Ryerson University

Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title:	Introduction To Computer Vision
Course Number:	COE 758
Semester/Year (e.g.F2016)	F2025

Instructor:	Dr. Lev Kirischian

Assignment/Lab Number:	Project #2
Assignment/Lab Title:	Simple Video Game Processor for VGA

Submission Date:	Saturday, November 15th, 2025
Due Date:	Monday, November 17th, 2025

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Sivapragasam	Sanjay	501045929	03	
Vo	Raymond	501129161	03	

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: http://www.ryerson.ca/senate/current/pol60.pdf

Abstract

The primary objective of this project is to design and implement a Simple Video-Game Processor for a VGA interface. This includes understanding the functionality of the Video Graphic Adaptor (VGA) standard, as well as gaining practical experience with on-chip (FPGA) to I/O device interfacing. The project provided hands-on experience in designing and implementing real-time, high-performance signal generators using custom logic circuits on a Xilinx Spartan-3E FPGA, as well as interfacing these on-chip circuits with an external VGA monitor. This project was completed using VHDL programming and the hardware evaluation program Xilinx ISE.

The high-level approach to this project involved designing a real-time video game processor capable of generating a 640x480 pixel display at a 60 Hz refresh rate using a 25 MHz pixel clock. The system consists of a VGA controller module that manages horizontal and vertical sync signals, a collision detection system for in-game physics, and manual switch-controlled player movement. The game features a green playing field with white boundaries and a goal opening, two colored paddles (blue and purple) controlled by FPGA board switches, and a yellow ball that changes trajectory upon collision and turns red when passing through goals.

The final result of this design is a fully functional video game processor for VGA that displays a playable two-player pong game. The implementation successfully demonstrates real-time frame generation at 60 Hz, proper synchronization with HSYNC and VSYNC signals, accurate collision detection between objects (the ball and paddles, and the ball and boundaries), and responsive player control through physical switches on the FPGA board. All game logic operates correctly, including ball trajectory changes at 90-degree angles upon collision and proper colour transitions based on game events.

Table of Contents

Introduction

The purpose of this project is to design and implement a Simple Video-Game Processor (SVGP) that interfaces with a VGA monitor to display a functional two-player pong game. Video systems operate on the fundamental concept of progressive scanning. This refers to when pixels are drawn sequentially from horizontally (left to right), and then vertically (top to bottom) to construct complete video frames. Each pixel is a small image element that has its own coordinates and, therefore, is addressed by rows and columns. The display's resolution represents the total number of pixels available horizontally and vertically.

The VGA display function can be broken down into 2 main synchronized scanning processes: horizontal scanning and vertical scanning. Horizontal scanning refers to the rate at which individual lines of pixels are drawn across the screen (measured in lines per second). In contrast, vertical scanning represents the rate at which complete video frames are refreshed (measured in frames per second, when all lines are scanned). These two processes work together through a progressive scan methodology to create smooth video output.

The VGA standard operates at specific timing parameters to ensure compatibility with displays. For a 60 Hz refresh rate, a pixel clock of 25 MHz is required. Each pixel is scanned within the active display area (measured as 640 pixels x 480 lines). Therefore, each horizontal line consists of 800 total clock cycles (640 visible pixels plus 160 blanking pixels for sync and porch periods), and each complete frame consists of 525 total lines (480 visible lines plus 45 blanking lines). This results in 420,000 clock cycles per frame, which at 25 MHz produces a frame time of 16.8 milliseconds, yielding approximately 59.5 Hz refresh rate. Surrounding this active area are timing intervals called front porches and back porches, which appear before and after synchronization pulses. The porches are essentially buffer or waiting periods before and after the synchronization pulse windows. The horizontal sync signal (HSYNC) and vertical sync signal (VSYNC) coordinate these scanning operations, but they have a negative polarity. At the beginning of each frame, both HSYNC and VSYNC are set high, and the first line of pixels is drawn. When the end of a line is reached, the HSYNC signal toggles to indicate that the electron beam (in CRT monitors) or pixel controller (in modern flat panels) should go back to the beginning of the next line. This process repeats for all visible lines in the frame. Once all 480 visible lines have been drawn, the VSYNC signal toggles from high to low, which indicates that a complete frame has been rendered, and the scanning should return to the very first pixel in the top-left corner of the display to begin the next frame on the first, top-most line.

The system must generate synchronization signals while simultaneously updating the positions of dynamic game elements, detecting collisions, and responding to user input from the switches on the FPGA board. This real-time processing requirement makes the project an excellent demonstration of FPGA capabilities for high-performance applications. This is because multiple parallel processes must execute simultaneously within strict timing constraints.

System Specifications

Behavioral Specifications

The Simple Video Game operates as a real-time VGA system that continuously renders and updates graphical elements at a refresh rate of 60 frames per second, sent to the monitor. Each frame is constructed line by line, using VGA horizontal and vertical synchronization timing signals derived from a 25 MHz pixel clock. The processor simultaneously performs multiple tasks such as generating synchronization pulses, producing RGB video signals, monitoring user inputs, and computing the ball motion trajectory. The paddle movement is controlled by physical switches on the FPGA board, allowing immediate response to user interaction.

The ball position and trajectory are calculated on every frame based on its existing direction and velocity. Collision detection logic is in charge of modifying the ball's path when it comes into contact with field boundaries or paddles, which ensures realistic reflection as per the game requirements. When the ball goes through the net and scores a goal, its colour changes to red to indicate a score event before then resetting to the center in yellow to restart play. The system is fully pipelined and operates in real time without any flicker, maintaining frame synchronization and stable output to the VGA display.

Functional Specifications

For the VGA Signal Generation, the processor produces precise horizontal and vertical synchronization signals that the VGA protocol uses for the visible resolution of 640×480 at 60 frames per second, resulting from the 25MHz clock, which drives each pixel generated. There is a non-active region that is part of the VGA resolution, resulting in a total of 800×525 resolution. The inactive region consists of front and back porches, as well as the synchronization clock cycles that are sandwiched between the porches. The same format is used for both horizontal and vertical portions of generating a frame. The 25MHz clock is derived from dividing the 50MHz signal in half, by only toggling the clock value change on every rising edge of the original 50MHz. Careful timing was implemented to ensure correct timing for all phases, consisting of front porch, sync pulse, and back porch for each frame in the active video.

Video Frame Rendering

The video game is represented by a static green background for the field. The white borders are drawn surrounding the play area, defining the upper, lower, and side boundaries. Specific screen regions are allocated for the goal zones on the left and right edges. A yellow ball that moves continuously across the screen is displayed, and two player paddles, colored blue and pink, are

positioned on opposite sides of the field. The paddle positions are updated in real time based on 4 switch inputs (SW0–SW3) from the FPGA board. Switches 0 and 1 are responsible for controlling the blue left paddle, with 2 and 3 controlling the right pink paddle, controlling the upwards enable, and downwards enable for each paddle, respectively.

