Utah State University - ECE 6930

Brick Breaker Final Project

Nathan Tipton and Erik Sargent

December 11, 2018

Contents

1 Introduction **2**

2 Requirements **2**

3 Procedure **2**
- 3.1 Ball Module . . . 3
- 3.2 Bricks Module . . . 3
- 3.3 Paddle Module . . . 3
- 3.4 Average ADC Module . . . 3
- 3.5 ADC module . . . 3
- 3.6 Tone Module . . . 4
- 3.7 sevenSeg Module . . . 4
- 3.8 Status Module . . . 4
- 3.9 VGA module . . . 4

4 Results **4**

5 Conclusion **5**

6 Appendix **5**
- 6.1 Ball Module . . . 6
- 6.2 Bricks Module . . . 11
- 6.3 Paddle Module . . . 15
- 6.4 Pixel Status Module . . . 18
- 6.5 Sound Module . . . 19
- 6.6 VGA display Module . . . 21
- 6.7 Seven Segment display Module . . . 25
- 6.8 Top Level Module . . . 27
- 6.9 Paddle Position Module . . . 33
- 6.10 ADC Module . . . 35

1 Introduction

The goal of this project is to develop a working Brick Breaker game for the DE-10 Lite FPGA module.

2 Requirements

Requirements for the Brick Breaker Final Project are:

- The digital design must be done in VHDL.
- The game image shall be 640 pixels wide by 480 pixels tall.
- Use the two push button to control game play. One button resets the game, the other drops a new ball. The user may request 5 new balls, after which the game is over and reset must be pressed to continue.
- Upon reset, the upper half of the screen (240 lines) shall be filled with red bricks separated by white mortar. Brick dimensions shall be 15 pixels wide by 7 pixels high. 1-pixel-wide strips of mortar completely separate every brick from each of its neighbors. Neighboring rows of bricks shall be offset by half a brick, thus every other row shall be aligned vertically.
- Upon reset, a brown paddle will appear in the middle of the bottom of the display. The paddle shall be 40 pixels wide by 5 pixels tall.
- Screen portions not occupied by bricks, mortar, ball, or paddle shall be black.
- When the drop ball button is pressed (and there are balls remaining), a ball will drop straight down from the (horizontal and vertical) center of the screen, so as to hit the middle of the paddle. The ball shall be a square or octagon with dimensions 10 pixels by 10 pixels. Pick an appropriate rate of movement for the ball.
- The user can move the paddle from side-to-side along the bottom of the screen. The paddle movement can be controlled by either a potentiometer interfaced to the FPGA's ADC, or via the on-board accelerometer.
- If the ball misses the paddle and falls out the bottom of the screen, the ball "dies" and a new ball must be requested. The ball bounces off the paddle, the sides of the screen, and the top of the screen.
- If the ball "hits" one or more bricks, the ball bounces off AND causes those bricks to disappear from the wall. Entire bricks must disappear (i.e. no partial bricks shall remain on the screen). Only bricks that have been hit by the ball disappear. The object of the game is knock all of the bricks out of the wall before all 5 balls have been lost.
- Produce 4 unique sounds using the FPGA and an external speaker. One when the ball hits the paddle, one when the ball hits the top or side of the screen, one when the ball breaks a brick, and one when the ball dies.

3 Procedure

Figure 1 shows the structure for the project. The details of these modules are described in more detail below.

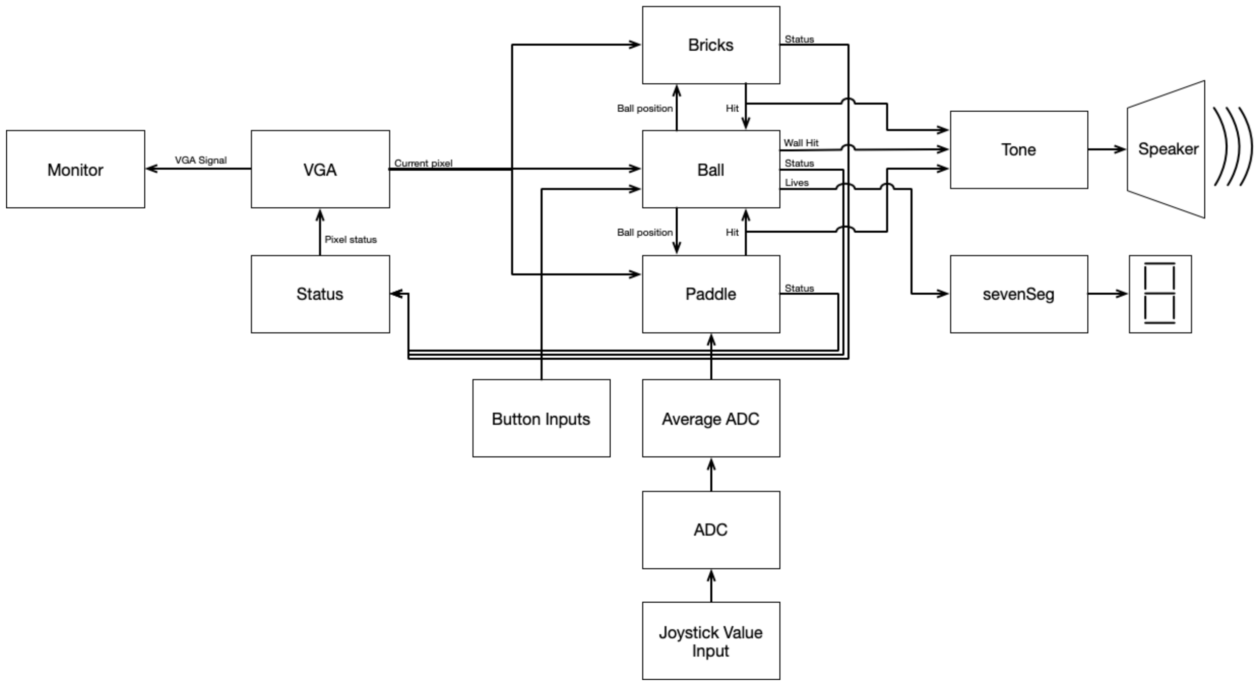


Figure 1: Structure of the project

3.1 Ball Module

The ball module determines the position and velocity of the ball on the screen. The position is output to the Bricks and Paddle module. The ball module uses the ball's current location to determine the status of the pixels to send to the Status Module. This ball also keeps track of the current number of lives as well as if the ball is alive or dead. A user input to the go button is needed to put the ball in an alive state and play the game.

3.2 Bricks Module

The bricks module tracks the status of each brick. The bricks are stored in a 2D `std_logic_vector`. The bricks module handles outputting brick or mortar statuses for display to the VGA module, and tracking ball collisions with the bricks.

3.3 Paddle Module

The paddle module takes the user input from the average ADC model to determine the location of the center of the paddle. The boundaries of the paddle are calculated and used to determine the status of the pixels sent to the Status module. The ball position is received by the paddle module and checked against it's current position. A hit signal is sent to the Ball module when a collision is detected.

3.4 Average ADC Module

This module receives the input from the ADC module. The input data is averaged and scaled to the monitor size before being sent to the Paddle module .This is to reduce the noise in the ADC value of the potentiometer position and keep the paddle on the screen.

3.5 ADC module

The ADC module uses the onboard ADC to read the position value from the joystick potentiometer. The ADC output is a 12-bit value.

3.6 Tone Module

The tone module handles playing sounds when the ball hits the paddle, a brick, either wall, or falls of the screen. Depending on the type of sound, the tone module outputs a square wave at 200-500Hz. The length of time the tone is played for is also dependent on the tone. The sound is output to a digital output pin on the board, and passed to a small speaker module. This is an easy method to produce several different tones.

3.7 sevenSeg Module

The sevenSeg module takes a 4 bit number from the ball module representing the number of lives and outputs it to one of the seven segment displays. This is used to display the remaining number of balls available to the player.

