Lingli Zeng

Professor Hussein

EE 271 Lab7

December 2nd, 2017

Lab7: Final Project—Flappy Bird

Abstract:

The purpose of this lab is to utilize my knowledge to implement a rather complicated system that has more functions. In my case, I implemented the game flappy bird. To build the game, I first wrote modules to randomly generate a 3-bit number to decide the shape of the green pipe. The pipe was then shifted column by column to generate a continuous game canvas. After that, I further wrote functions to control the movement of the bird, the game status, and display the user score. All modules are tested through modelsim and the final product was run on the FPGA board.

Introduction:

The game flappy bird was first implemented by Vietnamese programmer Dong Nguyen. The players in this game try to control the bird to avoid hitting on the moving pipes and falling on the floor. The players gain one point every time the bird passes a pipe, and lose the game once the bird hit a pipe or fall on the floor.

Materials:

- Altera's Terasic DE1-SoC Development Board
- DE1 Power Cord
- DE1 USB Cable
- PC with Quartus Prime Lite
- LED Matrix Board

Procedures:

Before implementing the lab, I drew a block diagram of the whole system which is shown below. The functionality of each part will be elaborated in the later part of this lab.

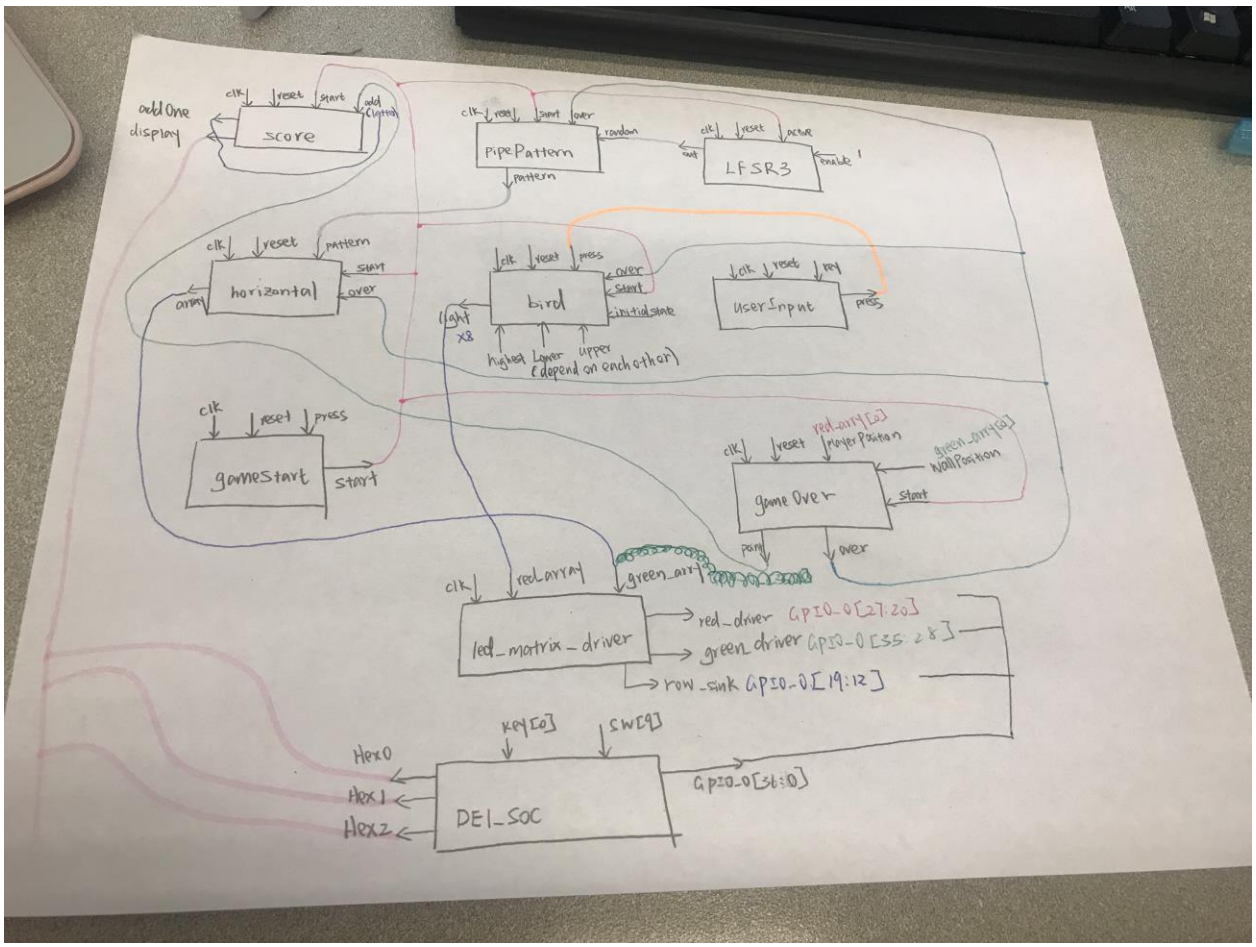


Figure 1: Block Diagram

After planning out design, I wrote the module userInput to avoid any possible metastability when the user presses the key. A snapshot of the userInput function is shown below:

```
1  // This module solves metastability
2  module userInput (clk, reset, key, press);
3     input logic clk, reset;
4     input logic key;
5     output logic press;
6     logic ps, ns;
7     // next state is the key press
8     always_comb begin
9        ns = key;
10    end
11
12    assign press = ps;
13
14
15    // pass input through flipflop
16    always @(posedge clk) begin
17       if (reset) ps <= 1'b0;
18       else ps <= ns;
19    end
20 endmodule
21
```

Figure 2: userInput Snapshot

Next, I used the LFSR system to generate a 3-bit input. The randomly generated number will decide which pattern of the wall will the system display. A snapshot of the LFSR module is shown below:

```
1  // generate random number
2  module LFSR3 (clk, reset, active, enable, out);
3     input logic clk, reset, enable, active;
4     output logic [2:0] out;
5     wire feedback;
6     assign feedback = !(out[2]^out[1]);
7      control the speed of the clock
8     logic[7:0] control = 0;
9
10    always @(posedge clk) begin
11       if (reset | ~active) begin
12          out <= 3'b0;
13          control <= 0;
14       end else if(enable) begin
15          if (control == 1) begin
16             out <= {out[1], out[0], feedback};
17          end
18          control <= control + 1;
