Cynthia Li
EE 371
June 3, 2021

Lab 6 Report

Procedure

This lab prompts us to design a competitive game using newly learned audio output, vga output and self-learning inputs keyboard, joystick, etc. Our original idea was to design a disk game that shoots the disk with different angles and allows the disk to bounce back from the upper and lower wall based on the angle of incidents. However, after assessing the workload, we realized that updating the location of the disk for that game logic requires too complex algorithms that might take more time than this lab. In this case, we decided to implement the easy version first, that the disk only goes diagonally but can still bounce back.

We divided the project into three tasks: 1. Create the game logic that is able to update the location of the disk, the locations of the players, and determine the winner and the scores of each player; 2. Use the output from the game logic to draw the game on VGA; 3. Change the user input to come from the keyboard.

Task 1: Task 1 is to set up the gamefield and the game logic. Considering the content of the game, we need one disk and two players. For the disk, we used module disk as the datapath, and module catch is the control. They together would be able to control the disk's movement and update its location on the screen. For the players, the module player can simply update the location of the player based on user input. Lastly, we want to put the players and the disk into the game field, the module game, count players' score and determine who wins. We also implemented the score to display the score of players on HEX 2 and HEX0, and display the winner on HEX5 and HEX4.

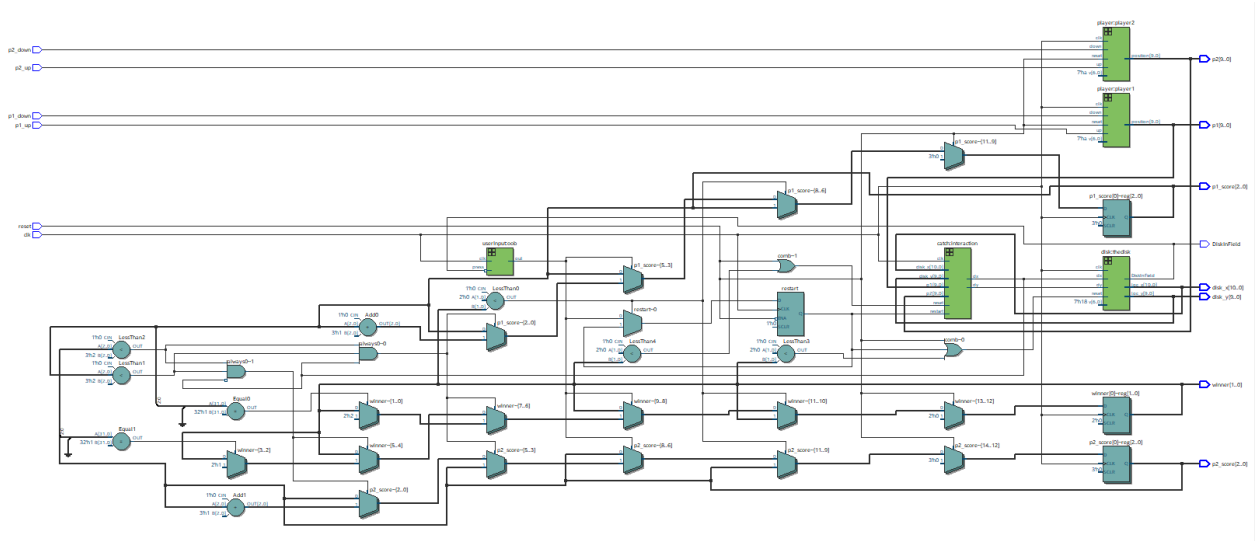

Fig. 1. Top block diagram Task 1

Task 2: Task2 is to write the color control for VGA to draw the players, the disk in white and the background to black and wire it correctly with VGA_framebuffer (given by professor from previous labs). Below is the top block diagram of Task 2.

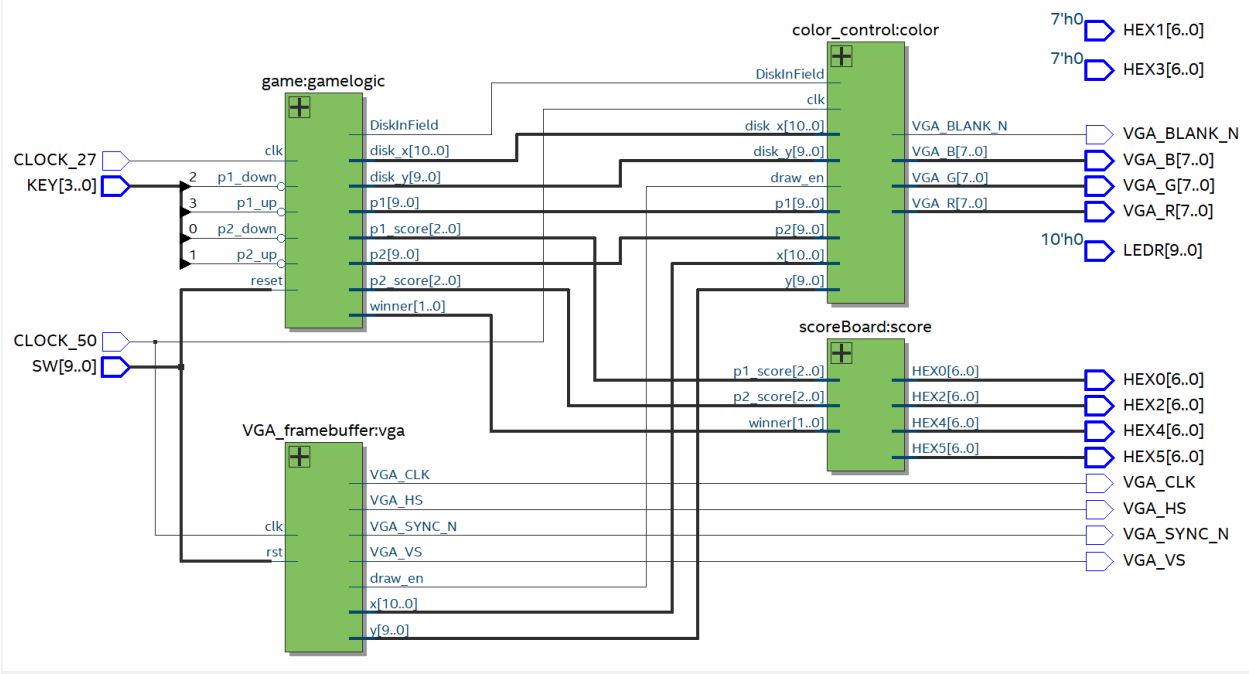

Fig. 2. Top block diagram Task 2

Task 3: Task 3 is to wire up with the N8 controller. We use the provided drivers on Canvas and make the N8 controller as our input device. Below is the top block diagram of Task 3.