3.8 Status Module

The Status module receives a pixel index from the VGA model which is then sent to the Paddle, Bricks, and Ball modules. It receives the status of that pixel and forwards that status to the VGA module in order to display the proper color at that pixel location.

3.9 VGA module

The VGA module is a modified version of the module developed for the flag lab. The module outputs the index of the horizontal and vertical pixel it is next going to write to the display, and uses the value the status module passes back to determine which color to write out to the monitor. This easily abstracts the VGA module and the pixel drawing logic from the logic to track the bricks, ball, and paddle.

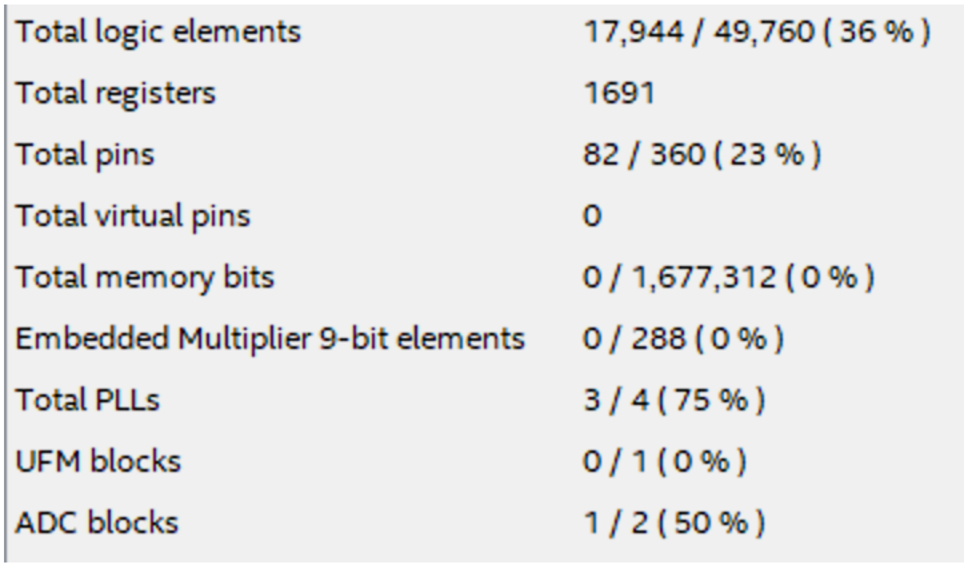
4 Results

Construction of the Brick Breaker game was successful, and the project meets all of the given requirements. Figure 2 shows the resource usage for the project. A large portion of the logic elements that are being used are used for handling and tracking the status of each of the bricks.

It was observed that the speaker module would produce noise on the ADC input to the paddle module when a tone was played. The paddle position on the screen would jitter when the ball hit was triggered on an object. This problem was addressed with a capacitor from the potentiometer input to ground.

The ball displayed as a square or octagon did not create the player experience that we desired. Anti-aliasing was used to allow the ball to display as a circle while still using the required 10 by 10 pixels as seen in Figure 3.

The ball speed seemed very slow and made the game tedious. We implemented a hit counter which gradually increases the ball speed until the ball dies, at which point the speed is reset to the initial speed. This improved the quality of game play for the player by making the game more difficult for better players.



Total logic elements	17,944 / 49,760 (36 %)
Total registers	1691
Total pins	82 / 360 (23 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	3 / 4 (75 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	1 / 2 (50 %)

Figure 2: Resource usage of the finished design

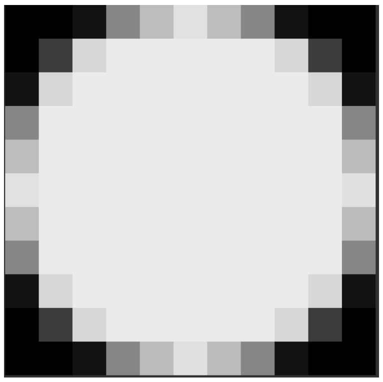


Figure 3: Ball Sprite blown up to visualize the anti-aliasing used on the edges to make it round.

5 Conclusion

The goal for this project was to build a Brick Breaker game on an FPGA. The game developed for the project is fully playable and meets all of the project requirements. The project also provided substantial experience in developing VHDL modules.

6 Appendix

The following sections contains the VHDL modules for the project. The modules can also be found at `https://github.com/eriksargent/BrickBreaker`.

6.1 Ball Module

Listing 1: Ball.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Ball is
	generic(
		BallUpdate :integer :=25000000


	);
	port(
		clk :in std_logic;
		hPos : in unsigned(10 downto 0);
		vPos : in unsigned(9 downto 0);
		ball_status : out std_logic_vector(3 downto 0);
		lives :out unsigned(3 downto 0);
		reset :in std_logic;
		go :in std_logic;
		die_sound :out std_logic;
		side_sound : out std_logic;
		BC_V :out unsigned(9 downto 0);
		BC_H :out unsigned(10 downto 0);
		PaddleHit :in std_logic_vector(4 downto 0);
		WallHit :in std_logic;
		WallHitSide :in std_logic_vector(5 downto 0);
		BallClk :out std_logic

	);
end entity Ball;

architecture RTL of Ball is
	type GameStatus is (dead,live);
	signal PS :GameStatus := dead;
	signal NS :GameStatus;
	signal ball_hPos : unsigned(12 downto 0) := b"0010011111100";
	signal ball_vPos :unsigned (11 downto 0) := b"001111010100";
	signal BCh :unsigned (10 downto 0) :=b"00100111111";
	signal BCv :unsigned (9 downto 0) := b"0011110101";
	signal BBh :unsigned (10 downto 0);
	signal BBv :unsigned (9 downto 0);
	signal BLh :unsigned (10 downto 0);
	signal BLv :unsigned (9 downto 0);
	signal BRh :unsigned (10 downto 0);
	signal BRv :unsigned (9 downto 0);
	signal BTh :unsigned (10 downto 0);
	signal BTv :unsigned (9 downto 0);
	signal count :integer:=0;
	signal update :std_logic;
	signal life :unsigned (3 downto 0) :=b"0101";
	signal die :std_logic:='0';
	constant speedLength : integer := 3;
	signal hSpeed :unsigned((speedLength-1) downto 0) := "000";
	signal vSpeed :unsigned((speedLength-1) downto 0) := "011";
	signal paddleHits : unsigned(23 downto 0) := (others => '0');
	begin

	BCh <= ball_hPos(12 downto 2);
	BCv <= ball_vPos(11 downto 2);

	BallStatus:process(hPos,vPos)
		variable md : signed(11 downto 0);
		variable dh : signed(11 downto 0);
```

```
        variable dv : signed(10 downto 0);
    begin

        dh := abs(signed('0' & hPos) - signed('0' & BCh));
        dv := abs(signed('0' & vPos) - signed('0' & BCv));
        md := dh + dv;

        if vPos >= BTv and vPos <= BBv then
            if hPos >= BLh and hPos <= BRh then
                if (dv <= 4 AND dh <= 2) OR (dv <= 2 AND dh <= 4) OR (dv <= 3 AND dh <= 3)
                    then
                    ball_status <= "0001";
                elsif (dv <= 4 AND dh <= 3) OR (dv <= 3 AND dh <= 4) then
                    ball_status <= "0010";
                elsif md <= 5 then
                    ball_status <= "0010";
                elsif md <= 6 then
                    ball_status <= "0100";
                else
                    ball_status <= "0000";
                end if;
            else
                ball_status<="0000";
            end if;
        else
            ball_status<="0000";
        end if;


    end process BallStatus;

