   @(posedge Clock);
                  gravity <= 1'b0; R <= 1'b1; NL <= 1'b0; NR <=1'b0; high <= 1'b0; low <= 1'b1; lose <= 1'b0;  repeat(3)   @(posedge Clock);
      Reset <= 1;

            @(posedge Clock);
      Reset <= 0; gravity <= 1'b0; R <= 1'b1; NL <= 1'b1; NR <=1'b1; high <= 1'b1; low <= 1'b0; lose <= 1'b1;  repeat(3)   @(posedge Clock);
                  gravity <= 1'b1; R <= 1'b0; NL <= 1'b1; NR <=1'b0; high <= 1'b0; low <= 1'b1; lose <= 1'b1;  repeat(3)   @(posedge Clock);
                  gravity <= 1'b0; R <= 1'b1; NL <= 1'b0; NR <=1'b1; high <= 1'b0; low <= 1'b1; lose <= 1'b1;  repeat(3)   @(posedge Clock);
                  gravity <= 1'b1; R <= 1'b0; NL <= 1'b0; NR <=1'b1; high <= 1'b1; low <= 1'b0; lose <= 1'b0;  repeat(3)   @(posedge Clock);

      Reset <= 1;
      Reset <= 0; gravity <= 1'b0; R <= 1'b1; NL <= 1'b1; NR <= 1'b0; high <= 1'b1; low <= 1'b0; lose <= 1'b0; repeat(3)   @(posedge Clock);
      Reset <= 0; gravity <= 1'b1; R <= 1'b0; NL <= 1'b1; NR <= 1'b0; high <= 1'b1; low <= 1'b0; lose <= 1'b1; repeat(4)   @(posedge Clock);

                              @(posedge Clock);
      $stop;// End of simulation
   end
endmodule
```

```
module playerInput (Clock, Reset, final_in, final_out);
   //The output for the player's input
   output logic final_out;
   //The inputs for the clock, the reset and the player input
   input logic Clock;
   input logic Reset;
   input logic final_in;

   enum {np, p} ps, ns; //state variables for pressed(p) and not pressed(np)

   always_comb begin
      case(ps)
         np:   if(final_in) begin //If it's not pressed, assign new state
                  final_out = 1'b1;
                  ns = p;
               end else begin
                  final_out = 1'b0;//Else it will remain not pressed
                  ns = np;//Remains in the current state
               end

         p:    if(final_in) begin//If it's pressed, output is 0 to prevent holding
                  final_out = 1'b0;
                  ns = p;//Remains in the same state if it's held
               end else begin
                  final_out = 1'b0;
                  ns = np;//Goes to not pressed state if player lets go of key
               end
      endcase
   end

   always_ff @(posedge Clock) begin
   if(Reset) begin
      ps <= np; //Start with not pressed when reset starts the game
   end else begin
      ps <= ns; //New state becomes present state
   end
end
endmodule

module playerInput_testbench ();
   logic final_out;

   logic Clock;
   logic Reset;
   logic final_in;

   playerInput dut (Clock, Reset, final_in, final_out);

   // Set up a simulated clock to toggle (from low to high or high to low)
   // every 50 time steps
   parameter  CLOCK_PERIOD=100;
   initial begin
      Clock <= 0;
      forever #(CLOCK_PERIOD/2) Clock <= ~Clock;//toggle the clock indefinitely
   end
//Sets up possible combinations for simulations
//Each line represents a clock cycle.
// Simulation sends the state machine into both possible states
   initial begin
                                                   @(posedge Clock);
      Reset <=1;                                   @(posedge Clock);//reset on
      Reset <=1;  final_in <= 16'b1;   repeat(4)   @(posedge Clock);
      Reset <=0;  final_in <= 16'b0;   repeat(4)   @(posedge Clock);//Checks for player presses key
                  final_in <= 16'b1;   repeat(4)   @(posedge Clock);
                  final_in <= 16'b0;   repeat(4)   @(posedge Clock);
                  final_in <= 16'b0;   repeat(4)   @(posedge Clock);
                  final_in <= 16'b1;   repeat(4)   @(posedge Clock);
                  final_in <= 16'b0;   repeat(4)   @(posedge Clock);
                  final_in <= 16'b1;   repeat(4)   @(posedge Clock);//Check for holding
                  final_in <= 16'b1;   repeat(4)   @(posedge Clock);
                  final_in <= 16'b1;   repeat(4)   @(posedge Clock);
                  final_in <= 16'b1;   repeat(4)   @(posedge Clock);
                  final_in <= 16'b1;   repeat(4)   @(posedge Clock);
      $stop; //end the simulation
   end
```

