BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

COURSE NO : EEE 304

PROJECT NAME: PONG GAME BY VERILOG

SUBMITTED TO-

RAJAT CHAKRABORTY,
ASSISTANT PROFESSOR,EEE,BUET
NAFIUL ISLAM(PT)
EEE,BUET.

SUBMITTED BY-

STUDENT ID: 1606106,1606118, 1606130
SECTION : B-2
DATE OF SUBMISSION :18 DECEMBER,2020
VIDEO LINK:https://youtu.be/sFBiwrYOwzo

INTRODUCTION:

For our minimum deliverable, we created a two player pong game based on existing one-player pong game architectures. Nevertheless, compared to traditional pong games where one player tries to compete to let the other player lose the game, we designed our game to be collaborative, so that the two players can play together to reach a mutual goal of getting a high score. Our team realized that nowadays most of the popular games, such as World of Warcraft, highlight collaboration as an essential game feature. Even in a single player game, collaboration can still exist when one person is watching another person play the game and giving advice about how to win the AI. Therefore, as our maximum deliverable, we wanted to create competition with the AI for a single player to add diversity and to maintain collaboration between the player and the viewer. The following are the rules of our pong game:

- Two Player Mode:
    i. After either player presses a button, the game starts.
    ii. The score increments each time either player hits the ball with the paddle.
    iii. When either player misses the ball, the game pauses and a new ball is provided. Three balls are provided in each session.
    iv. After three misses from both sides, the game is ended and displays "game over".

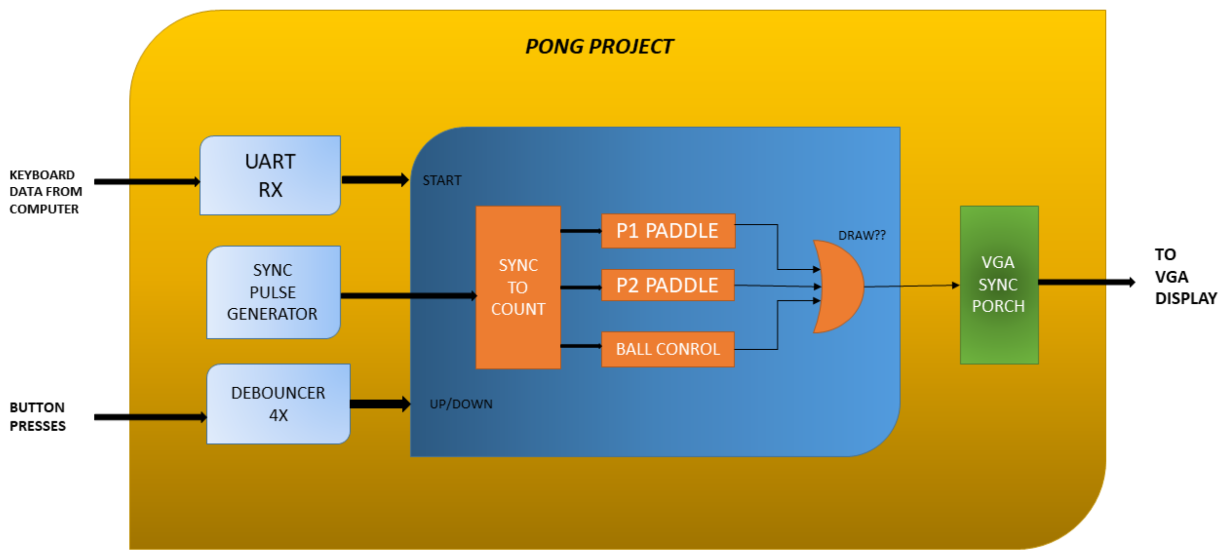
Being a good player requires good eye-hand coordination, precise control and the ability to predict the path of the ball is actually the key to score high. Moreover, we added variation to the ball speed to add unpredictability as a little challenge that requires a faster reaction of the player. Simple as our game design seems, we still believe that the players can have a lot of fun by the little challenges and the experience of coordination and collaboration.

OVERVIEW OF PONG GAME:

The way we decided to implement this game is by creating little functional blocks for each of the paddles and the ball itself. The Pong Board I set to be 40x30 pixels. I chose these because 40x30 is equal to 640x480 divided by 16 (640/16=40, 480/16=30). One thing about FPGAs, it's really hard to do division inside of an FPGA *unless* you are dividing by a power of 2. When dividing by a power of 2, simply drop the number of least significant bits that are required to represent the divisor. For our example, we are dividing by 16, which takes 4 bits to represent. So if we drop the least significant 4 bits off of the Row and Column indexes then we have divided them by 16.

Base-2 multiplying and dividing is done constantly inside of an FPGA, so it's important to understand. If you want to multiply a number by 2, simply add on a single bit on the right of the number. To multiply by 8, add on 3 bits (set to zero). Remove those bits to divide.

Each of the Paddle and Ball modules keeps track of their single component based on the current Row/Col index of the frame. If the current pixel is equal to the location of the paddle or the ball, those modules will drive their output high. This tells the Pong Top to draw white on that pixel. See below for a complete description of each module.

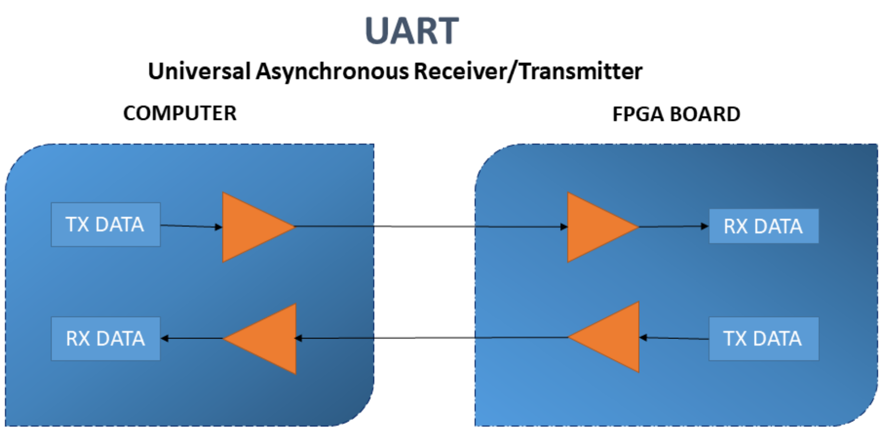


UART RX:

There needs to be a way to kick-off a new Pong Game, to tell the Go Board to start moving the ball. It would have been easier if there were five push-button switches on the Go Board, so that one of them could be used to start the game, but sadly there isn't. You could have chosen to use one of the other buttons to start a new game, but I just decided to reuse the UART RX module from a previous project. Any time the UART RX asserts its Data Valid (DV) pulse, then the game starts.