The system continuously monitors the ball's position to detect collisions with any field border or player paddle. When a collision occurs, the ball's trajectory is adjusted by ±90 degrees according to the direction of impact, while its speed remains constant to ensure smooth, uninterrupted motion. This is achieved with a simple inverse of the X or Y velocity, depending on whether the ball has collided against a vertical or horizontal surface. If the ball passes through a goal region on either the left or right side of the field, it triggers a scoring event in which the ball temporarily changes to red and disappears from the visible play area. After a short delay, it reappears at the center of the screen in yellow, signalling the start of a new round. Throughout the operation, the RGB output channels are controlled to produce the correct colours for all the components. Green for the field background, white for the borders, black for the dotted line down the middle, blue and pink for the paddles, yellow for the active ball, and red to indicate a goal event.

Device Design and Description

Symbol Diagram

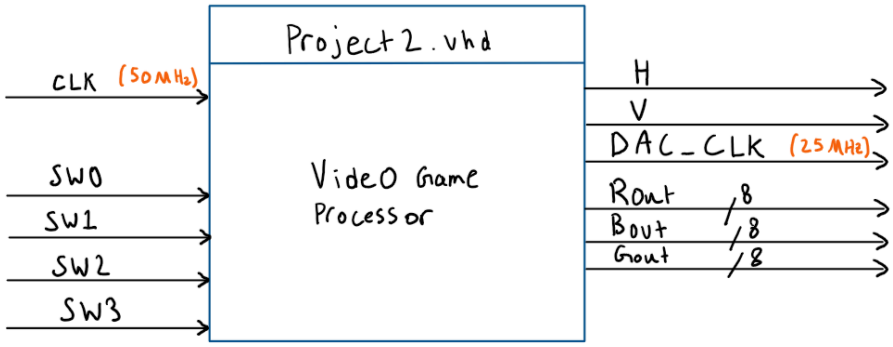

Figure 1.0: Symbol Diagram

Due to the simplicity and smaller scale of this project, we were able to implement the entire pong game within 400 lines of code in one VHD file. Thus, the above symbol diagram is created. The inputs and outputs are taken directly from the port declaration in the VHDL code, as it defines the interface of the entity. CLK is the external 50 MHz oscillator on the FPGA board, whereas DAC_CLK is the new clock that was divided for a lower frequency (50 MHz to 25 MHz). SW0-SW3 are the 4 switches on the FPGA board used for paddle movement. H and V are the

horizontal and vertical syncs, respectively. Lastly, RGB are the outputs for the drawing process, which applies colour to the VGA display.

VGA Specifications

The tables below show the VGA specifications for the projects:

VGA Horizontal Parameters	Clock Cycles	Signal Level
End of a Line (Complete)	800	H = ‘1’ (end of line is back Porch)
Active Region Width	640	H = ‘1’
Front Porch	16	H = ‘1’
Sync Pulse	96	H = ‘0’
Back Porch	48	H = ‘1’

Table 1.0: VGA Horizontal Parameters

VGA Vertical Parameters	Clock Cycles	Signal Level
End of a Line (Complete)	525	V = ‘1’ (end of line is back Porch)
Active Region Width	480	V = ‘1’
Front Porch	10	V = ‘1’
Sync Pulse	2	V = ‘0’
Back Porch	33	V = ‘1’

Table 2.0: VGA Vertical Parameters

The specifications of the clock cycles help determine the active region, which consequently helps map out what pixel coordinates are needed for the boundaries, paddles, gate and their corresponding widths. Another important aspect to consider is the horizontal and vertical sync pulses and how the VGA uses negative polarity. This means that the pulse is actually always high during every clock except the pulse window itself. When the pulse goes low, it signals that it is time for a new line (horizontally) or a new frame (vertically).

Block Diagram

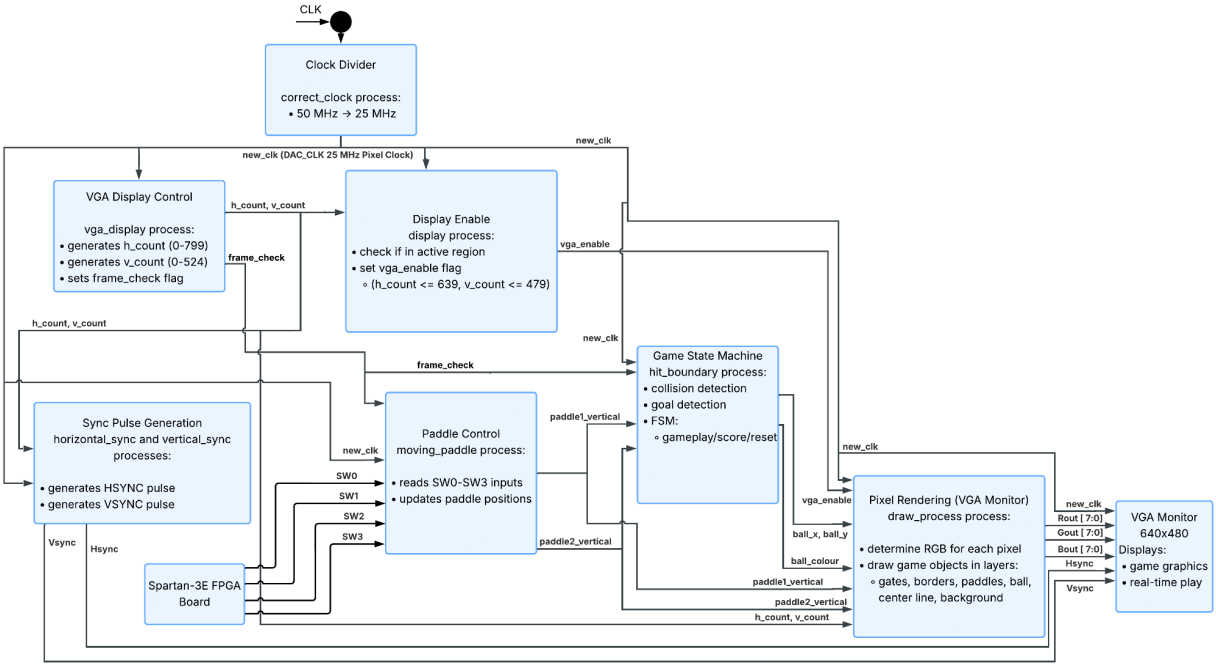

Figure 2.0: Block Diagram

The block diagram above highlights all of the key processes that were involved in creating the game. Each process represents a different aspect of the game. For the game logic, there is paddle control and a game state machine, but for the VGA display itself, there are the display control, display enable and sync pulse generation processes. All of which come together in the pixel rendering process that actually draws the frames by applying the specific RGB colours selected. One crucial component that brings everything together is the clock divider, which connects every single sequential component as the Pixel Clock runs on 25 MHz and not 50 MHz.