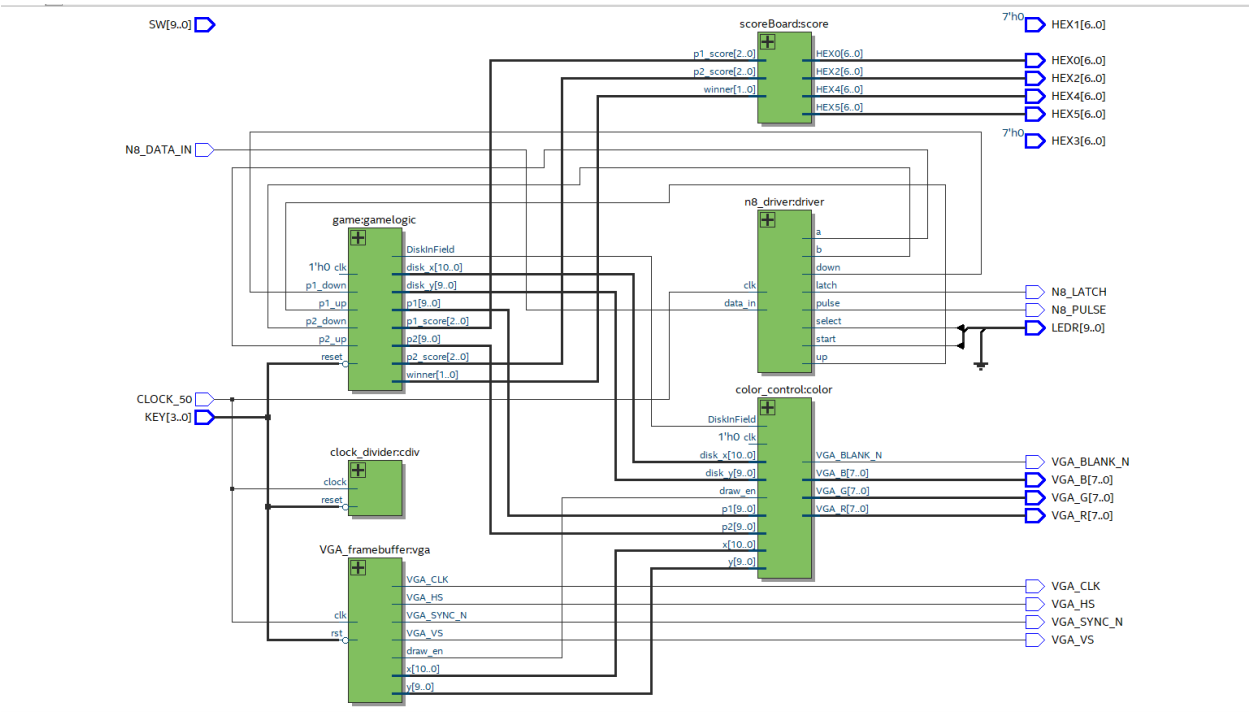

Fig. 3. Top block diagram Task 2

Results

Task 1:

We first chose to implement the module that represents a disk and updates the disk's location, which produced the following waveform below as in Fig.

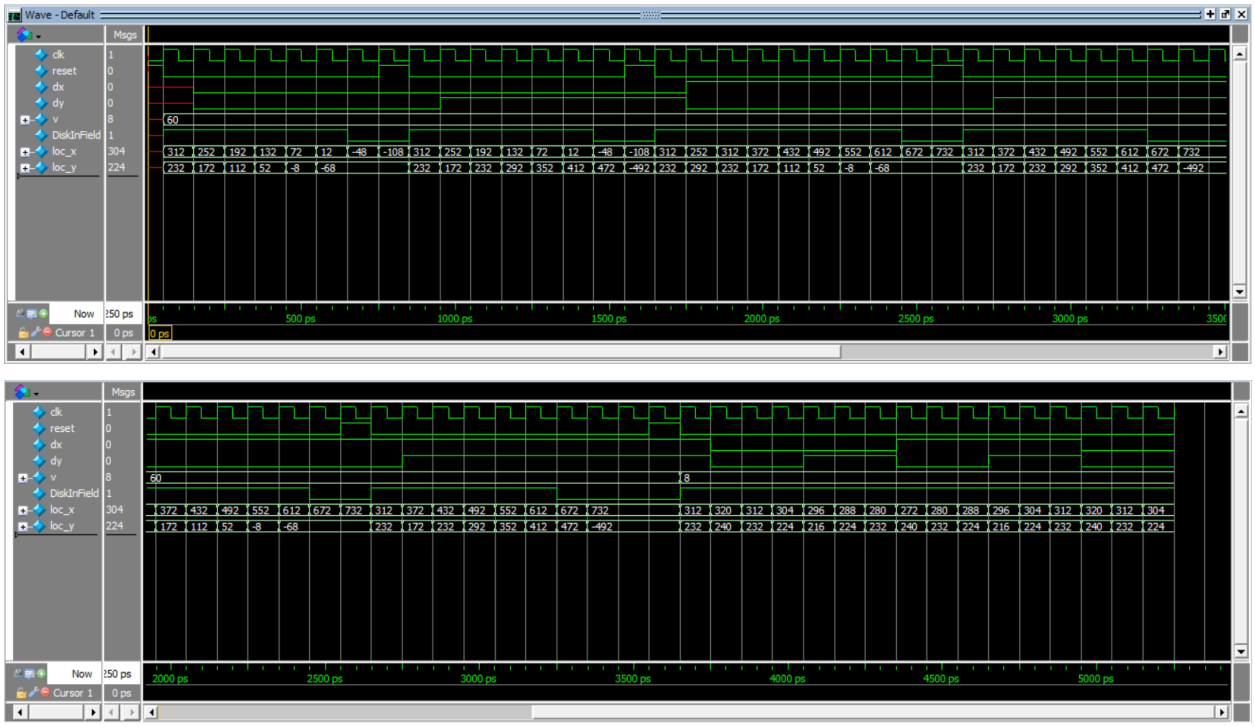

Fig. 4 disk_testbench simulation

We tested the cross boundary conditions for all combinations of dx, dy, and whether they update the coordinate loc_x, loc_y correctly based on the given input speed v, whether reset signal resets loc_x and loc_y correctly, and whether DiskInField is correct.

The expected output for loc_x and loc_y should have the position as the middle of the screen in the first cycle after reset, then in the first 6 cycles, we should observe loc_x and loc_y decrement. Then it is reset to the middle of the screen, in the next 6 cycles, loc_x decrement and loc_y increment. It is again set to the middle of the screen, loc_x increment and loc_y decrement. Then it's again set to the middle of the screen, loc_x and loc_y increment. In this procedure, each increment or decrement is by 60, and loc_x should stop changing after below 0 or exceeding 634, loc_y should stop changing after below 0 or exceeding 472. Then in the normal operation test, loc_x and loc_y both decrement for 3 cycles, loc_x decrement and loc_y increment for 3 cycles, loc_x increment and loc_y decrement for 3 cycles, and loc_x and loc_y increment for 3 cycles, with each increment or decrement by 8.

The expected output for DiskInField is that when loc_x is negative or larger than 634, it is false (in the same cycle).

From the simulation result in the fig. We can see that the disk's x, y coordinates and the status of DiskInField can be updated correctly. We can say this module has expected functionality.

Then we implemented the module that represents a player and updates the player's location. It produced the following waveform as in Fig.