```
endmodule

```

```
module metaStable (Clock, Reset, player_in, player_out);
   //The output for the player's input
   output logic player_out;
   //The inputs for the clock, the reset and the player input
   input logic Clock;
   input logic Reset;
   input logic player_in;

   //Assings the player input as the output based on the clock, unless it's reset it assings 0
   always_ff @(posedge Clock) begin
      if(Reset) begin
         player_out <= 1'b0; //Outputs 0 if it's in reset
      end else begin
         player_out <= player_in; //Outputs the player's input when it's time
      end
   end

endmodule

module metaStable_testbench ();
   logic player_out;

   logic Clock;
   logic Reset;
   logic player_in;

   metaStable dut(Clock, Reset, player_in, player_out);

   // Set up a simulated clock to toggle (from low to high or high to low)
   // every 50 time steps
   parameter  CLOCK_PERIOD=100;
   initial begin
      Clock <= 0;
      forever #(CLOCK_PERIOD/2) Clock <= ~Clock;//toggle the clock indefinitely
   end

   //Sets up possible combinations for simulations
   //Each line represents a clock cycle.
   // Simulation sends the state machine into both possible states
      initial begin

                                             repeat(1) @(posedge Clock);
      Reset <= 1;                            repeat(1) @(posedge Clock); //reset on
      Reset <= 1;    player_in <= 1'b1;      repeat(4) @(posedge Clock); //Test different player inputs
      Reset <= 0;    player_in <= 1'b0;      repeat(4) @(posedge Clock);
                     player_in <= 1'b1;      repeat(4) @(posedge Clock);
                     player_in <= 1'b0;      repeat(4) @(posedge Clock);
                     player_in <= 1'b1;      repeat(4) @(posedge Clock);
      Reset <= 1;    player_in <= 1'b1;      repeat(4) @(posedge Clock); //Test player inputs with reset
                     player_in <= 1'b0;      repeat(4) @(posedge Clock);
                     player_in <= 1'b1;      repeat(4) @(posedge Clock);
                     player_in <= 1'b0;      repeat(4) @(posedge Clock);
      $stop; // End the simulation.
   end
endmodule
```

```
//This module divides the on-board FPGA clock at 50Mhz to
//divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz,
//[24] = 1.5Hz, [25] = 0.75Hz, ...and so on.
module clock_divider (clock, reset, divided_clocks);
      input  logic reset, clock;
      output logic  [31:0]  divided_clocks = 0;
      always_ff @(posedge clock) begin
         divided_clocks <= divided_clocks + 1;
      end
endmodule
```

```systemverilog
// 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 = 0) (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];

    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
```

```systemverilog
module shift_pipes(Clock, Reset, newpattern, lose, out);
   input logic Clock, Reset, lose;
   input logic [15:0] newpattern;
   output logic [15:0] out;

   always_ff @(posedge Clock) begin
      if(Reset) begin
         out <= 16'b0; //16 is a 16 bit result
      end
      else if (lose) begin
         out <= out;
      end
      else begin
         out <= newpattern; //out becomes a new patter
      end
   end

endmodule

module shift_pipes_testbench();
   logic Clock, Reset, lose;
   logic [15:0] newpattern;
   logic [15:0] out;

   shift_pipes dut(Clock, Reset, newpattern, lose, out);

   // Set up a simulated clock to toggle (from low to high or high to low)
   // every 50 time steps
   parameter  CLOCK_PERIOD=100;
   initial begin
      Clock <= 0;
      forever #(CLOCK_PERIOD/2) Clock <= ~Clock;//toggle the clock indefinitely
   end

   //Sets up possible combinations for simulations
   //Each line represents a clock cycle.
   // Simulation sends the state machine into both possible states
   initial begin