    Ball_Update:process(clk)
    begin
        if rising_edge(clk) and PS=live then
            if ((paddleHits<600) and count = BallUpdate - to_integer(paddleHits sll 9)) or (
                paddleHits>=600 and count>=100000) then
                update<='1';
                count<=0;
            else
                update<='0';
                count<=count+1;
            end if;
        end if;
    end process Ball_Update;


    BallPosition:process(update,reset,PS)
    begin
        if reset='0' then
            life<=b"0101";
            ball_vPos<=b"001111010100";
            ball_hPos <= b"0010011111100";
        elsif rising_edge(update)then
            if vSpeed(speedLength-1) = '1' then
                ball_vPos <= to_unsigned(to_integer(ball_vPos) - to_integer(vSpeed(
                    speedLength-2 downto 0)), ball_vPos'length);
            else
                ball_vPos <= to_unsigned(to_integer(ball_vPos) + to_integer(vSpeed(
                    speedLength-2 downto 0)), ball_vPos'length);
            end if;

            if hSpeed(speedLength-1) = '1' then
                ball_hPos <= to_unsigned(to_integer(ball_hPos) - to_integer(hSpeed(
                    speedLength-2 downto 0)), ball_hPos'length);
            else
```

```
                    ball_hPos <= to_unsigned(to_integer(ball_hPos) + to_integer(hSpeed(
                        speedLength-2 downto 0)), ball_hPos'length);
                end if;

                if BCv>485 then
                    life<=life-1;
                    die<='1';
                    ball_vPos<=b"001111010100";
                    ball_hPos <= b"0010011111100";
                end if;

            end if;
            if PS=dead then
                die<='0';
            end if;


    end process BallPosition;

    GameState : process(clk,reset)
    begin
            if reset='0' then
                PS<=dead;
            elsif rising_edge(clk) then
                PS<=NS;
            end if;
    end process GameState;

    BallWait:process(PS,go,life,die)
    begin
            case PS is
                when dead =>
                    if go='0' and life /=b"0000" then
                        NS<=live;
                    else
                        NS<=dead;
                    end if;
                when live =>
                    if life=b"0000" or die='1'then
                        NS<=dead;
                    else
                        NS<=live;
                    end if;
            end case;

    end process BallWait;

    Paddle:process(update, reset)
    begin
            if reset = '0' then
                vSpeed <= "011";
                hSpeed <= "000";
                paddleHits <= (others => '0');
            elsif rising_edge(update) then
                if BLh = 0 then
                    hSpeed(speedLength-1) <= '0';
                    paddleHits <= paddleHits + 1;
                elsif BRh = 639 then
                    hSpeed(speedLength-1) <= '1';
                    paddleHits <= paddleHits + 1;

                elsif PaddleHit="10000" then
                    paddleHits <= paddleHits + 1;
                    vSpeed <= "101";
                    hSpeed <= "111";
```

```
                        elsif PaddleHit="01000" then
                                paddleHits <= paddleHits + 1;
                                vSpeed <= "110";
                                hSpeed <= "110";

                        elsif PaddleHit="00100" then
                                paddleHits <= paddleHits + 1;
                                vSpeed(speedLength-1) <= '1';

                        elsif PaddleHit="00010" then
                                paddleHits <= paddleHits + 1;
                                vSpeed <= "110";
                                hSpeed <= "010";

                        elsif PaddleHit="00001" then
                                paddleHits <= paddleHits + 1;
                                vSpeed <= "101";
                                hSpeed <= "011";

                        elsif WallHit='1' then
                                paddleHits <= paddleHits + 1;
                                if WallHitSide = "100000" then
                                        vSpeed(speedLength-1) <= '0';
                                elsif WallHitSide = "010000" then
                                        vSpeed(speedLength-1) <= '0';
                                        hSpeed(speedLength-1) <= '0';
                                elsif WallHitSide = "001000" then
                                        hSpeed(speedLength-1) <= '0';
                                elsif WallHitSide = "000100" then
                                        hSpeed(speedLength-1) <= '1';
                                elsif WallHitSide = "000010" then
                                        vSpeed(speedLength-1) <= '0';
                                        hSpeed(speedLength-1) <= '1';
                                elsif WallHitSide = "000001" then
                                        vSpeed(speedLength-1) <= '1';
                                end if;
                        elsif BCv>485 then
                                vSpeed <= "011";
                                hSpeed <= "000";
                                paddleHits <= (others => '0');
                        elsif BTv=0 then
                                vSpeed(speedLength-1) <= '0';
                                paddleHits <= paddleHits + 1;
                        end if;
                end if;
        end process Paddle;

        SIDESOUND : process(update)
        begin
                if rising_edge(update) then
                        if BLh = 0 OR BRh = 639 OR BTv=0 then
                                side_sound <= '1';
                        else
                                side_sound <= '0';
                        end if;
                end if;
        end process SIDESOUND;

BBh<=BCh;
BBv<=BCv+5;
BTh<=BCh;
BTv<=BCv-5;
BLh<=BCh-5;
BLv<=BCv;
BRh<=BCh+5;
BRv<=BCv;
```

```
lives<=life;
die_sound<=die;
BC_V<=BCv;
BC_H<=BCh;
BallClk<=update;