Process Diagram

The process diagram in Figure 3.0 below helps capture the overall flow of the entire game and the per-clock-cycle operation of the VGA Pong system running at 25 MHz. It begins by first dividing the clock to produce the 25 MHz pixel clock, which drives all of the different synchronous processes. On every rising edge, the horizontal counter increments until 799 pixels, and then resets, while the vertical counter increments until line 524, and then resets and sets frame_check = 1. This whole process is how the VGA display is able to create a frame. When frame_check = 1, it triggers the gameplay state of the state machine, including things like moving paddles, ball movement, collisions and transitioning FSM states. In parallel to this, the

horizontal and vertical sync pulses are generated to maintain VGA timing. These pulses are an active high when it is not in the pulse window, and an active low when they are in their pulse window. This is because they have negative polarity. Furthermore, once the active display region of (640x480) is reached, vga_enable = 1, and this is the trigger that activates the drawing process to render colours for the game objects, including the paddles, boundaries, ball, and background. Lastly, the RGB outputs, sync signals and the pixel clock are sent to the VGA monitor itself, producing one pixel every 40 ns at a 60 Hz refresh rate.

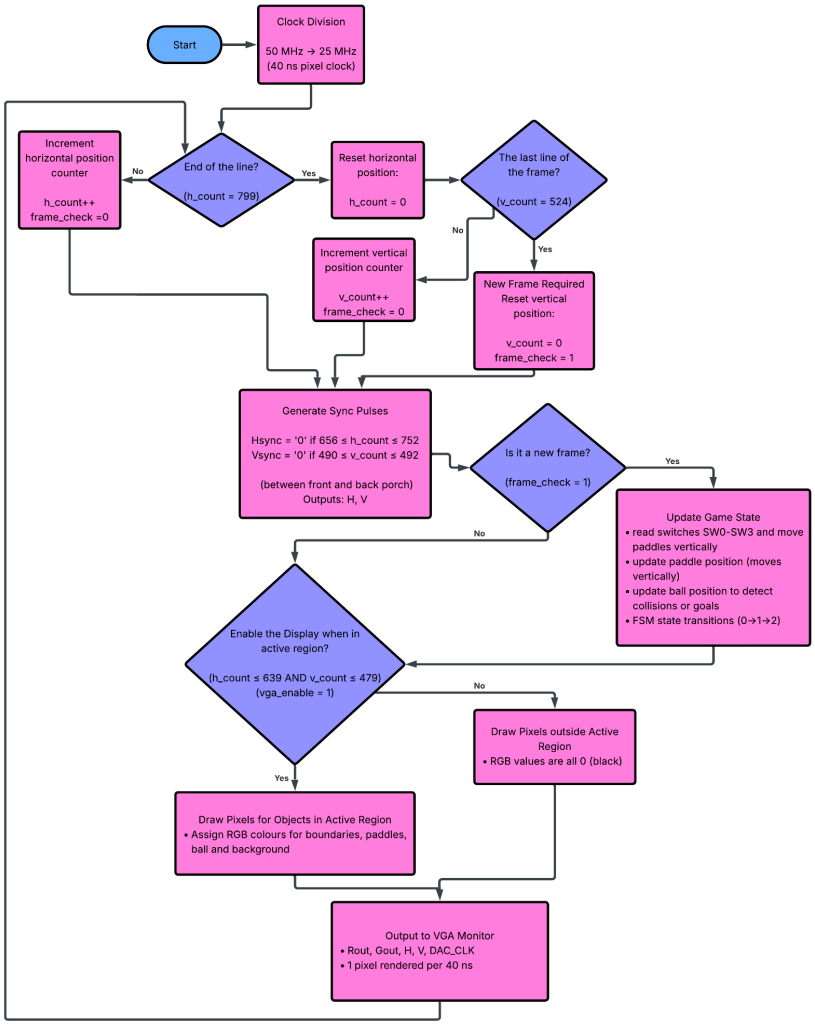

Figure 3.0: Process Diagram

State Diagram

The state diagram in Figure 4.0 below was implemented specifically for the ball trajectory when colliding with a boundary. If the ball is in a specific range, it should score a goal instead. This helped differentiate between what changes should occur visually when the game is still ongoing

versus when a goal is finally scored. In summary, there are 3 states: State 0: Gameplay, State 1: Scoring and State 2: Reset.

In state 0, this is simply for any of the regular gameplay, including if the ball hits a boundary or a paddle, as it should change its trajectory by 90 degrees or simply reverse its horizontal velocity, respectively. However, the conditional checks for the left and right boundaries were separated into considering the region of the gate (hole for scoring). Should a goal be scored, then the state will change to state 1. In state 1, the ball colour changes to red, and the ball naturally moves off-screen. A counter is included to add a buffer/waiting period before restarting the game. We decided to wait for approximately 3 seconds, which is determined by the calculation below:

$$Total\ Pixels\ for\ Frame\ =\ 800\ \times\ 525\ =\ 420000\ clock\ cycles$$
$$420000\ \times\ \frac{1}{25\ MHz}\ =\ 16.8\ \frac{ms}{frame}$$
$$Delay\ Time\ =\ 180\ frames\ \times\ 16.8\ \frac{ms}{frame}\ =\ 3024\ ms\ \approx\ 3\ seconds$$

When the waiting period is completed, it enters the reset state. In this state, the ball colour is changed back to yellow, the ball is relocated back to the center, and lastly, the velocity is reset to (2,2). The location of the paddles is also reset to signify that a new game has begun, and as a result, the game commences again, and it is now in state 0.

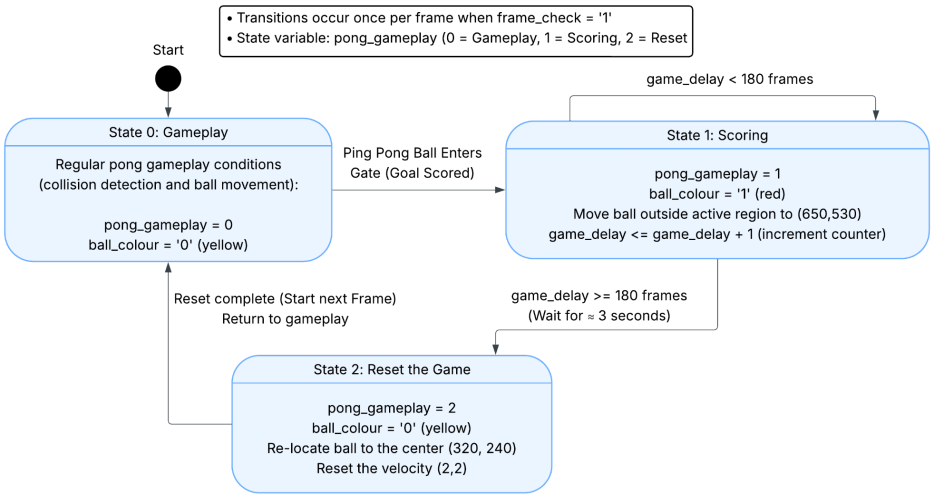

Figure 4.0: State Diagram

Results

Timing

The following waveforms were captured using ChipScope:

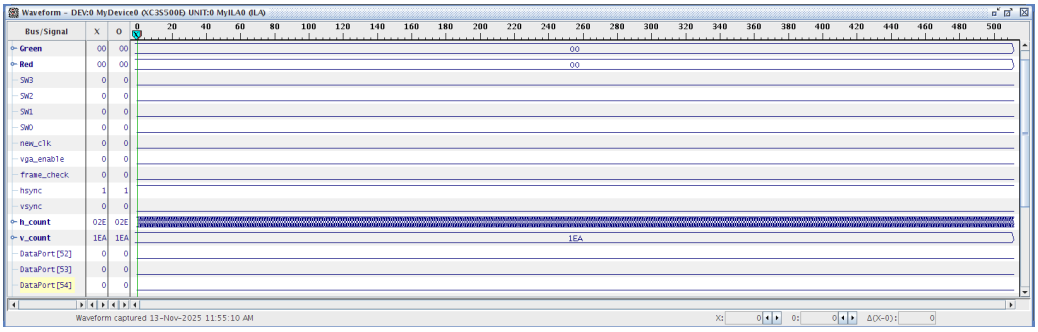

Figure 5.0

The vertical synchronization signal, established by vsync, remains low throughout Figure 5.0. However, because the capture window spans only approximately 550 clock cycles, it is therefore not large enough to display the full low duration. Vsync is expected to remain low for two complete VGA lines, which equates to 1600 clock cycles since there are 800 cycles per line. This occurrence happens once per frame, 60 times per second.

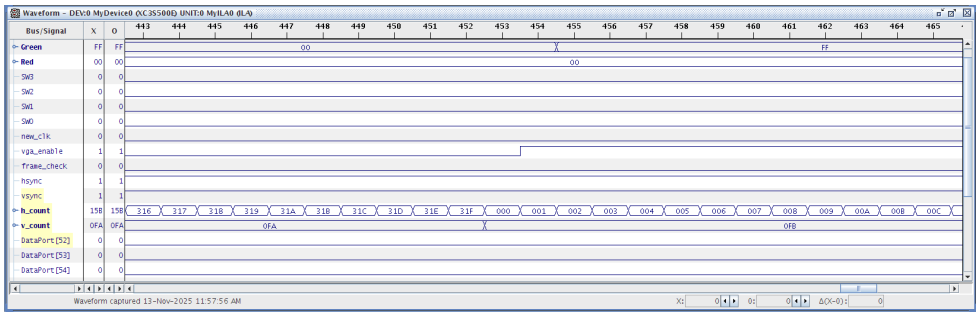

Figure 5.1

In Figure 5.1, it is shown that the VGA enable signal transitions high immediately after the start of a new line, indicated by an increment in the vertical counter. This corresponds to the moment the system exits the horizontal back porch interval and enters the active video region at pixel 0, displaying pixels on the monitor for the graphical interface to the user.

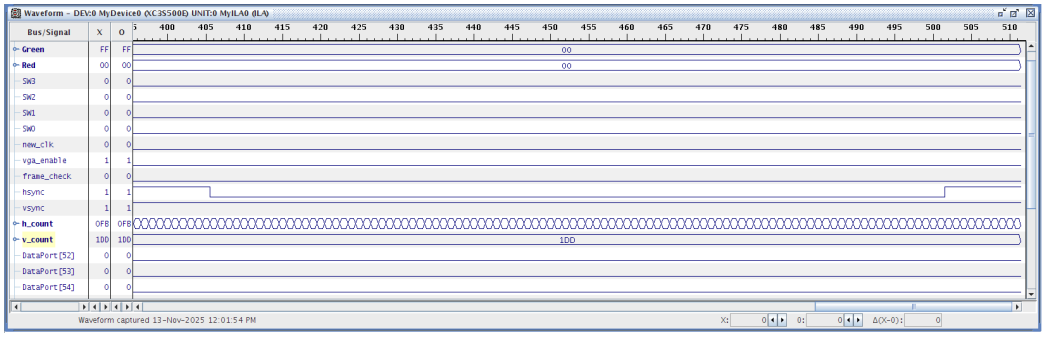

Figure 5.2

In Figure 5.2, the horizontal synchronization signal established as hsync asserts low for 96 clock cycles between the front and back porch intervals for the synchronization of the VGA protocol. After the back porch finishes, the next line begins and the vertical counter increments. This low period occurs when the horizontal counter advances from hexadecimal 0x291 to 0x2F0, as shown below in Figure 5.3, matching the VGA timing specification.