                                                                      repeat(1) @(posedge Clock);
      Reset <= 1;                                                     repeat(1) @(posedge Clock); //reset on
      Reset <= 0;    newpattern <= 16'b1100111111111111;             repeat(4) @(posedge Clock);
      Reset <= 1;                                                     repeat(4) @(posedge Clock);
      Reset <= 0;    newpattern <= 16'b1111111100111111;             repeat(4) @(posedge Clock);
                     newpattern <= 16'b1001111111111111;             repeat(4) @(posedge Clock);
                     newpattern <= 16'b1111111111110011;             repeat(4) @(posedge Clock);
                     newpattern <= 16'b1111100111111111;             repeat(4) @(posedge Clock);
                     newpattern <= 16'b1111111100111111;             repeat(4) @(posedge Clock);
                     newpattern <= 16'b1110011111111111;             repeat(4) @(posedge Clock);
                     newpattern <= 16'b1100111111111111;             repeat(4) @(posedge Clock);
      Reset <= 1;                                                     repeat(1) @(posedge Clock); //reset on
      $stop; // End the simulation.
   end
endmodule
```

```
module LFSR2(Clock, Reset, out);

   input logic Clock, Reset;
   output logic [3:0] out;
   logic connection;

   assign connection = ~(out[0] ^ out[1]); //XNOR out1 and out0

   always_ff @(posedge Clock) begin
      if(Reset) begin
         out <= 4'b0000; //4 bit output
      end
      else begin
         out <= {connection, out[3:1]};//DFf moves from connections to 2-3 and replaces Dffs q1 to be
                                       //the result of connection
      end
   end
endmodule


module LFSR2_testbench();
   logic Clock, Reset;
   logic [3:0] out;

   LFSR2 dut (Clock, Reset, out);

   // Set up a simulated clock to toggle (from low to high or high to low)
   // every 50 time steps
   parameter  CLOCK_PERIOD=100;
   initial begin
      Clock <= 0;
      forever #(CLOCK_PERIOD/2) Clock <= ~Clock; //toggle the clock indefinitely
   end

   //Sets up possible combinations for simulations
   //Each line represents a clock cycle.
   initial begin
                              @(posedge Clock);

      Reset <= 1;             @(posedge Clock); // Always reset FSMs at start
      Reset <= 0; repeat(3)   @(posedge Clock);
                              @(posedge Clock);
      Reset <= 1;             @(posedge Clock);
      Reset <= 0; repeat(3)   @(posedge Clock);
                              @(posedge Clock);
      Reset <= 1;             @(posedge Clock);
      Reset <= 0; repeat(3)   @(posedge Clock);
                              @(posedge Clock);
                              @(posedge Clock);
                              @(posedge Clock);
                              @(posedge Clock);
                              @(posedge Clock);
      $stop; // End the simulation.
   end
endmodule
```

```
module gameOver(pipe, bird, lose);
   input logic [15:0] bird, pipe;
   output logic lose;

   logic hit0, hit1, hit2, hit3, hit4, hit5;
   logic hit6, hit7, hit8, hit9, hit10, hit11;
   logic hit12, hit13, hit14, hit15;

   assign hit0 = pipe[0] & bird[0];
   assign hit1 = pipe[1] & bird[1];
   assign hit2 = pipe[2] & bird[2];
   assign hit3 = pipe[3] & bird[3];
   assign hit4 = pipe[4] & bird[4];
   assign hit5 = pipe[5] & bird[5];
   assign hit6 = pipe[6] & bird[6];
   assign hit7 = pipe[7] & bird[7];
   assign hit8 = pipe[8] & bird[8];
   assign hit9 = pipe[9] & bird[9];
   assign hit10 = pipe[10] & bird[10];
   assign hit11 = pipe[11] & bird[11];
   assign hit12 = pipe[12] & bird[12];
   assign hit13 = pipe[13] & bird[13];
   assign hit14 = pipe[14] & bird[14];
   assign hit15 = pipe[15] & bird[15];