end architecture RTL;
```

6.2 Bricks Module

Listing 2: Bricks.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity Bricks is

       port(
              clk : in std_logic;
              reset : in std_logic;
              hPos : in unsigned(10 downto 0);
       vPos : in unsigned(9 downto 0);
              brick_status : out std_logic_vector(1 downto 0);
              BCh : in unsigned(10 downto 0);
              BCv : in unsigned(9 downto 0);
              hit : out std_logic;
              hit_side : out std_logic_vector(5 downto 0);
              ball_update_clk : in std_logic;
              game_over : in std_logic
       );

end entity Bricks;


architecture rtl of Bricks is

       type BRICK_GRID is array (0 to 40, 0 to 30) of std_logic;

       signal grid : BRICK_GRID := (others => (others => '1'));

       signal count : integer := 0;
       signal hToAdd : integer := 0;
       signal vToAdd : integer := 0;

       signal BBh :unsigned (10 downto 0);
       signal BBv :unsigned (9 downto 0);
       signal BLh :unsigned (10 downto 0);
       signal BLv :unsigned (9 downto 0);
       signal BRh :unsigned (10 downto 0);
       signal BRv :unsigned (9 downto 0);
       signal BTh :unsigned (10 downto 0);
       signal BTv :unsigned (9 downto 0);
       signal BTCv :unsigned (9 downto 0);

       signal next_hit : std_logic := '0';
       signal next_hit_side : std_logic_vector(5 downto 0) := "000000";

begin

       process(hPos, vPos, grid)

              variable hIndex : integer := 0;
              variable hPosition : unsigned(10 downto 0);
              variable vIndex : integer := 0;

       begin

              vIndex := to_integer(vPos srl 3);
              if vPos(3) = '1' then
                     -- Shift the odd rows over
                     hIndex := to_integer((hPos + 8) srl 4);
                     hPosition := hPos + 8;
```

```
        else
                hIndex := to_integer(hPos srl 4);
                hPosition := hPos;
        end if;

        if vPos >= 239 then
                brick_status <= "00";
        else

                if (vPos AND "111") /= "111" AND (hPosition AND "1111") /= "1111" then
                        -- Brick
                        if grid(hIndex, vIndex) = '1' then
                                brick_status <= "01";
                        else
                                brick_status <= "00";
                        end if;
                elsif (vPos AND "111") /= "111" then
                        -- Vertical Mortar
                        if grid(hIndex, vIndex) = '1' AND grid(hIndex + 1, vIndex) = '1' then
                                brick_status <= "10";
                        else
                                brick_status <= "00";
                        end if;
                else
                        -- Horizontal Mortar
                        if vPos(3) = '0' then
                                if grid(hIndex, vIndex) = '1' AND grid(to_integer((hPos + 9) srl 4),
                                    vIndex + 1) = '1' then
                                        brick_status <= "10";
                                else
                                        brick_status <= "00";
                                end if;
                        else
                                if grid(hIndex, vIndex) = '1' AND grid(to_integer((hPos + 1) srl 4),
                                    vIndex + 1) = '1' then
                                        brick_status <= "10";
                                else
                                        brick_status <= "00";
                                end if;
                        end if;
                end if;

        end if;

end process;

process(reset, game_over, ball_update_clk)

        variable bTopIndexH : integer := 0;
        variable bTopIndexV : integer := 0;
        variable bTopCornerV : integer := 0;
        variable bTopRightIndexH : integer := 0;
        variable bTopLeftIndexH : integer := 0;
        variable bBottomIndexH : integer := 0;
        variable bBottomIndexV : integer := 0;
        variable bLeftIndexH : integer := 0;
        variable bLeftIndexV : integer := 0;
        variable bRightIndexH : integer := 0;
        variable bRightIndexV : integer := 0;

begin

        if reset = '0' then
                grid <= (others => (others => '1'));
        else
                bTopIndexV := to_integer(BTv srl 3);
```

```
                        if BTv(3) = '1' then
                                -- Shift the odd rows over
                                bTopIndexH := to_integer((BTh + 8) srl 4);
                        else
                                bTopIndexH := to_integer(BTh srl 4);
                        end if;

                        bTopCornerV := to_integer(BTCv srl 3);
                        if BTCv(3) = '1' then
                                -- Shift the odd rows over
                                bTopRightIndexH := to_integer((BRh + 5) srl 4);
                                bTopLeftIndexH := to_integer((BLh + 11) srl 4);
                        else
                                bTopRightIndexH := to_integer((BRh - 3) srl 4);
                                bTopLeftIndexH := to_integer((BLh + 3) srl 4);
                        end if;

                        bBottomIndexV := to_integer(BBv srl 3);
                        if BBv(3) = '1' then
                                -- Shift the odd rows over
                                bBottomIndexH := to_integer((BBh + 8) srl 4);
                        else
                                bBottomIndexH := to_integer(BBh srl 4);
                        end if;

                        bLeftIndexV := to_integer(BLv srl 3);
                        if BLv(3) = '1' then
                                -- Shift the odd rows over
                                bLeftIndexH := to_integer((BLh + 8) srl 4);
                        else
                                bLeftIndexH := to_integer(BLh srl 4);
                        end if;

                        bRightIndexV := to_integer(BRv srl 3);
                        if BRv(3) = '1' then
                                -- Shift the odd rows over
                                bRightIndexH := to_integer((BRh + 8) srl 4);
                        else
                                bRightIndexH := to_integer(BRh srl 4);
                        end if;


                        if BTv <= 239 then
                                if grid(bTopIndexH, bTopIndexV) = '1' then
                                        next_hit <= '1';
                                        next_hit_side <= "100000";
                                elsif grid(bTopLeftIndexH, bTopCornerV) = '1' then
                                        next_hit <= '1';
                                        next_hit_side <= "010000";
                                elsif grid(bTopRightIndexH, bTopCornerV) = '1' then
                                        next_hit <= '1';
                                        next_hit_side <= "000010";
                                elsif BCv <= 235 AND grid(bLeftIndexH, bLeftIndexV) = '1' then
                                        next_hit <= '1';
                                        next_hit_side <= "001000";
                                elsif BCv <= 235 AND grid(bRightIndexH, bRightIndexV) = '1' then
                                        next_hit <= '1';
                                        next_hit_side <= "000100";
                                elsif BBv <= 230 AND grid(bBottomIndexH, bBottomIndexV) = '1' then
                                        next_hit <= '1';
                                        next_hit_side <= "000001";
                                else
                                        next_hit <= '0';
                                        next_hit_side <= "000000";
                                end if;
                        else
```

```
                                next_hit <= '0';
                                next_hit_side <= "000000";
                        end if;

                        if rising_edge(ball_update_clk) then
                                hit <= next_hit;
                                hit_side <= next_hit_side;

                                if next_hit = '1' then
                                        if next_hit_side = "100000" then
                                                grid(bTopIndexH, bTopIndexV) <= '0';
                                        elsif next_hit_side = "010000" then
                                                grid(bTopLeftIndexH, bTopCornerV) <= '0';
                                        elsif next_hit_side = "001000" then
                                                grid(bLeftIndexH, bLeftIndexV) <= '0';
                                        elsif next_hit_side = "000100" then
                                                grid(bRightIndexH, bRightIndexV) <= '0';
                                        elsif next_hit_side = "000010" then
                                                grid(bTopRightIndexH, bTopCornerV) <= '0';
                                        elsif next_hit_side = "000001" then
                                                grid(bBottomIndexH, bBottomIndexV) <= '0';
                                        end if;
                                end if;
                        end if;
                end if;

        end process;

        BBh<=BCh;
        BBv<=BCv+5;
        BTh<=BCh;
        BTv<=BCv-5;
        BTCv<=BCv-2;
        BLh<=BCh-5;
        BLv<=BCv;
        BRh<=BCh+5;
        BRv<=BCv;

end architecture rtl;
```

6.3 Paddle Module

Listing 3: paddle.vhd

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;



entity paddle is
   generic(
      PaddleUpdate : integer


   );
   port(
         clk : in std_logic;
         hPos : in unsigned(10 downto 0);
         vPos : in unsigned(9 downto 0);
         paddle_status : out std_logic;
         ADC_in :in std_logic_vector(11 downto 0);
                  Pos_out :out std_logic_vector(11 downto 0);
                  BCv :in unsigned(9 downto 0);
                  BCh :in unsigned(10 downto 0);
                  PaddleHit :out std_logic_vector(4 downto 0);
                  BallUpdateClk :in std_logic

   );
end entity paddle;


architecture RTL of paddle is

   signal PC : unsigned(10 downto 0) :=b"00101000000"; --Center of paddle Horizontal position
   signal PL : unsigned(10 downto 0); --Upper left corner of paddle position
   signal DX : unsigned(10 downto 0);
   signal nPC : unsigned(10 downto 0);
   signal PR : unsigned(10 downto 0); --Upper right corner of paddle position
   signal count :integer :=0;
   signal update :std_logic :='0';
      signal ADC_Average : integer;
      signal ADC_count : integer :=0;
      signal ADC_in2 : unsigned(10 downto 0) :=b"00101000000";
      signal ADC_val : integer;
      signal BBh :unsigned (10 downto 0);
      signal BBv :unsigned (9 downto 0);
      signal BLh :unsigned (10 downto 0);
      signal BLv :unsigned (9 downto 0);
      signal BRh :unsigned (10 downto 0);
      signal BRv :unsigned (9 downto 0);
      signal BTh :unsigned (10 downto 0);
      signal BTv :unsigned (9 downto 0);

      component averageADC is
      port(
      data_in :in std_logic_vector (11 downto 0);
      data_out :out unsigned (10 downto 0);
      clk :in std_logic
      );
      end component averageADC;

   begin

      avg1 : averageADC
      port map(
             data_in=>ADC_in,
             data_out=>PC,
             clk=>clk



      );



  -- Determine if pixel is part of paddle or not
  P_status:process(hPos,vPos)
  begin
      -- check for vertical position to be one of last 5 lines
      if(vPos>474) then
          if hPos>=PL and hPos<=PR then
             -- Pixel is part of paddle between left and right edges
             paddle_status<='1';
          else
             paddle_status<='0';
          end if;
      else
          paddle_status<='0';
      end if;


  end process P_status;


