2021

EECS3216 - Project

CLASSIC PING PONG GAME

CHANDLER CABRERA 214407571

Submission

This project submission includes the following:

1. The Report (The document you are reading now)
2. Verilog files *PongProject.v* (top level) and *SevenSegment.v*
3. Demo Video

Snippets of the Verilog code are placed throughout this report for ease of reference. However, the full Verilog implementation will be submitted alongside it should further analysis be required.

Objective

To implement the specifications listed in the *Project Proposal.*

Specification

From Project Proposal

1. Game has the following functionality:
    a. Player controls a paddle
    b. Player prevents the ball from leaving the playable area by leaving the left side of the screen
2. Display the game state on a monitor using the VGA port on the DE-10 Lite board
3. Allow the player to control the paddle using a physical interface
4. Display the score on the DE-10 Lite board using the seven-segment display

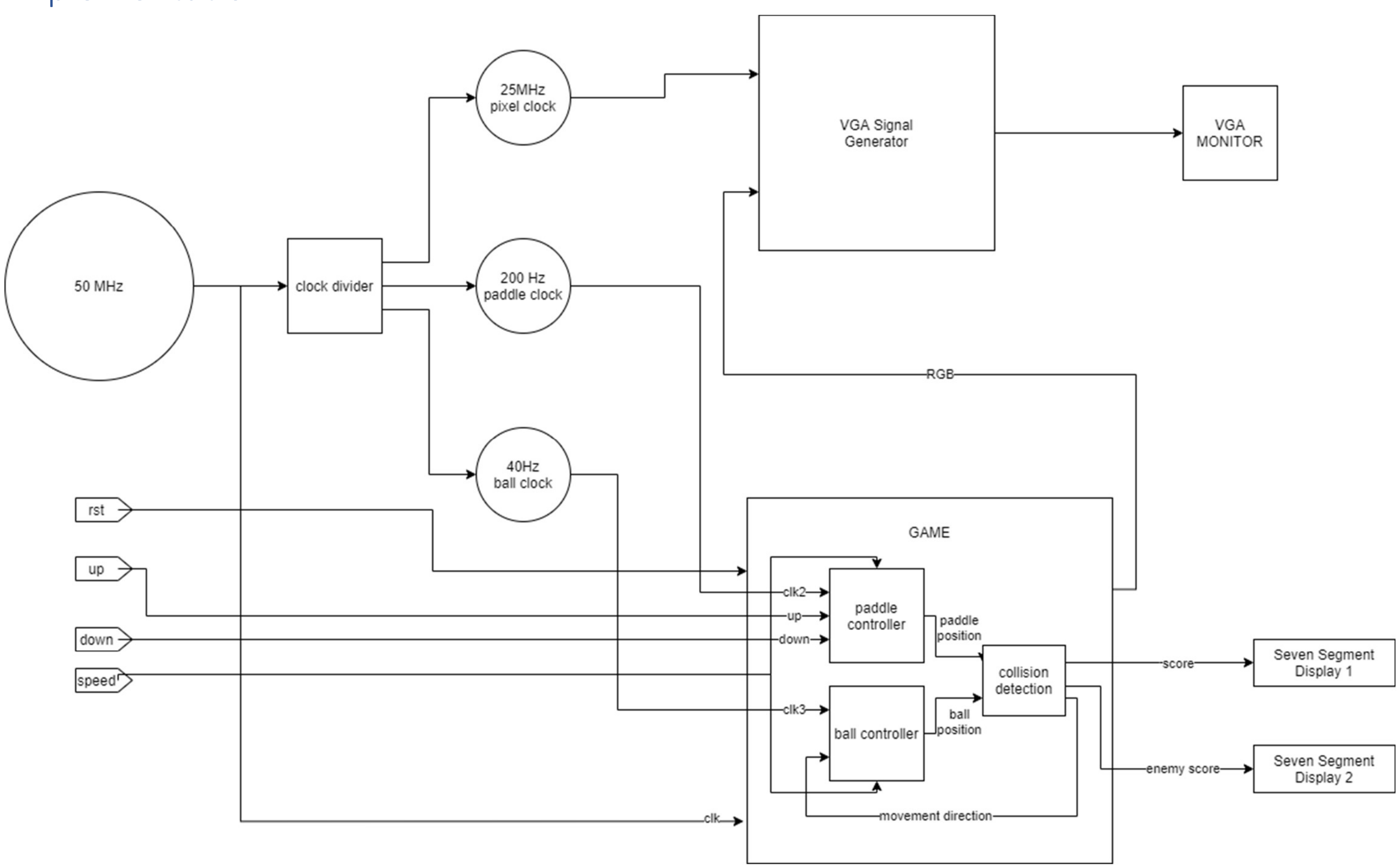
Additional Functionality Beyond the Proposal

5. AI controls another paddle located on the opposite
6. Score is determined by which "side" a goal is scored on

Components Used

1. DE-10 Lite board
2. Quartus Software
3. VGA Cable
4. VGA Monitor (ASUS VP228QG)

Implementation



Generating a VGA Signal

Generating a video signal on a monitor first requires a bit of information from the monitor first. The monitor I used was an ASUS VP228QG monitor, and according to its specification, supports the following resolution and refresh rates.