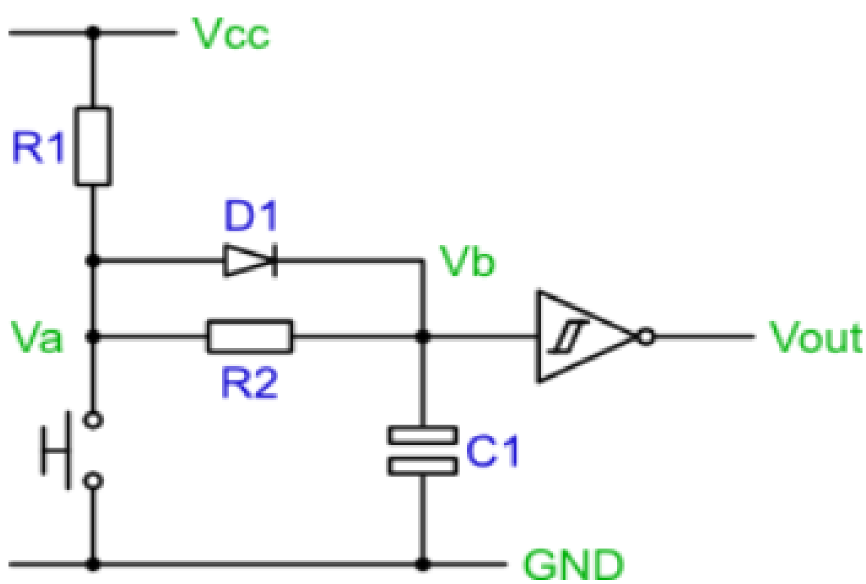
DESCRIPTION OF UART:

A UART is sometimes referred to as a Serial Port, RS-232 Interface, or COM Port. It's one of the simplest methods of communication with your FPGA. Other methods of communication you might have heard of are PCI, PCI-Express, USB, etc. But the Go Board uses a UART because it's the easiest and best to learn with. The UART consists of a receiver component and a transmitter component. For the UART Receiver portion, we need to take in the serial data from the computer. This is sent by the computer one bit at a time. The receiver converts it back to the original byte. This is a conversion of serial data to parallel data. The transmitter works in the opposite way. It sends out one byte at a time across a single wire. A byte is 8-bits wide, so in order to send a byte over a single wire, you need to convert the data from parallel data (as stored in the byte) to serial. This is what we will be doing in the UART Transmitter.



DEBOUNCE SWITCHS:

The Verilog code below introduces a few new concepts. The first thing you might notice is that there are two files. Debounce_Project_Top is the top level , which goes to the physical pins on the Go Board. Debounce_Switch is a lower level module which gets instantiated by the top level module. You can see how that's done below. Note that the signals in the parenthesis are the ones in the current file. The files before the parenthesis are the ones in the instantiated module. All we are adding is the ability to debounce the input switch, so rather than i_Switch_1 being used in the process, we need to use the output of the Debounce_Switch module. We need to create an intermediary signal (w_Switch_1), which will serve to wire up the output of Debounce_Switch to be able to be used in Debounce_Project_Top. For any signals that are wires, we use the prefix w_. Wires mean that the signal will not be turned into a register, it just is used to create a wire between two points. Putting a prefix on the signal is not required, but it helps me keep my code organized and clear.



- **Verilog Code(Debounce_Project_Top.v):**

```
module Debounce_Project_Top
     (input  i_Clk,
      input  i_Switch_1,
      output o_LED_1);

     reg  r_LED_1    = 1'b0;
     reg  r_Switch_1 = 1'b0;
     wire w_Switch_1;

     // Instantiate Debounce Module
     Debounce_Switch Debounce_Inst
     (.i_Clk(i_Clk),
      .i_Switch(i_Switch_1),
      .o_Switch(w_Switch_1));

     // Purpose: Toggle LED output when w_Switch_1 is released.
     always @(posedge i_Clk)
     begin
       r_Switch_1 <= w_Switch_1;         // Creates a Register

       // This conditional expression looks for a falling edge on
   w_Switch_1.
       // Here, the current value (i_Switch_1) is low, but the previous
   value
       // (r_Switch_1) is high.  This means that we found a falling edge.
       if (w_Switch_1 == 1'b0 && r_Switch_1 == 1'b1)
       begin
         r_LED_1 <= ~r_LED_1;         // Toggle LED output
       end
     end

     assign o_LED_1 = r_LED_1;

   endmodule
```

- **Verilog Code(Debounce_Switch.v):**

```
module Debounce_Switch (input i_Clk, input i_Switch, output o_Switch);

  parameter c_DEBOUNCE_LIMIT = 250000;  // 10 ms at 25 MHz

  reg [17:0] r_Count = 0;
  reg r_State = 1'b0;

  always @(posedge i_Clk)
  begin
    // Switch input is different than internal switch value, so an
input is
    // changing.  Increase the counter until it is stable for enough
time.
    if (i_Switch !== r_State && r_Count < c_DEBOUNCE_LIMIT)
      r_Count <= r_Count + 1;

    // End of counter reached, switch is stable, register it, reset
counter
    else if (r_Count == c_DEBOUNCE_LIMIT)
    begin
      r_State <= i_Switch;
      r_Count <= 0;
    end

    // Switches are the same state, reset the counter
    else
      r_Count <= 0;
  end

  // Assign internal register to output (debounced!)
  assign o_Switch = r_State;

endmodule
```

P1/P2 PADDLE:

Here's the actual first new piece of code thus far. The Paddle Control logic is exactly the same for Player 1 and Player 2, the only way to know which is which is by setting a Generic in VHDL or a Parameter in Verilog. This module takes in the Current Row/Col that we are drawing on the screen. It knows where the paddle is located and keeps track of that if the player moves their paddle up or down. If the current active row/col index is equal to the location of the paddle, it will draw the paddle on the screen. The output o_Draw_Paddle when 1 will draw a pixel at the current active row/col location.

BALL CONTROL METHOD:

This module keeps track of the ball location in Row/Col. The row/col that is being drawn on the VGA display is sent as an input to this module. If the current active pixel is the same pixel as where the ball is located, the output o_Draw_Ball becomes a 1. Otherwise it's a 0.

GATE:

This isn't its own module. The purpose of this Or Gate is to look at the outputs of P1 Paddle Control, P2 Paddle Control, and Ball Control. If any of them are telling the VGA display to draw at this Row/Col index, then it will allow the output to go high.

FUTURE IMPROVEMENT OF THE PROJECT:

1. We can implement 4 player mode (left-right, up-down).
2. By increasing ball speed we can improve its difficulty level.
3. By decreasing the length in terms of ball count it can create more fun.
4. AI Mode:
    a. After the player presses a button, the game starts.
    b. The score increments each time the AI misses the ball.
    c. When the player or AI misses the ball, the game pauses and a new ball is provide. Three balls are provided for the player.
    d. After three misses on the player’s side, the game is ended and displays “game over”.

PROJECT VERILOG CODE IS HERE !!!!