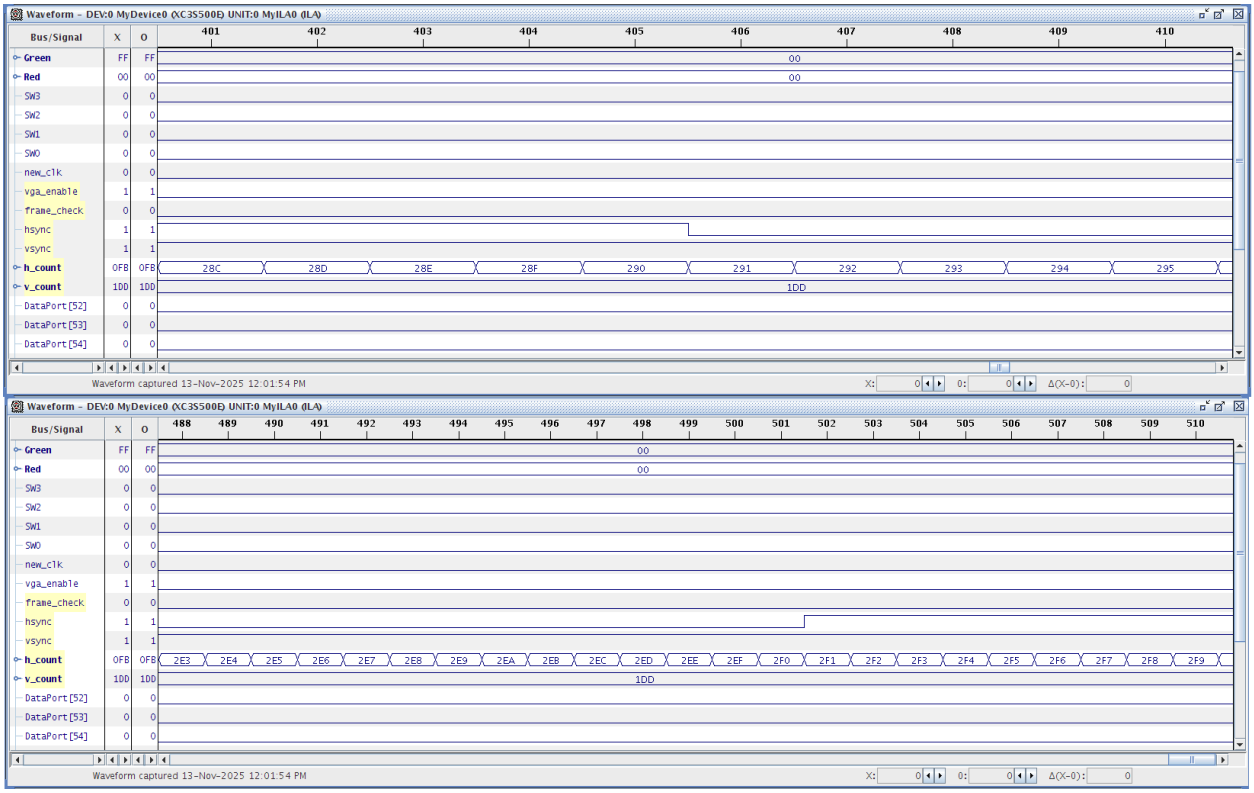

Figure 5.3

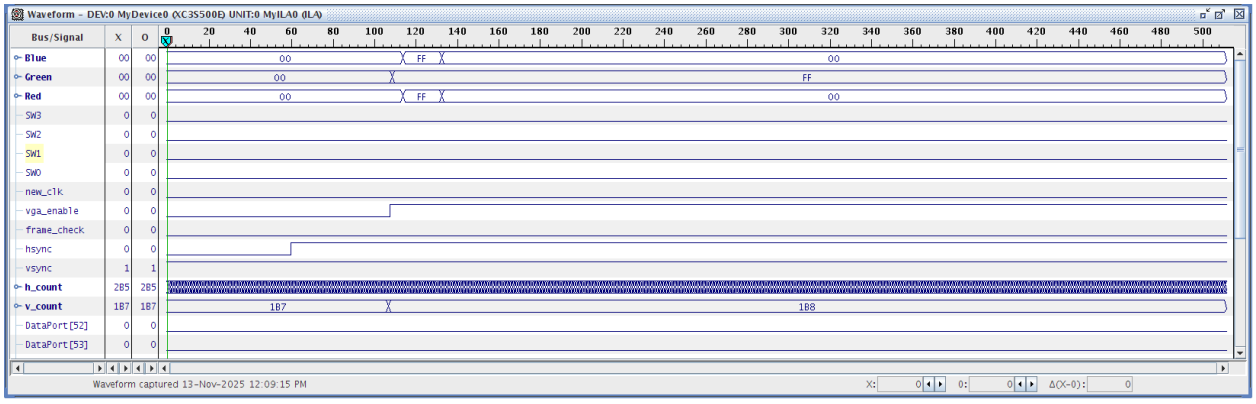

Figure 5.4

In Figure 5.4, it is confirmed that the pixel-colour generation immediately begins after a new line has started, with the RGB outputs reflecting the intended graphical display. For the first five

clock cycles, the green channel is set to 0xFF while red and blue remain at zero, producing five green pixels. For the next nineteen cycles, all three RGB channels are set to 0xFF, creating white pixels corresponding to the border region defined in the code. Afterward, the red and blue channels return to zero while green stays high, continuing the drawing of the green game field.

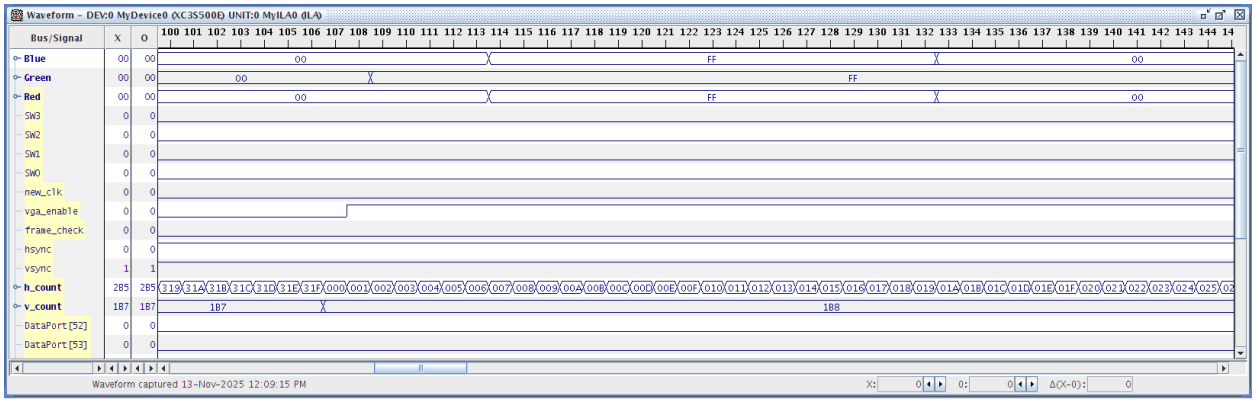

Figure 5.5

In Figure 5.5 above, the RGB values explained above are zoomed in. This helps visualize the h_count as the new line starts, and the green is displayed, followed by white, and green after for the field.

Screen Capture

Below is a screenshot of the functioning game in the lab room:

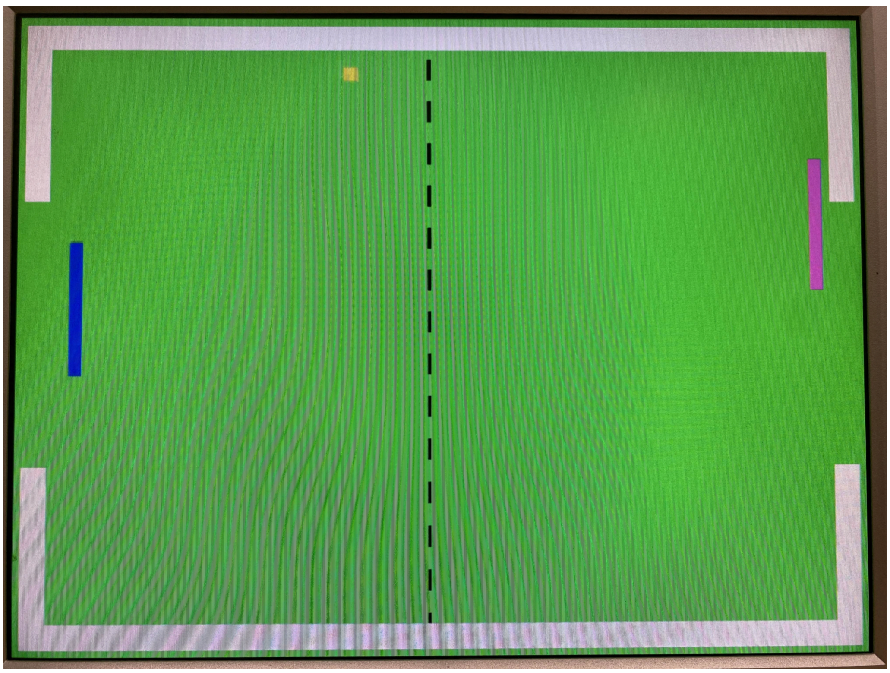

Figure 6.0

In Figure 6.0 above, the implementation of the game is seen with both paddles, a white border, a dotted center-line in the mid-section, as well as the yellow ball. There are green goal spaces on the left and right vertical edges, for the ball to be able to score in the goal post.

Conclusion

In conclusion, this Simple Video-Game Processor project successfully demonstrated the design and implementation of a real-time VGA controller with integrated game logic using VHDL on a Xilinx Spartan-3E FPGA. The system generated a 640×480 pixel display at 60 Hz with properly timed HSYNC and VSYNC signals adhering to VGA specifications. The game can be divided into 2 main categories: static elements (green field and gate, white borders, dashed center line) and dynamic elements (paddles and ball). Both of these types of elements rendered correctly with accurate collision detection and responsive switch-based control on the FPGA board.