      HitStatus:process(BallUpdateClk)
      begin
                  if rising_edge(BallUpdateClk) then
                          if (BBv>474 AND BBV<485) then
                                  if(BBh<(PR+5) and (BBh>(PL-5) or PL<5) ) then
                                          if BBH < PC - 20 then
                                                 PaddleHit <= "10000";
                                          elsif BBH<PC-5 then
                                                 PaddleHit<="01000";
                                          elsif BBH > PC + 20 then
                                                 PaddleHit <= "00001";
                                          elsif BBH>PC+5 then
                                                 PaddleHit <="00010";
                                          else
                                                 PaddleHit<="00100";
                                          end if;
                                  else
                                          PaddleHit<="00000";
                                  end if;
                          else
                                  PaddleHit<="00000";
                          end if;
                  end if;
      end process HitStatus;

      BBh<=BCh;
      BBv<=BCv+5;
      BTh<=BCh;
      BTv<=BCv-5;
      BLh<=BCh-5;
      BLv<=BCv;
      BRh<=BCh+5;
      BRv<=BCv;
  PL<=PC-20;
  PR<=PC+20;
      Pos_out(10 downto 0)<=std_logic_vector(PC(10 downto 0));
      Pos_out(11)<='0';
```

```
end architecture RTL;
```

6.4 Pixel Status Module

Listing 4: status.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity Status is

       port(
              ball_status : in std_logic_vector(3 downto 0);
              paddle_status : in std_logic;
              brick_status : in std_logic_vector(1 downto 0);
       status : out std_logic_vector(6 downto 0) := (others => '0');
               gameOver :in std_logic
       );


end entity Status;


architecture rtl of Status is


begin
       with gameOver select
              status <= brick_status & ball_status & paddle_status when '0',
                                                                                                                   "
                                                                                                                       0000C
                                                                                                                       "

                                                                                                                       when

                                                                                                                       '1';



end architecture rtl;
```

6.5 Sound Module

Listing 5: tone.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity Tone is

        port(
                clk : in std_logic;
                clk_audio : in std_logic;
                play_bounce_wall : in std_logic;
                play_bounce_brick : in std_logic;
                play_bounce_paddle : in std_logic;
                play_die : in std_logic;
                out_signal : out std_logic
        );

end entity Tone;


architecture rtl of Tone is

        type AUDIO_TONE is (NONE, BWALL, BBRICK, BPADDLE, DIE);
        signal current_tone, next_tone : AUDIO_TONE := NONE;
        signal current_count, next_count, target_count : integer := 0;
        signal current_tone_length, next_tone_length, target_tone_length : integer := 0;
        signal current_state, next_state : std_logic := '0';

begin

        out_signal <= current_state;

        process(clk)
        begin
                if rising_edge(clk) then
                        current_tone <= next_tone;
                end if;
        end process;

        process(clk_audio)
        begin
                if rising_edge(clk_audio) then

                        current_count <= next_count;
                        current_tone_length <= next_tone_length;
                        current_state <= next_state;

                end if;
        end process;

        process(current_tone, current_count, current_tone_length, current_state, play_bounce_wall,
             play_bounce_brick, play_bounce_paddle, play_die, target_tone_length, target_count)
        begin

                case current_tone is

                        when NONE =>
                                if play_bounce_wall = '1' then
                                        next_tone <= BWALL;
                                elsif play_bounce_brick = '1' then
                                        next_tone <= BBRICK;
                                elsif play_bounce_paddle = '1' then
```

```
                                    next_tone <= BPADDLE;
                              elsif play_die = '1' then
                                    next_tone <= DIE;
                              else
                                    next_tone <= NONE;
                              end if;

                              next_count <= 0;
                              next_tone_length <= 0;
                              next_state <= '0';

                        when others =>
                              if current_count = target_count then
                                    next_count <= 0;
                                    next_state <= not current_state;
                              else
                                    next_count <= current_count + 1;
                                    next_state <= current_state;
                              end if;

                              if current_tone_length = target_tone_length then
                                    next_tone_length <= 0;
                                    next_tone <= NONE;
                              else
                                    next_tone_length <= current_tone_length + 1;
                                    next_tone <= current_tone;
                              end if;
                  end case;
            end process;

            process(current_tone)
            begin
                  case current_tone is
                        when NONE =>
                              target_count <= 0;
                              target_tone_length <= 0;

                        when BWALL =>
                              target_count <= 4;
                              target_tone_length <= 175;

                        when BBRICK =>
                              target_count <= 5;
                              target_tone_length <= 175;

                        when BPADDLE =>
                              target_count <= 8;
                              target_tone_length <= 175;

                        when DIE =>
                              target_count <= 10;
                              target_tone_length <= 1000;
                  end case;


            end process;

      end architecture rtl;
```

6.6 VGA display Module

Listing 6: vga.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity VGA is

	port(
		clk : in std_logic;
		reset : in std_logic;
	VGA_B : out std_logic_vector(3 downto 0) := (others => '0');
	VGA_G : out std_logic_vector(3 downto 0) := (others => '0');
	VGA_HS : out std_logic      := '0';
	VGA_R : out std_logic_vector(3 downto 0) := (others => '0');
	VGA_VS : out std_logic      := '0';
	hPos : out unsigned(10 downto 0);
	vPos : out unsigned(9 downto 0);
	status : in std_logic_vector(6 downto 0) := (others => '0')
	);


end entity VGA;


architecture rtl of VGA is

	type VStateType is (V_FRONT_PORCH, V_SYNC, V_BACK_PORCH, DATA);
	type HStateType is (H_FRONT_PORCH, H_SYNC, H_BACK_PORCH, DATA);

	signal currentStateV, nextStateV : VStateType;
	signal currentStateH, nextStateH : HStateType;

	constant V_FRONT_PIXELS : integer := 8000;
	constant V_SYNC_PIXELS : integer := 1600;
	constant V_BACK_PIXELS : integer := 26400;
	constant V_LINES : integer := 480;

	constant H_FRONT_PIXELS : integer := 16;
	constant H_SYNC_PIXELS : integer := 96;
	constant H_BACK_PIXELS : integer := 48;
	constant H_PIXELS : integer := 640;

	signal pixelCount : integer := 0;
	signal nextPixelCount : integer := 0;
	signal pixelCountV : integer := 0;
	signal nextPixelCountV : integer := 0;
	signal lineCount : integer := 0;
	--signal nextLineCount : integer := 0;


begin

	hPos <= to_unsigned(nextPixelCount, hPos'length);
	vPos <= to_unsigned(lineCount, vPos'length);

	commit : process(clk)
	begin

		if rising_edge(clk) then

			currentStateV <= nextStateV;
			currentStateH <= nextStateH;
```

```
pixelCount <= nextPixelCount;
pixelCountV <= nextPixelCountV;

if currentStateH = DATA AND currentStateV = DATA then
        -- Paddle
        if status(0) = '1' then
                VGA_B <= "0011";
                VGA_G <= "0101";
                VGA_R <= "1000";
        -- Ball
        elsif status(1) = '1' then
                VGA_B <= "1111";
                VGA_G <= "1111";
                VGA_R <= "1111";
        elsif status(2) = '1' then
                VGA_B <= "1010";
                VGA_G <= "1010";
                VGA_R <= "1010";
        elsif status(3) = '1' then
                VGA_B <= "1000";
                VGA_G <= "1000";
                VGA_R <= "1000";
        elsif status(4) = '1' then
                VGA_B <= "0111";
                VGA_G <= "0111";
                VGA_R <= "0111";
        -- Brick
        elsif status(5) = '1' then
                VGA_B <= "0000";
                VGA_G <= "0000";
                VGA_R <= "1111";
        -- Mortar
        elsif status(6) = '1' then
                VGA_B <= "1111";
                VGA_G <= "1111";
                VGA_R <= "1111";
        -- Background
        else
                VGA_B <= "0000";
                VGA_G <= "0000";
                VGA_R <= "0000";
        end if;


else
        VGA_B <= (others => '0');
        VGA_G <= (others => '0');
        VGA_R <= (others => '0');
end if;

if currentStateV = V_SYNC then
        VGA_VS <= '0';
else
        VGA_VS <= '1';
end if;

if currentStateH = H_SYNC then
        VGA_HS <= '0';
else
        VGA_HS <= '1';
end if;

if currentStateH = DATA AND nextStateH = H_FRONT_PORCH then
        if currentStateV = DATA then
                if lineCount < V_LINES then
                        lineCount <= lineCount + 1;
```

```
                                else
                                        lineCount <= 0;
                                end if;
                        else
                                lineCount <= 0;
                        end if;
                end if;

        end if;

end process;

state : process(currentStateV, currentStateH, pixelCount, pixelCountV, lineCount)
begin

        case currentStateV is

                when V_FRONT_PORCH =>
                        if pixelCountV < V_FRONT_PIXELS then
                                nextPixelCountV <= pixelCountV + 1;
                                nextStateV <= V_FRONT_PORCH;
                        else
                                nextStateV <= V_SYNC;
                                nextPixelCountV <= 0;
                        end if;

                when V_SYNC =>
                        if pixelCountV < V_SYNC_PIXELS then
                                nextPixelCountV <= pixelCountV + 1;
                                nextStateV <= V_SYNC;
                        else
                                nextStateV <= V_BACK_PORCH;
                                nextPixelCountV <= 0;
                        end if;

                when V_BACK_PORCH =>
                        if pixelCountV < V_BACK_PIXELS then
                                nextPixelCountV <= pixelCountV + 1;
                                nextStateV <= V_BACK_PORCH;
                        else
                                nextStateV <= DATA;
                                nextPixelCountV <= 0;
                        end if;

                when DATA =>
                        if lineCount < V_LINES then
                                nextStateV <= DATA;
                        else
                                nextStateV <= V_FRONT_PORCH;
                        end if;

        end case;

        case currentStateH is

                when H_FRONT_PORCH =>
                        if pixelCount < H_FRONT_PIXELS then
                                nextPixelCount <= pixelCount + 1;
                                nextStateH <= H_FRONT_PORCH;
                        else
                                nextStateH <= H_SYNC;
                                nextPixelCount <= 0;
                        end if;

                when H_SYNC =>
                        if pixelCount < H_SYNC_PIXELS then
```

```
                                        nextPixelCount <= pixelCount + 1;
                                        nextStateH <= H_SYNC;
                                else
                                        nextStateH <= H_BACK_PORCH;
                                        nextPixelCount <= 0;
                                end if;

                        when H_BACK_PORCH =>
                                if pixelCount < H_BACK_PIXELS then
                                        nextPixelCount <= pixelCount + 1;
                                        nextStateH <= H_BACK_PORCH;
                                else
                                        nextStateH <= DATA;
                                        nextPixelCount <= 0;
                                end if;

                        when DATA =>
                                if pixelCount < H_PIXELS then
                                        nextPixelCount <= pixelCount + 1;
                                        nextStateH <= DATA;
                                else
                                        nextPixelCount <= 0;
                                        nextStateH <= H_FRONT_PORCH;
                                end if;

                end case;

        end process;

end architecture rtl;
```