I chose to generate a 640 x 480 resolution, at 60Hz, as it requires the least amount of calculation by the processor to generate a full frame.

The way a VGA signal is generated is as follows:

1. The screen, starting from the origin (top left of the screen), is scanned horizontally, pixel by pixel. If a signal called "Horizontal Sync" is pulsed HIGH.
2. After the horizontal sync has pulsed, and 48 more pixels have been scanned through (called a back porch), for a period of 640 pixels, the RGB data is asserted high. The front porch (16 pixels) are then scanned through.
3. Once each pixel in the horizontal row is iterated on, the iterator moves one pixel vertically, pulses the "Vertical Sync" signal, and repeats from Step 1.
4. Once the last pixel is reached, a full frame has been scanned, and the process repeats.

Resolution	Refresh Rate	Horizontal Frequency
640x480	60Hz	31.469kHz
640x480	72Hz	37.861kHz
640x480	75Hz	37.5kHz
800x600	56Hz	35.156kHz
800x600	60Hz	37.879kHz
800x600	72Hz	48.077kHz
800x600	75Hz	46.875kHz
1024x768	60Hz	48.363kHz
1024x768	70Hz	56.476kHz
1024x768	75Hz	60.023kHz
1152x864	75Hz	67.5kHz
1280x960	60Hz	60kHz
1280x1024	60Hz	63.981kHz
1280x1024	75Hz	79.976kHz
1440x900	60Hz	55.935kHz
1440x900	75Hz	70.635kHz
1680x1050	60Hz	65.29kHz
1920x1080	60Hz	67.5kHz
1920x1080(for VP228QG)	75Hz	83.894kHz

Figure 1: Supported resolutions/refresh rate of monitor from ASUS monitor user manual

The timing specification for the chosen resolution and refresh rate are also given by the DE-10 manual,

As seen from figure 2, in order to generate a 60Hz signal at this resolution, a pixel clock of 25 MHz is required. The display interval (when the RGB data is valid) is also given by Table 3-9 and Table 3-10 in the DE-10 manual.

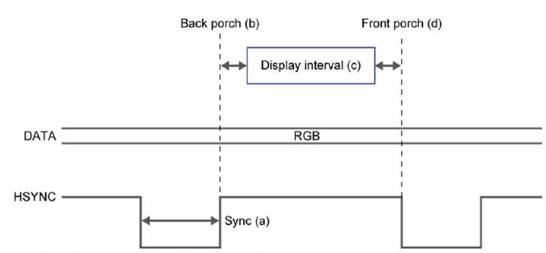


Figure 3-22 VGA horizontal timing specification

Table 3-9 VGA Horizontal Timing Specification

VGA mode		Horizontal Timing Spec				
Configuration	Resolution(HxV)	a(pixel clock cycle)	b(pixel clock cycle)	c(pixel clock cycle)	d(pixel clock cycle)	Pixel clock(MHz)
VGA(60Hz)	640x480	96	48	640	16	25

Table 3-10 VGA Vertical Timing Specification

VGA mode		Vertical Timing Spec				
Configuration	Resolution(HxV)	a(lines)	b(lines)	c(lines)	d(lines)	Pixel clock(MHz)
VGA(60Hz)	640x480	2	33	480	10	25