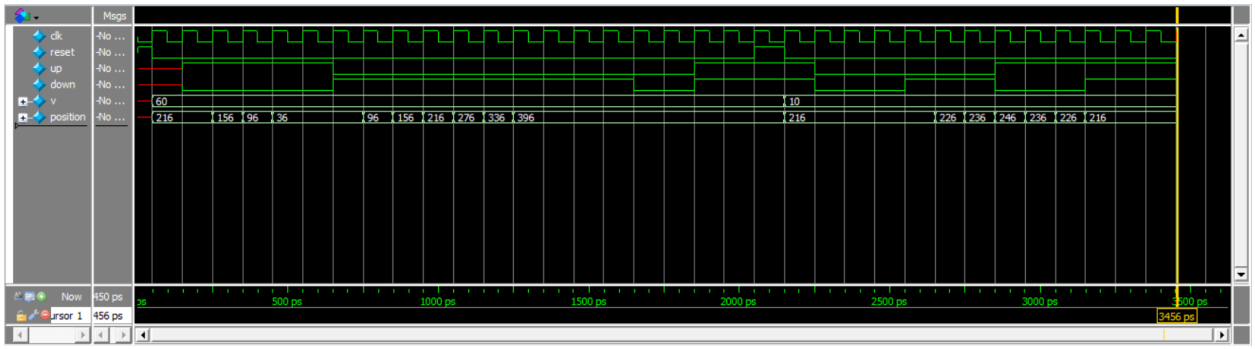

Fig. 5 player_testbench simulation

We tested the cross boundary circumstances of the player, whether it moves to the correct direction given the different up/down instruction, whether it updates location correctly with different speed, and whether it's correctly reset. We expect to see the position originally start at the middle of the screen after reset, and gradually decrease until it reaches the upper boundary and stay at 36, then it should increment until reaching the lower boundary and then stay at 396. Then, when up/down are both true or both false, the position doesn't change. Then, the speed is changed to 8, it will be reset to the middle of the screen, and then it should stay still for 3 cycles, increment by 8 for 3 cycles, decrement by 8 for 3 cycles, and stay still for another 3 cycles.

From the waveform in Fig. we can see that the position's update matches with what we have expected. Therefore, this module is working properly.

After having the disk and the player, we need a logic that determines the direction of the disk's movement. Therefore, based on the players' positions and the disk's position, we wrote the module catch to give the disk's movement instruction. This module produced the waveform as below in Fig.

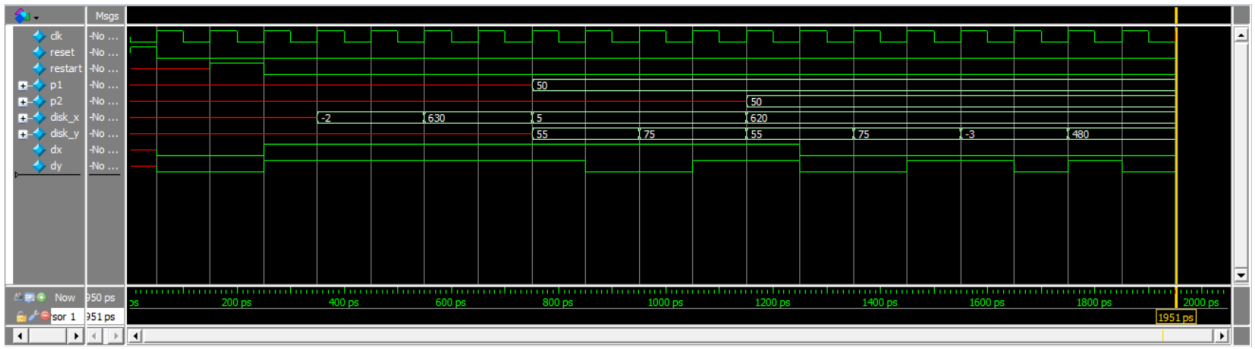

Fig. 6 catch_testbench simulation

We tested the reset function, restart function, crossing boundary situations, caught by player1, caught by player2, and hitting the upper/lower boundaries. We expect to see reset set dx to 0, dy to 1, restart reverses dx, dy (first 4 cycles). When disk_x is out of the left/right boundary, direction doesn't change. When it is caught by player1, it sets dx to 1, and sets dy based on if it's in the upper half of the player or lower half of the player. When it is caught by player2, it sets dx to 0, and sets dy based on if it's in the upper or lower half of the player as well. Last, we should see that when disk_y is out of the upper/lower boundary, dy will be reversed. From the

simulation in Fig. we see that the waveform it produced matches our expectation. This module is working properly in this case.

We also used userInput (from the previous lab) to process DiskInField to get OutOfBound, Since DiskInField can be false for more than 1 cycle. It produced the following waveform.

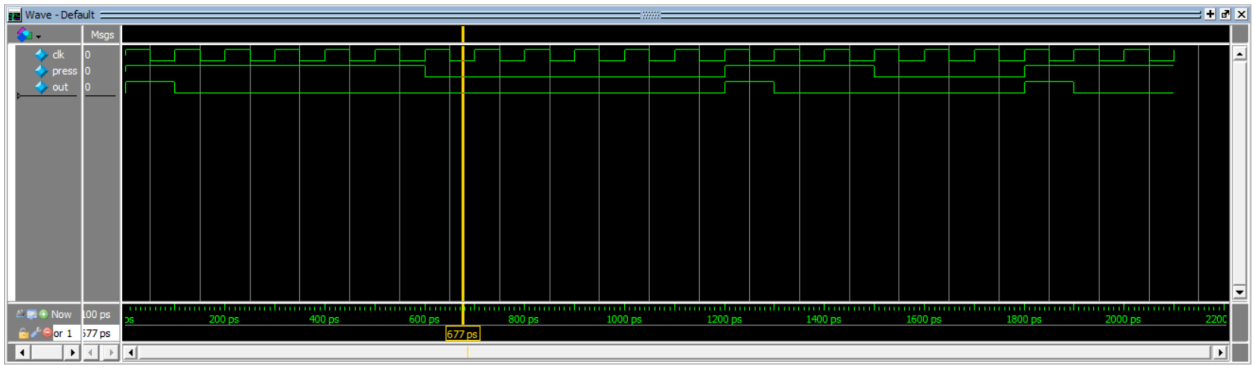

Fig. 7 userInput_testbench simulation

We basically expect the module to be able to process a long pulse in the input to a short pulse that is only true for 1 cycle. And from the simulation, we can see that it matches our expectation.

Then we set up the whole game using two players, one disk and the catch logic in the module game. This module also updates players' scores and the winner, which produced the waveform as below in Fig.

