ABSTRACT

The main objective of the project was to create a simple video game written in VHDL. The game involves paddles and a ball; users are to move the paddles up and down (with switches) and score on one another, much like the classic arcade game *Pong*. To accomplish this, an understanding of the Video Graphic Adaptor (VGA) had to be met so that an appropriate VGA controller could be created. Once that was completed, certain logic and mechanisms (in the form of various VHDL processes) had to be implemented to ensure that the game was displayed correctly and the objects interacted with each other in a desirable fashion. Upon completion of the project, a successful game was created. The ball moved on the screen correctly and bounced off of the boundaries and paddles as expected. Furthermore, the scoring mechanism also functioned properly. Overall, the project was a success.

INTRODUCTION

The main objective of the project was to create a simple game having much semblance to the classic *Pong* arcade game. In order to complete said task, a Video Graphic Adaptor (VGA) controller had to first be implemented; an understanding of the VGA sub-system was in order. The active image area for a standard VGA display contains 480 rows by 640 columns of picture elements (pixels). Each pixel contains red, green, and blue components which are illuminated at different intensities to produce a single coloured "dot" on the display (see figure 1).

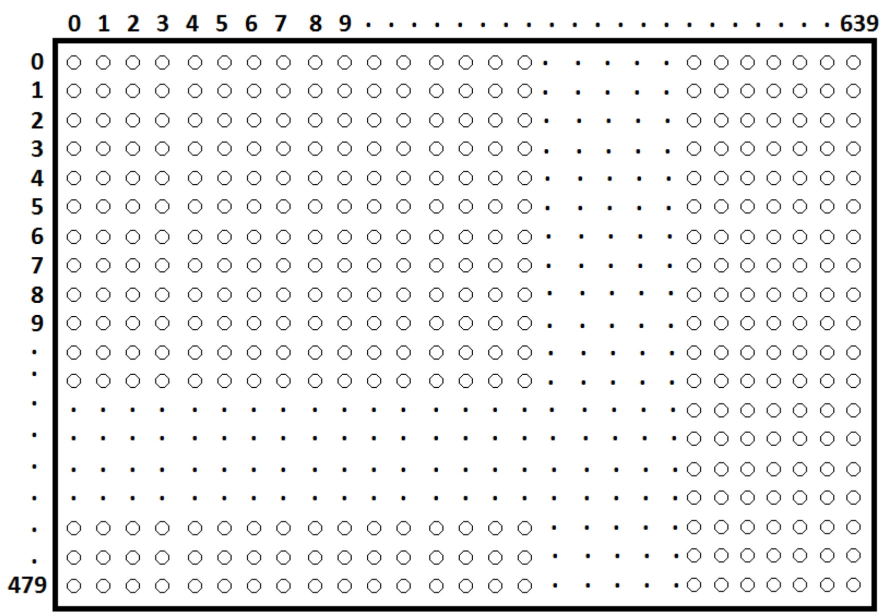


Figure 1: Pixel matrix for the active image area

A single image frame is created by traversing every pixel in the matrix and setting the appropriate RGB values; to create a moving picture without any flicker, this has to be done at 60 times per second (i.e., with a refresh rate of 60 Hz). In order for the display to scan the screen at 60 Hz and display an accurate moving picture, three signals are necessary: a pixel clock, horizontal synchronization (H-sync) signal, and vertical synchronization (V-sync) signal. After every row of pixels (a single line) is set, time must be allotted for a horizontal retrace; similarly, time must be allotted for a vertical retrace upon completion of all columns (a single frame). For both the H-sync and V-sync signals, the retrace time comes in the form of a front porch, sync pulse, and back porch. During each retrace, no RGB signals should be sent out in order to prevent an undesirable display (such as incorrect colours). After the VGA controller was implemented, logic for the game had to be created so that the various components (such as the border, paddles, and ball) were displayed in their correct positions and collision detection could be performed.

SPECIFICATIONS

The game must be run on a 640 x 480 VGA display with a refresh rate of 60 Hz and a pixel clock of 25 MHz. The overall system must conform to the input and output specifications shown in the symbol diagram of figure 2 (below). The game takes inputs from the 50 MHz system clock and four input switches found on the Spartan 3E board. The switches are used to control the up and down movement of the paddles on each side of the screen. The system must output the *DAC_CLK* signal, which is a pixel clock signal of 25 MHz. The horizontal and vertical synchronization signals are outputted as *H* and *V*, respectively. Each of the red, green, and blue colour outputs (for each pixel) are 8 bits each, and are mapped to *Rout*, *Gout*, and *Bout*, respectively.

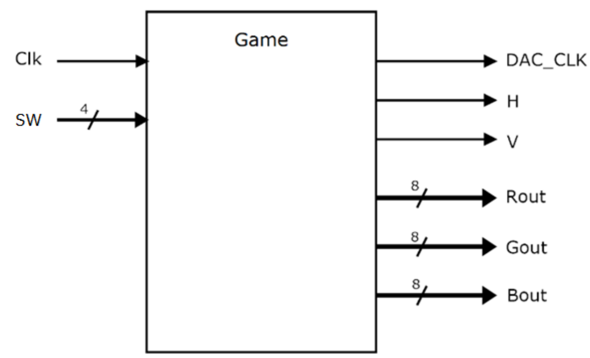


Figure 2: Symbol diagram of the game

The structure of the horizontal and vertical synchronization signals (*H* and *V*) are shown in figure 3; the signal is set to high unless it is in the sync pulse region. The corresponding durations for each of the parameters is seen is tables 1 and 2; in terms of the pixel clock, a complete line takes 800 cycles, while a single frame takes 525 cycles.

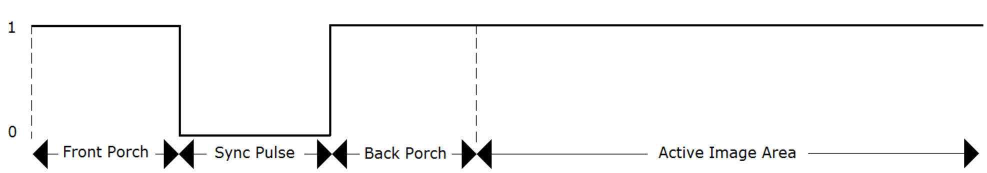


Figure 3: Structure of the H-sync and V-sync Signals

Parameter	Pixel Clock Cycles
Front Porch	16
Sync Pulse	96
Back Porch	48
Active Image area	640

Table 1: H-sync parameters

Parameter	Pixel Clock Cycles
Front Porch	10
Sync Pulse	2
Back Porch	33
Active Image area	480

Table 2: V-sync parameters

The block diagram for the entire system can be seen in figure 4 (below). Two counters are used: one for the H-sync, and another for the V-sync. The values of the counters are fed into comparators to check which regions the H-sync and V-sync signals are in. The counter on the left is for the H-sync, and is synchronized with the pixel clock (which is a division of the system clock by 2, as indicated by the *Div 2* block). Each of the comparators (from left to right) notify the controller that the H-sync has moved past the front porch, sync pulse, back porch, and active image regions. The H-sync counter resets when it reaches 800, indicating that a line has been completed; at the same time, the counter for the V-sync (right) increments. The comparators for the V-sync counter perform the same checks as the comparators for the H-sync counter (but with different values). Upon reaching a count of 525, the V-sync counter resets. The switch inputs from the board are sent to the controller block as a 4-bit signal. Inside the controller block is where the logic for paddle movement and collision detection is implemented.