Figure 2: Timing data from DE-10 User Manual

Implementing the VGA Signal Generator

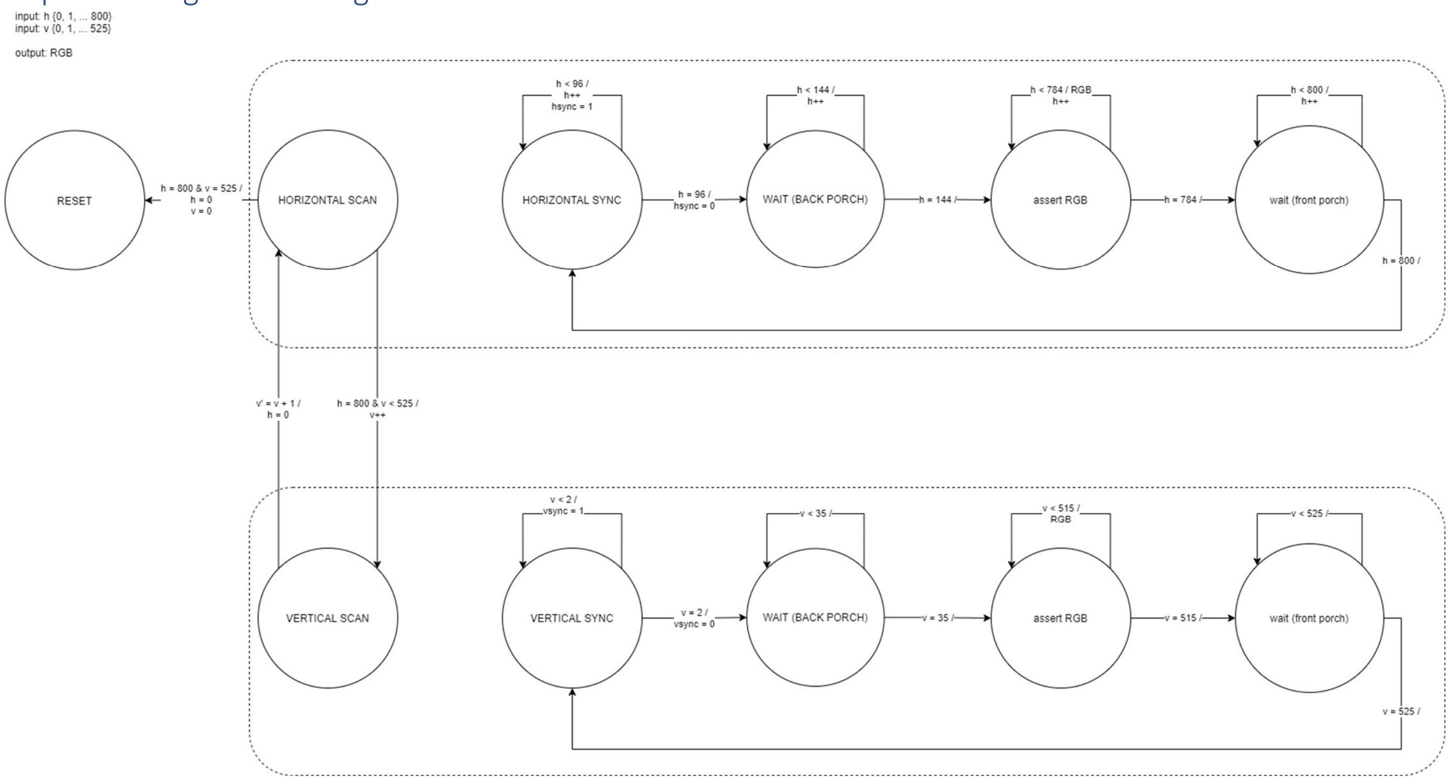


Figure 3: Hierarchal FSM of VGA Signal Generation

This Mealy Finite State Machine shows the deterministic pattern of how a VGA signal is created. Its Verilog implementation follows.

Shown below are snippets of the Verilog code. The full implementation will be submitted alongside the report.

```
//VARIOUS CLOCKS
reg clk25 = 0;
// HORIZONTAL SYNC AND VERTICAL SYNC RELATED
reg enable_v_counter;
reg [15:0] h_counter = 0;
reg [15:0] v_counter = 0;
// colour registers
reg [3:0] r;
reg [3:0] g;
reg [3:0] b;
always @(posedge clk) begin
    clk25 <= ~clk25;
```

registers to store colour data

clock divider to convert 50MHz source clock to 25MHz pixel clock

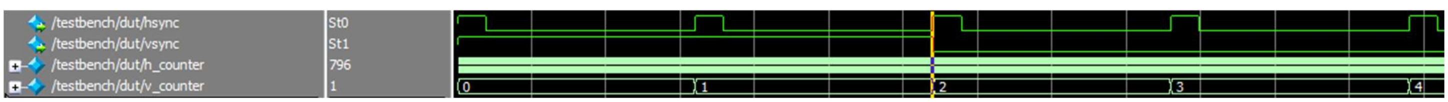
```verilog
//horizontal counter
always @(posedge clk25) begin
    if (h_counter < 799) begin
        h_counter <= h_counter + 1;
        enable_v_counter <= 0;
    end
    else begin
        h_counter <= 0;
        enable_v_counter <= 1;
    end
end

//vertical counter
always @(posedge clk25) begin
    if (enable_v_counter == 1'b1) begin
        if (v_counter < 524)
            v_counter <= v_counter + 1;
        else
            v_counter <= 0;
    end
end
assign hsync = (h_counter < 96) ? 1 : 0;
assign vsync = (v_counter < 2) ? 1 : 0;

assign red = r;
assign green = g;
assign blue = b;
```

The iterator counts 800 pixels in every horizontal row, and 525 pixels in every vertical column.

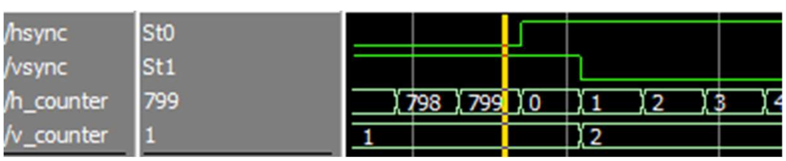
horizontal sync is pulsed to sync timing
vertical sync is pulsed to sync timing

colour registers are wired to colour output

VGA Generator Simulation



Using a testbench, the values of hsync and vsync can be seen. The horizontal and vertical sync pulse at the correct time.