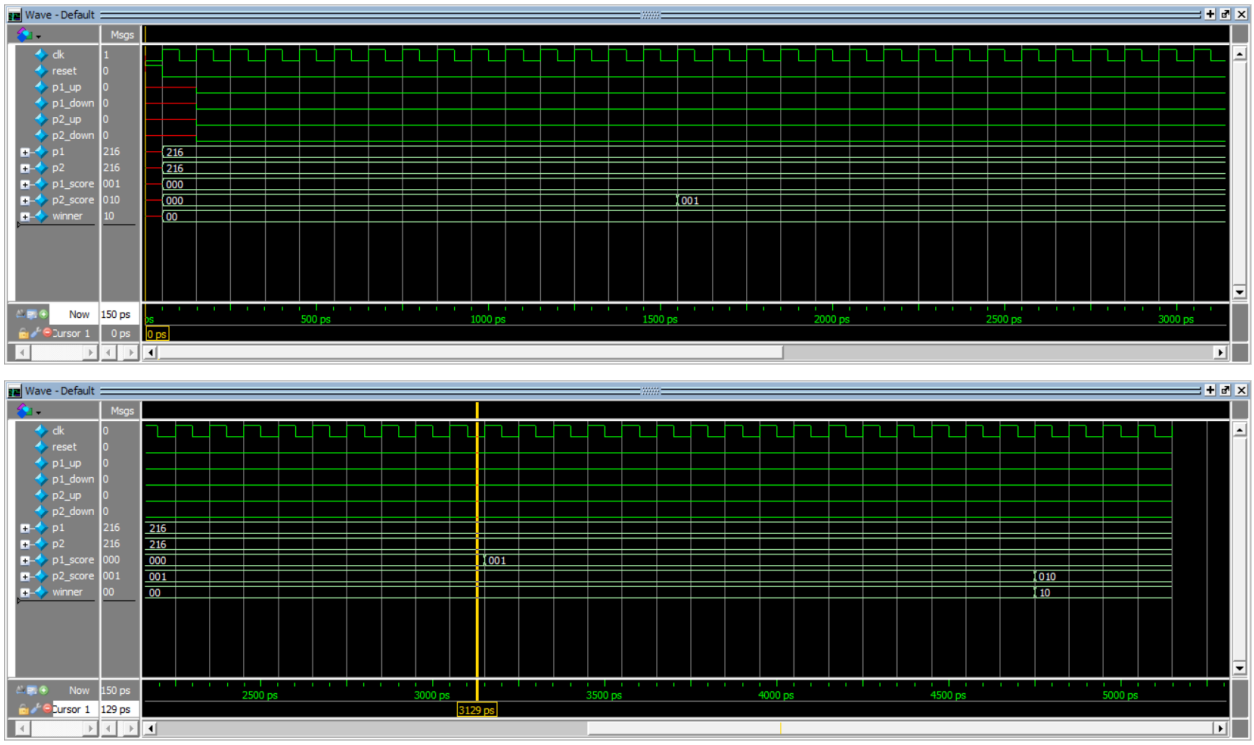

Fig. 8 game_testbench simulation

From previous simulations of disk, player, catch, we see that the location can be updated correctly, thus, we just want to check after one reset, if the game is able to correctly assign points to players, whether it is able to restart by itself before either player gaining certain points, and whether it is able to determine the winner after one player gained certain amount of points. Let the players stand still (no user control). We expect to see player 2 first gains 1 point, then player 1 gains 1 point (since the pitching machine takes turns for each player), player 2 takes another point and becomes the winner (we used full score of 2 for the simulation purpose). From the simulation, we can see that it outputs correctly and thus, this module works correctly.

After getting the player score and winner status, we are able to write the combinational logic to display these results on HEX. We wrote the scoreBoard module for this purpose. It produced the waveform below in Fig.

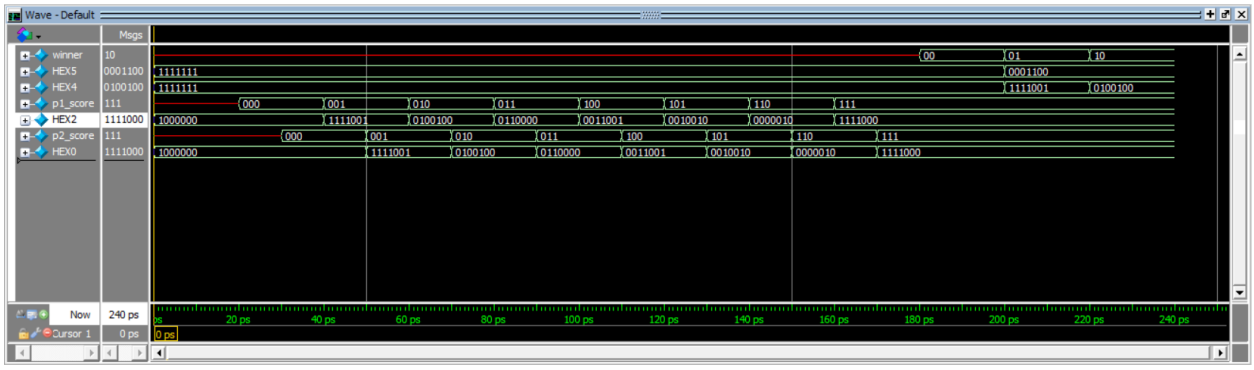

Fig. 9 scoreBoard_testbench simulation

We tested the default display, HEX2 and HEX0's display for p1_score and p2_score for each score (0-7), and HEX5 and HEX4's display for winner. We expect to see by default, HEX5, HEX4, is off, HEX2, and HEX0 display the 7-segment value of 0. Then when p1_score and p2_score are updated from 0 to 7 , HEX2 and HEX0 produce the corresponding 7-segment value at the same cycle. Lastly, when the value of the winner is larger than 0, HEX5 should display the 7-segment value of "P," and HEX4 should display the 7-segment value of 1, or 2 depending on the winner being 1 or 2. From the waveform in Fig. we see that the output matches our expectation. Thus, we can conclude that this module has our needed functionality.

In task 1, the basic game logic is set up to be output onto VGA. Below is the synthesis report of Task 1.

Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins	Virtual Pins
1	|game	267 (13)	45 (8)	0	0	56	0
1	|catch:interaction|	106 (106)	3 (3)	0	0	0	0
2	|disk:thedisk|	49 (49)	15 (15)	0	0	0	0
3	|player:player1|	49 (49)	9 (9)	0	0	0	0
4	|player:player2|	49 (49)	9 (9)	0	0	0	0
5	|userInput:oob|	1 (1)	1 (1)	0	0	0	0

Fig. 10 Resource Utilization Report of Task1

Task 2:

There are really not many things in this task. We just added the color_control module to change the color to white for the VGA display for the ball and the player, and turned the color to black for the background. It produces the following waveform.

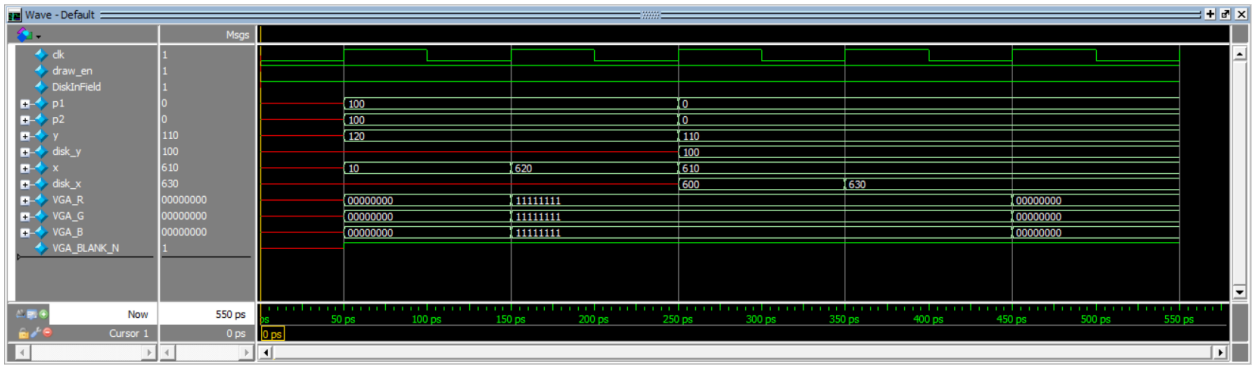

Fig. 11 color_control_testbench simulation

We simply tested if the VGA_R, VGA_G, VGA_B will turn to 8'b1 when it is in the defined region of the player or the disk, and be 8'b0 for the rest. From the simulation, we see that it produces the needed result. We didn't have a new top module yet. We will have the new synthesis report in Task 3.

Task 3:

The only thing needed for this task is to wire the N8 in the top module as the user input. Since it is the given code, and only modification is the top module. There is no simulation for this task. Below is the synthesis report for this task.

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Block Memory Bits	DSP Blocks	Pins
1	▾ |DE1_SoC	76 (0)	56 (0)	0	0	99
1	|VGA_framebuffer:vga|	40 (40)	21 (21)	0	0	0
2	|n8_driver:driver|	36 (36)	35 (35)	0	0	0

Fig. 12 Resource Utilization Report of Task3

Conclusion:

In this lab we practiced using VGA and N8 for the user input and output.Though minor improvements could have been done such as adding a delay before the score display, since we are using a fast clock. But overall, everything worked as expected.