   //you losse if any of these logic combinations happen
   assign lose = hit0 | hit1 | hit2 | hit3 | hit4 | hit5 |
                 hit6 | hit7 | hit8 | hit9 | hit10 | hit11 |
                 hit12 | hit13 | hit14| hit15;
endmodule

module gameOver_testbench ();
   logic [15:0] bird, pipe;
   logic lose;

gameOver dut (bird, pipe, lose);

   initial begin
      bird = 16'b0000000000010000 ; pipe = 16'b0011111111111111 ; #10; //hit
      bird = 16'b0000000000000001 ; pipe = 16'b1111111111111100 ; #10; //pass
      bird = 16'b0000000010000000 ; pipe = 16'b1001111111111111 ; #10; //hit
      bird = 16'b0000010000000000 ; pipe = 16'b1111100111111111 ; #10; //pass
      bird = 16'b0000000000000010 ; pipe = 16'b1111001111111111 ; #10; //hit
   end
endmodule
```

```
module clock_counter (Clock, Reset, out);
   input logic Clock, Reset;
   output logic out;

   //state variables for present states and new states of pipe gaps
   enum {pattern, gap1, gap2, gap3, gap4} ps, ns;

// This logic describes all the possible state transitions from ps to ns
always_comb begin
    case (ps)
            pattern:    begin
                           out = 1'b1;
                           ns = gap1;
                        end
            gap1:       begin
                           out = 1'b0;
                           ns = gap2;
                        end
            gap2:       begin
                           out = 1'b0;
                           ns = gap3;
                        end
            gap3:       begin
                           out = 1'b0;
                           ns = gap4;
                        end
            gap4:       begin
                           out = 1'b0;
                           ns = pattern;
                        end
      endcase
   end

   // D Flip Flop implementation (DFFs)
   always_ff @(posedge Clock) begin
      if (Reset) begin
            ps <= pattern;
      end
      else begin
         ps <= ns; //Otherwise, advances to next state in state diagram
      end
   end
endmodule

   module clock_counter_testbench();
   logic Clock, Reset;
   logic out;

   clock_counter dut(Clock, Reset, out);

   // Set up a simulated clock to toggle (from low to high or high to low)
   // every 50 time steps
   parameter  CLOCK_PERIOD=100;
   initial begin
      Clock <= 0;
      forever #(CLOCK_PERIOD/2) Clock <= ~Clock;//toggle the clock indefinitely
   end

   //Sets up possible combinations for simulations
   //Each line represents a clock cycle.
    initial begin

                     repeat(2) @(posedge Clock);
      Reset <= 1;    repeat(3) @(posedge Clock);
      Reset <= 0;    repeat(3) @(posedge Clock);
                     repeat(3) @(posedge Clock);
                     repeat(3) @(posedge Clock);
                     repeat(3) @(posedge Clock);
      Reset <= 1;    repeat(3) @(posedge Clock);
      Reset <= 0;    repeat(3) @(posedge Clock);
                     repeat(3) @(posedge Clock);
                     repeat(3) @(posedge Clock);
                     repeat(3) @(posedge Clock);
      $stop; // End the simulation.
   end
endmodule
```

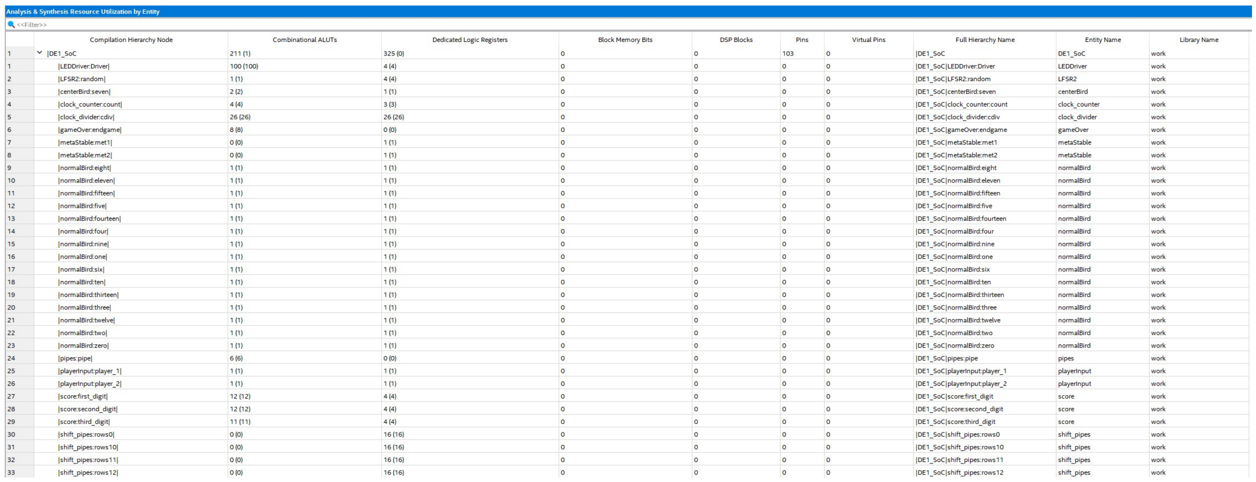
```
module score(Clock, Reset, lose, increaseIN, increaseout, HEX);
   input logic Clock, Reset;
   input logic lose, increaseIN;