19       end
20    end
21 endmodule
22
```

Figure 3:LFSR3 snapshot

After the random number is generated, I later used the numbers to generate patterns of the walls. The logic is implemented in the module pipePattern, which is shown below:

```
module pipePattern(clk, reset, start, over, random, pattern);

    input logic clk, reset, start, over;
    input logic [2:0] random;
    // this variable decides how much gap is between two walls
    logic [1:0] gap;
    logic [7:0] ps, ns;
    logic [7:0] count;
    output logic [7:0] pattern;


    always_comb
        if (over)
            ns = ps;
        else begin
            if (gap == 0)
                case(random)
                    // select the wall patterns here
                    3'b000: ns = 8'b00001111;
                    3'b001: ns = 8'b11001111;
                    3'b010: ns = 8'b11100111;
                    3'b011: ns = 8'b11110011;
                    3'b100: ns = 8'b11110001;
                    3'b101: ns = 8'b10001111;
                    3'b110: ns = 8'b11000111;
                    3'b111: ns = 8'b11100011;
                    default: ns = 8'b00000000;
                endcase
            else
                ns = 8'b00000000;
        end

    assign pattern = ps;
    // flip flop
    always_ff @(posedge clk)
    // reset
        if (reset | ~start) begin
            gap <= 0;
            ps <= 8'b00000000;
            count <= 8'b00000000;
        end
        else begin
            // count controls the speed of the update
            // the state update everytime count is 0
            if (count == 0) begin

                ps <= ns;
                gap <= gap + 2'b01;
            end
            count <= count + 8'b00000001;
        end

endmodule
```

Figure 4: pipePattern snapshot

Once the patterns of the wall are successfully generated, I further implemented a module to continuously shift the walls horizontally. This logic is implemented in the module horizontal which is shown below:

```
module horizontal(clk, reset, pattern, start, over, array);
   input logic clk, reset, start, over;
   input logic [7:0] pattern;
   // this variable controls the update speed
   logic [7:0] control;
   logic [7:0] [7:0] ps, ns;
   // this is the entire LED matrix output
   output logic [7:0] [7:0] array;

   always_comb
      if (over)
         ns [7:0] = ps[7:0];
      else begin
      // shift by one column and push in the newly generated pattern
         ns[6:0] = ps[7:1];
         ns[7] = pattern;
      end

   assign array = ps;

   always_ff @(posedge clk)
      if (reset | ~start) begin
         control <= 0;
         ps <= 0;
      end
      else begin
      // controls update speed
      // update when cpmtrol is 0
         if (control == 0) ps <= ns;
         control <= control + 8'b00000001;
      end
endmodule
```