By viewing the waveform with greater precision, the vertical counter increments correctly alongside the horizontal counter, according to the behaviour from the Hierarchal Finite State Machine shown in *Figure 3.*

Implementing the Ping Pong Game

```
module PongProject(
    input clk,
    input rst,
    input up,
    input down,
    input speed,
    output hsync,
    output vsync,
    output [3:0] red,
    output [3:0]green,
    output [3:0]blue,
    output [15:0] player_scoreboard,
    output [15:0] enemy_scoreboard
    );

    // register and wire reset
    reg rst_reg = 0;
    wire reset;

    // four counters for clock divider
    reg [20:0] i = 0;
    reg [20:0] j = 0;
    reg [25:0] k = 0;

    // colour registers
    reg [3:0] r;
    reg [3:0] g;
    reg [3:0] b;

    //BALL coordinates and direction of travel
    reg [15:0] ball_x = 220;
    reg [15:0] ball_y = 275;
    reg ball_x_dir = 1;
    reg ball_y_dir = 1;

    //PADDLE position, top of paddle, bottom of paddle
    reg [15:0] paddle_x = 200;
    reg [15:0] paddle_y = 275;
    reg [15:0] paddle_bottom;
    reg [15:0] paddle_top;

    //ENEMY paddle position, top of paddle, bottom of paddle
    reg [15:0] paddle2_x = 710;
    reg [15:0] paddle2_y = 275;
    reg [15:0] paddle2_bottom;
    reg [15:0] paddle2_top;

    reg [4:0] paddle_width = 5'b10100;

    //SCORE
    reg [6:0] score       = 7'b0000000;
    reg [6:0] enemy_score = 7'b0000000;
```

[x,y] coordinates of the ball

ball will travel in the positive x direction and positive y direction

player's paddle starting position

registers to store the position of the top and bottom part of the paddle

enemy player paddle's starting position

paddle width is the amount of space extending from the top and bottom of the paddle's origin

```
//BALL SPAWN LOCATION
parameter ball_start_x = 220;
parameter ball_start_y = 275;

//SCREEN BOUNDARY PARAMETERS
parameter h_min = 143, h_max = 784, v_min = 34, v_max = 515;

//GAME PARAMETERS and MODIFIERS
parameter ball_slow = 2, ball_fast = 5, enemy_paddle_bonus = 5;
reg[1:0] bounce_variance_x = 2'b01;
reg[1:0] bounce_variance_y = 2'b01;
```

screen boundary parameters for 480p display via VGA signal

modifiers and parameters to change game behaviour

```
//VARIOUS CLOCKS
reg clk25 = 0;
reg clk2  = 0;
reg clk3  = 0;
reg clkslow = 0;
reg enemy_clk = 0;

// HORIZONTAL SYNC AND VERTICAL SYNC RELATED
reg enable_v_counter;
reg [15:0] h_counter = 0;
reg [15:0] v_counter = 0;

SevenSegment p2_score_tracker(.clk(clk), .value(enemy_score),  .display(enemy_scoreboard));
SevenSegment p1_score_tracker(.clk(clk), .value(score),        .display(player_scoreboard));

// clock dividers to achieve 25MHz clock (for 60Hz display), slower clock for paddle control, and even slower clock for ball movement
always @(posedge clk) begin
   clk25 <= ~clk25;

   if (i >= 625_000) begin
      clk2 <= ~ clk2;
      i <= 0;
   end else begin
      if (speed)
         i <= i + 10;
      else
         i <= i + 5;
   end

   if (j >= 625_000) begin
      clk3 <= ~clk3;
      j <= 0;
   end else begin
      if (speed)
         j <= j + ball_fast;
      else
         j <= j + ball_slow;
   end

   if (k >= 25_000_000) begin
      clkslow = ~clkslow;
      k <= 0;
   end else
      k <= k + 1;
end
```

Seven Segment Display

The seven-segment display was implemented in a manner similar to previous labs done in this course.

```
module SevenSegment(
   input clk,
   input [6:0] value,
   output[15:0] display
   );

   reg [15:0] encoding = 16'b00000000_00000000;

   always @ (posedge clk) begin

      case(value)
         // encoding pattern:  dotgfedcba
          4'd0 : encoding[15:0] = 16'b11000000_11000000; //0
          4'd1 : encoding[15:0] = 16'b11000000_11111001; //1
          4'd2 : encoding[15:0] = 16'b11000000_10100100; //2
          4'd3 : encoding[15:0] = 16'b11000000_10110000; //3
          4'd4 : encoding[15:0] = 16'b11000000_10011001; //4
          4'd5 : encoding[15:0] = 16'b11000000_10010010; //5
          4'd6 : encoding[15:0] = 16'b11000000_10000010; //6
          4'd7 : encoding[15:0] = 16'b11000000_11111000; //7
          4'd8 : encoding[15:0] = 16'b11000000_10000000; //8
          4'd9 : encoding[15:0] = 16'b11000000_10010000; //9
          4'd10: encoding[15:0] = 16'b11111001_11000000; //10
      endcase
   end
   assign display = encoding[15:0];
endmodule
```

Game-Related Functions

All the following code was implemented with this simple Finite State Machine in mind. It describes the general behaviour of game state flow.