PROJECT10 PONG_TOP.v:

```
module Project10_Pong_Top
  (input  i_Clk,       // Main Clock
   input  i_UART_RX, // UART RX Data

   // Push BUttons
   input  i_Switch_1,
   input  i_Switch_2,
   input  i_Switch_3,
   input  i_Switch_4,

   // VGA
   output o_VGA_HSync,
   output o_VGA_VSync,
   output o_VGA_Red_0,
   output o_VGA_Red_1,
   output o_VGA_Red_2,
   output o_VGA_Grn_0,
   output o_VGA_Grn_1,
   output o_VGA_Grn_2,
   output o_VGA_Blu_0,
   output o_VGA_Blu_1,
   output o_VGA_Blu_2
   );

  // VGA Constants to set Frame Size
  parameter c_VIDEO_WIDTH = 3;
  parameter c_TOTAL_COLS  = 800;
  parameter c_TOTAL_ROWS  = 525;
  parameter c_ACTIVE_COLS = 640;
  parameter c_ACTIVE_ROWS = 480;

  // Common VGA Signals
  wire [c_VIDEO_WIDTH-1:0] w_Red_Video_Pong, w_Red_Video_Porch;
  wire [c_VIDEO_WIDTH-1:0] w_Grn_Video_Pong, w_Grn_Video_Porch;
  wire [c_VIDEO_WIDTH-1:0] w_Blu_Video_Pong, w_Blu_Video_Porch;

  // 25,000,000 / 115,200 = 217
  UART_RX #(.CLKS_PER_BIT(217)) UART_RX_Inst
  (.i_Clock(i_Clk),
   .i_RX_Serial(i_UART_RX),
   .o_RX_DV(w_RX_DV),
   .o_RX_Byte());

  // Generates Sync Pulses to run VGA
  VGA_Sync_Pulses #(.TOTAL_COLS(c_TOTAL_COLS),
                    .TOTAL_ROWS(c_TOTAL_ROWS),
                    .ACTIVE_COLS(c_ACTIVE_COLS),
                    .ACTIVE_ROWS(c_ACTIVE_ROWS)) VGA_Sync_Pulses_Inst
  (.i_Clk(i_Clk),
   .o_HSync(w_HSync_VGA),
   .o_VSync(w_VSync_VGA),
```

```
 .o_Col_Count(),
 .o_Row_Count()
);

// Debounce All Switches
Debounce_Switch Switch_1
  (.i_Clk(i_Clk),
   .i_Switch(i_Switch_1),
   .o_Switch(w_Switch_1));

Debounce_Switch Switch_2
  (.i_Clk(i_Clk),
   .i_Switch(i_Switch_2),
   .o_Switch(w_Switch_2));

Debounce_Switch Switch_3
  (.i_Clk(i_Clk),
   .i_Switch(i_Switch_3),
   .o_Switch(w_Switch_3));

Debounce_Switch Switch_4
  (.i_Clk(i_Clk),
   .i_Switch(i_Switch_4),
   .o_Switch(w_Switch_4));

Pong_Top #(.c_TOTAL_COLS(c_TOTAL_COLS),
           .c_TOTAL_ROWS(c_TOTAL_ROWS),
           .c_ACTIVE_COLS(c_ACTIVE_COLS),
           .c_ACTIVE_ROWS(c_ACTIVE_ROWS)) Pong_Inst
(.i_Clk(i_Clk),
 .i_HSync(w_HSync_VGA),
 .i_VSync(w_VSync_VGA),
 .i_Game_Start(w_RX_DV),
 .i_Paddle_Up_P1(w_Switch_1),
 .i_Paddle_Dn_P1(w_Switch_2),
 .i_Paddle_Up_P2(w_Switch_3),
 .i_Paddle_Dn_P2(w_Switch_4),
 .o_HSync(w_HSync_Pong),
 .o_VSync(w_VSync_Pong),
 .o_Red_Video(w_Red_Video_Pong),
 .o_Grn_Video(w_Grn_Video_Pong),
 .o_Blu_Video(w_Blu_Video_Pong));

VGA_Sync_Porch  #(.VIDEO_WIDTH(c_VIDEO_WIDTH),
                  .TOTAL_COLS(c_TOTAL_COLS),
                  .TOTAL_ROWS(c_TOTAL_ROWS),
                  .ACTIVE_COLS(c_ACTIVE_COLS),
                  .ACTIVE_ROWS(c_ACTIVE_ROWS))
VGA_Sync_Porch_Inst
 (.i_Clk(i_Clk),
  .i_HSync(w_HSync_Pong),
  .i_VSync(w_VSync_Pong),
  .i_Red_Video(w_Red_Video_Pong),
  .i_Grn_Video(w_Grn_Video_Pong),
  .i_Blu_Video(w_Blu_Video_Pong),
  .o_HSync(o_VGA_HSync),
```

```
    .o_VSync(o_VGA_VSync),
    .o_Red_Video(w_Red_Video_Porch),
    .o_Grn_Video(w_Grn_Video_Porch),
    .o_Blu_Video(w_Blu_Video_Porch));

  assign o_VGA_Red_0 = w_Red_Video_Porch[0];
  assign o_VGA_Red_1 = w_Red_Video_Porch[1];
  assign o_VGA_Red_2 = w_Red_Video_Porch[2];

  assign o_VGA_Grn_0 = w_Grn_Video_Porch[0];
  assign o_VGA_Grn_1 = w_Grn_Video_Porch[1];
  assign o_VGA_Grn_2 = w_Grn_Video_Porch[2];

  assign o_VGA_Blu_0 = w_Blu_Video_Porch[0];
  assign o_VGA_Blu_1 = w_Blu_Video_Porch[1];
  assign o_VGA_Blu_2 = w_Blu_Video_Porch[2];

endmodule
```