6.7 Seven Segment display Module

Listing 7: sevenSeg.vhd

```
------------------------------------------------------------------------
-- Seven Segment Driver for De-lite 10
--
-- Nathan Tipton Copyright 2018
--
--
-- inputs:
-- num_in - single hex digit of desired value
-- dec - 0 : no decimal point, 1 : decimal point
--
-- outputs:
-- seg_out - mapping for value on single seven segment display
--
--
--
--
--
-------------------------------------------------------------------------
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sevenSeg is
       port(
              clk  :in std_logic;
              num_in :in std_logic_vector(3 downto 0);
              seg_out :out std_logic_vector(7 downto 0);
              dec  :in std_logic

       );
       end sevenSeg;


architecture RTL of sevenSeg is
       type seven_Seg is array (0 to 15) of std_logic_vector(7 downto 0);
       constant sevSeg: seven_Seg := (X"C0", X"F9", X"A4",X"B0",X"99",X"92",X"82",X"F8",X"80",X"98",x"88"
            ,x"83",x"C6",x"A1",x"86",x"8E");
       type sevenSeg2 is array (0 to 15) of std_logic_vector(7 downto 0);
       constant sevSegDec: sevenSeg2 := (X"40",X"79",X"24",X"30",X"19",X"12",X"02",X"78",X"00",X"18",x"08
            ",x"03",x"46",x"21",x"06",x"0E");
       begin

       seven :process(clk)
       begin
              if rising_edge(clk) then
                     if dec='0' then
                            case num_in is
                                          when X"0" =>
                                          seg_out <= sevSeg(0);
                                   when X"1" =>
                                   seg_out <= sevSeg(1);
                                   when X"2" =>
                                   seg_out <= sevSeg(2);
                                   when X"3" =>
                                   seg_out <= sevSeg(3);
                                   when X"4" =>
                                   seg_out <= sevSeg(4);
                                   when X"5" =>
                                   seg_out <= sevSeg(5);
                                   when X"6" =>
                                   seg_out <= sevSeg(6);
                                   when X"7" =>
```

```
                                    seg_out <= sevSeg(7);
                                    when X"8" =>
                                    seg_out <= sevSeg(8);
                                    when X"9" =>
                                    seg_out <= sevSeg(9);
                                    when X"A" =>
                                    seg_out <= sevSeg(10);
                                    when X"B" =>
                                    seg_out <= sevSeg(11);
                                    when X"C" =>
                                    seg_out <= sevSeg(12);
                                    when X"D" =>
                                    seg_out <= sevSeg(13);
                                    when X"E" =>
                                    seg_out <= sevSeg(14);
                                    when X"F" =>
                                    seg_out <= sevSeg(15);
                                    when others =>
                                    seg_out <= x"FF";
                             end case;
                      else
                             case num_in is
                                    when X"0" =>
                                    seg_out <= sevSegDec(0);
                                    when X"1" =>
                                    seg_out <= sevSegDec(1);
                                    when X"2" =>
                                    seg_out <= sevSegDec(2);
                                    when X"3" =>
                                    seg_out <= sevSegDec(3);
                                    when X"4" =>
                                    seg_out <= sevSegDec(4);
                                    when X"5" =>
                                    seg_out <= sevSegDec(5);
                                    when X"6" =>
                                    seg_out <= sevSegDec(6);
                                    when X"7" =>
                                    seg_out <= sevSegDec(7);
                                    when X"8" =>
                                    seg_out <= sevSegDec(8);
                                    when X"9" =>
                                    seg_out <= sevSegDec(9);
                                    when X"A" =>
                                    seg_out <= sevSegDec(10);
                                    when X"B" =>
                                    seg_out <= sevSegDec(11);
                                    when X"C" =>
                                    seg_out <= sevSegDec(12);
                                    when X"D" =>
                                    seg_out <= sevSegDec(13);
                                    when X"E" =>
                                    seg_out <= sevSegDec(14);
                                    when X"F" =>
                                    seg_out <= sevSegDec(15);
                                    when others =>
                                    seg_out <= x"FF";
                             end case;
                      end if;
               end if;
        end process seven;
end architecture RTL;
```