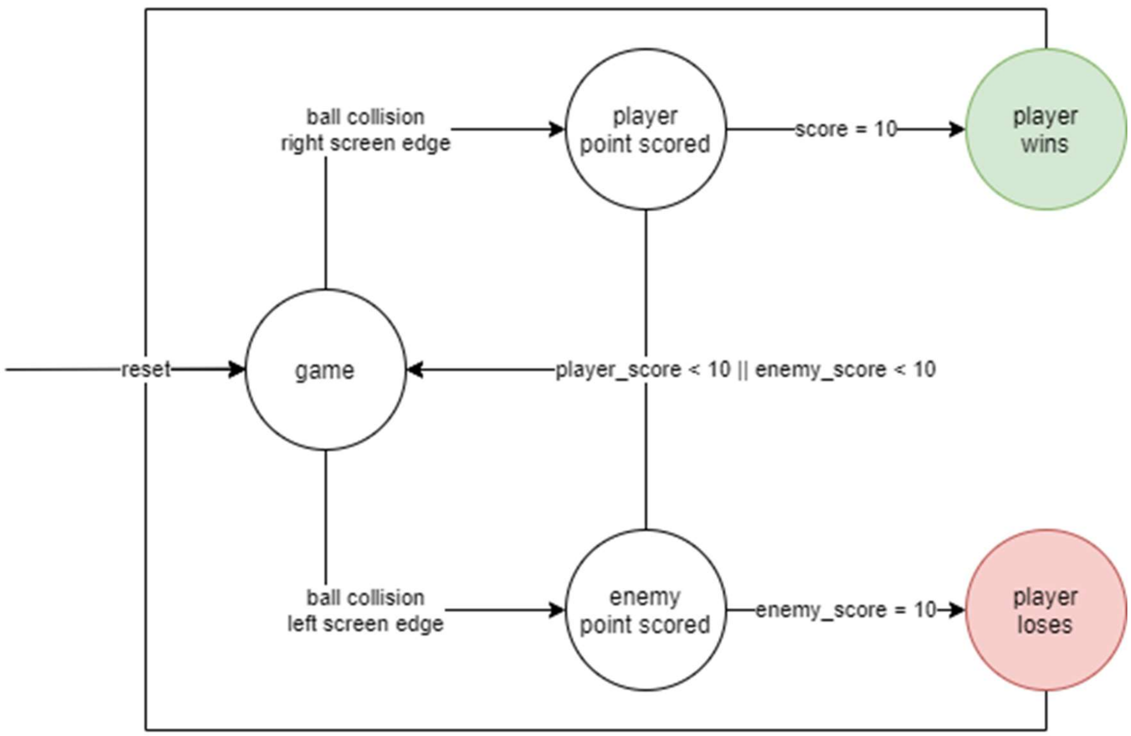


Figure 4: General Game State flow. Assume game state self-loops until one of the conditions are met.

According to this plan, the game must have:

- registers to store the player score, and the enemy score
- Collision Detection
  - Collisions with the left and right edges of the screen score points
  - Collisions with the top and bottom edges of the screen reverse the ball's y-direction
  - Collisions with the player or enemy paddle reverse the ball's x-direction
- Movement
  - Ball must move on its own according to a slower clock (so that it is humanly possible to predict the ball's movement)
  - Paddles must be able to move to give the player agency to win/lose the game
- Win state or lose state
  - Flash the screen GREEN when the player wins
  - Flash the screen RED when the player loses

Ball Movement and Collision Handling

```
// BALL MOVEMENT AND COLLISION HANDLING
always @(posedge clk3) begin
   if (reset) begin
      ball_x = ball_start_x;
      ball_y = ball_start_y;
      ball_x_dir = 1;
      ball_y_dir = 1;
      score <= 0;
      enemy_score = 0;
   end

   if (ball_x <= h_min + 1) begin
         ball_x = ball_start_x;
         ball_y = ball_start_y;
         ball_x_dir = 1;
         ball_y_dir = 1;
         enemy_score = enemy_score + 1;
   end

   // ball exited screen right
      if (ball_x >= h_max - 1) begin
         ball_x = ball_start_x + 400;
         ball_y = ball_start_y;
         ball_x_dir = 0;
         ball_y_dir = ~ball_y_dir;
         score <= score + 1;
      end

   // collision occurred
      if (ball_x >= paddle_x && ball_x <= paddle_x + 1 && ball_y >= paddle_bottom && ball_y <= paddle_top) begin
         if ((up && ball_y_dir) || down && ~ball_y_dir) begin
            bounce_variance_y <= 2'b10;
            bounce_variance_x <= 2'b01;
         end else if ((up && ~ball_y_dir) || (down && ~ball_y_dir)) begin
            bounce_variance_x <= 2'b10;
            bounce_variance_y <= 2'b01;
         end else begin
            bounce_variance_x <= 2'b01;
            bounce_variance_y <= 2'b01;
         end

         ball_x_dir = ~ball_x_dir;

      end else if (ball_x <= paddle2_x && ball_x >= paddle2_x - 1 && ball_y >= paddle2_bottom && ball_y <= paddle2_top) begin
         ball_x_dir = ~ball_x_dir;
      end

      if (ball_y >= (v_max - 3) || ball_y <= (v_min + 3))
         ball_y_dir = ~ball_y_dir;

      ball_x = (ball_x_dir) ? ball_x + bounce_variance_x : ball_x - bounce_variance_x;
      ball_y = (ball_y_dir) ? ball_y + bounce_variance_y : ball_y - bounce_variance_y;
end
```