PONG_TOP.v:

```
module Pong_Top
  #(parameter c_TOTAL_COLS=800,
    parameter c_TOTAL_ROWS=525,
    parameter c_ACTIVE_COLS=640,
    parameter c_ACTIVE_ROWS=480)
  (input            i_Clk,
   input            i_HSync,
   input            i_VSync,
   // Game Start Button
   input            i_Game_Start,
   // Player 1 and Player 2 Controls (Controls Paddles)
   input            i_Paddle_Up_P1,
   input            i_Paddle_Dn_P1,
   input            i_Paddle_Up_P2,
   input            i_Paddle_Dn_P2,
   // Output Video
   output reg       o_HSync,
   output reg       o_VSync,
   output [3:0] o_Red_Video,
   output [3:0] o_Grn_Video,
   output [3:0] o_Blu_Video);

  // Local Constants to Determine Game Play
  parameter c_GAME_WIDTH  = 40;
  parameter c_GAME_HEIGHT = 30;
  parameter c_SCORE_LIMIT = 9;
  parameter c_PADDLE_HEIGHT = 6;
  parameter c_PADDLE_COL_P1 = 0;  // Col Index of Paddle for P1
  parameter c_PADDLE_COL_P2 = c_GAME_WIDTH-1; // Index for P2

  // State machine enumerations
  parameter IDLE    = 3'b000;
  parameter RUNNING = 3'b001;
```

```
parameter P1_WINS = 3'b010;
parameter P2_WINS = 3'b011;
parameter CLEANUP = 3'b100;

reg [2:0] r_SM_Main = IDLE;

wire       w_HSync, w_VSync;
wire [9:0] w_Col_Count, w_Row_Count;

wire       w_Draw_Paddle_P1, w_Draw_Paddle_P2;
wire [5:0] w_Paddle_Y_P1, w_Paddle_Y_P2;
wire       w_Draw_Ball, w_Draw_Any;
wire [5:0] w_Ball_X, w_Ball_Y;

reg [3:0] r_P1_Score = 0;
reg [3:0] r_P2_Score = 0;

// Divided version of the Row/Col Counters
// Allows us to make the board 40x30
wire [5:0] w_Col_Count_Div, w_Row_Count_Div;

Sync_To_Count #(.TOTAL_COLS(c_TOTAL_COLS),
                .TOTAL_ROWS(c_TOTAL_ROWS)) Sync_To_Count_Inst
  (.i_Clk(i_Clk),
   .i_HSync(i_HSync),
   .i_VSync(i_VSync),
   .o_HSync(w_HSync),
   .o_VSync(w_VSync),
   .o_Col_Count(w_Col_Count),
   .o_Row_Count(w_Row_Count));

// Register syncs to align with output data.
always @(posedge i_Clk)
begin
  o_HSync <= w_HSync;
  o_VSync <= w_VSync;
end

// Drop 4 LSBs, which effectively divides by 16
assign w_Col_Count_Div = w_Col_Count[9:4];
assign w_Row_Count_Div = w_Row_Count[9:4];


// Instantiation of Paddle Control + Draw for Player 1
Pong_Paddle_Ctrl #(.c_PLAYER_PADDLE_X(c_PADDLE_COL_P1),
                   .c_GAME_HEIGHT(c_GAME_HEIGHT)) P1_Inst
  (.i_Clk(i_Clk),
   .i_Col_Count_Div(w_Col_Count_Div),
   .i_Row_Count_Div(w_Row_Count_Div),
   .i_Paddle_Up(i_Paddle_Up_P1),
   .i_Paddle_Dn(i_Paddle_Dn_P1),
   .o_Draw_Paddle(w_Draw_Paddle_P1),
   .o_Paddle_Y(w_Paddle_Y_P1));

// Instantiation of Paddle Control + Draw for Player 2
```

```
Pong_Paddle_Ctrl #(.c_PLAYER_PADDLE_X(c_PADDLE_COL_P2),
                   .c_GAME_HEIGHT(c_GAME_HEIGHT)) P2_Inst
  (.i_Clk(i_Clk),
   .i_Col_Count_Div(w_Col_Count_Div),
   .i_Row_Count_Div(w_Row_Count_Div),
   .i_Paddle_Up(i_Paddle_Up_P2),
   .i_Paddle_Dn(i_Paddle_Dn_P2),
   .o_Draw_Paddle(w_Draw_Paddle_P2),
   .o_Paddle_Y(w_Paddle_Y_P2));

// Instantiation of Ball Control + Draw
Pong_Ball_Ctrl Pong_Ball_Ctrl_Inst
  (.i_Clk(i_Clk),
   .i_Game_Active(w_Game_Active),
   .i_Col_Count_Div(w_Col_Count_Div),
   .i_Row_Count_Div(w_Row_Count_Div),
   .o_Draw_Ball(w_Draw_Ball),
   .o_Ball_X(w_Ball_X),
   .o_Ball_Y(w_Ball_Y));


// Create a state machine to control the state of play
always @(posedge i_Clk)
begin
  case (r_SM_Main)

  // Stay in this state until Game Start button is hit
  IDLE :
  begin
    if (i_Game_Start == 1'b1)
      r_SM_Main <= RUNNING;
  end

  // Stay in this state until either player misses the ball
  // can only occur when the Ball is at 0 to c_GAME_WIDTH-1
  RUNNING :
  begin
    // Player 1 Side
    if (w_Ball_X == 0 &&
        (w_Ball_Y < w_Paddle_Y_P1 ||
         w_Ball_Y > w_Paddle_Y_P1 + c_PADDLE_HEIGHT))
      r_SM_Main <= P2_WINS;

    // Player 2 Side
    else if (w_Ball_X == c_GAME_WIDTH-1 &&
             (w_Ball_Y < w_Paddle_Y_P2 ||
              w_Ball_Y > w_Paddle_Y_P2  + c_PADDLE_HEIGHT))
      r_SM_Main <= P1_WINS;
  end

  P1_WINS :
  begin
    if (r_P1_Score == c_SCORE_LIMIT-1)
      r_P1_Score <= 0;
    else
    begin
```

```
        r_P1_Score <= r_P1_Score + 1;
        r_SM_Main <= CLEANUP;
      end
    end

    P2_WINS :
    begin
      if (r_P2_Score == c_SCORE_LIMIT-1)
        r_P2_Score <= 0;
      else
      begin
        r_P2_Score <= r_P2_Score + 1;
        r_SM_Main <= CLEANUP;
      end
    end

    CLEANUP :
      r_SM_Main <= IDLE;

    endcase
  end

  // Conditional Assignment based on State Machine state
  assign w_Game_Active = (r_SM_Main == RUNNING) ? 1'b1 : 1'b0;

  assign w_Draw_Any = w_Draw_Ball | w_Draw_Paddle_P1 | w_Draw_Paddle_P2;

  // Assign colors. Currently set to only 2 colors, white or black.
  assign o_Red_Video = w_Draw_Any ? 4'b1111 : 4'b0000;
  assign o_Grn_Video = w_Draw_Any ? 4'b1111 : 4'b0000;
  assign o_Blu_Video = w_Draw_Any ? 4'b1111 : 4'b0000;

endmodule
```

PONG_PADDLE_CONTROL:

```
module Pong_Paddle_Ctrl
  #(parameter c_PLAYER_PADDLE_X=0,
    parameter c_PADDLE_HEIGHT=6,
    parameter c_GAME_HEIGHT=30)
  (input            i_Clk,
   input [5:0]      i_Col_Count_Div,
   input [5:0]      i_Row_Count_Div,
   input            i_Paddle_Up,
   input            i_Paddle_Dn,
   output reg       o_Draw_Paddle,
   output reg [5:0] o_Paddle_Y);

  // Set the Speed of the paddle movement.
  // In this case, the paddle will move one board game unit
  // every 50 milliseconds that the button is held down.
  parameter c_PADDLE_SPEED = 1250000;

  reg [31:0] r_Paddle_Count = 0;
```

```
  wire w_Paddle_Count_En;

  // Only allow paddles to move if only one button is pushed.
  // ^ is an XOR bitwise operation.
  assign w_Paddle_Count_En = i_Paddle_Up ^ i_Paddle_Dn;

  always @(posedge i_Clk)
  begin
    if (w_Paddle_Count_En == 1'b1)
    begin
      if (r_Paddle_Count == c_PADDLE_SPEED)
        r_Paddle_Count <= 0;
      else
        r_Paddle_Count <= r_Paddle_Count + 1;
    end

    // Update the Paddle Location slowly.  Only allowed when the
    // Paddle Count reaches its limit.  Don't update if paddle is
    // already at the top of the screen.
    if (i_Paddle_Up == 1'b1 && r_Paddle_Count == c_PADDLE_SPEED &&
        o_Paddle_Y !== 0)
      o_Paddle_Y <= o_Paddle_Y - 1;
    else if (i_Paddle_Dn == 1'b1 && r_Paddle_Count == c_PADDLE_SPEED &&
             o_Paddle_Y !== c_GAME_HEIGHT-c_PADDLE_HEIGHT-1)
      o_Paddle_Y <= o_Paddle_Y + 1;

  end


  // Draws the Paddles as determined by input parameter
  // c_PLAYER_PADDLE_X as well as o_Paddle_Y
  always @(posedge i_Clk)
  begin
    // Draws in a single column and in a range of rows.
    // Range of rows is determined by c_PADDLE_HEIGHT
    if (i_Col_Count_Div == c_PLAYER_PADDLE_X &&
        i_Row_Count_Div >= o_Paddle_Y &&
        i_Row_Count_Div <= o_Paddle_Y + c_PADDLE_HEIGHT)
      o_Draw_Paddle <= 1'b1;
    else
      o_Draw_Paddle <= 1'b0;
  end

endmodule
```