Figure 5: horizontal snapshot

After the above work to generate the game canvas, I moved on to control the position of the bird. For the bird module, instead of checking the light column by column, I focused on one single light, and called this module in the top level many times to check the status of each LED. A snapshot of the module bird is shown below:

```
module bird(clk, reset, press, over, start, initialState, upper, lower, light, highest);
   input logic clk, reset, press, over, start, initialState, upper, lower, highest;
   logic ns;
   output logic light;
   // update speed controler
   logic [6:0] count;

   always_comb
      if (over)
         ns = light;
      else begin
         case(light)
            // when currently light is off
            // if lower light is on and key pressed --> light on
            // if upper light is on and key not pressed --> light on
            1'b0: if ((press & lower) | (~press & upper)) ns = 1'b1;
                  else ns = 1'b0;
            // when currnet light is on
            // if this light is the upper limit and the key is pressed --> light is on
            1'b1: if (highest & press) ns = 1'b1;
                  else ns = 1'b0;
         endcase
      end

   always_ff @(posedge clk)
   // reset the game
      if (reset | !start) begin
         light <= initialState;
         count <= 6'b000000;
      end
      // update the status
      else begin
         if (count == 0) light <= ns;
         count <= count + 6'b000001;
      end
   end
endmodule
```

Figure 6: bird snapshot

Now that the bird and wall positions are all generated, it is time to decide the status of the game. Controlling the starting of the game is easy. The game simply starts when the key is pressed. The game will remain before the user loses the game. The logic gameStart is shown below:

```
module gameStart(clk, reset, press, start);
   input logic clk, reset, press;
   logic ps, ns;
   output logic start;

   // The game starts by pressing the key, and stays on
   always_comb
      ns = ps | press;

   // update the status
   always_ff @(posedge clk)
      if (reset)
         ps <= 1'b0;
      else
         ps <= ns;

   assign start = ps;

endmodule
```

Figure 7: gameStart snapshot

The player loses the game when the bird hits a pipe or hit the floor. The gameOver module is implemented as shown below:

```
module gameOver(clk, reset, playerPosition, wallPosition, start, point, over);
   // basic inputs
   input logic clk, reset, start;
   input logic [7:0] playerPosition, wallPosition;

   // positions of the wall and the bird
   logic nsplayerPosition, psplayerPosition;
   logic [7:0] nswall, pswall;
   output logic over, point;

   always_comb begin
      // below are cases when the player looses the game
      nsplayerPosition = playerPosition[0] // this is when the bird hits the floor
               // below is the cases when birds hits the pipe
               | (playerPosition[7] & wallPosition[7])
               | (playerPosition[6] & wallPosition[6])
               | (playerPosition[5] & wallPosition[5])
               | (playerPosition[4] & wallPosition[4])
               | (playerPosition[3] & wallPosition[3])
               | (playerPosition[2] & wallPosition[2])
               | (playerPosition[1] & wallPosition[1]);
      nswall = wallPosition;
   end

   assign over = psplayerPosition;
   // when the bird jump over one wall, the player gets 1 point
   assign point = ~(pswall == 8'b00000000) & (wallPosition == 8'b00000000);

   always_ff @(posedge clk)
      // reset the game
      if (reset | !start) begin
         psplayerPosition <= 1'b0;
         pswall <= 8'b00000000;
      end
      // update
      else begin
         psplayerPosition <= nsplayerPosition;
         pswall <= nswall;
      end
endmodule
```