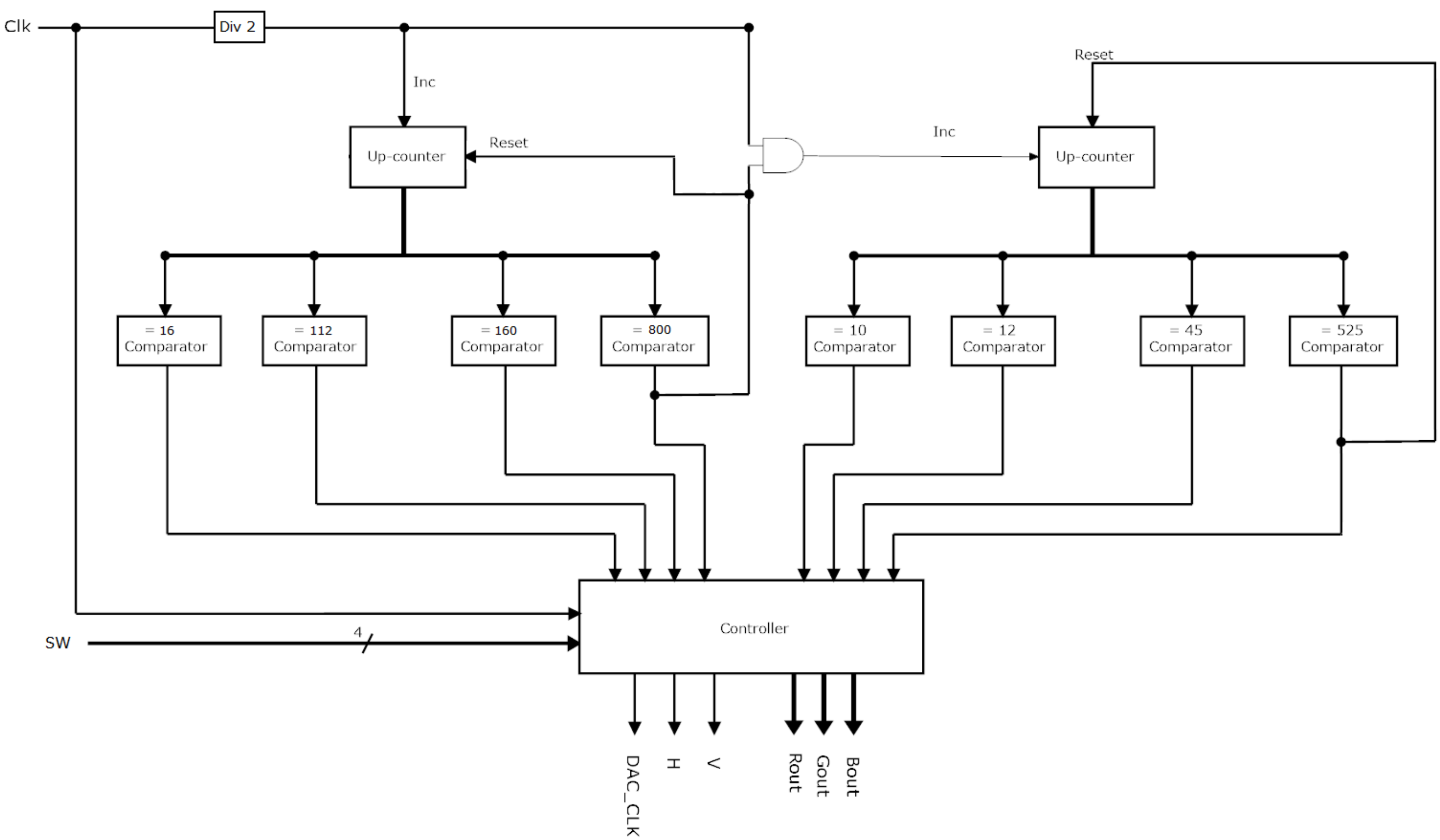


Figure 4: The block diagram for the game

VHDL IMPLEMENTATION

Using the above symbol diagram as a guide, the top-level entity is instantiated with the following VHDL code:

```
entity Game is
  Port
  (
     clk : in  STD_LOGIC;
      H : out  STD_LOGIC;
      V : out  STD_LOGIC;
      DAC_CLK : out  STD_LOGIC;
      Bout : out  STD_LOGIC_VECTOR (7 downto 0);
      Gout : out  STD_LOGIC_VECTOR (7 downto 0);
      Rout : out  STD_LOGIC_VECTOR (7 downto 0);
      Switches : in  STD_LOGIC_VECTOR (3 downto 0)
  );
end Game;
```

Since a pixel clock of 25 MHz is required, but only the 50 MHz clock is available, the *clk* signal must be divided by two and mapped to *DAC_CLK*. This is done using an internal signal called *pixel_clk* in the following process:

```
process (clk)
  begin
  if clk'event and clk='1' then
    pixel_clk <= NOT(pixel_clk);
  end if;
end process;
DAC_CLK <= pixel_clk;
```

Simply using the NOT operator on *pixel_clk* at each rising of the system clock divides it by two. The *pixel_clk* signal is then mapped to DAC_CLK to be outputted to the VGA.

The horizontal and vertical synchronization signals were set up using their respective counters, as well as the *pixel_clk* signal. The following VHDL code was used:

```
process (pixel_clk, hcounter, vcounter)

    begin
        if pixel_clk'event and pixel_clk = '1' then

            -- horizontal counter counts from 0 to 799
            hcounter <= hcounter+1;

            if (hcounter = 799) then

                vcounter <= vcounter+1;
                hcounter <= 0;

            end if;

            -- vertical counter counts from 0 to 524
            if (vcounter = 524) then

                vcounter <= 0;

            end if;
        end if;
end process;

H <= '0' when hcounter <= 96
        else '1';

V <= '0' when vcounter <= 2
        else '1';
```

As per the specifications, the H-sync counter is synchronized with the *pixel_clk* signal; it increases by 1 at each rising edge of the pixel clock. It resets after 800 pixel clock cycles, which causes the V-sync counter to increase by 1; the V-sync counter resets after 525 pixel clock cycles. The H-sync and V-sync signals themselves are made asynchronous to avoid any delays that may cause their timing to be inaccurate if they were inside a process with sequential statements. Referring to figure 3, it can be seen that the only time the H-sync and V-sync signals should be set to low is during the sync pulse. This is implemented with the code following the end of the process.

Internal signals were also created for the red, green, and blue colour outputs (named *R*, *G*, and *B*). To ensure that colour signals were only sent inside the active image region, the following process was used:

```
process
  begin

        -- Active image region values are calculated as follows:
        -- Beginning of active region = sync pulse + back porch - 1
        -- End of active region = end of line - front porch - 1

        --// For H-sync:
        -- Beginning of active region = 96 + 48 - 1 = 143
        -- End of active region = 800 - 16 - 1 = 783

        --// For V-sync:
        -- Beginning of active region = 2 + 33 - 1 = 34
        -- End of active region = 525 - 10 - 1  = 514


        if(hcounter >= 143 and hcounter <= 783 and vcounter >= 34 and vcounter <= 514) then

                Rout <= R;
                Gout <= G;
                Bout <= B;

        else
                Rout <= (others => '0');
                Gout <= (others => '0');
                Bout <= (others => '0');

        end if;
end process;
```

The *hcounter* and *vcounter* signals are integers with ranges of 0 – 799 and 0 – 524, respectively. This explains why a 1 was subtracted from the values used to check for the active image boundaries. In the subsequently created processes where the images are created, values are directly assigned to the *R*, *G*, and *B* signals. The outputs *Rout*, *Gout*, and *Bout* only receive their values while in the active region, else they are assigned values of zero. As previously mentioned, the process was created to prevent skewed and discoloured images from being displayed, which were encountered early in the development of the game.