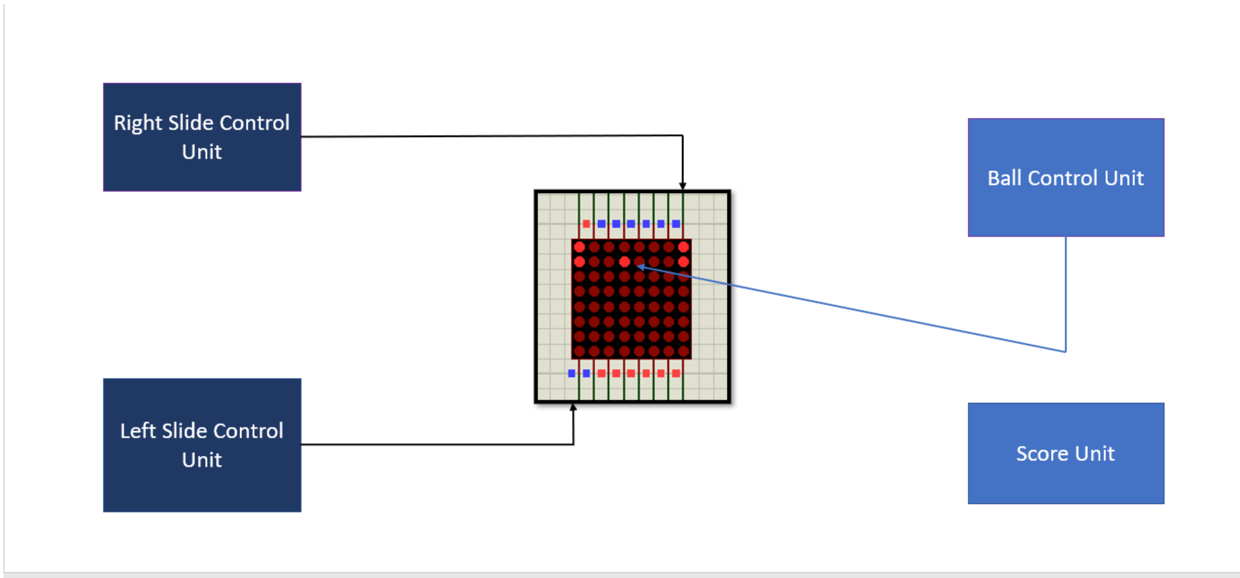
PONG_BALL_CONTROL:

```
module Pong_Ball_Ctrl
  #(parameter c_GAME_WIDTH=40,
    parameter c_GAME_HEIGHT=30)
  (input            i_Clk,
   input            i_Game_Active,
   input [5:0]      i_Col_Count_Div,
   input [5:0]      i_Row_Count_Div,
   output reg       o_Draw_Ball,
```

```
 output reg [5:0] o_Ball_X = 0,
 output reg [5:0] o_Ball_Y = 0);

// Set the Speed of the ball movement.
// In this case, the ball will move one board game unit
// every 50 milliseconds that the button is held down.
parameter c_BALL_SPEED = 1250000;

reg [5:0]  r_Ball_X_Prev = 0;
reg [5:0]  r_Ball_Y_Prev = 0;
reg [31:0] r_Ball_Count = 0;

always @(posedge i_Clk)
begin
  // If the game is not active, ball stays in the middle of
  // screen until the game starts.
  if (i_Game_Active == 1'b0)
  begin
    o_Ball_X      <= c_GAME_WIDTH/2;
    o_Ball_Y      <= c_GAME_HEIGHT/2;
    r_Ball_X_Prev <= c_GAME_WIDTH/2 + 1;
    r_Ball_Y_Prev <= c_GAME_HEIGHT/2 - 1;
  end

  // Update the ball counter continuously.  Ball movement
  // update rate is determined by input parameter
  // If ball counter is at its limit, update the ball position
  // in both X and Y.
  else
  begin
    if (r_Ball_Count < c_BALL_SPEED)
      r_Ball_Count <= r_Ball_Count + 1;
    else
    begin
      r_Ball_Count <= 0;

      // Store Previous Location to keep track of movement
      r_Ball_X_Prev <= o_Ball_X;
      r_Ball_Y_Prev <= o_Ball_Y;

      // When Previous Value is less than current value, ball is moving
      // to right.  Keep it moving to the right unless we are at wall.
      // When Prevous value is greater than current value, ball is moving
      // to left.  Keep it moving to the left unless we are at a wall.
      // Same philosophy for both X and Y.

      if ((r_Ball_X_Prev < o_Ball_X && o_Ball_X == c_GAME_WIDTH-1) ||
          (r_Ball_X_Prev > o_Ball_X && o_Ball_X != 0))
        o_Ball_X <= o_Ball_X - 1;
      else
        o_Ball_X <= o_Ball_X + 1;

      if ((r_Ball_Y_Prev < o_Ball_Y && o_Ball_Y == c_GAME_HEIGHT-1) ||
          (r_Ball_Y_Prev > o_Ball_Y && o_Ball_Y != 0))
        o_Ball_Y <= o_Ball_Y - 1;
      else
```

```
          o_Ball_Y <= o_Ball_Y + 1;
      end
    end
  end // always @ (posedge i_Clk)


  // Draws a ball at the location determined by X and Y indexes.
  always @(posedge i_Clk)
  begin
    if (i_Col_Count_Div == o_Ball_X && i_Row_Count_Div == o_Ball_Y)
      o_Draw_Ball <= 1'b1;
    else
      o_Draw_Ball <= 1'b0;
  end


endmodule
```