APPENDIX

I. Task1:

A. disk.sv

```
// Cynthia Li, Simon Chen
// EE 371 LAB 6

// This module has a parameter SIZE to specify the disk size, it takes a
// reset signal, movement direction instruction dx for horizontal movement,
// dy for vertical movement, a 5-bit value that specifies the speed of
// movement, and output the 10-bit y coordinate and 11-bit x coordinate of
// the disk's current position. It simply simulate a disk and updates the
// location.
module disk #(parameter SIZE = 16) (clk, reset, v, dx, dy, DiskInField, loc_x, loc_y);
   input logic clk, reset;
   // 0 for left in dx, up in dy
   // 1 for right in dx, down in dy
   input logic dx, dy;
   // v is the speed the disk travels
   input logic signed [6:0] v;
   output logic DiskInField;
   // loc_x, loc_y is the coordinate of disk on VGA
   // lox_x is the left bound of the disk
   // loc_y is the upper bound of the disk
   // we need them as signed values because of the outOfBound situation
   output logic signed [10:0] loc_x;
   output logic signed [9:0] loc_y;

   localparam width = 640;
   localparam height = 480;

   // the following ff block update the position of the disk
   always_ff @(posedge clk) begin
      if (reset) begin     // reset to initial position
         loc_x <= width / 2 - SIZE / 2;
         loc_y <= height / 2 - SIZE / 2;
      end else begin
         // the following updates x coordinate
         if (loc_x >= 0 - v && loc_x + SIZE <= width + v) begin         // in the screen
            if (dx) loc_x <= loc_x + v;    // move to the right
            else loc_x <= loc_x - v;       // move to the left
         end else loc_x <= loc_x;

         // the following updates y coordinate
         if (loc_y >= 0 - v && loc_y + SIZE <= height + v) begin        // in the screen
            if (dy) loc_y <= loc_y + v;    // move down
            else loc_y <= loc_y - v;       // move up
         end else loc_y <= loc_y;
      end
   end

   assign DiskInField = ((loc_x >= 0) && (loc_x <= width-SIZE));
endmodule
```

```
50
51  module disk_testbench ();
52     logic clk, reset;
53     logic dx, dy;
54     logic signed [6:0] v;
55     logic DiskInField;
56     logic signed [10:0] loc_x;
57     logic signed [9:0] loc_y;
58
59     parameter CLK_Period = 100;
60     initial begin
61        clk <= 1'b0;
62        forever #(CLK_Period/2) clk <= ~clk;
63     end
64
65     disk #(16) dut (.*);
66
67     integer i;
68     initial begin
69                                                                    // cross bounds conditions
70        reset <= 1;                                                 @(posedge clk);
71        reset <= 0; v <= 60;                                        @(posedge clk);
72        {dx, dy} <= 2'b00;                        repeat(6)         @(posedge clk);   // upper left
73        reset <= 1;                                                 @(posedge clk);
74        reset <= 0; v <= 60;                                        @(posedge clk);
75        {dx, dy} <= 2'b01;                        repeat(6)         @(posedge clk);   // lower left
76        reset <= 1;                                                 @(posedge clk);
77        reset <= 0; v <= 60;                                        @(posedge clk);
78        {dx, dy} <= 2'b10;                        repeat(8)         @(posedge clk);   // upper right
79        reset <= 1;                                                 @(posedge clk);
80        reset <= 0; v <= 60;                                        @(posedge clk);
81        {dx, dy} <= 2'b11;                        repeat(8)         @(posedge clk);   // lower right
82
83                                                                    // normal updates
84        reset <= 1;                                                 @(posedge clk);
85        reset <= 0; v <= 8;                                         @(posedge clk);
86        for (i = 0; i < 5; i++) begin
87           {dx, dy} <= i;                         repeat(3)         @(posedge clk);
88        end
89        $stop;
90     end
91
92  endmodule
93
```

B. player.sv

```
1   // Cynthia Li, Simon Chen
2   // EE 371, LAB 6
3
4   // This module has two parameters player_w as player width and player_h as player height, it takes a reset signal
5   // the movement instruction up and down, a 5-bit value represeting the speed, and output a 9 bit value
6   // representing the y coordinate of the player. It's role is to simulate one player and update its position
7   // in terms of height.
8   module player #(parameter player_w = 24, player_h = 48) (clk, reset, up, down, v, position);
9      input logic clk, reset;
10     // up/down: to track the movement of the player
11     // need both as it can stay still
12     input logic up, down;
13     // v: the speed of the player's movement
14     input logic signed [6:0] v;
15     // the y coordinate of the top edge of the player
16     output logic signed [9:0] position;
17
18     // the vga screen height
19     localparam height = 480;
20
21     // the following ff block update the y position of one player
22     always_ff @(posedge clk) begin
23        if (reset) begin     // reset to initial position
24           position <= height / 2 - player_h / 2;
25        end else begin
26           if (up && ~down) begin
27              // not attempt to pass the upper screen boundary
28              if ((position - v) < 0) begin
29                 position <= position;
30              end else begin
31                 position <= position - v;
32              end
33           end else if (~up && down) begin
34              // not attempt to pass the lower screen boundary
35              if ((position + v) > (height - player_h)) begin
36                 position <= position;
37              end else begin
38                 position <= position + v;
39              end
40           end else begin // up and down is both true or both false
41              position <= position;
42           end
43        end
44     end
45  endmodule
46
```

```
47  module player_testbench();
48     logic clk, reset;
49     logic up, down;
50     logic signed [6:0] v;
51     logic signed [9:0] position;
52
53     parameter CLK_Period = 100;
54     initial begin
55        clk <= 1'b0;
56        forever #(CLK_Period/2) clk <= ~clk;
57     end
58
59     player #(24, 48) dut (.*);
60
61     integer i;
62     initial begin
63        reset <= 1;                                          @(posedge clk);
64        reset <= 0; v <= 60;                                 @(posedge clk);
65        // test cross bound operation
66        up <= 1; down <= 0;                    repeat(5)     @(posedge clk);
67        up <= 0; down <= 1;                    repeat(10)    @(posedge clk);
68        up <= 0; down <= 0;                    repeat(2)     @(posedge clk);
69        up <= 1; down <= 1;                    repeat(2)     @(posedge clk);
70        reset <= 1;                                          @(posedge clk);
71        reset <= 0; v <= 10;                                 @(posedge clk);
72        // test normal operation
73        for (i = 0; i < 4; i++) begin
74           {up, down} <= i;                       repeat(3)   @(posedge clk);
75        end
76        $stop;
77     end
78  endmodule
```