If the position changes of the ball and paddles were synchronized with the pixel clock, the game wouldn't produce smooth animations that a human player could recognize. To ensure that smooth movement of the paddles and the ball, a new clock was created using the following process:

```
--Generate a 60 Hz refresh clock to produce smooth
--animations. The 25 MHz clock runs at 25,000,000 cycles
--per second. To get 60 Hz, the refresh clock changes
--every 25000000/60 = 416667 cycles of the 25 MHz clock

process(pixel_clk)

    begin
        if pixel_clk'event and pixel_clk='1' then
            if (refresh_cntr >= 416667) then

                refresh_clk <= not(refresh_clk);
                refresh_cntr <= 0;
            else
                refresh_cntr <= refresh_cntr + 1;

            end if;
        end if;
end process;
```

The ball movement process essentially allows the ball to continue in its direction, moving a predetermined amount of pixels every time it is updated. This is done by synchronizing the ball movement with the *refresh_clk* signal, which is a 60 Hz clock signal transformed from the *pixel_clk* in a previous process. At the rising edge of the *refresh_clk*, we move the ball by 5 pixels in the direction it is previously heading; the ball is first initialized to move towards the positive x and y directions. The variables *ball_x_inc* and *ball_y_inc* are used to indicate the ball's direction. Once the ball comes into contact with a paddle or an outlying border, the direction(s) become reversed. An example is shown in figure 5: when the ball encounters a border on the right hand side of the field, we can assume that the *ball_x_inc* is in the positive direction previous to the encounter. Once the collision has been made, and the ball's x-coordinates (*ball_x2 and ball_x1*) have passed the x co-ordinates of the wall (*top_border_x5*), we know that the ball has hit the wall, and not the paddle. Since the values of *ball_y1* and *ball_y2* are in between *top_border_y4* and *top_border_y6*, we know that only the ball's path in the x direction must be reversed, thereby setting *ball_x_inc* to 0; this causes the ball to move in the negative x-direction. This can similarly be said for the left hand wall, as well as the top and bottom borders. In the case of the ball colliding with the top and bottom borders, the y-direction of the ball is reversed. However, to determine if the ball collides with the paddle, we must check if the ball's current position (*ball_x1, ball_x2, ball_y1, ball_y*) intersect with the paddles' x and y values(for the case of the blue paddle, *b_paddle_x1,b_paddle_x2,b_paddle_y1,b_paddle_y2*). If a collision occurs, then the x-direction of the ball is once again reversed. Similarly, for hitting the goal zone of the paddles, we must also verify that the x position (*ball_x2*) of the ball has managed to get past the goal line. Assuming that the walls and paddles are working correctly (i.e., they will repel the ball), if the ball is to pass a certain location of the map (for the case of the blue paddle, any value less than *b_border_x2*), then the player has scored. Once a someone has scored, the ball is reset, and is set to head towards the player who has scored.

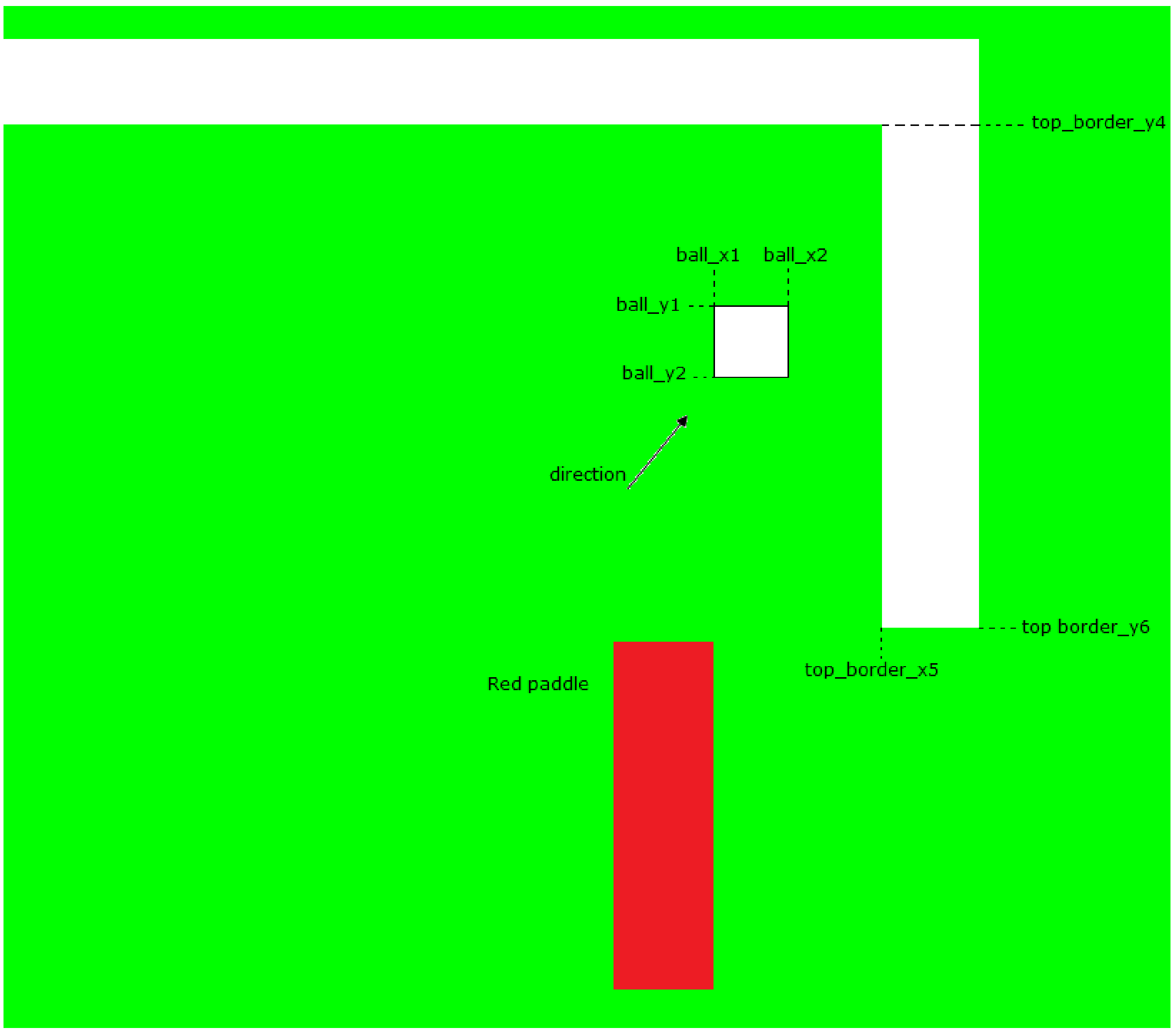


Figure 5: Cross-section of the game, showing the ball before it hits the top right section of the border

The code used to implement the ball movement and collision detection process is as follows:

```
process(refresh_clk)
begin
            if refresh_clk'event and refresh_clk = '1' then

            --//Wall collision detection

                 --Ball has hit object on the left, send ball to positive x direction

                if (ball_x1 <= top_border_x2 and (ball_y1 >= top_border_y4 and ball_y2<= top_border_y6)) then

                        --Ball hits top-left boundary
                        ball_x_inc <= '1';

               elsif(ball_x1 <= b_border_x2 and (ball_y1 >= b_border_y5 and ball_y2<= b_border_y6)) then

                        --Ball hits bottom-left boundary
                        ball_x_inc <= '1';

                        --Ball has hit object on the right, send ball to negative x direction

              elsif (ball_x2 >= top_border_x5 and (ball_y2 >= top_border_y4 and ball_y1<= top_border_y6)) then

                        --Ball hits top-right boundary
                        ball_x_inc <= '0';

              elsif (ball_x2 >= b_border_x5 and (ball_y2 >= b_border_y1 and ball_y1<= b_border_y6)) then

                        --Ball hits bottom-right boundary
                        ball_x_inc <= '0';

                                    end if;

                                    --Ball and paddle collision detection

            if (ball_x1 <= b_paddle_x2 and (ball_y1 >= b_paddle_y1 and ball_y2 <= b_paddle_y2)) then

                        --Ball has collided with blue paddle (left); send ball in the positive x direction
                        ball_x_inc <= '1';

            elsif (ball_x2 >= r_paddle_x1 and (ball_y1 >= r_paddle_y1 and ball_y2 <= r_paddle_y2)) then

                        --Ball has collided with red paddle (right); send ball in the positive x direction
                        ball_x_inc <= '0';

            end if;
```

```
if (ball_y1 <= top_border_y4) then

        --Ball has hit top boundary; send ball in negative y direction
        ball_y_inc <= '0';

elsif (ball_y2 >= b_border_y3) then

        --Ball has hit bottom boundary; send ball in positive y direction
        ball_y_inc <= '1';

end if;


--//Goal collision detection

if (ball_x1 <= b_goal_x) then

--Ball reaches left goal line; red paddle scores
score <= '1';

elsif (ball_x2 >= r_goal_x) then

--Ball reaches right goal line; blue paddle scores
score <= '1';

else

--Otherwise, if none of the conditions have been met, there is no scoring
score <= '0';
end if;

if (ball_x1 <= 10) then

--Ball has reached end of screen (left); reset ball location
ball_x1 <= 310;
ball_x2 <= 330;
ball_y1 <= 230;
ball_y2 <= 250;
ball_x_inc <= '1';

ball_y_inc <= '1';

elsif (ball_x2 >= 630) then

--Ball has reached end of screen (right); reset ball location
ball_x1 <= 310;
```

```
        ball_x2 <= 330;
        ball_y1 <= 230;
        ball_y2 <= 250;
        ball_x_inc <= '0';
        ball_y_inc <= '1';

        else
            --Ball movement
            if (ball_x_inc = '1') then
                --Move ball in positive x direction
                ball_x1 <= ball_x1 + 3;
                ball_x2 <= ball_x2 + 3;

            else
            --Move ball in negative x direction
            ball_x1 <= ball_x1 - 3;
            ball_x2 <= ball_x2 - 3;

            end if;

            if ( ball_y_inc = '1') then
                --Move ball in positive y direction
                ball_y1 <= ball_y1 - 3;
                ball_y2 <= ball_y2 - 3;

            else
                --Move ball in negative y direction
                ball_y1 <= ball_y1 + 3;
                ball_y2 <= ball_y2 + 3;
            end if;
        end if;
    end if;

end process;
```