Figure 8: gameOver snapshot

After the system knows how to judge the game status, it can now report the score of the player on the hex display. The function score report is implemented with a simple state machine shown below:

```
module score(clk, reset, start, add, display, addOne);
   // basic input
   input logic clk, reset, start, add;
   logic [6:0] ps, ns;
   // whether to add one point
   output logic addOne;
   // hex display
   output logic [6:0] display;

   always_comb
      if (add)
         case(ps)
            7'b1000000: ns = 7'b1111001; // 0-1
            7'b1111001: ns = 7'b0100100; // 1-2
            7'b0100100: ns = 7'b0110000; // 2-3
            7'b0110000: ns = 7'b0011001; // 3-4
            7'b0011001: ns = 7'b0010010; // 4-5
            7'b0010010: ns = 7'b0000010; // 5-6
            7'b0000010: ns = 7'b1111000; // 6-7
            7'b1111000: ns = 7'b0000000; // 7-8
            7'b0000000: ns = 7'b0010000; // 8-9
            7'b0010000: ns = 7'b1000000; // 9-0
            default ns = 7'b1111001; // 1
         endcase
      else
         ns = ps;

   assign display[6:0] = ps[6:0];
   assign addOne = (ps[6:0] == 7'b0010000) & add; // nine and increment
   // update
   always_ff @(posedge clk)
      if(reset | !start)
         ps <= 7'b1000000;
      else
         ps <= ns;

endmodule
```

Figure 9: score snapshot

After the main logic is done, I found some source code to connect my implementation to the LED board. A snapshot of this module is shown below:

```
module led_matrix_driver (clock, red_array, green_array, red_driver, green_driver, row_sink);
  input clock;
  input [7:0][7:0] red_array, green_array;
  output reg [7:0] red_driver, green_driver, row_sink;
  reg [2:0] count;

  always @(posedge clock)
      count <= count + 3'b001;
  always @(*)
      case (count)
         3'b000: row_sink = 8'b11111110;
         3'b001: row_sink = 8'b11111101;
         3'b010: row_sink = 8'b11111011;
         3'b011: row_sink = 8'b11110111;
         3'b100: row_sink = 8'b11101111;
         3'b101: row_sink = 8'b11011111;
         3'b110: row_sink = 8'b10111111;
         3'b111: row_sink = 8'b01111111;
      endcase

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

Figure 10: led_matrix_driver snapshot

Last but not the least, I connected all my modules in the top-level entity. The connection of the top level is explained in the block diagram. The detailed implementation is shown below:

```
module DE1_SoC (CLOCK_50, HEX0, HEX1, HEX2, KEY, LEDR, SW, GPIO_0);
    input logic CLOCK_50; // 50MHz CLOCK_50.
    output logic [6:0] HEX0, HEX1, HEX2;
    output logic [9:0] LEDR;
    input logic [3:0] KEY; // True when not pressed, False when pressed
    input logic [9:0] SW;
    output logic [35:0] GPIO_0;
    logic w;
    logic start;
    logic over;
    logic player;
    logic [7:0][7:0] red_array, green_array;
    logic[7:0] pattern;
    logic reset;
    logic [2:0]  randomNum;

    // Generate clk off of CLOCK_50, whichClock picks rate.
    logic [31:0] clk;
    parameter whichClock = 15;
    clock_divider cdiv(CLOCK_50, clk);

    always_ff @(posedge clk[whichClock])
      w <= SW[9];
    always_ff @(posedge clk[whichClock])
      reset <= w; // Reset when SW[9] is pressed.



    userInput  user(.reset(reset), .clk(CLOCK_50), .key(~KEY[0]), .press(player));
    gameStart gameActivate(.clk(clk[whichClock]), .reset, .press(player), .start);