Furthermore, the project reinforced key concepts in real-time digital system design, including strict timing requirements for video generation and parallel process implementation. The VHDL architecture was completed with separate processes for clock generation, sync signals, collision detection, paddle movement, and pixel rendering for effective debugging and maintainability. Lastly, descriptive comments, clear signal naming and constant definitions enhanced code readability.

Overall, the project successfully bridged the gap between theoretical knowledge of concepts such as VGA timing with practical FPGA-based real-time system design. The functional two-player pong game demonstrates proper application of digital design principles from low-level timing control to high-level game logic. Thus, every aspect of this project meets all project requirements and provides valuable experience applicable to future digital systems engineering work.

References

[1] L. Kirischian, Coe 758 Xilinx Ise 13.4 tutorial 1, https://www.ecb.torontomu.ca/~lkirisch/ele758/handouts/Tutorial1_ISE_Project_Creation.pdf (accessed 2025).

[2] L. Kirischian, Torontomu, https://www.ee.torontomu.ca/~lkirisch/ele758/labs/Cache%20Project%5B12-09-10%5D.pdf (accessed Nov. 5, 2025).

[3] W. Storr, “Frequency division using divide-by-2 toggle flip-flops,” Basic Electronics Tutorials, https://www.electronics-tutorials.ws/counter/count_1.html (accessed Nov. 4, 2025).

[4] J. Darvill, “Using VHDL process blocks to model sequential logic,” FPGA Tutorial, https://fpgatutorial.com/using-vhdl-process-blocks-to-model-sequential-logic/ (accessed Nov. 3, 2025).

[5] paras@ugrad. cs. ualberta. ca Paras Mehta, VHDL syntax reference, https://webdocs.cs.ualberta.ca/~amaral/courses/329/labs/VHDL_Reference.html (accessed Nov. 4, 2025).