- If reset, the ball is placed at a predetermined location, directions of movement are reset, and scores are reset
- First, check for collisions:
  - collision with screen edges:
    - If the x-coordinate of the ball reaches the left edge of the screen, the ball position/movement direction are reset, and the enemy is given a point
    - If the x-coordinate of the ball reaches the right edge of the screen, the ball position/movement direction are reset, and the player is given a point.
    - If the y-coordinate of the ball collides with the top or bottom of the screen, reverse its y direction
  - collision with game objects:
    - If the ball collides with the paddle, its movement in the x direction is reversed, and depending on the movement of the paddle at impact:
      - If the paddle was moving in the same direction as the ball, the ball will travel faster in the y direction
      - If the paddle was moving in the opposite direction as the ball, the ball will travel faster in the x direction
      - If the paddle was still, the ball moves equally fast in the x and y directions
- Second, move the ball in the x direction (amount of movement affected by *bounce variance x*)
- Third, move the ball in the y direction (amount of movement affected by *bounce variance y)*

Controlling the Player's Paddle

```
// CONTROLLING THE PLAYER'S PADDLE
always @(posedge clk2) begin

      if (up && ~down) begin
         if (paddle_y < v_max - paddle_width)
            paddle_y <= paddle_y + 1;
      end
      else if (~up && down) begin
         if (paddle_y > v_min + paddle_width)
            paddle_y <= paddle_y - 1;
      end

   paddle_top    <= paddle_y + paddle_width;
   paddle_bottom <= paddle_y - paddle_width;
end
```

To control the player's paddle, this always block listens for an up or down input signal and moves the paddle in the corresponding direction.

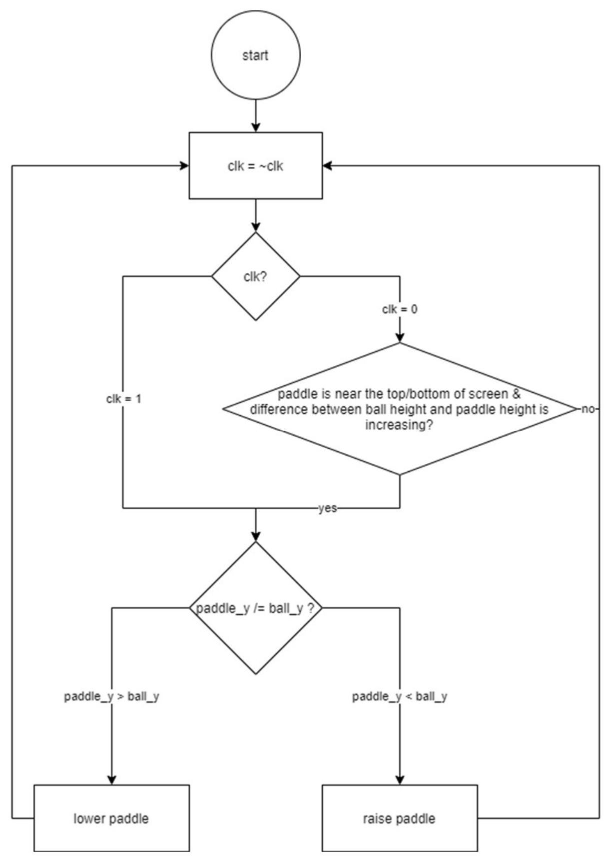
Since the paddle can only move on one axis, only two buttons are needed to control its movement.

It operates on a much slower clock than the source clock in order to be feasibly controlled.

Enemy Paddle's Behaviour

The behaviour of the enemy paddle (whose implementation is given below) is best described with the following decision tree:

```
//ENEMY AI BEHAVIOUR
always @(posedge clk2) begin
   enemy_clk = ~enemy_clk;
   if (enemy_clk) begin
      if (ball_y > paddle2_y)
         paddle2_y <= paddle2_y + 1;
      else if (ball_y < paddle2_y)
         paddle2_y <= paddle2_y - 1;
   end else begin
      if (paddle2_y >= v_max - 250 && ~ball_y_dir && paddle2_y > ball_y)
         paddle2_y <= paddle2_y - 1;
      else if (ball_y <= v_min + 250 && ball_y_dir && paddle2_y < ball_y)
         paddle2_y <= paddle2_y + 1;
   end
   paddle2_top <= paddle2_y + paddle_width + enemy_paddle_bonus;
   paddle2_bottom <= paddle2_y - paddle_width - enemy_paddle_bonus;
end
```



Generally, if the y-coordinate of the ball and the y-coordinate of the paddle are different, the paddle will move to close the gap.