6.8 Top Level Module

Listing 8: DE10.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity DE10 is

       port(
              ADC_CLK_10 : in std_logic;
              MAX10_CLK1_50 : in std_logic;
              KEY : in std_logic_vector(1 downto 0) := (others => '0');
       VGA_B : out std_logic_vector(3 downto 0) := (others => '0');
       VGA_G : out std_logic_vector(3 downto 0) := (others => '0');
       VGA_HS : out std_logic := '0';
       VGA_R : out std_logic_vector(3 downto 0) := (others => '0');
       VGA_VS : out std_logic := '0';
              HEX0 : out std_logic_vector(7 downto 0);
              HEX1 : out std_logic_vector(7 downto 0);
              HEX2 : out std_logic_vector(7 downto 0);
              HEX3 : out std_logic_vector(7 downto 0);
              HEX4 : out std_logic_vector(7 downto 0);
              HEX5 : out std_logic_vector(7 downto 0);
              ARDUINO_IO : out std_logic_vector(15 downto 0)
       );


end entity DE10;


architecture rtl of DE10 is

    component pll is
       port (
                     inclk0 : IN STD_LOGIC;
                     c0   : OUT STD_LOGIC
       );
    end component pll;

    component pll_audio is
       port (
                     inclk0 : IN STD_LOGIC;
                     c0   : OUT STD_LOGIC
       );
    end component pll_audio;

    component VGA is
       port (
          clk : in std_logic;
          reset : in std_logic;
          VGA_B : out std_logic_vector(3 downto 0) := (others => '0');
          VGA_G : out std_logic_vector(3 downto 0) := (others => '0');
          VGA_HS : out std_logic := '0';
          VGA_R : out std_logic_vector(3 downto 0) := (others => '0');
          VGA_VS : out std_logic := '0';
          hPos : out unsigned(10 downto 0);
          vPos : out unsigned(9 downto 0);
          status : in std_logic_vector(6 downto 0) := (others => '0')
       );
    end component VGA;

    component Status is
       port (
          ball_status : in std_logic_vector(3 downto 0);
          paddle_status : in std_logic;
          brick_status : in std_logic_vector(1 downto 0);
          status : out std_logic_vector(6 downto 0) := (others => '0');
                            gameOver :in std_logic
       );
    end component Status;


       component sevenSeg is
              port (
                      clk  :in std_logic;
                  num_in :in std_logic_vector(3 downto 0);
                  seg_out :out std_logic_vector(7 downto 0);
                  dec  :in std_logic
              );
       end component sevenSeg;

    component ADC_COMM is
       port (
          clk : in std_logic;
          reset : in std_logic;
          adc_value : out std_logic_vector(11 downto 0)
       );
    end component ADC_COMM;

       component paddle is
              generic (
                     PaddleUpdate : integer
              );
              port (
                     clk : in std_logic;
                     hPos : in unsigned(10 downto 0);
                     vPos : in unsigned(9 downto 0);
                     paddle_status : out std_logic;
                     ADC_in :in std_logic_vector(11 downto 0);
                     Pos_out :out std_logic_vector(11 downto 0);
                     BCv :in unsigned(9 downto 0);
                     BCh :in unsigned(10 downto 0);
                     PaddleHit :out std_logic_vector(4 downto 0);
                     BallUpdateClk :in std_logic

              );
       end component paddle;

    component Bricks is
       port (
          clk : in std_logic;
          reset : in std_logic;
          hPos : in unsigned(10 downto 0);
          vPos : in unsigned(9 downto 0);
          brick_status : out std_logic_vector(1 downto 0);
          BCh : in unsigned(10 downto 0);
          BCv : in unsigned(9 downto 0);
          hit : out std_logic;
          hit_side : out std_logic_vector(5 downto 0);
          ball_update_clk : in std_logic;
          game_over  : in std_logic
       );
    end component Bricks;

       component Ball is
              generic(
                     BallUpdate :integer
```

```
              );
              port(
                     clk :in std_logic;
                     hPos : in unsigned(10 downto 0);
                     vPos : in unsigned(9 downto 0);
                     ball_status : out std_logic_vector(3 downto 0);
                     lives :out unsigned(3 downto 0);
                     reset :in std_logic;
                     go :in std_logic;
                     die_sound :out std_logic;
                     side_sound : out std_logic;
                     BC_V :out unsigned(9 downto 0);
                     BC_H :out unsigned(10 downto 0);
                     PaddleHit :in std_logic_vector(4 downto 0);
                     WallHit :in std_logic;
                     WallHitSide :in std_logic_vector(5 downto 0);
                     BallClk :out std_logic

              );
       end component Ball;

    component Tone is
       port (
          clk    : in std_logic;
                     clk_audio   : in std_logic;
          play_bounce_wall : in std_logic;
          play_bounce_brick : in std_logic;
          play_bounce_paddle : in std_logic;
          play_die : in std_logic;
          out_signal : out std_logic
       );
    end component Tone;

    signal clockVGA : std_logic := '0';
    signal hPos : unsigned(10 downto 0);
    signal vPos : unsigned(9 downto 0);
        signal BV :unsigned(9 downto 0);
        signal BH :unsigned(10 downto 0);
    signal pixel_status : std_logic_vector(6 downto 0);
       signal livesNum :unsigned(3 downto 0):="0101";
       signal ADC1 :std_logic_vector(3 downto 0);
       signal ADC2 :std_logic_vector(3 downto 0);
       signal ADC3 :std_logic_vector(3 downto 0);
       signal ADC_reset :std_logic;
       signal ADC_DATA :std_logic_vector (11 downto 0);
       signal Paddle_Pos :std_logic_vector(11 downto 0);
    signal ball_status : std_logic_vector(3 downto 0) := (others => '0');
    signal paddle_status : std_logic := '0';
    signal brick_status : std_logic_vector(1 downto 0) := (others => '0');
       signal PaddleBallHit :std_logic_vector(4 downto 0);
    signal audio_clk : std_logic := '0';
    signal audio_signal : std_logic := '0';
       signal play_bounce_wall_sound : std_logic := '0';
    signal play_bounce_brick_sound : std_logic := '0';
    signal play_bounce_paddle_sound : std_logic := '0';
    signal play_die_sound : std_logic := '0';

    signal wall_hit : std_logic := '0';
    signal wall_hit_side : std_logic_vector(5 downto 0) := "000000";
    signal ball_update_clk : std_logic := '0';
    signal game_over : std_logic := '0';


begin

       pll1 : pll
```

```
              port map (
                     inclk0 => MAX10_CLK1_50,
                     c0 => clockVGA
              );

       pll2 : pll_audio
              port map (
                     inclk0 => MAX10_CLK1_50,
                     c0 => audio_clk
              );

    VGA_1 : VGA
       port map (
          clk => clockVGA,
          reset => KEY(0),
          VGA_B => VGA_B,
          VGA_G => VGA_G,
          VGA_HS => VGA_HS,
          VGA_R => VGA_R,
          VGA_VS => VGA_VS,
          hPos => hPos,
          vPos => vPos,
          status => pixel_status
       );

    Status_1 : Status
       port map (
          ball_status => ball_status,
          paddle_status => paddle_status,
          brick_status => brick_status,
          status => pixel_status,
                           gameOver=>game_over

       );

       lives: sevenSeg
              port map(
                     clk => MAX10_CLK1_50,
                     num_in (3 downto 0) => std_logic_vector(livesNum(3 downto 0)),
                     seg_out(7 downto 0) =>HEX0(7 downto 0),
                     dec => '0'
              );

       paddle_1 : paddle
              generic map(
                     PaddleUpdate => 25000000

              )
              port map(
                     clk =>MAX10_CLK1_50,
                     hPos => hPos,
                     vPos => vPos,
                     paddle_status => paddle_status,
                     ADC_in => ADC_DATA,
                     Pos_out =>Paddle_Pos,
                     BCv=>BV,
                     BCh=>BH,
                     PaddleHit=>PaddleBallHit,
                     BallUpdateClk=>ball_update_clk

              );

       ADCcount1: sevenSeg
              port map(
                     clk => MAX10_CLK1_50,
                     num_in (3 downto 0) => ADC1(3 downto 0),
                     seg_out(7 downto 0) =>HEX3(7 downto 0),
                     dec => '0'
                 );

             ADCcount2: sevenSeg
              port map(
                     clk => MAX10_CLK1_50,
                     num_in (3 downto 0) => ADC2(3 downto 0),
                     seg_out(7 downto 0) =>HEX4(7 downto 0),
                     dec => '0'
              );