The switch inputs from the Spartan 3E board are used to control the paddle movement. A single process was made to control paddle movement and also detect (and avoid) collisions. The process is outlined in the following code:

```
process(refresh_clk)

	begin

		--Once a switch input has been set for up or down movement,
		--it cannot be interrupted by another switch for the opposite
		--movement direction. For example, if the red paddle is set
		--to move upward, it will keep moving upward, even if the
		--down movement switch is pressed. Once the switch is set to
		--low, the paddle can move in the opposite direction.
		--NOTE: When encountering a boundary, the boundary will act like a
		--"force-field" and push the paddle back.

		if refresh_clk'event and refresh_clk = '1' then

			if (SW(1) = '1') then

				--Move red paddle up

				if (r_paddle_y1 < top_border_y4) then

					r_paddle_y1 <= r_paddle_y1 + 1;
					r_paddle_y2 <= r_paddle_y2 + 1;

				else
					r_paddle_y1 <= r_paddle_y1 - 5;
					r_paddle_y2 <= r_paddle_y2 - 5;
				end if;

			elsif(SW(0) = '1') then

				--Move red paddle down
				if (r_paddle_y2 > b_border_y3) then

					r_paddle_y1 <= r_paddle_y1 - 1;
					r_paddle_y2 <= r_paddle_y2 - 1;
				else
					r_paddle_y1 <= r_paddle_y1 + 5;
					r_paddle_y2 <= r_paddle_y2 + 5;

				end if;
```

```
                elsif (SW(3) = '1') then

                        --Move blue paddle up
                        if (b_paddle_y1 < top_border_y4) then

                                b_paddle_y1 <= b_paddle_y1 + 1;
                                b_paddle_y2 <= b_paddle_y2 + 1;
                        else
                                b_paddle_y1 <= b_paddle_y1 - 5;
                                b_paddle_y2 <= b_paddle_y2 - 5;
                        end if;

                elsif(SW(2) = '1') then

                        --Move blue paddle down
                        if (b_paddle_y2 > b_border_y3) then

                                b_paddle_y1 <= b_paddle_y1 - 1;
                                b_paddle_y2 <= b_paddle_y2 - 1;
                        else
                                b_paddle_y1 <= b_paddle_y1 + 5;
                                b_paddle_y2 <= b_paddle_y2 + 5;

                        end if;
                end if;
        end if;
end process;
```

The process is based off of the refresh clock to ensure that the animation of the paddles is smooth. Switches 3 and 2 on the board are used for moving the blue paddle (left side) up and down, respectively. Similarly, Switches 1 and 0 are used to control the movement of the red paddle (right side). When either paddle touches the upper or lower boundaries of the wall, their vertical positions are changed such that they are moved away from the wall; visually, this creates a force-field effect.

Finally, the process to display the images on the monitor was created. In order to map out the locations of all the objects on screen, *x* and *y* variables were created and assigned values such that they map to an intuitive locations. This makes it so that the top-left location on the physical monitor corresponds to an (*x,y*) value of (0,0) instead of (143,34). The latter coordinates would be if they were based off of the H-sync and V-sync counters (see earlier implementation of the process which multiplexes the RGB colour output for calculation details). The VHDL code for the process is as follows:

```
process(hcounter, vcounter)

        variable x: integer range 0 to 639;
        variable y: integer range 0 to 479;

begin

--To isolate the active region, we subtract the number of cycles it takes
--for H-sync and V-sync to reach their respective active regions and place
--the values into x and y coordinates.This helps to intuitively determine
--the placement of objects on the physical screen

        x := hcounter - 143;
        y := vcounter - 34;

--Every pixel that isn't part of an object on the screen is set to display green
        R <= "00000000";
        G <= "11111111";
        B <= "00000000";

--//Displaying the ball

 if (x > ball_x1 and x < ball_x2 and y > ball_y1 and y < ball_y2) then

                --Changing the ball colour to red when either side has scored
                if (score = '1') then

                        R <= "11111111";
                        G <= "00000000";
                        B <= "00000000";
                else
                        R <= "11111111";
                        G <= "11111111";
                        B <= "11111111";
                end if;

--//Displaying the boundaries of the field
  -- Top Boundary
  elsif (x > top_border_x1 and x < top_border_x2  and y > top_border_y1 and y < top_border_y2) then

                R <= "11111111";
                G <= "11111111";
                B <= "11111111";
```

```
elsif (x > top_border_x3 and x < top_border_x4  and y > top_border_y3 and y < top_border_y4) then
			R <= "11111111";
			G <= "11111111";
			B <= "11111111";
elsif (x > top_border_x5 and x < top_border_x6  and y > top_border_y5 and y < top_border_y6) then
			R <= "11111111";
			G <= "11111111";
			B <= "11111111";


	--Bottom Boundaries
elsif (x > b_border_x1 and x < b_border_x2  and y > b_border_y1 and y < b_border_y2) then
			R <= "11111111";
			G <= "11111111";
			B <= "11111111";
elsif (x > b_border_x3 and x < b_border_x4  and y > b_border_y3 and y < b_border_y4) then
			R <= "11111111";
			G <= "11111111";
			B <= "11111111";
elsif (x > b_border_x5 and x < b_border_x6  and y > b_border_y5 and y < b_border_y6) then
			R <= "11111111";
			G <= "11111111";
			B <= "11111111";


--//Displaying the line in the middle of the field
elsif (x > m_line_x1 and x < m_line_x2 and y > m_line_y1 and y < m_line_y2) then
			R <= "00000000";
			G <= "00000000";
			B <= "00000000";
```

```
--//Displaying the paddles

        --Red paddle
  elsif (x > r_paddle_x1 and x < r_paddle_x2 and y > r_paddle_y1 and y < r_paddle_y2) then

                R <= "11111111";
                G <= "00000000";
                B <= "00000000";

        --Blue paddle
  elsif (x > b_paddle_x1 and x <b_paddle_x2 and y > b_paddle_y1 and y < b_paddle_y2) then

                R <= "00000000";
                G <= "00000000";
                B <= "11111111";
        end if;

--//Changing the ball colour to red when either side has scored

        if (score = '1') then

                R <= "11111111";
                G <= "00000000";
                B <= "00000000";
        end if;
end process;
```

Using the knowledge that the active region for the H-sync and V-sync signals begin at 143 and 34, respectively, the values assigned to *x* and *y* are set accordingly. Next, creation of the images is done by setting pixel colours based on the coordinates assigned to each object. When the *x* and *y* values are within the regions bounded by said coordinates, the appropriate colour values are outputted. For example, the ball is displayed by using the earlier assigned coordinates (see Appendix for full code) *ball_x1, ball_x2, ball_y1,* and *ball_y2*; these can be thought of as the end points for a horizontal and vertical line which define a rectangular shape (see ball coordinates of figure 3 for better insight). By filling in the areas inside of these values white, the ball can be displayed on-screen. To display a white ball, once we have ensured that *x* and *y* are contained in these regions, *R, G,* and *B* values of 255 are set. Note that this logic is applied for all other objects on the display. In addition to this, we can also change the colour of the ball upon scoring. This is done by using the score flag (found inside the ball movement and collision detection process): if the flag is set, the ball is displayed red instead of white.