However, this alone does not make for a sufficiently smart AI, as the paddle is often too late to catch the ball. In order to combat this, but still make the game winnable, I implemented a simple catchup mechanic that operates when the clock is LOW.

Every other clock cycle, if the paddle is closer to the top or bottom of the screen, and the ball's y-coordinate is moving away from the paddle, it will perform an additional move to help the paddle catch up to the ball.

Game State Handling

```
// GAME STATE HANDLING
always @(posedge clkslow) begin
   if (rst_reg)
      rst_reg = 0;

   if ((score >= 10 || enemy_score >= 10) && (up || down)) begin
      rst_reg = 1;
   end
end
```

If the reset switch was toggled recently, set it back to LOW.

If either player's score reaches the score limit, and the player inputs up or down, reset the game.

Drawing the Scenario

```
// DRAWING THE SCENARIO ON THE BOARD
always @(posedge clk) begin
        if (score >= 10) begin
           r <= 4'h0;
           g <= 4'hF;
           b <= 4'h0;
        end else if (enemy_score >= 10) begin
           r <= 4'hF;
           g <= 4'h0;
           b <= 4'h0;
        //draw ball
        end else if ((h_counter <= ball_x + 2 && h_counter >= ball_x && v_counter <= ball_y + 1 && v_counter >= ball_y - 1)) begin
           r <= 4'hF;
           g <= 4'hF;
           b <= 4'hF;
        // draw paddle
        end else if (v_counter >= paddle_bottom && v_counter <= paddle_top && h_counter == paddle_x) begin
           r <= 4'h0;
           g <= 4'hF;
           b <= 4'hF;
        // draw enemy paddle
        end else if (v_counter >= paddle2_bottom && v_counter <= paddle2_top && h_counter == paddle2_x) begin
           r <= 4'hF;
           g <= 4'h0;
           b <= 4'h0;
        // draw background
        end else if (h_counter < h_max && h_counter > h_min && v_counter < v_max && v_counter > v_min) begin
           r <= 4'h0;
           g <= 4'h0;
           b <= 4'h0;
        end
end
```

If the coordinate of the scanned pixel on screen corresponds to the position of the ball, paddle, or enemy paddle, change the colour of the pixel. Otherwise, draw the black background.

Wiring

```
   assign hsync = (h_counter < 96) ? 1 : 0;
   assign vsync = (v_counter < 2) ? 1 : 0;

   assign red = r;
   assign green = g;
   assign blue = b;

   assign reset = (rst_reg || rst);

endmodule
```

Pin Assignments

blue[3]	Output	PIN_N2	2	B2_N0	PIN_N2	2.5 V	12mA (default)	2 (default)
blue[2]	Output	PIN_P4	2	B2_N0	PIN_P4	2.5 V	12mA (default)	2 (default)
blue[1]	Output	PIN_T1	2	B2_N0	PIN_T1	2.5 V	12mA (default)	2 (default)
blue[0]	Output	PIN_P1	2	B2_N0	PIN_P1	2.5 V	12mA (default)	2 (default)
clk	Input	PIN_P11	3	B3_N0	PIN_P11	2.5 V	12mA (default)	
down	Input	PIN_A7	7	B7_N0	PIN_A7	2.5 V	12mA (default)	
enemy_scoreboard[15]	Output	PIN_A16	7	B7_N0	PIN_A16	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[14]	Output	PIN_B17	7	B7_N0	PIN_B17	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[13]	Output	PIN_A18	7	B7_N0	PIN_A18	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[12]	Output	PIN_A17	7	B7_N0	PIN_A17	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[11]	Output	PIN_B16	7	B7_N0	PIN_B16	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[10]	Output	PIN_E18	6	B6_N0	PIN_E18	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[9]	Output	PIN_D18	6	B6_N0	PIN_D18	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[8]	Output	PIN_C18	7	B7_N0	PIN_C18	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[7]	Output	PIN_D15	7	B7_N0	PIN_D15	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[6]	Output	PIN_C17	7	B7_N0	PIN_C17	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[5]	Output	PIN_D17	7	B7_N0	PIN_D17	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[4]	Output	PIN_E16	7	B7_N0	PIN_E16	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[3]	Output	PIN_C16	7	B7_N0	PIN_C16	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[2]	Output	PIN_C15	7	B7_N0	PIN_C15	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[1]	Output	PIN_E15	7	B7_N0	PIN_E15	2.5 V	12mA (default)	2 (default)
enemy_scoreboard[0]	Output	PIN_C14	7	B7_N0	PIN_C14	2.5 V	12mA (default)	2 (default)
green[3]	Output	PIN_R1	2	B2_N0	PIN_R1	2.5 V	12mA (default)	2 (default)
green[2]	Output	PIN_R2	2	B2_N0	PIN_R2	2.5 V	12mA (default)	2 (default)
green[1]	Output	PIN_T2	2	B2_N0	PIN_T2	2.5 V	12mA (default)	2 (default)
green[0]	Output	PIN_W1	2	B2_N0	PIN_W1	2.5 V	12mA (default)	2 (default)
hsync	Output	PIN_N3	2	B2_N0	PIN_N3	2.5 V	12mA (default)	2 (default)
player_scoreboard[15]	Output	PIN_L19	6	B6_N0	PIN_L19	2.5 V	12mA (default)	2 (default)
player_scoreboard[14]	Output	PIN_N20	6	B6_N0	PIN_N20	2.5 V	12mA (default)	2 (default)
player_scoreboard[13]	Output	PIN_N19	6	B6_N0	PIN_N19	2.5 V	12mA (default)	2 (default)
player_scoreboard[12]	Output	PIN_M20	6	B6_N0	PIN_M20	2.5 V	12mA (default)	2 (default)
player_scoreboard[11]	Output	PIN_N18	6	B6_N0	PIN_N18	2.5 V	12mA (default)	2 (default)
player_scoreboard[10]	Output	PIN_L18	6	B6_N0	PIN_L18	2.5 V	12mA (default)	2 (default)
player_scoreboard[9]	Output	PIN_K20	6	B6_N0	PIN_K20	2.5 V	12mA (default)	2 (default)
player_scoreboard[8]	Output	PIN_J20	6	B6_N0	PIN_J20	2.5 V	12mA (default)	2 (default)
player_scoreboard[7]	Output	PIN_F17	6	B6_N0	PIN_F17	2.5 V	12mA (default)	2 (default)
player_scoreboard[6]	Output	PIN_F20	6	B6_N0	PIN_F20	2.5 V	12mA (default)	2 (default)
player_scoreboard[5]	Output	PIN_F19	6	B6_N0	PIN_F19	2.5 V	12mA (default)	2 (default)