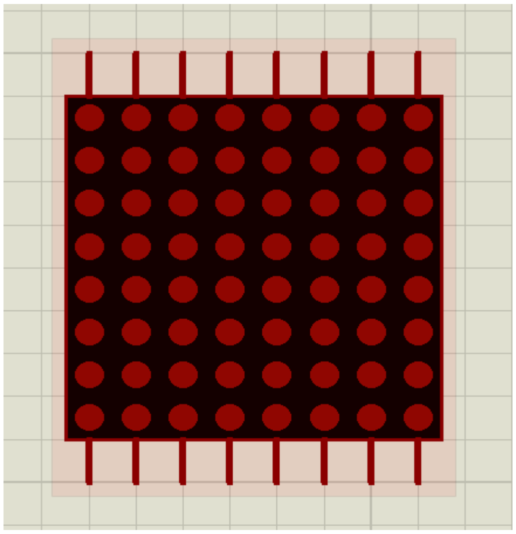
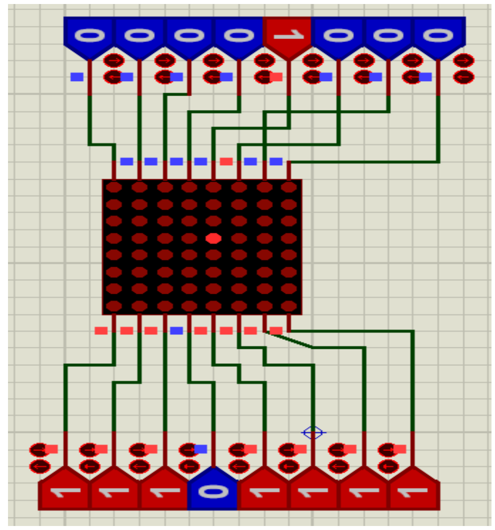
Proteus Implementation



Display:

In Proteus 8x8 LED matrix is used as the display. Here columns and rows of the LED matrix are internally shorted. The LED matrix is like the matrix of LEDs as below.

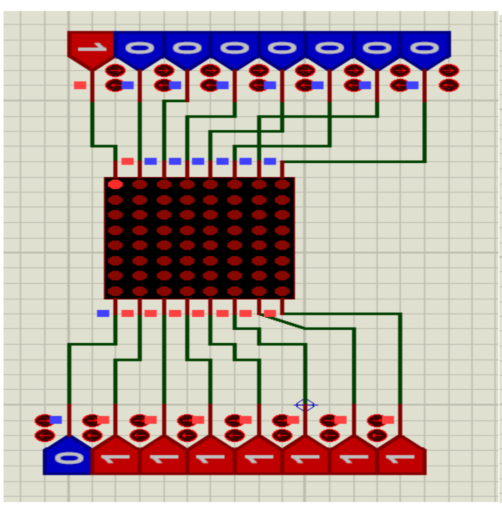
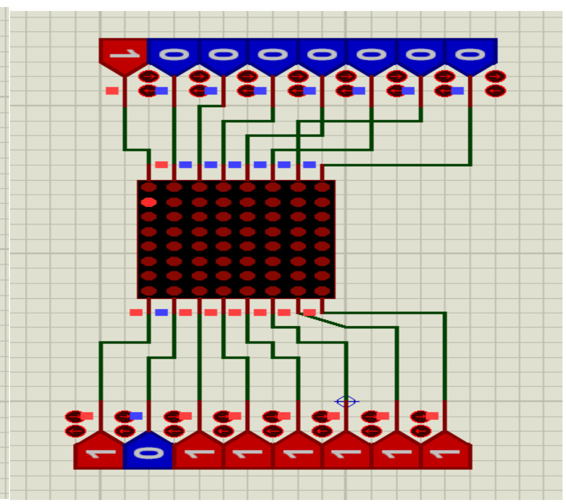
0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7
6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7
7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7

So for turning on the (3,4)th LED the 4th column has to be in HIGH state and the 3rd row in the LOW state.

Left or Right Slide/Paddle Control:

For the left or right paddle control, a UP/DOWN counter has been used. The outputs of the counter are taken as the input of a 3to8 decoder. As a result if the counter counts 1 from 0 then the second row of the LED matrix will be in the LOW state. So the LED will move downward as shown in the next figures.

For displaying the paddle 2 LEDs are used. So for turning on another LED, 1 is added to the counter and then the sum is taken as the inputs of another 3to8 decoder. Similarly, Right Paddle is created. But the difference is now column 7 will be in high state instead of column 0.

Ball Control:

For controlling the ball movement somewhat similar approach like the paddles is taken. The difference is in case of paddles the columns are in the constant state whereas for ball control both rows and columns will be changing the logic states. For example, column 0 is given the HIGH state and other columns are at LOW states for left paddle. As the ball moves diagonally, 4 kind of motion can be possible.

i) Diagonally right upward (row will decrease and column will increase)
ii) Diagonally left upward (both row and column will decrease)
iii) Diagonally right downward (both row and column will increase)
iv) Diagonally left downward (row will increase and column will decrease)

So, both the row and column counters count from 0 to 7 and again from 7 to 0. The motion of the ball is detected by comparing the previous position of the ball to the present position and the counter changes its direction of counting accordingly. For this purpose, D flip-flops are used for storing the previous bits.

Now, if the ball collides with the paddle then ball will change its direction with 90-degree angle instead of moving to the 0^{th} or 7^{th} column. To detect the collision the row position of the ball at 1^{st} and 6^{th} column and the row position of the left and right paddles were compared.

Score Unit:

The score system is based on the logic that if the ball goes to the 7^{th} column, left player will be awarded a point. Again, the right player will be awarded a point if the ball goes to the 0^{th} column. The scores are displayed with the help of two 7 segment displays.

Persistence of vision (POV):

Here persistence of vision, one type of optical illusion, is used. The persistence of vision of normal human eye is 1/16 seconds. So, if any LED blinks in a regular interval of less than the POV, our human eye will consider the blinking as the continuous light. In this project, the right paddle, the ball and the left paddle blink respectively with 1/500 seconds interval. As a result, the management of the LEDs becomes easier and power efficient.

Schematic Diagram:

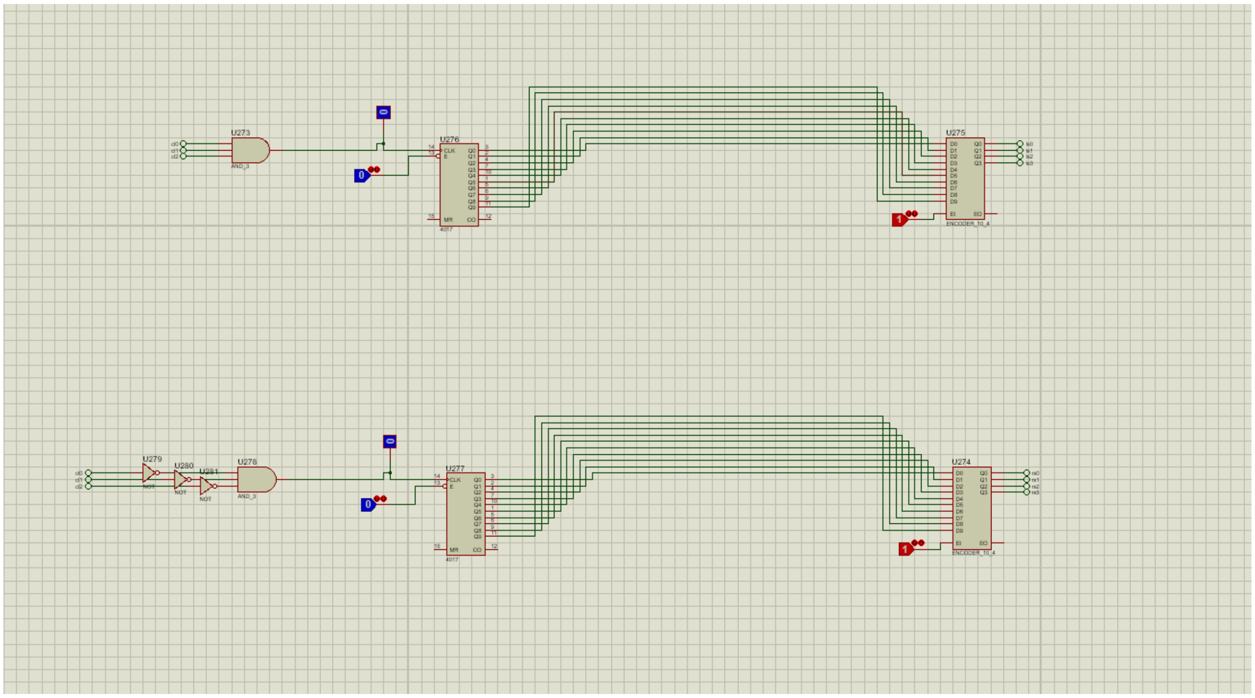


Figure 1:Score Unit

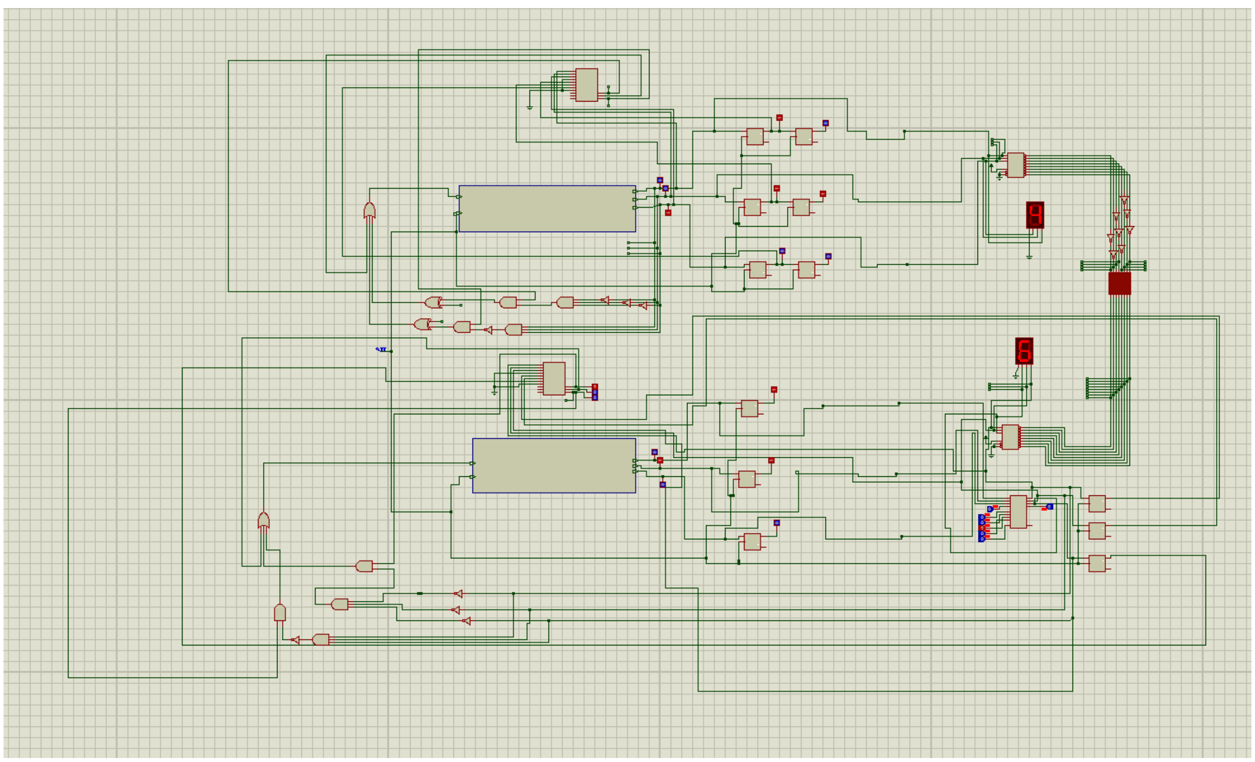


Figure 2: Ball Control (Part 1)