   output logic [6:0] HEX;
   output logic increaseout;

   // State variables, resets to 0 after 9
   enum { zero, one, two, three, four, five, six, seven, eight, nine } ps, ns;

   // Next State logic
   always_comb begin
      case (ps)
         zero: if(increaseIN) begin
                  HEX = 7'b0000110; //1
                  increaseout = 1'b0;
                  ns = one;
               end
               else begin
                  HEX = 7'b0111111;
                  increaseout = 1'b0; //0
                  ns = zero;
               end
         one:  if(increaseIN) begin
                  HEX = 7'b1011011; //2
                  increaseout = 1'b0;
                  ns = two;
               end
               else begin
                  HEX = 7'b0000110; //1
                  increaseout = 1'b0;
                  ns = one;
               end
         two:  if(increaseIN) begin
                  HEX = 7'b1001111; //3
                  increaseout = 1'b0;
                  ns = three;
               end
               else begin
                  HEX = 7'b1011011;//2
                  increaseout = 1'b0;
                  ns = two;
               end
         three:   if(increaseIN) begin
                  HEX = 7'b1100110; //4
                  increaseout = 1'b0;
                  ns = four;
               end
               else begin
                  HEX = 7'b1001111; //3
                  increaseout = 1'b0;
                  ns = three;
               end
         four: if(increaseIN) begin
                  HEX = 7'b1101101;//5
                  increaseout = 1'b0;
                  ns = five;
               end
               else begin
                  HEX = 7'b1100110; //4
                  increaseout = 1'b0;
                  ns = four;
               end
         five: if(increaseIN) begin
                  HEX = 7'b1111101; //6
                  increaseout = 1'b0;
                  ns = six;
               end
               else begin
                  HEX = 7'b1101101; //5
                  increaseout = 1'b0;
                  ns = five;
               end
         six: if(increaseIN) begin
                  HEX = 7'b0000111; //7
                  increaseout = 1'b0;
```

```
                     ns = seven;
                  end
                  else begin
                     HEX = 7'b1111101; //6
                     increaseout = 1'b0;
                     ns = six;
                  end
            seven: if(increaseIN) begin
                     HEX = 7'b1111111; //8
                     increaseout = 1'b0;
                     ns = eight;
               end
               else begin
                  HEX = 7'b0000111; //7
                  increaseout = 1'b0;
                  ns = seven;
               end
            eight: if(increaseIN) begin
                  HEX = 7'b1101111; //9
                  increaseout = 1'b0;
                  ns = nine;
               end
               else begin
                  HEX = 7'b1111111; //8
                  increaseout = 1'b0;
                  ns = eight;
               end
            nine: if(increaseIN) begin
                  HEX = 7'b0111111; //0
                  increaseout = 1'b1;
                  ns = zero;
               end
               else begin
                  HEX = 7'b1101111; //9
                  increaseout = 1'b0;
                  ns = nine;
               end
      endcase
   end

      always_ff @(posedge Clock) begin
         if(Reset) begin
            ps <= zero;
         end
         else if(lose) begin
            ps <= ps;
         end
         else begin
            ps <= ns;
         end
      end
   endmodule

   module score_testbench();
       logic Clock, Reset;
       logic lose, increaseIN;

       logic [6:0] HEX;
       logic increaseout;


   score dut(.Clock, .Reset, .lose, .increaseIN, .increaseout, .HEX);

      // Set up a simulated clock to toggle (from low to high or high to low)
      // every 50 time steps
      parameter  CLOCK_PERIOD=100;
      initial begin
      Clock <= 0;
      forever #(CLOCK_PERIOD/2) Clock <= ~Clock; // Forever toggle the clock
   end