RESULTS

The first functionality check of the game was to see if it displayed on the screen with the correct alignment and colours. As shown in figure 6, it is clearly seen that the game is displayed correctly, on a green field with white borders, a black mid-line, a white ball, a blue paddle, and a red paddle.

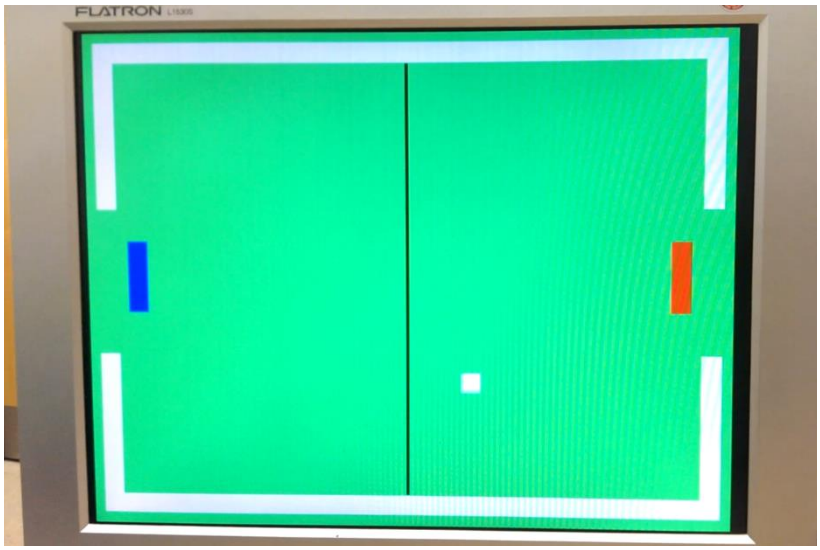


Figure 6: Shows the game displayed with the correct positioning and colour of all objects

The second check involved ball movement and collision detection. Figure 7 shows the ball coming in from the left side and headed south-east, just before it collides with the bottom border. Figure 8 shows the ball after the collision; it heads north-east, as expected.

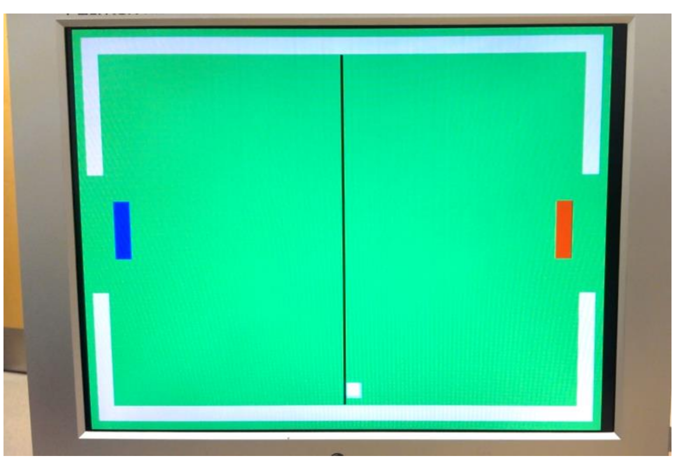


Figure 7: Shows the ball before the collision

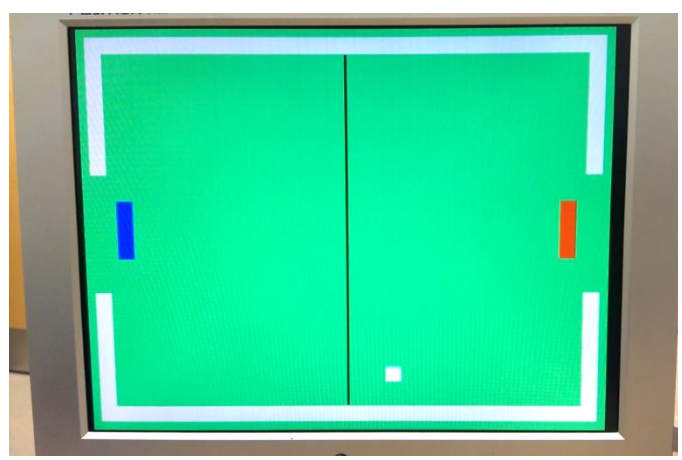


Figure 8: Shows the ball after the collision

The ball was also expected to bounce off of the paddles. Figure 9 shows the ball headed in the south-west direction, just before it collides with the blue paddle. Figure 10 shows the ball after the collision; it heads south-east, as expected. Note that both figures also showcase the paddle movement functionality: both paddles are moved out of their default positions (shown in figure 6) using switches 3 and 1 to move the blue and red paddles, respectively. Figure 11 shows the collision detection functionality for the paddles: here, the blue paddle has been moved up by leaving switch 3 set to high. As expected, the paddle is stopped from moving further.