player_scoreboard[4]	Output	PIN_H19	6	B6_N0	PIN_H19	2.5 V		12mA (default)	2 (default)
player_scoreboard[3]	Output	PIN_J18	6	B6_N0	PIN_J18	2.5 V		12mA (default)	2 (default)
player_scoreboard[2]	Output	PIN_E19	6	B6_N0	PIN_E19	2.5 V		12mA (default)	2 (default)
player_scoreboard[1]	Output	PIN_E20	6	B6_N0	PIN_E20	2.5 V		12mA (default)	2 (default)
player_scoreboard[0]	Output	PIN_F18	6	B6_N0	PIN_F18	2.5 V		12mA (default)	2 (default)
red[3]	Output	PIN_Y1	3	B3_N0	PIN_Y1	2.5 V		12mA (default)	2 (default)
red[2]	Output	PIN_Y2	3	B3_N0	PIN_Y2	2.5 V		12mA (default)	2 (default)
red[1]	Output	PIN_V1	2	B2_N0	PIN_V1	2.5 V		12mA (default)	2 (default)
red[0]	Output	PIN_AA1	3	B3_N0	PIN_AA1	2.5 V		12mA (default)	2 (default)
rst	Input	PIN_C10	7	B7_N0	PIN_C10	2.5 V		12mA (default)	
speed	Input	PIN_F15	7	B7_N0	PIN_F15	2.5 V		12mA (default)	
up	Input	PIN_B8	7	B7_N0	PIN_B8	2.5 V		12mA (default)	
vsync	Output	PIN_N1	2	B2_N0	PIN_N1	2.5 V		12mA (default)	2 (default)

Conclusion

The project satisfies the *Specifications* in the following way:

Specification 1a

The player has the ability to move the paddle up and down.

Specification 1b

The player can prevent the ball from leaving the playable area by moving the paddle. Additionally, the player can try to make the ball leave the right side of the screen to score a point by using the paddle to bounce the ball back and affect its trajectory.

Specification 2

The game state is displayed on the monitor using the method described above in *Section: Implementing the VGA Generator* and *Drawing the Scenario.*

Specification 3

The player controls the paddle via the KEY0 and KEY1 buttons on the DE-10 board. Additionally, the player can reset the game using SW0 or change the difficulty of the game with SW9.

Specification 4

The player's score is displayed on the left-most seven segment displays. The enemy's score is displayed on the right-most displays.

Tools I used that I learned from EECS3216

- How to represent complex Finite State Machines as a Hierarchal State Machine
- Seven Segment Displays
- Clock Dividers
- Counters

What I learned:

- Implementing a Hierarchal Finite State Machine
- How to generate a Video Graphics Array signal
- How to create a player controlled entity
- How to handle collisions on a 2D grid

If I were to expand on this project, I would like to do the following:

- Add more game parameters to vary gameplay
  - Currently, the only variable to gameplay (that is set by the player) is the difficulty. SW9 controls how fast the ball and paddle move
- Add an external method to control paddles
  - As of right now, the only method to control the paddles is the KEY0 and KEY1 buttons. If I could use external devices to control the paddle, it creates options for the game to be multiplayer AND potentially finer control over the paddle.
- Score / statistics on screen instead of on the DE-10 board