C. catch.sv

```
1  // Cynthia Li, Simon Chen
2  // EE 371, Lab 6
3
4  // This module takes a parameter diskSize that specifies the disk size, player_w for player's width
5  // player_h for player's height. It takes the reset and restart signal, the 10-bit value for player1&2's
6  // position and a 11-bit value and a 10-bit value for the ball's position. It is able to analyze whether
7  // the disk hits the wall, cross the boundary or is caught by a player, and output the movement instruction
8  // for the disk.
9  module catch #(parameter diskSize = 16, player_w = 24, player_h = 48)
10       (clk, reset, restart, p1, p2, disk_x, disk_y, dx, dy);
11
12    input logic clk, reset, restart;
13    // players' player y location
14    input logic signed [9:0] p1, p2;
15    // the disk's x/y location
16    input logic signed [10:0] disk_x;
17    input logic signed [9:0] disk_y;
18    // movement direction of the disk
19    output logic dx, dy;
20
21    // parameters for the screen size
22    localparam width = 640;
23    localparam height = 480;
24    localparam wall = 20;
25
26    logic pre_dx;
27
28    always_ff @(posedge clk) begin
29       if (reset) begin                 // reset: same direction to start the game
30          dx <= 0;
31          dy <= 0;
32          pre_dx <= 0;
33       end else if (restart)begin       // restart (within a game):let 2 players take turns
34          dx <= ~pre_dx;                // let 2 players take turns
35          dy <= ~dy;                    // dy depends on the end motion of last move from previous inning
36          pre_dx <= ~pre_dx;
37       end else begin
38
39          // define the catch region as completely overlap with player
40          if (disk_x + diskSize <= player_w && disk_x >= 0 && disk_y >= p1
41                      && disk_y + diskSize <= p1 + player_h) begin           // caught by player1
42             dx <= 1;    // reverse horizontal direction
43             if (disk_y + diskSize / 2 >= p1 + player_h / 2) begin    // caught in the lower half of player, go
44                dy <= 1;
45             end else begin    // caught in the upper half of player, go up
46                dy <= 0;
47             end
48          end else if (disk_x >= width - player_w && disk_x + diskSize <= width && disk_y >= p2
49                      && disk_y + diskSize <= p2 + player_h) begin            // caught by player2
50             dx <= 0;    // reverse horizontal direction
51             if (disk_y + diskSize / 2 >= p2 + player_h / 2) begin    // caught in the lower half of player, go down
52                dy <= 1;
53             end else begin    // caught in the upper half of player, go up
54                dy <= 0;
55             end
56          end
57
58          if (disk_y <= wall) begin     // hit upper bound
59             dy <= ~dy; // reverse direction
60          end else if (disk_y + diskSize >= height - wall) begin         // hit lower bound
61             dy <= ~dy;  // reverse direction
62          end
63       end
64
65    end
66 endmodule
67
68 module catch_testbench();
69    logic clk, reset, restart;
70    logic signed [9:0] p1, p2;
71    logic signed [10:0] disk_x;
72    logic signed [9:0] disk_y;
73    logic dx, dy;
74
75    parameter CLOCK_PERIOD = 100;
76    initial begin
77       clk <= 0;
78       forever #(CLOCK_PERIOD/2) clk <= ~clk;
79    end
80
81    catch dut (.*);
82
83    initial begin
84       reset <= 1;                                    @(posedge clk);    // reset function
85       reset <= 0;                                    @(posedge clk);
86       restart <= 1;                                  @(posedge clk);    // restart function
87       restart <= 0;                                  @(posedge clk);
88       disk_x <= -2;                       repeat(2)  @(posedge clk);    // out of bound on the left
89       disk_x <= 630;                      repeat(2)  @(posedge clk);    // out of bound on the right
90
```

```
        disk_x <= 5; disk_y <= 55; p1 <= 50;            @(posedge clk);   // caught in upper half of p1's player
                                                        @(posedge clk);
        disk_x <= 5; disk_y <= 75; p1 <= 50;            @(posedge clk);   // caught in bottom half of p1's player
                                                        @(posedge clk);
        disk_x <= 620; disk_y <= 55; p2 <= 50;          @(posedge clk);   // caught in upper half of p2's player
                                                        @(posedge clk);
        disk_x <= 620; disk_y <= 75; p2 <= 50;          @(posedge clk);   // caught in bottom half of p2's player
                                                        @(posedge clk);
        disk_y <= -3;                       repeat(2)   @(posedge clk);
        disk_y <= 480;                      repeat(2)   @(posedge clk);
        $stop;
    end
endmodule
```

D. game.sv

```
// Cynthia Li, Simon Chen
// EE 371, Lab 6

// The module game takes two parameters diskspeed and playerspeed (defaultly set to 40 for modelSim
// simulation, use different speed for demostration). It takes 4 1-bit input p1_up, p1_down, p2_up
// p2_down as movmement direction instruction for each player's player. It outputs the status of whether
// the disk is still in the game field, the 11-bit x location of the disk, the 10-bit y location of the
// disk and the two players, and two 3-bit value as score for player 1 and 2, and a 2-bit value representing
// the winner.
module game #(parameter diskspeed = 24, playerspeed = 10, width = 4) (clk, reset, p1_up, p1_down, p2_up, p2_down,
              p1, p2, DiskInField, disk_x, disk_y, p1_score, p2_score, winner);

    input logic clk, reset;
    // the following is the user input to control player1/2's player's movement
    input logic p1_up, p1_down, p2_up, p2_down;

    // diskInField: status of whether the disk is in the visible scale of screen
    output logic DiskInField;
    // disk's x, y coordinates
    output logic signed [10:0] disk_x;
    output logic signed [9:0] disk_y;
    // players' players y coordinates
    output logic signed [9:0] p1, p2;
    // p1_score, p2_score and winner updates current score for each player,
    // whether the game has a winner yet, and who the winner is.
    output logic [2:0] p1_score;
    output logic [2:0] p2_score;
    output logic [1:0] winner;

    // the following are some local logic to store outputs of instantiations
    logic dx, dy;
    logic OutOfBound;
    logic restart;
    logic collide;


    // the following sets the initial p1_score/p2_score/winner for each inning,
    // and starts the counter
    always_ff @(posedge clk) begin
        if (reset) begin                    // reset the score and winner
            p1_score <= 0;
            p2_score <= 0;
            winner <= 0;
        end else if (winner > 0) begin      // the game is finished, score and winner remain unchanged
            p1_score <= p1_score;
            p2_score <= p2_score;
            winner <= winner;
```

```
48          end else if (OutOfBound) begin      // unfinished game, someone gains a point
49          // the following updates player 1 & 2's score when the game is not finished
50
51             // use full score 2 for simulation
52             if (dx && p1_score < 2 && p2_score < 2) begin      // disk cross right bound --> p2 misses the disk
53             // use full score 7 for demostration
54             //if (dx && p1_score < 7 && p2_score < 7) begin    // disk cross right bound --> p2 misses the disk
55
56                p1_score <= p1_score + 1;
57
58                // for simulation
59                if (p1_score == 1) begin
60                // if (p1_score == 6) begin
61
62                   winner <= 2'b01;
63                end
64             end
65
66             else if (~dx && p1_score < 2 && p2_score < 2) begin   // disk cross left bound --> p1 misses the disk
67             // else if (~dx && p1_score < 7 && p2_score < 7) begin   // disk cross left bound --> p1 misses the disk
68
69                p2_score <= p2_score + 1;
70
71                if (p2_score == 1) begin
72                // if (p2_score == 6) begin
73
74                   winner <= 2'b10;
75                end
76             end
77
78             restart <= 1;
79
80          end else begin    // no one missed
81             if (restart) begin   // keep restart only true for one cycle
82                restart <= 0;
83             end
84             p1_score <= p1_score;
85             p2_score <= p2_score;
86             winner <= winner;
87          end
```

```
88
89       end
90
91       userInput oob (.clk, .press(~DiskInField), .out(OutOfBound));
92       // the following instantiates the disk
93       disk thedisk (.clk, .reset(reset || restart || (winner > 0)), .v(diskspeed),
94                     .dx, .dy, .DiskInField, .loc_x(disk_x), .loc_y(disk_y));
95       // the following instantiates player1's player
96       player player1 (.clk, .reset, .up(p1_up), .down(p1_down), .v(playerspeed), .position(p1));
97       // the following instantiates player2's player
98       player player2 (.clk, .reset, .up(p2_up), .down(p2_down), .v(playerspeed), .position(p2));
99       // the following instantiates the hit module to define the disk/player/wall interaction
100      catch interaction (.clk, .reset(reset || (winner > 0)), .restart,
101                                    .p1, .p2, .disk_x, .disk_y, .dx, .dy);
102
103   endmodule
104
105   module game_testbench();
106      logic clk, reset;
107      logic p1_up, p1_down, p2_up, p2_down;
108      logic DiskInField;
109      logic signed [10:0] disk_x;
110      logic signed [9:0] disk_y;
111      logic signed [9:0] p1, p2;
112      logic [2:0] p1_score;
113      logic [2:0] p2_score;
114      logic [1:0] winner;
115
116      parameter CLOCK_PERIOD = 100;
117      initial begin
118         clk <= 0;
119         forever #(CLOCK_PERIOD/2) clk <= ~clk;
120      end
121
122      game dut (.*);
123
124      initial begin
125         reset <= 1;                                    @(posedge clk);
126         reset <= 0;                                    @(posedge clk);
127         p1_up <= 0; p1_down <= 0; p2_up <= 0; p2_down <= 0;
128                                       repeat(50)       @(posedge clk);
129         $stop;
130      end
131   endmodule
132
```