   //Sets up possible combinations for simulations
   //Each line represents a clock cycle.
   initial begin
                                                                    @(posedge
 Clock);
```

```
      Reset <= 1;                                                                  @(posedge Clock);
      Reset <= 0; increaseIN <= 1'b0; lose <= 1'b0;   repeat(3)                  @(posedge Clock);
      Reset <= 0; increaseIN <= 1'b1; lose <= 1'b0;   repeat(3)                  @(posedge Clock);
      Reset <= 1;                                                                  @(posedge Clock);
      Reset <= 0; increaseIN <= 1'b0; lose <= 1'b0;   repeat(3)                  @(posedge Clock);
      Reset <= 0; increaseIN <= 1'b0; lose <= 1'b1;   repeat(3)                  @(posedge Clock);
      Reset <= 1;                                                                  @(posedge Clock);
      Reset <= 0; increaseIN <= 1'b0; lose <= 1'b1;   repeat(3)                  @(posedge Clock);
      Reset <= 0; increaseIN <= 1'b1; lose <= 1'b0;   repeat(3)                  @(posedge Clock);
      Reset <= 1;                                                                  @(posedge Clock);
      Reset <= 0;                                                                  @(posedge Clock);
      Reset <= 0; increaseIN <= 1'b1; lose <= 1'b1;   repeat(3)                  @(posedge Clock);
                                                                                   @(posedge Clock);

      $stop; // End the simulation.
   end
endmodule

```

```
module pipes(LFSR2_out, pattern, generate_clock);
   input logic [3:0] LFSR2_out;
   output logic [15:0] pattern;
   input logic generate_clock;

   always_comb begin
      case (LFSR2_out)
         4'b0000:
            if(generate_clock) begin
               pattern = 16'b1111110011111111; end // 0
            else begin
               pattern = 16'b0;
            end

         4'b0001:
         if(generate_clock) begin
            pattern = 16'b1001111111111111; end // 1
         else begin
            pattern = 16'b0;
            end

         4'b0010:
         if(generate_clock) begin
            pattern = 16'b1100111111111111; end// 2
         else begin
            pattern = 16'b0;
            end

         4'b0011:
         if(generate_clock) begin
            pattern = 16'b1110011111111111; end// 3
         else begin
            pattern = 16'b0;
            end

         4'b0100:
         if(generate_clock) begin
            pattern = 16'b1111001111111111; end// 4
         else begin
            pattern = 16'b0;
            end

         4'b0101:
         if(generate_clock) begin
            pattern = 16'b1111100111111111; end// 5
         else begin
            pattern = 16'b0;
            end

         4'b0110:
         if(generate_clock) begin
            pattern = 16'b1111110011111111; end// 6
         else begin
            pattern = 16'b0;
            end

         4'b0111:
         if(generate_clock) begin
            pattern = 16'b1111111001111111; end// 7
         else begin
            pattern = 16'b0;
            end

         4'b1000:
         if(generate_clock) begin
            pattern = 16'b1111111100111111; end// 8
         else begin
            pattern = 16'b0;
            end

         4'b1001:
         if(generate_clock) begin
            pattern = 16'b1111111110011111; end// 9
         else begin
            pattern = 16'b0;
            end

            4'b1010:
            if(generate_clock) begin
               pattern = 16'b1111111111001111; end// 10
            else begin
               pattern = 16'b0;
               end

            4'b1011:
            if(generate_clock) begin
               pattern = 16'b1111111111100111; end// 11
            else begin
               pattern = 16'b0;
               end

            4'b1100:
            if(generate_clock) begin
               pattern = 16'b1111111111110011; end// 12
            else begin
               pattern = 16'b0;
               end

            4'b1101:
            if(generate_clock) begin
               pattern = 16'b1111111111111001; end // 13
            else begin
               pattern = 16'b0;
               end