    bird l0 (.clk(clk[whichClock]), .reset, .press(player), .over(over), .start, .initialState(1'b0), .upper(1'b0), .lower(red_array[0][6]), .light(red_array[0][7]), .highest(1'b1));
    bird l1 (.clk(clk[whichClock]), .reset, .press(player), .over(over), .start, .initialState(1'b0), .upper(red_array[0][7]), .lower(red_array[0][5]), .light(red_array[0][6]), .highest(1'b0));
    bird l2 (.clk(clk[whichClock]), .reset, .press(player), .over(over), .start, .initialState(1'b0), .upper(red_array[0][6]), .lower(red_array[0][4]), .light(red_array[0][5]), .highest(1'b0));
    bird l3 (.clk(clk[whichClock]), .reset, .press(player), .over(over), .start, .initialState(1'b0), .upper(red_array[0][5]), .lower(red_array[0][3]), .light(red_array[0][4]), .highest(1'b0));
    bird l4 (.clk(clk[whichClock]), .reset, .press(player), .over(over), .start, .initialState(1'b1), .upper(red_array[0][4]), .lower(red_array[0][2]), .light(red_array[0][3]), .highest(1'b0));
    bird l5 (.clk(clk[whichClock]), .reset, .press(player), .over(over), .start, .initialState(1'b0), .upper(red_array[0][3]), .lower(red_array[0][1]), .light(red_array[0][2]), .highest(1'b0));
    bird l6 (.clk(clk[whichClock]), .reset, .press(player), .over(over), .start, .initialState(1'b0), .upper(red_array[0][2]), .lower(red_array[0][0]), .light(red_array[0][1]), .highest(1'b0));
    bird l7 (.clk(clk[whichClock]), .reset, .press(player), .over(over), .start, .initialState(1'b0), .upper(red_array[0][1]), .lower(1'b0), .light(red_array[0][0]), .highest(1'b0));

    assign red_array[7][7:0] = 8'b00000000;
    assign red_array[6][7:0] = 8'b00000000;
    assign red_array[5][7:0] = 8'b00000000;
    assign red_array[4][7:0] = 8'b00000000;
    assign red_array[3][7:0] = 8'b00000000;
    assign red_array[2][7:0] = 8'b00000000;
    assign red_array[1][7:0] = 8'b00000000;

    LFSR3 random(.clk(clk[whichClock]), .reset(reset), .active(start), .enable(1), .out(randomNum));
    pipePattern pipe0(.clk(clk[whichClock]), .reset(reset), .start(start), .over(over), .random(randomNum), .pattern(pattern));
    horizontal pipe1(.clk(clk[whichClock]), .reset(reset), .pattern(pattern), .start(start), .over(over), .array(green_array));

    logic [7:0] rowsink, redDriver, greenDriver;
    assign GPIO_0 [35:28] = greenDriver;
    assign GPIO_0 [27:20] = redDriver;
    assign GPIO_0 [19:12] = rowsink;
    led_matrix_driver led_array(.clock(clk[10]), .red_array, .green_array, .red_driver(redDriver), .green_driver(greenDriver), .row_sink(rowsink));
    logic point;
    gameOver isOver(.clk(clk[whichClock]), .reset(reset), .playerPosition(red_array[0]), .wallPosition(green_array[0]), .start, .point, .over);

    logic add0, add1, add2;
    score display0(.clk(clk[whichClock]), .reset, .start, .add(point), .display(HEX0[6:0]), .addOne(add0));
    score display1(.clk(clk[whichClock]), .reset, .start, .add(add0), .display(HEX1[6:0]), .addOne(add1));
    score display2(.clk(clk[whichClock]), .reset, .start, .add(add1), .display(HEX2[6:0]), .addOne(add2));
endmodule

   // 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;

   initial
      divided_clocks <= 0;