E. userInput.sv

```
// Cynthia Li, Simon Chen
// EE 371 LAB6

// Module userInput takes a input press that indicates the status
// of a KEY, and should process it so that every press is true for
// only one cycle and otherwise stays false. It will output this
// processed KEY status to out.
module userInput (clk, press, out);
   input logic clk, press;
   // press: true when the KEY is pressed
   output logic out;

   enum {on, off} ps, ns;

   // the next state and output logic
   always_comb
      case(ps)
         on: if (press) begin // key contiuously being pressed
                  ns = on;    // & out has been true for 1 cycle
                  out = 0;
               end
             else begin       // key is released
                  ns = off;
                  out= 0;
               end
         off:  if (press) begin  // key is pressed (the action)
                     ns = on;    // make out true for one cycle
                     out= 1;
                  end
               else begin        // key is not pressed
                     ns = off;
                     out = 0;
                  end
         default: begin
                     ns = off;
                     out = 0;
                  end
      endcase

   always_ff @(posedge clk)
      ps <= ns;

endmodule
```

```
module userInput_testbench();
   logic clk, press;
   logic out;

   userInput dut (.clk, .press, .out);

   // Set up the clock.
   parameter CLOCK_PERIOD=100;
   initial clk=1;
   always begin
      #(CLOCK_PERIOD/2);
      clk = ~clk;
   end

   // Set up the inputs to the design. Each line is a clock cycle.
   initial begin
   press <= 1;   repeat(3) @(posedge clk); // off state: key is pressed
   press <= 1;   repeat(3) @(posedge clk); // on state: key is pressed
   press <= 0;   repeat(3) @(posedge clk); // on state: key is released
   press <= 0;   repeat(3) @(posedge clk); // off state: key is unpressed
   press <= 1;   repeat(3) @(posedge clk); // off state: key is pressed
   press <= 0;   repeat(3) @(posedge clk); // on state: key is released
   press <= 1;   repeat(3) @(posedge clk); // off state: key is pressed

   $stop; // End the simulation.
   end
endmodule
```

F. scoreBoard.sv

```
// Cynthia Li, Simon Chen
// EE 371, Lab 6

// This module takes two 3-bit value as player 1 and 2's score, and a 2-bit value as
// the winnder, it will output 4 7-bit value for the 7 segment display on HEX5, HEX4,
// HEX2 and HEX0. It is used as the scoreboard for the game.
module scoreBoard (p1_score, p2_score, winner, HEX5, HEX4, HEX2, HEX0);
   input logic [2:0] p1_score, p2_score;
   input logic [1:0] winner;
   output logic [6:0] HEX5, HEX4, HEX2, HEX0;

   // the following controls HEX5 and HEX4's display for winner of the two players
   always_comb begin
      case (winner)
         // when there is no winner, turns off display
         2'b00: begin
            HEX5 = 7'b1111111;
            HEX4 = 7'b1111111;
            end
         // when player 1 wins, display "P1"
         2'b01: begin
            HEX5 = 7'b0001100; // P
            HEX4 = 7'b1111001; // 1
            end
         // when player 2 wins, display "P2"
         2'b10: begin
            HEX5 = 7'b0001100; // P
            HEX4 = 7'b0100100; // 2
            end
         // default turns off display
         default: begin
            HEX5 = 7'b1111111;
            HEX4 = 7'b1111111;
         end
      endcase
   end
```

```
   // the following controls HEX0 display for Player2's score
   always_comb begin
      case (p2_score)
                  // Light: 6543210
         4'b0000: HEX0 = 7'b1000000; // 0
         4'b0001: HEX0 = 7'b1111001; // 1
         4'b0010: HEX0 = 7'b0100100; // 2
         4'b0011: HEX0 = 7'b0110000; // 3
         4'b0100: HEX0 = 7'b0011001; // 4
         4'b0101: HEX0 = 7'b0010010; // 5
         4'b0110: HEX0 = 7'b0000010; // 6
         4'b0111: HEX0 = 7'b1111000; // 7
         default: HEX0 = 7'b1000000;
      endcase
   end
endmodule

module scoreBoard_testbench();
   logic [2:0] p1_score, p2_score;
   logic [1:0] winner;
   logic [6:0] HEX5, HEX4, HEX2, HEX0;

   scoreBoard dut (.*);

   integer i;
   initial begin
                                 repeat(2)   #10;  // test default display
      for (i = 0; i < 8; i++) begin
         p1_score <= i;                      #10;  // test p1 increment
         p2_score <= i;                      #10;  // test p2 increment
      end
      winner <= 0;               repeat(2)   #10;  // test no winner display
      winner <= 1;               repeat(2)   #10;  // test after winning display (p1 wins)
      winner <= 2;               repeat(2)   #10;  // test after winning display (p2 wins)
      $stop;
   end
endmodule

```

II. Task2:

A. VGA_framebuffer.sv

```
1  // VGA driver: provides I/O timing for the VGA port.
2
3  module VGA_framebuffer(
4     input logic clk, rst,
5     output logic signed [10:0] x,
6     output logic signed [9:0] y,
7     output logic draw_en,
8     output logic frame_start,   // Pulse is fired at the start of a frame.
9
10    // Outputs to the VGA port.
11    output logic VGA_CLK, VGA_HS, VGA_VS, VGA_SYNC_N
12 );
13
14    /*
15    *
16    * HCOUNT 1599 0             1279       1599 0
17    *            _______________              ________
18    * ___________|    Video      |____________|  Video
19    *
20    *
21    * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
22    *       _____________________      _____________
23    * |____|       VGA_HS          |____|
24    *
25    */
26
27    // Constants for VGA timing.
28    localparam HPX = 11'd640*2, HFP = 11'd16*2, HSP = 11'd96*2, HBP = 11'd48*2;
29    localparam VLN = 11'd480,   VFP = 10'd11,   VSP = 10'd2,    VBP = 10'd31;
30    localparam HTOTAL = HPX + HFP + HSP + HBP; // 800*2=1600
31    localparam VTOTAL = VLN + VFP + VSP + VBP; // 524
32
33    // Horizontal counter.
34    logic [10:0] h_count;
35    logic end_of_line;
36
37    assign end_of_line = h_count == HTOTAL - 1;
38
39    always_ff @(posedge clk)
40       if (rst) h_count <= 0;
41       else if (end_of_line) h_count <= 0;
42       else h_count <= h_count + 11'd1;
43
44    // Vertical counter & buffer swapping.
45    logic [9:0] v_count;
46    logic end_of_field;
47    logic front_odd; // whether odd address is the front buffer.
48
49    assign end_of_field = v_count == VTOTAL - 1;
50    assign frame_start = !h_count && !v_count;
51
52    always_ff @(posedge clk)
53       if (rst) begin
54          v_count <= 0;
55          front_odd <= 0;
56       end else if (end_of_line)
57          if (end_of_field) begin
58             v_count <= 0;
59             front_odd <= !front_odd;
60          end else
61             v_count <= v_count + 10'd1;
62
63    // Sync signals.
64    assign VGA_CLK = h_count[0]; // 25 MHz clock: pixel latched on rising edge.
65    assign VGA_HS = !(h_count - (HPX + HFP) < HSP);
66    assign VGA_VS = !(v_count - (VLN + VFP) < VSP);
67    assign VGA_SYNC_N = 1; // Unused by VGA
68
69    assign x = h_count - HSP - HBP;  // the x-pixel coordinate in the active area
70    assign y = v_count - VSP - VBP;  // the y-pixel coordinate in the active area
71    assign draw_en = (h_count >= (HSP+HBP) && h_count < (HSP+HBP+HPX)
72                      && v_count >= (VSP+VBP) && v_count < (VSP+VBP+VLN));
73
74 endmodule
75
```