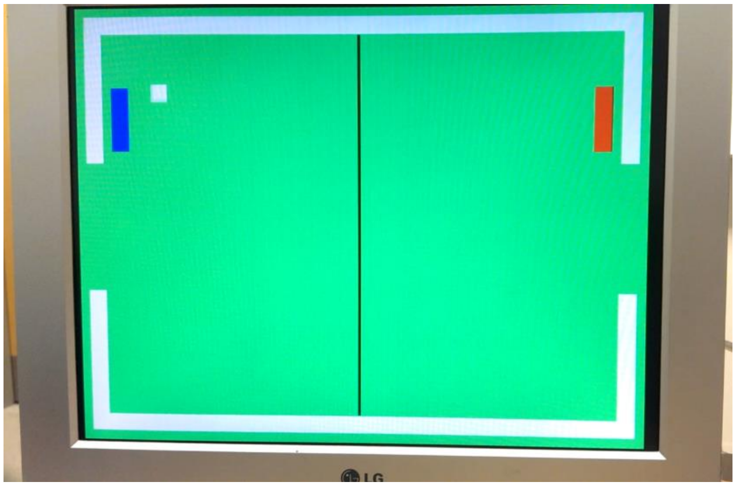


Figure 9: The ball before hitting the paddle

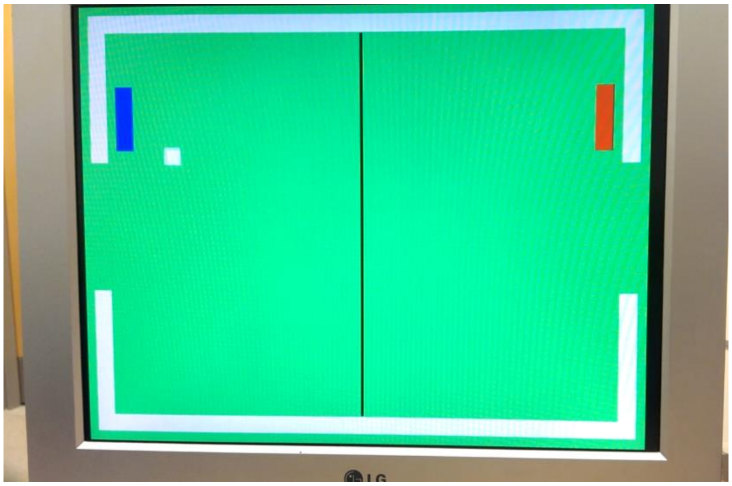


Figure 10: The ball after hitting the paddle

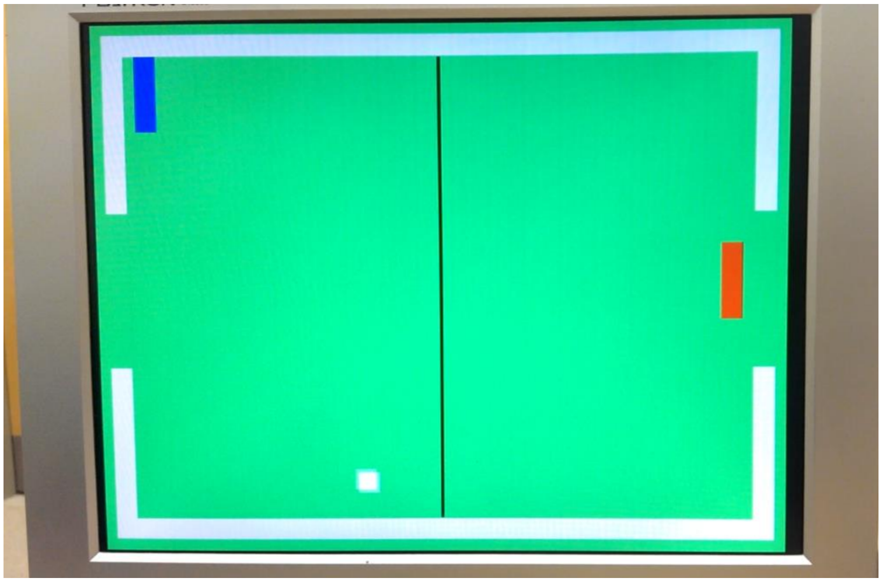


Figure 11: Showing the paddle collision avoidance

The final function of the game to be checked was the scoring mechanism: upon reaching the goal line, the ball is expected to change its colour to red. After it reaches the end of the screen (on either side), it is supposed to reset to the middle and head toward the paddle that scored. In figure 12, the blue paddle has scored a point; the ball changes to red upon reaching the red paddle's goal zone. Figure 13 shows that since the blue paddle has scored, the ball is sent towards it after it reaches the end of the screen (right side) and resets.

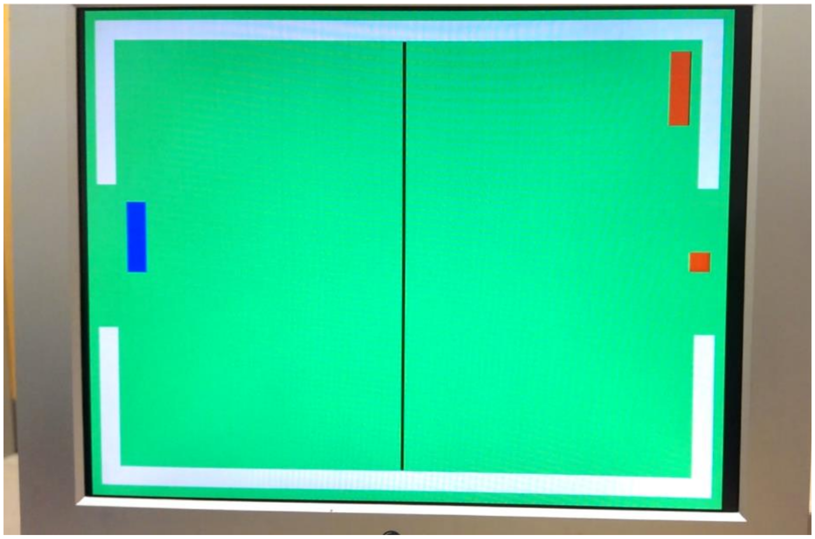


Figure 12: Shows the scoring mechanism used. The ball turns red once it reaches the goal line

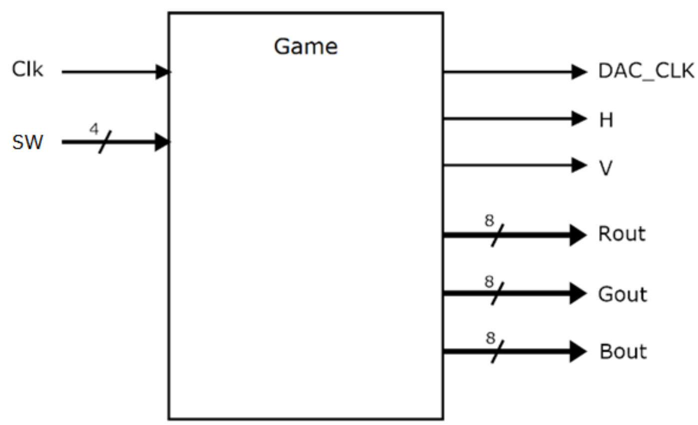
Figure 13: Shows the ball headed towards the scorer's side after resetting

CONCLUSION

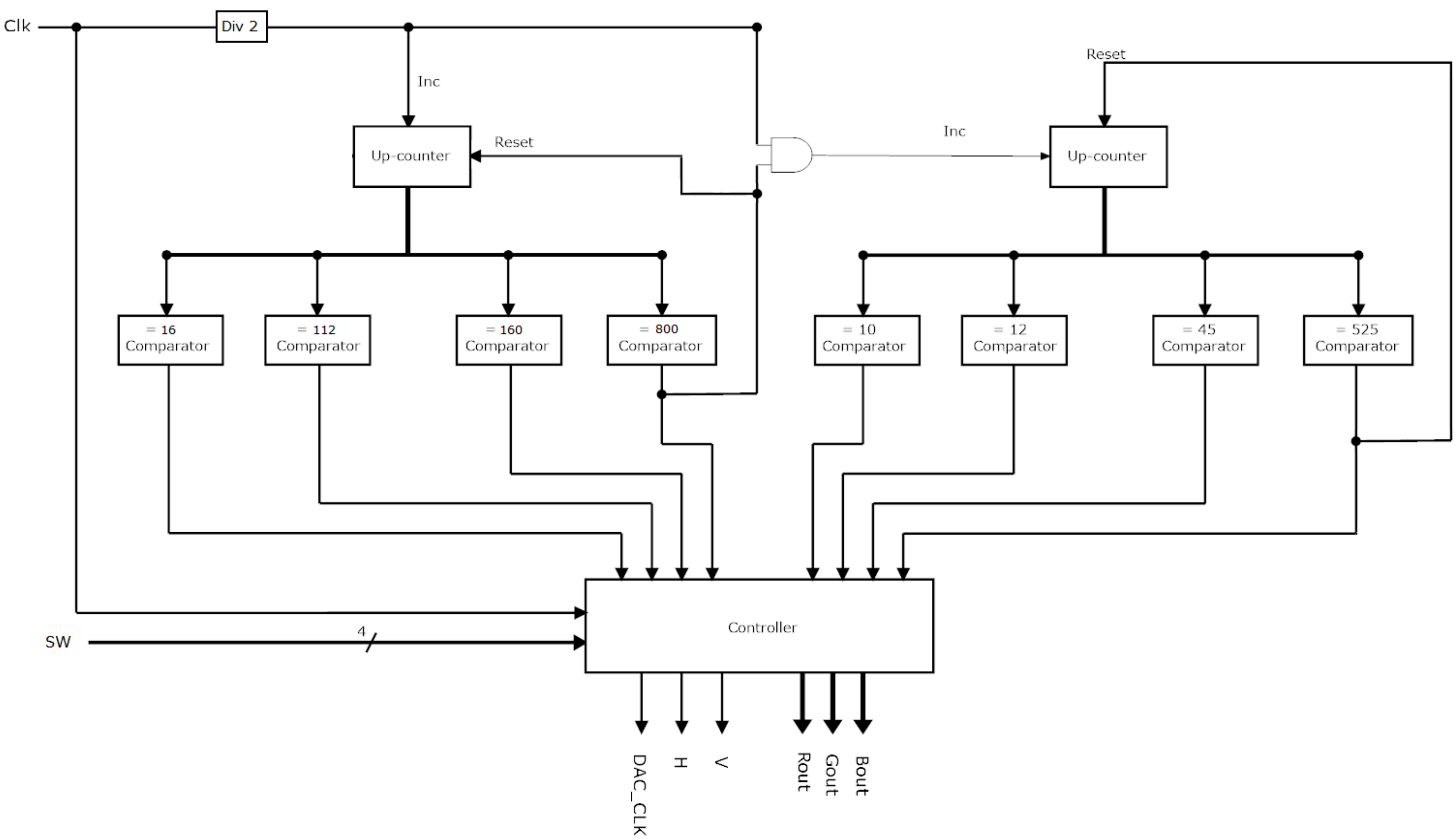
The purpose of the project was to recreate a simple *Pong*-like game that was capable of being displayed onto a monitor using the Video Graphic Adaptor (VGA). Through the use of horizontal and vertical synchronization signals, a pixel clock, and colour signals, the correct timing and image outputs were achieved. In addition, logic was implemented to create animated objects and provide mechanisms for paddle and ball movement, collision detection, collision avoidance, and scoring. The game was successfully created: all of the objects were displayed on the screen with correct colouring and positioning, animations were smooth, and all of the mechanisms implemented worked as expected. The aesthetics of the game are quite simple and could have been improved upon, but time constraints did not allow for time to be allotted for such improvements. The game is functionally sound, but improvements on the collision detection and avoidance mechanisms could have also been made. Overall, the project served as a useful way to gain foundational knowledge of real-time video subsystems that will prove to be useful in future engineering work that may be encountered.