   always_ff @(posedge clock)
      divided_clocks <= divided_clocks + 1;
endmodule
```

Figure 11: DE1_SoC

After writing all the code, I watched a tutorial to connect the LED board to my FPGA. The connection is shown below:

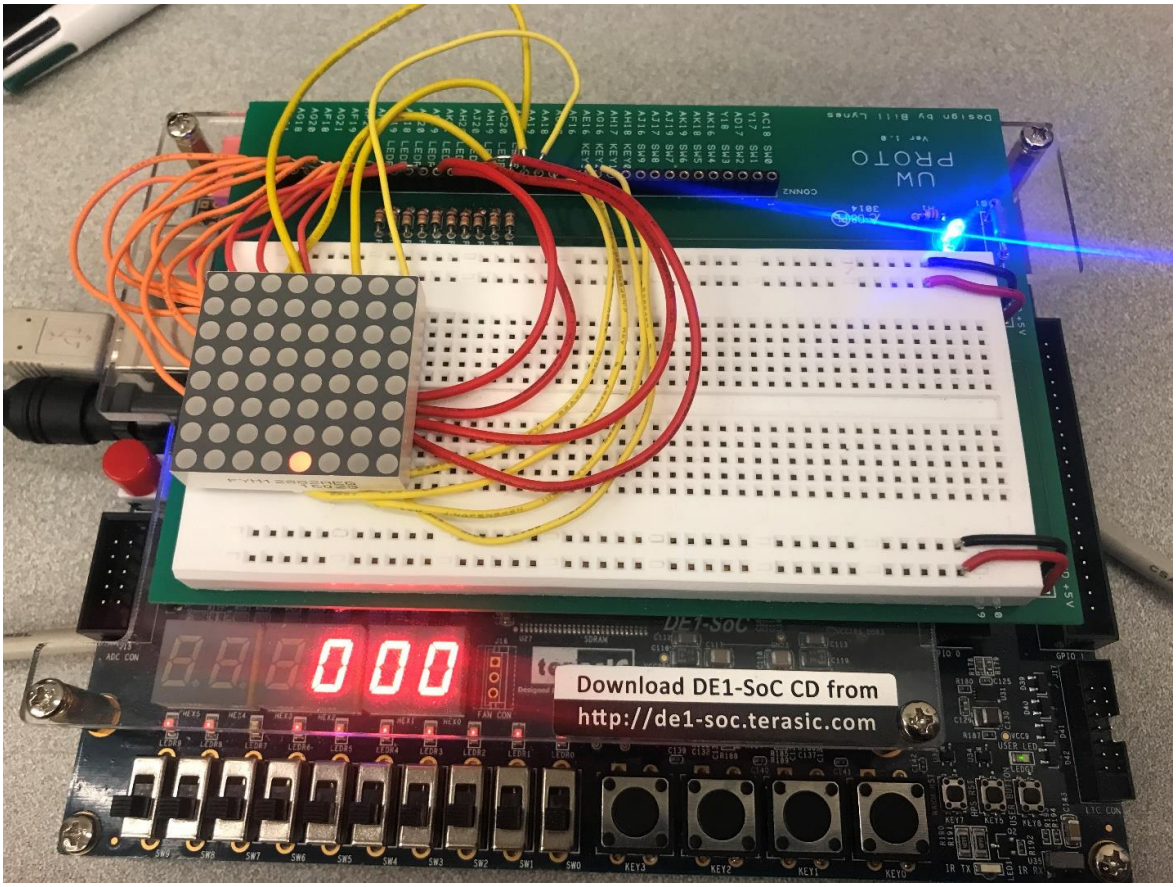
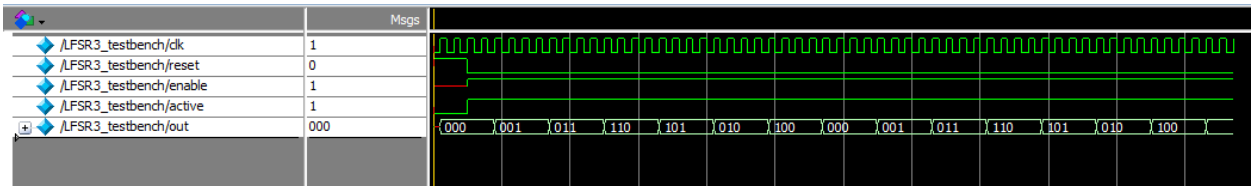


Figure 12: LED Board connection

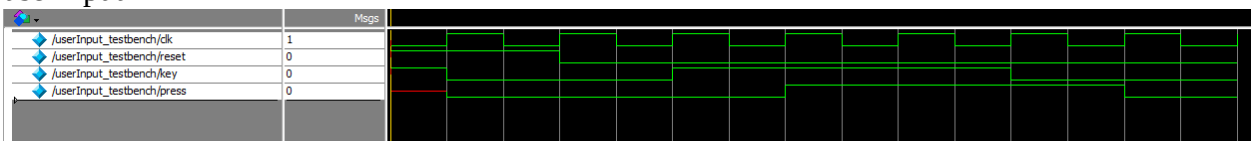
Results:

The above submodules are all tested through modelsim. Pictures of modelsim result is shown below:

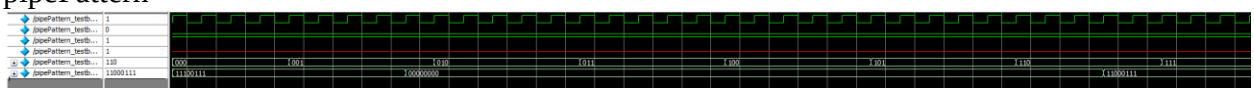
- LFSR



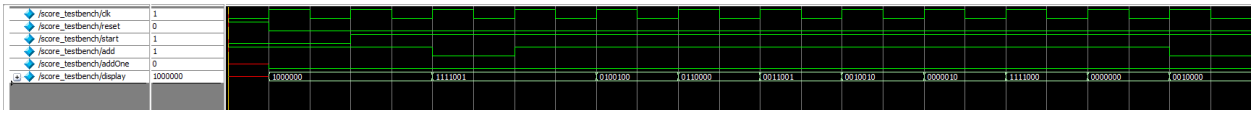
- userInput



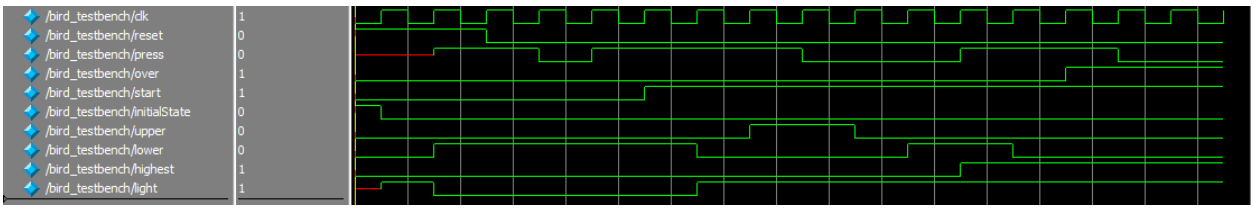