            4'b1110:
            if(generate_clock) begin
               pattern = 16'b1111111111111100; end// 14
            else begin
               pattern = 16'b0;
               end

            default: begin
               pattern = 16'b0000000000000000;  // default
            end
         endcase
      end
   endmodule

   module pipes_testbench();
    logic [3:0] LFSR2_out;
    logic [15:0] pattern;
    logic generate_clock;

    pipes dut(LFSR2_out ,pattern);

    integer i;
    integer j;
    initial begin
      generate_clock = 1'b1;
      for(i = 0; i < 16; i++)
         begin LFSR2_out = i; #10;
      end

      generate_clock = 1'b0;
      for(j = 0; j < 16; j++)
         begin LFSR2_out = j; #10;
      end

   end
endmodule



```

Resource Utilization:

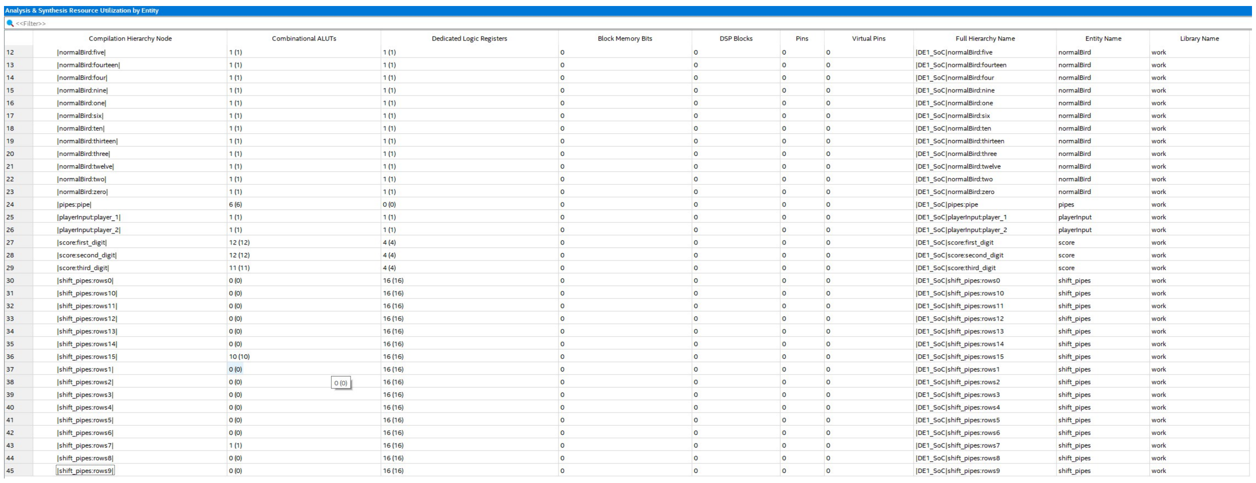


Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins	Full Hierarchy Name	Entity Name	Library Name
1	|DE1_SoC	211 (1)	325 (0)	0	0	103	0	|DE1_SoC	DE1_SoC	work
1	|LEDDriver:Driver|	100 (100)	4 (4)	0	0	0	0	|DE1_SoC|LEDDriver:Driver	LEDDriver	work
2	|LFSR2:random|	1 (1)	4 (4)	0	0	0	0	|DE1_SoC|LFSR2:random	LFSR2	work
3	|centerBird:seven|	2 (2)	1 (1)	0	0	0	0	|DE1_SoC|centerBird:seven	centerBird	work
4	|clock_counter:count|	4 (4)	3 (3)	0	0	0	0	|DE1_SoC|clock_counter:count	clock_counter	work
5	|clock_divider:cdiv|	26 (26)	26 (26)	0	0	0	0	|DE1_SoC|clock_divider:cdiv	clock_divider	work
6	|gameOver:endgame|	8 (8)	0 (0)	0	0	0	0	|DE1_SoC|gameOver:endgame	gameOver	work
7	|metaStable:met1|	0 (0)	1 (1)	0	0	0	0	|DE1_SoC|metaStable:met1	metaStable	work
8	|metaStable:met2|	0 (0)	1 (1)	0	0	0	0	|DE1_SoC|metaStable:met2	metaStable	work
9	|normalBird:eight|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:eight	normalBird	work
10	|normalBird:eleven|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:eleven	normalBird	work
11	|normalBird:fifteen|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:fifteen	normalBird	work
12	|normalBird:five|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:five	normalBird	work
13	|normalBird:fourteen|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:fourteen	normalBird	work
14	|normalBird:four|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:four	normalBird	work
15	|normalBird:nine|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:nine	normalBird	work
16	|normalBird:one|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:one	normalBird	work
17	|normalBird:six|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:six	normalBird	work
18	|normalBird:ten|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:ten	normalBird	work
19	|normalBird:thirteen|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:thirteen	normalBird	work
20	|normalBird:three|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:three	normalBird	work
21	|normalBird:twelve|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:twelve	normalBird	work
22	|normalBird:two|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:two	normalBird	work
23	|normalBird:zero|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:zero	normalBird	work
24	|pipes:pipe|	6 (6)	0 (0)	0	0	0	0	|DE1_SoC|pipes:pipe	pipes	work
25	|playerInput:player_1|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|playerInput:player_1	playerInput	work
26	|playerInput:player_2|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|playerInput:player_2	playerInput	work
27	|score:first_digit|	12 (12)	4 (4)	0	0	0	0	|DE1_SoC|score:first_digit	score	work
28	|score:second_digit|	12 (12)	4 (4)	0	0	0	0	|DE1_SoC|score:second_digit	score	work
29	|score:third_digit|	11 (11)	4 (4)	0	0	0	0	|DE1_SoC|score:third_digit	score	work
30	|shift_pipes:rows0|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows0	shift_pipes	work