B. color_control.sv

```
// Cynthia Li, Simon Chen
// EE 371 Lab 6

// This module takes a parameter diskSize that specifies the disk size, player_w for player's width
// player_h for player's height. When draw_en is asserted, it is able to output white color to VGA_R,
// VGA_G, VGA_B in the region of players and balls (11/10-bit location value as p1, p2, disk_x, disk_y),
// and black color for the rest.
module color_control #(DiskSize = 12, player_w = 32, player_h = 48) (clk, draw_en, DiskInField, x, y,
                                                                     p1, p2, disk_x, disk_y, VGA_R, VGA_G, VGA_B, VGA_BLANK_N);
    // define input and output logic
    input logic clk, draw_en, DiskInField;
    input logic signed [9:0] p1, p2;
    input logic signed [9:0] y, disk_y;
    input logic signed [10:0]x, disk_x;
    output logic [7:0]  VGA_R, VGA_G, VGA_B;
    output logic VGA_BLANK_N;

    // horizontal visible screen size.
    localparam HPX = 640;

    always_ff @(posedge clk) begin
        // when we can draw
        if (draw_en) begin
            VGA_BLANK_N <= 1;
            // player1's white shape
            if (x < player_w && y > p1 && y < p1 + player_h) begin
                {VGA_R, VGA_G, VGA_B} <= ~24'b0;
            // player2's white shape
            end else if (x > HPX - player_w && y > p2 && y < p2 + player_h) begin
                {VGA_R, VGA_G, VGA_B} <= ~24'b0;
            // the disk's white shape
            end else if (DiskInField && x > disk_x &&x < disk_x + DiskSize && y > disk_y && y < disk_y + DiskSize) begin
                {VGA_R, VGA_G, VGA_B} <= ~24'b0;
            // the background
            end else begin
                {VGA_R, VGA_G, VGA_B} <= 24'b0;
            end
        end else begin
            VGA_BLANK_N <= 0;
        end
    end
endmodule
```

```
module color_control_testbench();
    logic clk, draw_en, DiskInField;
    logic signed [9:0] p1, p2;
    logic signed [9:0] y, disk_y;
    logic signed [10:0]x, disk_x;
    logic [7:0]  VGA_R, VGA_G, VGA_B;
    logic VGA_BLANK_N;

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

    color_control dut (.*);

    integer i;
    initial begin
    draw_en <= 1; DiskInField <= 1;                                                 @(posedge clk);
    p1 <= 100; p2 <= 100; x <= 10; y <= 120;                                        @(posedge clk);
    p1 <= 100; p2 <= 100; x <= 620; y <= 120;                                       @(posedge clk);
    p1 <= 0; p2 <= 0; disk_x <= 600; disk_y <= 100; x <= 610; y <= 110;             @(posedge clk);
    disk_x <= 630; disk_y <= 100;                                                   @(posedge clk);
                                                                                    @(posedge clk);
    $stop;
    end
endmodule
```

III. Task3

A. DE1_SoC.sv

```
// Cynthia Li, Simon Chen
// EE 371 Lab 6

module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, CLOCK_50,
    VGA_R, VGA_G, VGA_B, VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, VGA_VS, N8_PULSE,N8_LATCH, N8_DATA_IN);

    // standard FPGA board I/O
    output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
    output logic [9:0] LEDR;
    input logic [3:0] KEY;
    input logic [9:0] SW;

    // CLOCK_50 for VGA and CLOCK_27 for game logic
    input CLOCK_50;
    output [7:0] VGA_R;
    output [7:0] VGA_G;
    output [7:0] VGA_B;
    output VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, VGA_VS;

    // N8 driver
    input logic N8_DATA_IN;
    output logic N8_LATCH;
    output logic N8_PULSE;

    n8_driver driver(
         .clk(CLOCK_50),
         .data_in(N8_DATA_IN),
         .latch(N8_LATCH),
         .pulse(N8_PULSE),
         .up(up),
         .down(down),
         .left(left),
         .right(right),
         .select(LEDR[9]),
         .start(LEDR[8]),
         .a(a),
         .b(b)
     );

     wire up;
     wire down;
     wire left;
     wire right;
     wire a;
     wire b;


    // Generate slower clks off of CLOCK_50, whichClock picks rate.

    // Generate slower clks off of CLOCK_50, whichClock picks rate.

    logic reset;
    logic [31:0] div_clk;

    assign reset = ~KEY[0];
    parameter whichClock = 25; // 0.75 Hz clock
    clock_divider cdiv (.clock(CLOCK_50),
                        .reset(reset),
                        .divided_clocks(div_clk));

    // the following defines the local variables used for wiring
    // Location of pixel to draw
    logic signed [10:0] x;
    logic signed [9:0] y;
    // Bats locations
    logic signed [9:0] p1;
    logic signed [9:0] p2;
    // disk location
    logic signed [10:0] disk_x;
    logic signed [9:0] disk_y;
    // Scores and winner
    logic [2:0] p1_score;
    logic [2:0] p2_score;
    logic [1:0] winner;          // 0 = none, 1 = P1, 2 = P2

    logic draw_en, DiskInField, frame_start;

    // The following instantiates the VGA_framebuffer
    VGA_framebuffer vga (.clk(CLOCK_50), .rst(reset), .x, .y, .draw_en, .frame_start, .VGA_CLK, .VGA_HS, .VGA_VS, .VGA_SYNC_N);

    // the following instantiates the color_control module to control the VGA drawing color
    color_control color (.clk(CLOCK_), .draw_en, .DiskInField, .x, .y, .p1, .p2, .disk_x, .disk_y, .VGA_R, .VGA_G, .VGA_B, .VGA_BLANK_N);

    // the following instantiates the game module to setup the game field
    game gamelogic (.clk(CLOCK_27), .reset(reset), .p1_up(up), .p1_down(down), .p2_up(a), .p2_down(b),
                    .p1, .p2, .DiskInField, .disk_x, .disk_y, .p1_score, .p2_score, .winner);

    // the following isntantiates the scoreboard module to update the scores for players and display the winner
    scoreBoard score (.p1_score, .p2_score, .winner, .HEX5, .HEX4, .HEX2, .HEX0);
endmodule
```