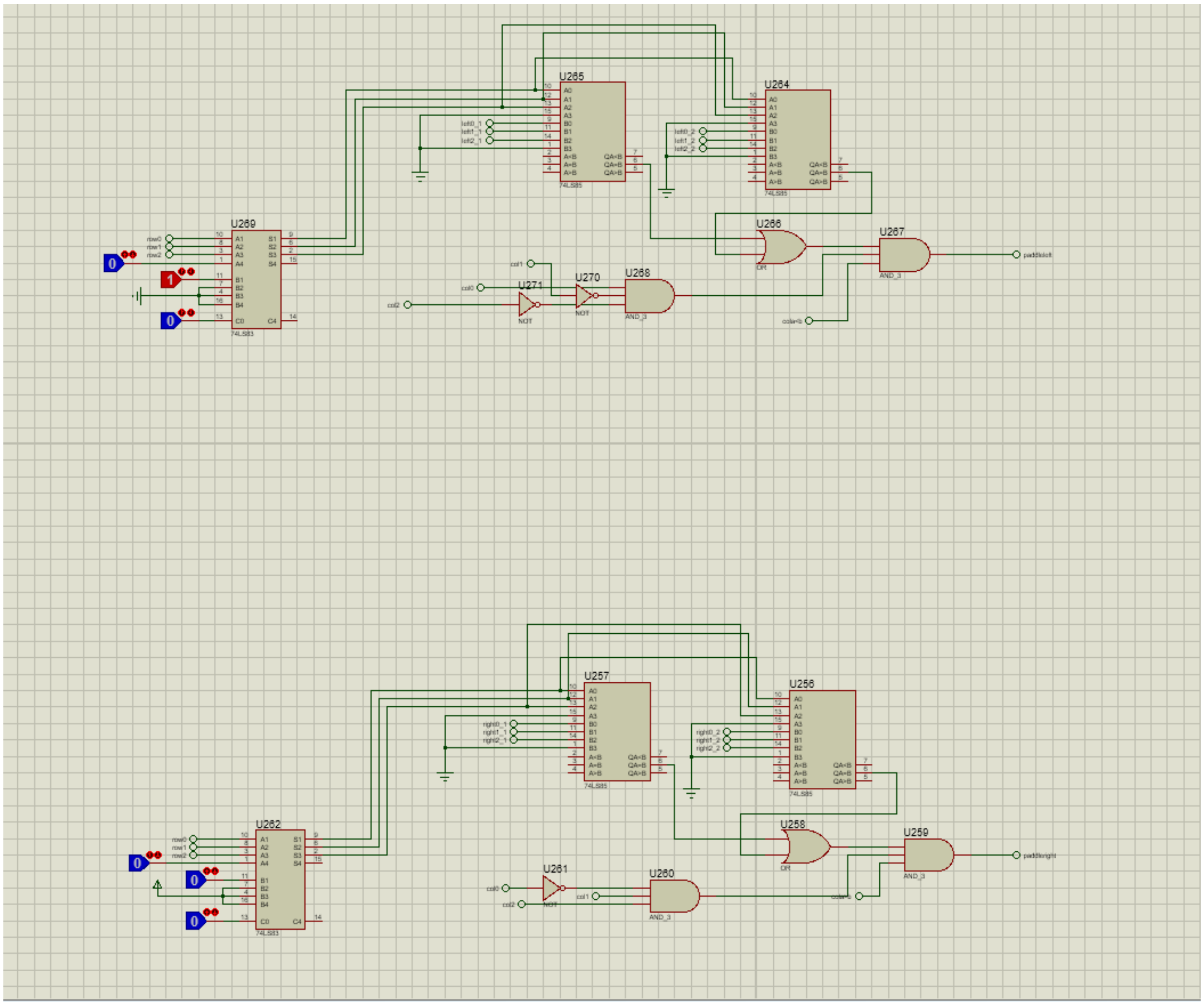


Figure 3: Ball Control part 2 (For detecting the collision)

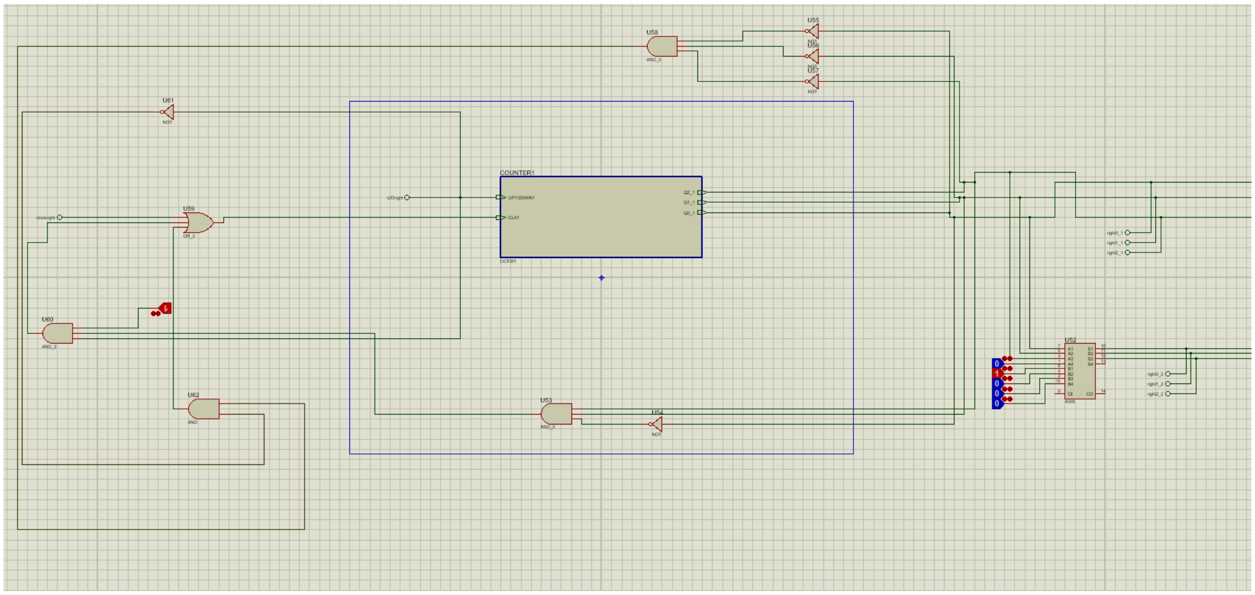


Figure 4: Paddle Control Counter Part

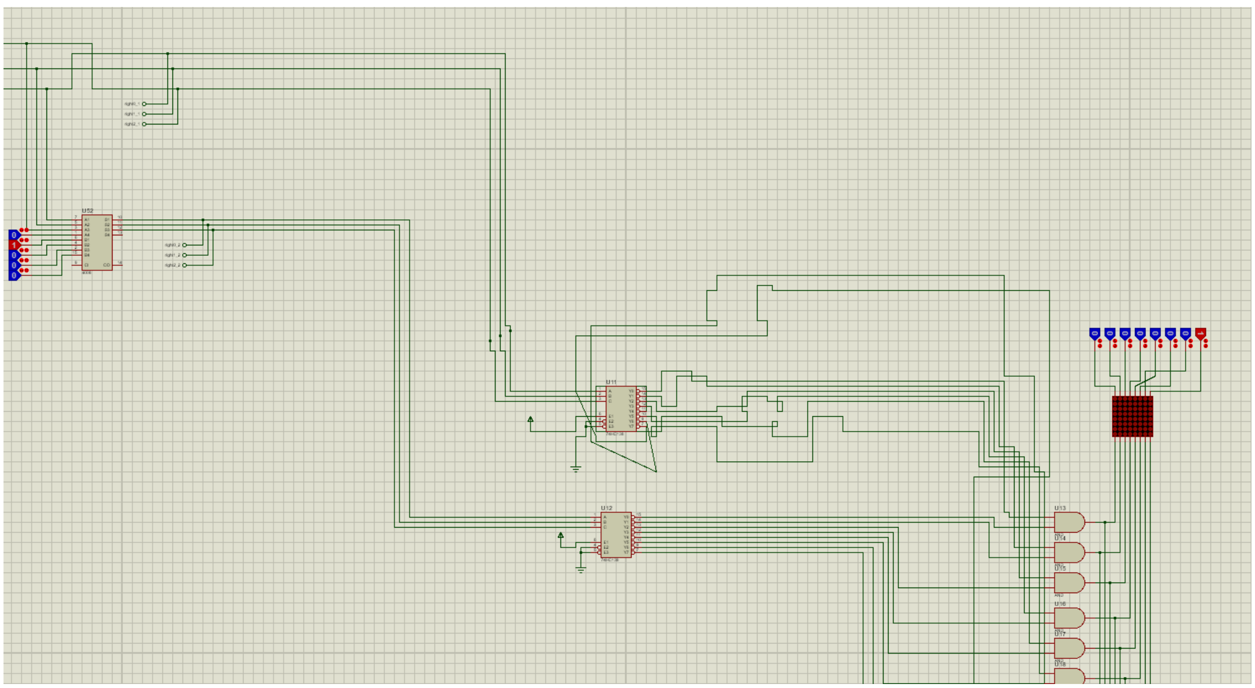


Figure 5: Paddle Control from counter to LED matrix part

Contribution:

1606118: Proteus Implementation + Report Writing+ Video recording

1606130: Idea implementation + Report Writing + Video editing

1606106: Coding + Report Writing+ Video recording