APPENDIX A – SYMBOL AND BLOCK DIAGRAM

Symbol



Block Diagram



APPENDIX B – VHDL CODE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Game is

   Port ( clk : in  STD_LOGIC;
        H : out  STD_LOGIC;
        V : out  STD_LOGIC;
        DAC_CLK : out  STD_LOGIC;
        Bout : out  STD_LOGIC_VECTOR (7 downto 0);
        Gout : out  STD_LOGIC_VECTOR (7 downto 0);
        Rout : out  STD_LOGIC_VECTOR (7 downto 0);
        SW : in  STD_LOGIC_VECTOR (3 downto 0));

end Game;

architecture Behavioral of Game is

--H-sync counter
signal hcounter : integer range 0 to 799;

--V-sync counter
signal vcounter : integer range 0 to 524;

--Pixel clock
signal pixel_clk : std_logic;

--Refresh rate clock & counter
signal refresh_clk : std_logic;
signal refresh_cntr: integer := 0;

--Colour Vectors

signal R                    : std_logic_vector(7 downto 0);
signal G                    : std_logic_vector(7 downto 0);
signal B                    : std_logic_vector(7 downto 0);
```

```
--Variables declaring the boundaries of the field.
--The field can be thought of as a concatenation
--of six rectangles

--Vertical bar 1

signal top_border_x1: integer := 10;
signal top_border_x2: integer := 30;
signal top_border_y1: integer := 10;
signal top_border_y2: integer := 170;

--Horizontal bar 1

signal top_border_x3: integer := 10;
signal top_border_x4: integer := 630;
signal top_border_y3: integer := 10;
signal top_border_y4: integer := 30;

--Vertical bar 2

signal top_border_x5: integer := 610;
signal top_border_x6: integer := 630;
signal top_border_y5: integer := 10;
signal top_border_y6: integer := 170;

--Vertical bar 3

signal b_border_x1  : integer := 10;
signal b_border_x2  : integer := 30;
signal b_border_y1  : integer := 310;
signal b_border_y2  : integer := 470;

--Horizontal bar 2

signal b_border_x3  : integer := 10;
signal b_border_x4  : integer := 630;
signal b_border_y3  : integer := 450;
signal b_border_y4  : integer := 470;

--Vertical bar 4

signal b_border_x5  : integer := 610;
signal b_border_x6  : integer := 630;
```

```
signal b_border_y5  : integer := 310;
signal b_border_y6  : integer := 470;

--Mid-field line

signal m_line_x1   : integer := 318;
signal m_line_x2   : integer := 322;
signal m_line_y1   : integer := 30;
signal m_line_y2   : integer := 450;

--Paddle dimensions for the red paddle

signal r_paddle_x1          : integer := 580;
signal r_paddle_x2          : integer := 600;
signal r_paddle_y1          : integer := 200;
signal r_paddle_y2          : integer := 270;
signal r_paddle_x_inc       : integer := 0;
signal r_paddle_y_inc       : integer := 0;

--Paddle dimensions for the blue paddle

signal b_paddle_x1          : integer := 40;
signal b_paddle_x2          : integer := 60;
signal b_paddle_y1          : integer := 200;
signal b_paddle_y2          : integer := 270;
signal b_paddle_x_inc       : integer := 0;
signal b_paddle_y_inc       : integer := 0;


--Dimensions for the ball

signal ball_x1              : integer := 310;
signal ball_x2              : integer := 330;
signal ball_y1              : integer := 230;
signal ball_y2              : integer := 250;

--Goal lines for the red and blue sides

signal r_goal_x             : integer := 620;
signal b_goal_x             : integer := 20;
```

```
--Flags for score detection and reset

signal score                    : std_logic;
signal reset                    : std_logic;


--The following signals determine the direction of the ball by
--Increasing or decreasing its x and y positions

signal ball_x_inc        : std_logic;
signal ball_y_inc        : std_logic;

begin

--Generate a 25Mhz clock using the 50 MHz system clock
process (clk)
   begin

   if clk'event and clk='1' then

      pixel_clk <= NOT(pixel_clk);

   end if;
end process;

--Mapping the pixel clock to the DAC_CLK output

DAC_CLK <= pixel_clk;


--H-sync and V-sync counter setup

process (pixel_clk, hcounter, vcounter)

        begin
                if pixel_clk'event and pixel_clk = '1' then

                        -- horizontal counter counts from 0 to 799
                        hcounter <= hcounter+1;

                        if (hcounter = 799) then

                                vcounter <= vcounter+1;
                                hcounter <= 0;

                        end if;
```

```
                    -- vertical counter counts from 0 to 524
                    if (vcounter = 524) then

                        vcounter <= 0;

                    end if;
                end if;
end process;

--H-sync and V-sync are low during the sync pulse
--Otherwise, they are set to high

H <= '0' when hcounter <= 96
        else '1';

V <= '0' when vcounter <= 2
        else '1';

-- Only output colours when in the active region
process
  begin

    -- Active image region values are calculated as follows:
    -- Beginning of active region = sync pulse + back porch - 1
    -- End of active region = end of line - front porch - 1

    --// For H-sync:
    -- Beginning of active region = 96 + 48 - 1 = 143
    -- End of active region = 800 - 16 - 1 = 783

    --// For V-sync:
    -- Beginning of active region = 2 + 33 - 1 = 34
    -- End of active region = 525 - 10 - 1  = 514
    if(hcounter >= 143 and hcounter <= 783 and vcounter >= 34 and vcounter <= 514) then
                    Rout <= R;
                    Gout <= G;
                    Bout <= B;
                else
                    Rout <= (others => '0');
                    Gout <= (others => '0');
                    Bout <= (others => '0');
    end if;

end process;
```

```
--Generate a 60 Hz refresh clock to produce smooth
--animations. The 25 MHz clock runs at 25,000,000 cycles
--per second. To get 60 Hz, the refresh clock changes
--every 25000000/60 = 416667 cycles of the 25 MHz clock

process(pixel_clk)

    begin
        if pixel_clk'event and pixel_clk='1' then
            if (refresh_cntr >= 416667) then

                refresh_clk <= not(refresh_clk);
                refresh_cntr <= 0;
            else
                refresh_cntr <= refresh_cntr + 1;

            end if;
        end if;
end process;


--Ball movement and collision detection
process(refresh_clk)
begin
    if refresh_clk'event and refresh_clk = '1' then

    --//Wall collision detection

       --Ball has hit object on the left, send ball to positive x direction

      if (ball_x1 <= top_border_x2 and (ball_y1 >= top_border_y4 and ball_y2<= top_border_y6)) then

        --Ball hits top-left boundary
        ball_x_inc <= '1';

     elsif(ball_x1 <= b_border_x2 and (ball_y1 >= b_border_y5 and ball_y2<= b_border_y6)) then

        --Ball hits bottom-left boundary
        ball_x_inc <= '1';

        --Ball has hit object on the right, send ball to negative x direction

     elsif (ball_x2 >= top_border_x5 and (ball_y2 >= top_border_y4 and ball_y1<= top_border_y6)) then