Appendix

project2.vhd

```
----------------------------------------------------------------------------------
-- Company:
-- Engineers: Sanjay Sivapragasam and Raymond Vo
-- Create Date:    14:25:52 11/04/2025
-- Module Name:    project2 - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
```

```
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity project2 is
Port (
   CLK : in std_logic;
         -- sync pulses for horizontal and vertical
   H : out std_logic; -- horizontal sync from .ucf
   V : out std_logic;  -- vertical sync from .ucf
         -- RGB vectors based on given .ucf file
         Rout : out std_logic_vector (7 downto 0);
         Gout : out std_logic_vector (7 downto 0);
         Bout : out std_logic_vector (7 downto 0);
         -- switches used for paddle movement
         SW0 : in std_logic;                    -- paddle1 up
   SW1 : in std_logic;                  -- paddle1 down
   SW2 : in std_logic;                  -- paddle2 up
   SW3 : in std_logic;                  -- paddle2 down
         DAC_CLK : out std_logic              -- pixel clock output
);
end project2;


architecture Behavioral of project2 is

-- setting up the VGA monitor display

-- horizontal VGA
constant total_horizontal : integer := 800; -- 800 clock cycles wide
constant active_horizontal : integer := 639; -- 640 clock cycles (count from 0)
constant front_porch_horizontal : integer := 16; -- 16 clock cycles
constant Hsync_pulse : integer := 96; -- 96 clock cycles
constant back_porch_horizontal : integer := 48; -- 48 clock cycles
```

```
-- vertical VGA
constant total_vertical : integer := 525; -- 525 clock cycles long
constant active_vertical : integer := 479; -- 480 clock cycles (count from 0)
constant front_porch_vertical : integer := 10; -- 10 clock cycles
constant Vsync_pulse : integer := 2; -- 2 clock cycles
constant back_porch_vertical : integer := 33; -- 33 clock cycles

-- signals for the sync pulses
signal Hsync, Vsync : std_logic := '0';


-- setting up the boundaries for the game
constant top_boundary : integer := 5; -- near top of the active region starting position of
boundary
constant bottom_boundary : integer := 475; -- near the bottom of the active region
constant left_boundary : integer := 5; -- left of the active region
constant right_boundary : integer := 635; -- right of the active region
constant boundary_width : integer := 20; -- giving all boundaries an equal width of 20 pixels
constant halfline : integer := 309; -- setting up the line at the halfway point
constant gate_top : integer := 140; -- the top of the gate for goals
constant gate_height : integer := 200; -- height of the gate


-- counting variables for the horizontal position
signal h_count : integer range 0 to total_horizontal := 0;

-- counting variables for the vertical position
signal v_count : integer range 0 to total_vertical := 0;

-- variables for the ping-pong ball
-- ping pong ball starts at the center of the game
signal ball_x : integer := 320; -- 640/2
signal ball_y : integer := 240; -- 480/2
signal ball_x_velocity : integer := 2; -- horizontal velocity of the ball
signal ball_y_velocity : integer := 2; -- vertical velocity of the ball
constant ball : integer := 12; -- setting the size of the ball
signal ball_colour: std_logic := '0'; -- initializing the ball colour

-- setup of the ping-pong rackets
```

```
constant paddle_width : integer := 10; -- width of the paddle
constant paddle_length : integer := 100; -- length of the paddle

-- player 1 paddle located on the left side
signal paddle1_horizontal : integer := 40;
signal paddle1_vertical : integer := 300;

-- player 2 paddle located on the right side
signal paddle2_horizontal : integer := 600;
signal paddle2_vertical : integer := 300;

-- flagging variables to indicate changes
signal frame_check : std_logic := '0'; -- status variable for if it is a new frame or not
signal vga_enable : std_logic := '0'; -- status variable for if the VGA display visible or not (active
region)

-- a variable for the new clock that will be used since VGA runs at 25MHz
signal new_clk : std_logic := '0';

-- setting up the RGB colour system
signal R, G, B : std_logic_vector(7 downto 0);


-- setting up the state machine for the states of play
signal pong_gameplay : integer := 0; -- 0 is gameplay, 1 is scoring, 2 is resetting the game
signal game_delay : integer := 0; -- used to delay between goal scored and resetting the game



-- ChipScope Components
component proj2_icon
  port (
    CONTROL0 : inout std_logic_vector(35 downto 0)
  );
end component;

component proj2_ila
  port (
    CONTROL : inout std_logic_vector(35 downto 0);
    CLK     : in  std_logic;
```

```
   DATA    : in  std_logic_vector(63 downto 0);
   TRIG0   : in  std_logic_vector(7 downto 0)
  );
end component;

signal CONTROL0 : std_logic_vector(35 downto 0);
signal ILA_DATA : std_logic_vector(63 downto 0);
signal ILA_TRIG : std_logic_vector(7 downto 0);



---------- beginning of the project----------
begin

-- changing the 50 MHz frequency clock to 25 MHz clock
correct_clock : process (CLK)
begin
   if (rising_edge(CLK)) then
          -- inverting the rising edge can divide frequency by 2
          -- if the value itself is divided, then the amplitude would change
          -- which is why the NOT command is used
      new_clk <= not new_clk;
   end if;
end process correct_clock;

-- this is the output of the newly divided clock for DAC pixel clock
DAC_CLK <= new_clk;


-- the frame is filled in horizontally first from the beginning to the end of the line
-- then this repeats for all the lines in the display
-- once the last pixel of the full frame is filled in (on the last line -  line 525)
-- the process needs to repeat again at index (0,0)
vga_display : process (new_clk)
begin
   if (rising_edge(new_clk)) then
      if (h_count = total_horizontal - 1) then -- at the end of the line, clock cycle 799
         h_count <= 0; -- reset back to beginning of a line, clock cycle 0
         if (v_count = total_vertical - 1) then -- at the last line, bottom of the display
            v_count <= 0; -- go back to first line, at the top of the display
```

```
            frame_check <= '1'; -- now requires a new frame since there are no more lines below
this current line
          else
            v_count <= v_count + 1; -- go to next line (move down vertically by 1)
            frame_check <= '0'; -- still on the same frame, not a new frame
          end if;
       else
          h_count <= h_count + 1; -- not at the end of the line, increment to next clock cycle to the
right
          frame_check <= '0'; -- still on current frame
       end if;
    end if;
end process vga_display;

-- for the sync pulses, VGA convention uses active-low. This means that when the voltage
-- drops to 0 (low pulse) to signal the end of a line or frame, it would be high (1) anytime else
-- this is because based on CRT standards, when the electron beam finished scanning a line
-- it would turn off, move to the left, and resume. So the pulse needs to be low between the
porches,
-- and high in the active region and the porches itself

-- setup of the horizontal sync
horizontal_sync : process (new_clk)
begin
    if (rising_edge(new_clk)) then
       if (h_count <= (active_horizontal + front_porch_horizontal)) or
          (h_count > (active_horizontal + front_porch_horizontal + Hsync_pulse)) then
          Hsync <= '1'; -- not in the pulse window
       else
          Hsync <= '0'; -- in the pulse window of 656 to 752 clock cycles
       end if;
    end if;
end process horizontal_sync;


-- setup of the vertical sync
vertical_sync : process (new_clk)
begin
    if (rising_edge(new_clk)) then
       if (v_count <= (active_vertical + front_porch_vertical)) or
```

```
        (v_count > (active_vertical + front_porch_vertical + Vsync_pulse)) then
         Vsync <= '1'; -- not in the pulse window
      else
         Vsync <= '0'; -- in the pulse window
      end if;
   end if;
end process vertical_sync;


-- the trajectory of the ball needs to change by 90 degrees when it hits a boundary
hit_boundary : process (new_clk)
begin
   if (rising_edge(new_clk)) then
      if (frame_check = '1') then



                  -- setting up a state machine to differentiate different stages of the game
                  -- there is the regular game play, when a player scores, and when the game resets
                  case pong_gameplay is

                  -- regular play
                  when 0 =>
                  -- if the ball hits the top or bottom boundaries
         if (ball_y <= top_boundary + boundary_width) then
            ball_y_velocity <= abs(ball_y_velocity); -- travel downwards
         elsif (ball_y + ball >= bottom_boundary - boundary_width) then
            ball_y_velocity <= (-1)* abs(ball_y_velocity); -- travel upwards
         end if;



         -- if the ball hits the left boundary
         if (ball_x <= left_boundary + boundary_width and not(ball_y > gate_top and ball_y <=
gate_top + gate_height)) then
            ball_x_velocity <= abs(ball_x_velocity);
                                    -- if the ball hits the right boundary
         elsif (ball_x + ball >= right_boundary - boundary_width and not(ball_y > gate_top and
ball_y <= gate_top + gate_height)) then
            ball_x_velocity <= (-1)* abs(ball_x_velocity);
         end if;
```

```
        -- when the ball hits paddle1
        if (ball_x <= paddle1_horizontal + paddle_width and ball_x >= paddle1_horizontal and
           ball_y + ball >= paddle1_vertical and ball_y <= paddle1_vertical + paddle_length)
then
           ball_x_velocity <= abs (ball_x_velocity);
        end if;

                                -- when the ball hits paddle2
        if (ball_x + ball >= paddle2_horizontal and ball_x <= paddle2_horizontal + paddle_width
and
           ball_y + ball >= paddle2_vertical and ball_y <= paddle2_vertical + paddle_length)
then
           ball_x_velocity <= (-1)* abs(ball_x_velocity);
        end if;

        -- provide the ball velocity so it can move
        ball_x <= ball_x + ball_x_velocity;
        ball_y <= ball_y + ball_y_velocity;



                                -- if the ball hits the left or right boundaries
        if (((ball_x <= left_boundary + boundary_width) and
                                (ball_y > gate_top and ball_y <= gate_top + gate_height)) or
                                ((ball_x + ball >= right_boundary - boundary_width) and
                                (ball_y > gate_top and ball_y <= gate_top + gate_height))) then
           ball_colour <= '1';
                                        pong_gameplay <= 1;
                                        game_delay <= 0;
        end if;


                -- when they score a goal
                when 1 =>
                game_delay <= game_delay +1;

                ball_x <= ball_x + ball_x_velocity;
                ball_y <= ball_y + ball_y_velocity;
```

```
                --ball_x <= 650;
                --ball_y <= 530;

                if game_delay >= 180 then
                        pong_gameplay <= 2;
                end if;


                -- to reset the game
                when 2 =>
                ball_x <= 320;
                ball_y <= 240;
                ball_x_velocity <= 2;
                ball_y_velocity <= 2;
                ball_colour <= '0';
                pong_gameplay <= 0;

                -- player 1 paddle located on the left side
                        --paddle1_horizontal <= 40;
                        --paddle1_vertical <= 300;

                        -- player 2 paddle located on the right side
                        --paddle2_horizontal <= 600;
                        --paddle2_vertical <= 300;
                        when others =>
                                        pong_gameplay <= 0;
                        end case;
      end if;
   end if;
end process hit_boundary;


-- enabling the display of the content on the monitor
display : process (new_clk)
begin
   if (rising_edge(new_clk)) then
      if (h_count <= active_horizontal and v_count <= active_vertical) then
         vga_enable <= '1'; -- flag variable set to 1 to indicate it is in the active region
      else
```

```
            vga_enable <= '0';
        end if;
    end if;
end process display;


-- moving the paddles (using switches)
moving_paddle : process (new_clk)
begin
    if (rising_edge(new_clk)) then
        if (frame_check = '1') then
            -- player 1 paddle movement
            if (SW0 = '1' and SW1 = '1') then
                                                paddle1_vertical <= paddle1_vertical;
                                        elsif (SW0 = '1' and paddle1_vertical > top_boundary +
boundary_width) then
                paddle1_vertical <= paddle1_vertical - 2; -- move up
            elsif (SW1 = '1' and paddle1_vertical + paddle_length < bottom_boundary -
boundary_width) then
                paddle1_vertical <= paddle1_vertical + 2; -- move down
            end if;

            -- player 2 paddle movement
                                        if (SW2 = '1' and SW3 = '1') then
                                                paddle2_vertical <= paddle2_vertical;
            elsif (SW2 = '1' and paddle2_vertical > top_boundary + boundary_width) then
                paddle2_vertical <= paddle2_vertical - 2; -- move up
            elsif (SW3 = '1' and paddle2_vertical + paddle_length < bottom_boundary -
boundary_width) then
                paddle2_vertical <= paddle2_vertical + 2; -- move down
            end if;
        end if;
    end if;
end process moving_paddle;


-- the final portion of this project is to draw everything with the correct colour schemes
draw_process : process(new_clk)
begin
  if rising_edge(new_clk) then
```

```
   if (vga_enable = '1') then -- if the vga is ready to display


 -- colour of ball
     if (h_count >= ball_x and h_count < ball_x + ball and
           v_count >= ball_y and v_count < ball_y + ball) then
       if (ball_colour = '1') then
         R <= (others => '1'); G <= (others => '0'); B <= (others => '0'); -- red (goal)
       else
         R <= (others => '1'); G <= (others => '1'); B <= (others => '0'); -- yellow (normal)
       end if;


           -- making the gates green (drawn before borders so they are visible)
     elsif ((h_count < left_boundary + boundary_width and
           v_count > gate_top and v_count < gate_top + gate_height) or
         (h_count > right_boundary - boundary_width and
           v_count > gate_top and v_count < gate_top + gate_height)) then
       R <= (others => '0'); G <= (others => '1'); B <= (others => '0');

     -- making all the boundaries white as RGB [111] white
     elsif (v_count < top_boundary + boundary_width and v_count > top_boundary and h_count
> left_boundary and h_count < right_boundary) or
           (v_count > bottom_boundary - boundary_width and v_count < bottom_boundary and
h_count > left_boundary and h_count < right_boundary) or
           (h_count < left_boundary + boundary_width and h_count > left_boundary and v_count >
top_boundary and v_count < bottom_boundary) or
           (h_count > right_boundary - boundary_width and h_count < right_boundary and v_count
> top_boundary and v_count < bottom_boundary) then
       R <= (others => '1'); G <= (others => '1'); B <= (others => '1');
                   elsif (v_count < top_boundary + 2) then
                     R <= (others => '0'); G <= (others => '1'); B <= (others => '0');
--                   elsif (v_count < top_boundary or
--                               v_count > bottom_boundary or
--                                     h_count < left_boundary or
--                                     h_count > right_boundary) then
--                     R <= (others => '0'); G <= (others => '1'); B <= (others => '0');

     -- colouring the paddles
                   -- player 1's paddle
```

```
      elsif (h_count >= paddle1_horizontal and
            h_count < paddle1_horizontal + paddle_width and
            v_count >= paddle1_vertical and
            v_count < paddle1_vertical + paddle_length) then
        R <= (others => '0'); G <= (others => '0'); B <= (others => '1'); -- blue

                    -- player 2's paddle
      elsif (h_count >= paddle2_horizontal and
            h_count < paddle2_horizontal + paddle_width and
            v_count >= paddle2_vertical and
            v_count < paddle2_vertical + paddle_length) then
        R <= (others => '1'); G <= (others => '0'); B <= (others => '1'); -- purple

--      -- colour of ball
--      elsif (h_count >= ball_x and h_count < ball_x + ball and
--            v_count >= ball_y and v_count < ball_y + ball) then
--        if (ball_colour = '1') then
--          R <= (others => '1'); G <= (others => '0'); B <= (others => '0'); -- red (goal)
--        else
--          R <= (others => '1'); G <= (others => '1'); B <= (others => '0'); -- yellow (normal)
--        end if;

      -- center line
      elsif (h_count >= halfline and h_count < halfline + 3) then
                              if ((v_count mod 32) < 16) then
        R <= (others => '0'); G <= (others => '0'); B <= (others => '0'); -- black
                    else
                    R <= (others => '0'); G <= (others => '1'); B <= (others => '0'); -- black
                    end if;

      -- background (green field)
      else
        R <= (others => '0'); G <= (others => '1'); B <= (others => '0');
      end if;
    else
      R <= (others => '0'); G <= (others => '0'); B <= (others => '0');
    end if;
  end if;
end process draw_process;
```

```
-- Map the RGB outputs to match .ucf pins
Rout <= R;
Gout <= G;
Bout <= B;
-- Map the sync and clock signals to the ports based on .ucf
H <= Hsync;
V <= Vsync;
DAC_CLK <= new_clk;


-- chipscope waveform setup
u_icon : proj2_icon
  port map (
    CONTROL0 => CONTROL0
  );

u_ila : proj2_ila
  port map (
    CONTROL => CONTROL0,
    CLK     => new_clk,   -- pixel clock sampling
    DATA    => ILA_DATA,
    TRIG0   => ILA_TRIG
  );

-- map internal signals to ILA DATA inputs
ILA_DATA(51 downto 42) <= std_logic_vector(to_unsigned(h_count, 10));
ILA_DATA(41 downto 33) <= std_logic_vector(to_unsigned(v_count, 9));
ILA_DATA(32) <= Vsync;
ILA_DATA(31) <= Hsync;
ILA_DATA(30) <= frame_check;
ILA_DATA(29) <= vga_enable;
ILA_DATA(28) <= new_clk;
ILA_DATA(27) <= SW0;
ILA_DATA(26) <= SW1;
ILA_DATA(25) <= SW2;
ILA_DATA(24) <= SW3;
ILA_DATA(23 downto 16) <= R(7 downto 0);
ILA_DATA(15 downto 8) <= G(7 downto 0);
ILA_DATA(7 downto 0) <= B(7 downto 0);
```

```
end Behavioral;
```