             ADCcount3: sevenSeg
              port map(
                     clk => MAX10_CLK1_50,
                     num_in (3 downto 0) => ADC3(3 downto 0),
                     seg_out(7 downto 0) =>HEX5(7 downto 0),
                     dec => '0'
              );

   ADC_COMM_1 : ADC_COMM
             port map (
          clk => MAX10_CLK1_50,
          reset => ADC_reset,
          adc_value => ADC_DATA
             );

              Ball1 : Ball
              generic map(

              BallUpdate=>337500
              )
              port map(
                            clk=>MAX10_CLK1_50,
                            hPos=>hPos,
                            vPos=>vPos,
                            ball_status=>ball_status,
                            lives=>livesNum,
                            reset=>KEY(0),
                            go=>KEY(1),
                            die_sound=>play_die_sound,
                            side_sound=>play_bounce_wall_sound,
                            BC_V=>BV,
                            BC_H=>BH,
                            PaddleHit=>PaddleBallHit,
                            WallHit =>wall_hit,
                            WallHitSide =>wall_hit_side,
                            BallClk=>ball_update_clk



              );



              HEX1(7 downto 0)<=x"FF";
    HEX2(7 downto 0)<=x"FF";
             -- livesNum<=b"0101";
              ADC_reset<=KEY(0);
              ADC1(3 downto 0)<=Paddle_Pos(3 downto 0);
              ADC2(3 downto 0)<=Paddle_Pos(7 downto 4);
              ADC3(3 downto 0)<=Paddle_Pos(11 downto 8);

   Bricks_1 : Bricks
             port map (
          clk => clockVGA,
            reset    => KEY(0),
            hPos => hPos,
            vPos => vPos,
            brick_status => brick_status,
            BCh => BH,
            BCv => BV,
            hit => wall_hit,
            hit_side => wall_hit_side,
            ball_update_clk => ball_update_clk,
            game_over  => game_over
        );

    Tone_1 : Tone
        port map (
            clk => MAX10_CLK1_50,
            clk_audio  => audio_clk,
            play_bounce_wall => play_bounce_wall_sound,
            play_bounce_brick => wall_hit,
            play_bounce_paddle => PaddleBallHit(0) OR PaddleBallHit(1) OR PaddleBallHit(2) OR PaddleBallHit
                 (3) OR PaddleBallHit(4),
            play_die => play_die_sound,
            out_signal => audio_signal
        );

    ARDUINO_IO(7) <= audio_signal;

    with livesNum select
        game_over <= '1' when "0000",
                             '0' when others;


end architecture rtl;
```

6.9 Paddle Position Module

Listing 9: averageADC.vhd

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity averageADC is
       port(
       data_in :in std_logic_vector (11 downto 0);
       data_out :out unsigned (10 downto 0);
       clk :in std_logic




       );
end entity averageADC;

architecture RTL of averageADC is
       signal sum :unsigned (31 downto 0);
       signal count :integer :=0;
       signal ADCcount :integer;
       signal horiz :unsigned(31 downto 0);
       signal data :unsigned(10 downto 0);

begin
       ADCcount<=to_integer(unsigned(data_in(11 downto 5)));

-- process(clk)
-- begin
-- if ADCcount>=500 then
--  data_out<=to_unsigned(619,data_out'length);
-- elsif ADCcount<=20 then
--  data_out<=to_unsigned(20,data_out'length);
-- else
--  data_out<=to_unsigned(ADCcount,data_out'length);
-- end if;
--
-- end process;
-- process(clk)
-- begin
-- if rising_edge(clk) then
--  sum<=sum+unsigned(data_in);
--
--  if count=65536 then
--   sum<=shift_right(sum,16);
--   data_out<=sum(10 downto 0);
--   sum<=(others=>'0');
--   count<=0;
--  else
--   count<=count+1;
--
--  end if;
-- end if;
-- end process;
       horiz<=to_unsigned(638*ADCcount,horiz'length);
       sum<=shift_right(horiz,7);
       data<=sum(10 downto 0)+20;
       process(data)
       begin
       if data<620 then
              data_out<=data;
       else
              data_out<=to_unsigned(620,data_out'length);
       end if;
       end process;


-- data_out<=to_unsigned(0+((600-0)/(255-0))*(ADCcount-0)+20,data_out'length);
-- data_out<=to_unsigned(to_integer(shift_right(x"258",12))*ADCcount+20,data_out'length);
-- data_out<=to_unsigned((0.14652*ADCcount)+20,data_out'length);

end architecture RTL;
```

6.10 ADC Module

Listing 10: adc.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity ADC_COMM is

       port(
              clk : in std_logic;
              reset : in std_logic;
       adc_value : out std_logic_vector(11 downto 0)
       );

end entity ADC_COMM;

architecture behavioral of ADC_COMM is

    component adc is
       port (
           clk_clk : in std_logic := 'X'; -- clk
           clk_out_clk : out std_logic; -- clk
           adc_command_valid : in std_logic := 'X'; -- valid
           adc_command_channel : in std_logic_vector(4 downto 0) := (others => 'X'); -- channel
           adc_command_startofpacket : in std_logic := 'X'; -- startofpacket
           adc_command_endofpacket : in std_logic := 'X'; -- endofpacket
           adc_command_ready : out std_logic; -- ready
           adc_response_valid : out std_logic; -- valid
           adc_response_channel : out std_logic_vector(4 downto 0); -- channel
           adc_response_data : out std_logic_vector(11 downto 0); -- data
           adc_response_startofpacket : out std_logic; -- startofpacket
           adc_response_endofpacket : out std_logic; -- endofpacket
           reset_reset_n : in std_logic := 'X' -- reset_n
       );
    end component adc;

    signal adc_data : std_logic_vector(11 downto 0) := (others => '0');
    signal adc_valid : std_logic := '0';
    signal adc_sop : std_logic := '0';
    signal adc_eop : std_logic := '0';
    signal clk_10M : std_logic := '0';
    signal resp_sop : std_logic := '0';
    signal resp_eop : std_logic := '0';
    signal resp_valid : std_logic := '0';

    type STATE is (IDLE, REQ, RD);
    signal current_state, next_state : STATE;
    signal count, next_count : integer;

begin


    adc_1 : adc
       port map (
           clk_clk => clk,
           clk_out_clk => clk_10M,
           adc_command_valid => adc_valid,
           adc_command_channel => "00001",
           adc_command_startofpacket => adc_sop,
           adc_command_endofpacket => adc_eop,
           adc_command_ready => open,
           adc_response_valid => resp_valid,
           adc_response_channel => open,
           adc_response_data => adc_data,
           adc_response_startofpacket => resp_sop,
           adc_response_endofpacket => resp_eop,
           reset_reset_n => reset
       );

   process(clk_10M, reset)
   begin

       if reset = '0' then
           current_state <= IDLE;
           count <= 0;

       elsif rising_edge(clk) then

           current_state <= next_state;
           count <= next_count;

           if current_state = RD then
               adc_value <= adc_data;
           end if;

           if current_state = REQ then
               adc_valid <= '1';
               adc_sop <= '1';
               adc_eop <= '1';
           else
               adc_valid <= '0';
               adc_sop <= '0';
               adc_eop <= '0';
           end if;

       end if;

   end process;


   process(current_state, count, resp_valid)
   begin

       case current_state is

           when IDLE =>
               if count < 499998 then
                  next_state <= IDLE;
                  next_count <= count + 1;
               else
                  next_state <= REQ;
                  next_count <= 0;
               end if;

           when REQ =>
               if resp_valid = '1' then
                  next_state <= RD;
                  next_count <= 0;
               else
                  next_state <= REQ;
               end if;

           when RD =>
               next_state <= IDLE;
               next_count <= 0;

       end case;
    end process;


end architecture behavioral;
```