        --Ball hits top-right boundary
        ball_x_inc <= '0';
```

```
elsif (ball_x2 >= b_border_x5 and (ball_y2 >= b_border_y1 and ball_y1<= b_border_y6)) then

    --Ball hits bottom-right boundary
    ball_x_inc <= '0';

        end if;

        --Ball and paddle collision detection

if (ball_x1 <= b_paddle_x2 and (ball_y1 >= b_paddle_y1 and ball_y2 <= b_paddle_y2)) then

    --Ball has collided with blue paddle (left); send ball in the positive x direction
    ball_x_inc <= '1';

elsif (ball_x2 >= r_paddle_x1 and (ball_y1 >= r_paddle_y1 and ball_y2 <= r_paddle_y2)) then

    --Ball has collided with red paddle (right); send ball in the positive x direction
    ball_x_inc <= '0';

end if;


if (ball_y1 <= top_border_y4) then

    --Ball has hit top boundary; send ball in negative y direction
    ball_y_inc <= '0';

elsif (ball_y2 >= b_border_y3) then

    --Ball has hit bottom boundary; send ball in positive y direction
    ball_y_inc <= '1';

end if;


--//Goal collision detection

if (ball_x1 <= b_goal_x) then

--Ball reaches left goal line; red paddle scores
score <= '1';

elsif (ball_x2 >= r_goal_x) then

--Ball reaches right goal line; blue paddle scores
score <= '1';
```

```
else

         --Otherwise, if none of the conditions have been met, there is no scoring
          score <= '0';
end if;

if (ball_x1 <= 10) then

--Ball has reached end of screen (left); reset ball location
ball_x1 <= 310;
ball_x2 <= 330;
ball_y1 <= 230;
ball_y2 <= 250;
ball_x_inc <= '1';

ball_y_inc <= '1';

elsif (ball_x2 >= 630) then

--Ball has reached end of screen (right); reset ball location
ball_x1 <= 310;

              ball_x2 <= 330;
              ball_y1 <= 230;
              ball_y2 <= 250;
              ball_x_inc <= '0';
              ball_y_inc <= '1';


              else
                            --Ball movement
                            if (ball_x_inc = '1') then
                                          --Move ball in positive x direction
                                          ball_x1 <= ball_x1 + 3;
                                          ball_x2 <= ball_x2 + 3;

                            else
                            --Move ball in negative x direction
                            ball_x1 <= ball_x1 - 3;
                            ball_x2 <= ball_x2 - 3;

                            end if;

                            if ( ball_y_inc = '1') then
                                          --Move ball in positive y direction
```

```
                ball_y1 <= ball_y1 - 3;
                ball_y2 <= ball_y2 - 3;

            else
                --Move ball in negative y direction
                ball_y1 <= ball_y1 + 3;
                ball_y2 <= ball_y2 + 3;
            end if;
        end if;
    end if;

end process;


--Paddle movement and collision detection

process(refresh_clk)

    begin

        --Once a switch input has been set for up or down movement,
        --it cannot be interrupted by another switch for the opposite
        --movement direction. For example, if the red paddle is set
        --to move upward, it will keep moving upward, even if the
        --down movement switch is pressed. Once the switch is set to
        --low, the paddle can move in the opposite direction.
        --NOTE: When encountering a boundary, the boundary will act like a
        --"force-field" and push the paddle back.

        if refresh_clk'event and refresh_clk = '1' then

            if (SW(1) = '1') then

                --Move red paddle up

                if (r_paddle_y1 < top_border_y4) then

                    r_paddle_y1 <= r_paddle_y1 + 1;
                    r_paddle_y2 <= r_paddle_y2 + 1;

                else
                    r_paddle_y1 <= r_paddle_y1 - 5;
                    r_paddle_y2 <= r_paddle_y2 - 5;
                end if;
```

```
elsif(SW(0) = '1') then

        --Move red paddle down
        if (r_paddle_y2 > b_border_y3) then

                r_paddle_y1 <= r_paddle_y1 - 1;
                r_paddle_y2 <= r_paddle_y2 - 1;
        else
                r_paddle_y1 <= r_paddle_y1 + 5;
                r_paddle_y2 <= r_paddle_y2 + 5;

        end if;



elsif (SW(3) = '1') then

        --Move blue paddle up
        if (b_paddle_y1 < top_border_y4) then

                b_paddle_y1 <= b_paddle_y1 + 1;
                b_paddle_y2 <= b_paddle_y2 + 1;
        else
                b_paddle_y1 <= b_paddle_y1 - 5;
                b_paddle_y2 <= b_paddle_y2 - 5;
        end if;

elsif(SW(2) = '1') then

        --Move blue paddle down
        if (b_paddle_y2 > b_border_y3) then

                b_paddle_y1 <= b_paddle_y1 - 1;
                b_paddle_y2 <= b_paddle_y2 - 1;
        else
                b_paddle_y1 <= b_paddle_y1 + 5;
                b_paddle_y2 <= b_paddle_y2 + 5;

        end if;
end if;
    end if;
end process;
```

```
--Image display

process(hcounter, vcounter)

        variable x: integer range 0 to 639;
        variable y: integer range 0 to 479;

begin

--To isolate the active region, we subtract the number of cycles it takes
--for H-sync and V-sync to reach their respective active regions and place
--the values into x and y coordinates.This helps to intuitively determine
--the placement of objects on the physical screen

        x := hcounter - 143;
        y := vcounter - 34;

--Every pixel that isn't part of an object on the screen is set to display green
        R <= "00000000";
        G <= "11111111";
        B <= "00000000";

--//Displaying the ball

  if (x > ball_x1 and x < ball_x2 and y > ball_y1 and y < ball_y2) then

                --Changing the ball colour to red when either side has scored
                if (score = '1') then

                        R <= "11111111";
                        G <= "00000000";
                        B <= "00000000";
                else
                        R <= "11111111";
                        G <= "11111111";
                        B <= "11111111";
                end if;
```

```
--//Displaying the boundaries of the field

   -- Top Boundary
   elsif (x > top_border_x1 and x < top_border_x2  and y > top_border_y1 and y < top_border_y2) then

                  R <= "11111111";
                  G <= "11111111";
                  B <= "11111111";
   elsif (x > top_border_x3 and x < top_border_x4  and y > top_border_y3 and y < top_border_y4) then

                  R <= "11111111";
                  G <= "11111111";
                  B <= "11111111";

   elsif (x > top_border_x5 and x < top_border_x6  and y > top_border_y5 and y < top_border_y6) then

                  R <= "11111111";
                  G <= "11111111";
                  B <= "11111111";


         --Bottom Boundaries
   elsif (x > b_border_x1 and x < b_border_x2  and y > b_border_y1 and y < b_border_y2) then

                  R <= "11111111";
                  G <= "11111111";
                  B <= "11111111";

   elsif (x > b_border_x3 and x < b_border_x4  and y > b_border_y3 and y < b_border_y4) then

                  R <= "11111111";
                  G <= "11111111";
                  B <= "11111111";
   elsif (x > b_border_x5 and x < b_border_x6  and y > b_border_y5 and y < b_border_y6) then

                  R <= "11111111";
                  G <= "11111111";
                  B <= "11111111";
```

```
--//Displaying the line in the middle of the field
   elsif (x > m_line_x1 and x < m_line_x2 and y > m_line_y1 and y < m_line_y2) then
                    R <= "00000000";
                    G <= "00000000";
                    B <= "00000000";
--//Displaying the paddles
            --Red paddle
   elsif (x > r_paddle_x1 and x < r_paddle_x2 and y > r_paddle_y1 and y < r_paddle_y2) then
                    R <= "11111111";
                    G <= "00000000";
                    B <= "00000000";


            --Blue paddle
   elsif (x > b_paddle_x1 and x <b_paddle_x2 and y > b_paddle_y1 and y < b_paddle_y2) then
                    R <= "00000000";
                    G <= "00000000";
                    B <= "11111111";
            end if;
--//Changing the ball colour to red when either side has scored
            if (score = '1') then
                    R <= "11111111";
                    G <= "00000000";
                    B <= "00000000";
            end if;
end process;
end Behavioral;
```