31	|shift_pipes:rows10|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows10	shift_pipes	work
32	|shift_pipes:rows11|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows11	shift_pipes	work
33	|shift_pipes:rows12|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows12	shift_pipes	work

Figure 5. Screenshot of the Resource Utilization



Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins	Full Hierarchy Name	Entity Name	Library Name
12	|normalBird:five|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:five	normalBird	work
13	|normalBird:fourteen|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:fourteen	normalBird	work
14	|normalBird:four|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:four	normalBird	work
15	|normalBird:nine|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:nine	normalBird	work
16	|normalBird:one|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:one	normalBird	work
17	|normalBird:six|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:six	normalBird	work
18	|normalBird:ten|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:ten	normalBird	work
19	|normalBird:thirteen|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:thirteen	normalBird	work
20	|normalBird:three|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:three	normalBird	work
21	|normalBird:twelve|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:twelve	normalBird	work
22	|normalBird:two|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:two	normalBird	work
23	|normalBird:zero|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|normalBird:zero	normalBird	work
24	|pipes:pipe|	6 (6)	0 (0)	0	0	0	0	|DE1_SoC|pipes:pipe	pipes	work
25	|playerInput:player_1|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|playerInput:player_1	playerInput	work
26	|playerInput:player_2|	1 (1)	1 (1)	0	0	0	0	|DE1_SoC|playerInput:player_2	playerInput	work
27	|score:first_digit|	12 (12)	4 (4)	0	0	0	0	|DE1_SoC|score:first_digit	score	work
28	|score:second_digit|	12 (12)	4 (4)	0	0	0	0	|DE1_SoC|score:second_digit	score	work
29	|score:third_digit|	11 (11)	4 (4)	0	0	0	0	|DE1_SoC|score:third_digit	score	work
30	|shift_pipes:rows0|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows0	shift_pipes	work
31	|shift_pipes:rows10|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows10	shift_pipes	work
32	|shift_pipes:rows11|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows11	shift_pipes	work
33	|shift_pipes:rows12|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows12	shift_pipes	work
34	|shift_pipes:rows13|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows13	shift_pipes	work
35	|shift_pipes:rows14|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows14	shift_pipes	work
36	|shift_pipes:rows15|	10 (10)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows15	shift_pipes	work
37	|shift_pipes:rows1|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows1	shift_pipes	work
38	|shift_pipes:rows2|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows2	shift_pipes	work
39	|shift_pipes:rows3|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows3	shift_pipes	work
40	|shift_pipes:rows4|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows4	shift_pipes	work
41	|shift_pipes:rows5|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows5	shift_pipes	work
42	|shift_pipes:rows6|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows6	shift_pipes	work
43	|shift_pipes:rows7|	1 (1)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows7	shift_pipes	work
44	|shift_pipes:rows8|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows8	shift_pipes	work
45	|shift_pipes:rows9|	0 (0)	16 (16)	0	0	0	0	|DE1_SoC|shift_pipes:rows9	shift_pipes	work

Figure 6. Screenshot of the Resource Utilization Continued

A total of 536 resources were used in this project