- pipePattern



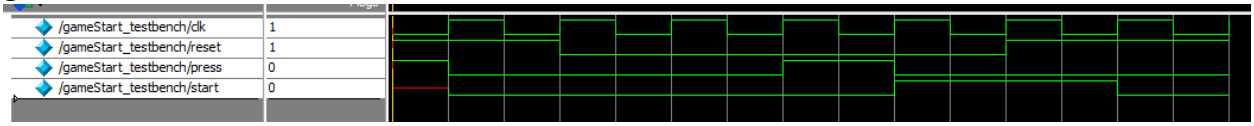
- score



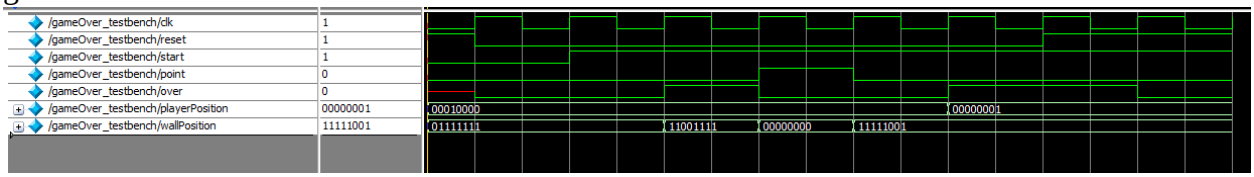
- bird



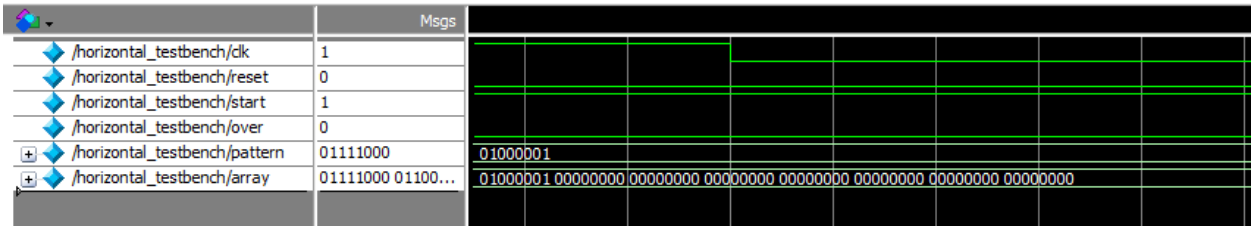
- gameStart



- gameOver



- horizontal



The top-module was finally tested on the FPGA board, which works perfectly fine.

Conclusion and Analysis:

This project gave me a chance to use Verilog to implement a rather complex system. It allows me to design the project from scratch and gradually implement the detail of the lab. After the lab is done, I am more confident with Verilog and now have a deeper understanding of hardware descriptive language.