project2.ucf

```
# 50 MHz input clock.
NET "clk" LOC = "c9";
# Synchronization signals.
NET "H" LOC = "c5";
NET "V" LOC = "d5";
# Pixel clock for the video DAC.
NET "DAC_CLK" LOC = "a4";
# Blue channel pins.
NET "Bout<7>" LOC = "b16";
NET "Bout<6>" LOC = "a16";
NET "Bout<5>" LOC = "d14";
NET "Bout<4>" LOC = "c14";
NET "Bout<3>" LOC = "b14";
NET "Bout<2>" LOC = "a14";
NET "Bout<1>" LOC = "b13";
NET "Bout<0>" LOC = "a13";
# Green channel pins.
NET "Gout<0>" LOC = "f9";
NET "Gout<1>" LOC = "e9";
NET "Gout<2>" LOC = "d11";
NET "Gout<3>" LOC = "c11";
NET "Gout<4>" LOC = "f11";
NET "Gout<5>" LOC = "e11";
NET "Gout<6>" LOC = "e12";
NET "Gout<7>" LOC = "f12";
# Red channel pins.
NET "Rout<0>" LOC = "a6";
NET "Rout<1>" LOC = "b6";
NET "Rout<2>" LOC = "e7";
NET "Rout<3>" LOC = "f7";
NET "Rout<4>" LOC = "d7";
NET "Rout<5>" LOC = "c7";
NET "Rout<6>" LOC = "f8";
NET "Rout<7>" LOC = "e8";
```

```
# On-board switches.
NET "SW0" LOC = "N17";
NET "SW1" LOC = "H18";
NET "SW2" LOC = "L14";
NET "SW3" LOC = "L13";
```