

“Pong” the Game, Implemented on an Altera Flex EPF10K70CR240-4 FPGA

Imre Knausz

EE650 @ RIT
Dr. Eric Peskin
November 12, 2004

Introduction.....	2
Top Level.....	2
Sub Blocks.....	3
kb_main.....	3
keyboard.....	3
kb_control.....	3
Test.....	4
VGAsync.....	4
pongmain.....	4
Test.....	5
ball.....	5
Test.....	5
paddle.....	5
Test.....	5
paddle2.....	5
Test.....	6
Bug(s).....	6
VHDL Code.....	6
kb_main.....	6
Keyboard.....	7
kb_control.....	8
VGAsync.....	10
pongmain.....	11
ball.....	14
paddle.....	15
paddle2.....	17

Introduction

This project consists of creating an entirely synthesizable “pong” core that can be implemented on an Altera Flex EPF10k70cr240-4 FPGA. The game uses a VGA monitor as output and a PS/2 keyboard as input. The game is one player only, and that player plays against the core’s artificial intelligence. The object of the game is to keep the ball in play by using the “paddle” to keep the ball in play. The ball is out of bounds when it reaches either the left or right side of the screen.

It is possible to beat the computer because the ball actually moves faster than the paddles. When the player wins, the screen turns green and when the FPGA wins, the screen turns red.

Top Level

The top level of this project is a GDF file. It ties together the keyboard controller, game core and the VGA controller. Below is a screen capture of the GDF:

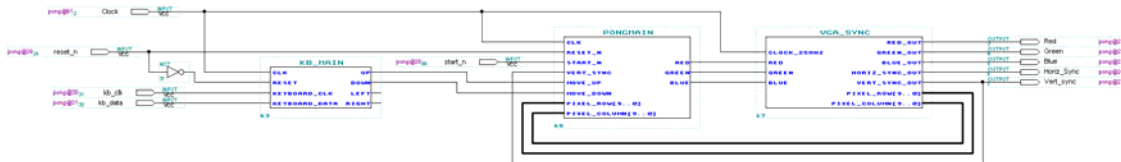


Figure 1: Top level of pong core.

This game was designed using a hierarchical approach which has several advantages. The game is easier to code when it is broken apart into fundamental functions, it is easier to understand and it is easier to debug. The chart below shows the hierarchical structure of the pong core:

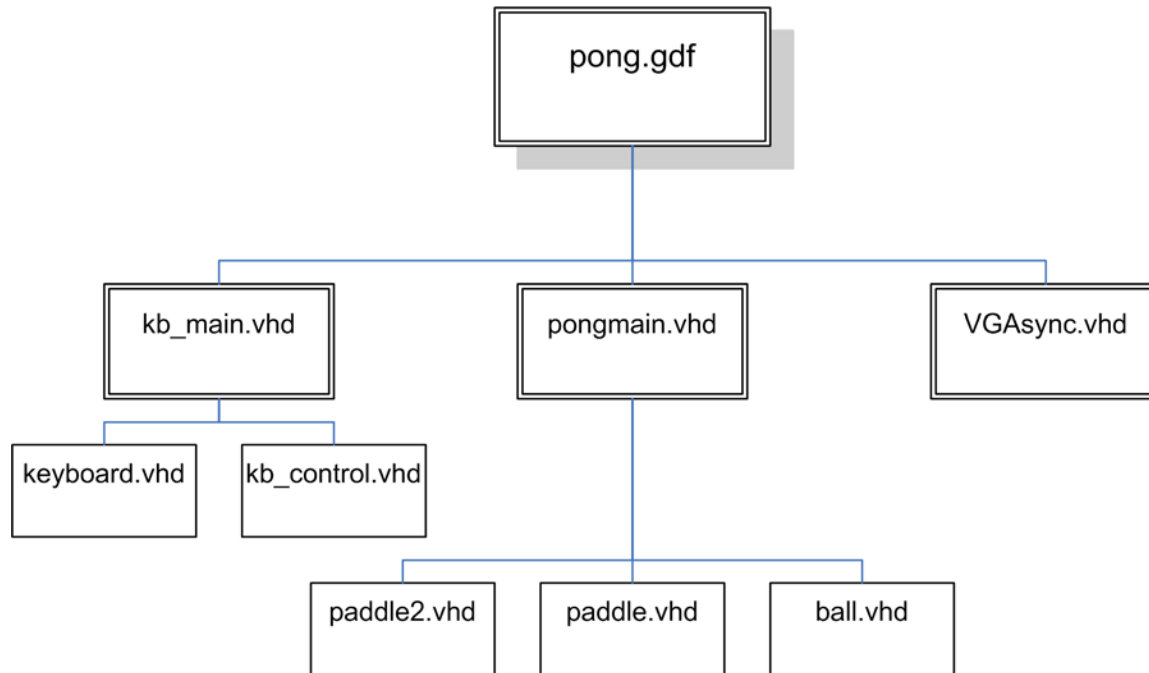


Figure 2: Hierarchical representation of pong core

The remaining part of this paper will describe each of these blocks in detail. The code will be listed at the end of the paper.

Sub Blocks

kb_main

There is no RTL in this block, it simply ties together the *keyboard* and *kb_control* blocks.

keyboard

This block interfaces directly with the data and clock pins on the PS/2 keyboard. This code was supplied by the UP2 manual.

kb_control

This is the keyboard controller block. It controls the *keyboard* block and reads the keyboard codes. It interprets keyboard code sequences and decides whether certain keys are depressed or not. It can detect four keys – up, down, left and right. Below is an ASM chart describing its operation:

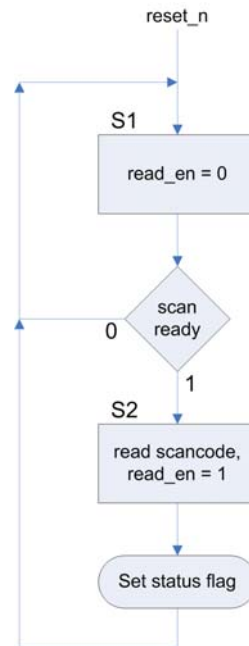


Figure 3: ASM chart for keyboard controller.

The status flag can have three values: E0 code, F0 code and key code. E0 and F0 codes designate a make or break and the key code designates which key was hit. A more efficient implementation might have been to use two separate state machines – one to interface with the *keyboard* block and another to keep track of scancodes.

Test

This block was tested visually. It was hooked up in a GDF test bench where its four outputs were tied to LEDs. The moment a key (up, down, left or right) was depressed, the associated LED would turn on and as soon as the key was let go, the LED would turn off.

The ideal way to test the block (and the other blocks for that matter) would have been to create a test bench and use some ASSERT statements to check whether things were working. Unfortunately, the Altera software does not support this essential feature of VHDL.

VGAsync

This block interfaces directly with the red, green, blue, hsync and vsync signals on the UP2 board. It generates all the timings required by a VGA monitor. This code was supplied by the UP2 manual.

pongmain

This is the main module for the pong core. It ties together the ball, paddle and paddle2 state machines. The state machine contained in this module controls the game start and end. Below is an ASM chart describing its operation:

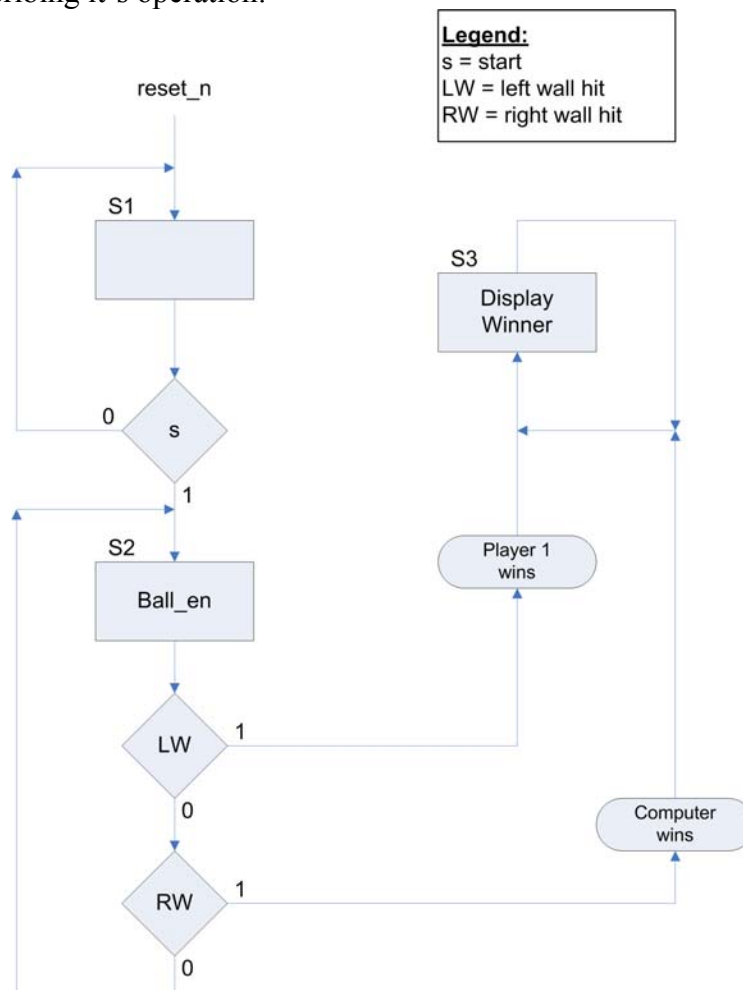


Figure 4: Main state machine ASM chart

Pongmain also OR's together the RGB outputs from the *ball*, *paddle* and *paddle2* blocks so that all of their outputs may be displayed simultaneously on the screen. Also, the *player1win* signal is OR'ed in with Green so that the screen turns green when the player has won. Similarly, *player2win* is OR'ed with Red so that a win by the computer is indicated.

Test

This block was tested using a simple test bench. Since it does not have many signals (once the *ball*, *paddle* and *paddle2* components are removed) it was possible to test it using the waveform editor in the MaxPlus2 software.

ball

The start of this code came from the ball example in the UP2 manual, however, it was heavily modified and doesn't look anything like the original code. The original code had several problems – the two most significant were the fact that it didn't compile because the CONV_STD_LOGIC_VECTOR functions were incorrectly used and it had WAIT statements which are not supposed to be used in synthesizable code (however, for some reason the Altera software accepts the use of WAIT).

Ball is a very simple single state state machine, so it's really not even a machine. It checks the *reset_n* signal to reset it's variables. Otherwise, it moves the ball based on the current direction *speedX* and *speedY* and checks for collisions with the walls and paddles.

Ball (and also the following blocks *paddle* and *paddle2*) has a separate PROCESS to “draw” the ball using RGB signals.

Test

This block was difficult to test. The philosophy for this block, *paddle* and *paddle2* was to test the individually and incrementally. The first version of the *ball* block only bounced in the Y direction. Once that was working, the other dimension was added. Then the *paddle* block was created and tested. Finally, I brought the Y position output of the *paddle* block into the *ball* block and wrote the section that detects the paddle collision.

paddle

This code is similar to the *ball* code, however the paddle moves only in one direction and based on user input from the keyboard.

Test

This ENTITY was tested by trial and error, since setting up a test bench with waveforms would have been too difficult. This block was designed and tested incrementally.

paddle2

This code is essentially the same as the *paddle* block with the exception of an extra PROCESS which is used to control the paddle. The *Track_ball* process checks the position and then moves the paddle according to the following rules: if the ball is “north” of the paddle then the *move_up* flag is set and if the ball is “south” of the paddle then the *move_down* flag is set. Then in the *Move_paddle* process, the paddle is moved according to the flags and whether or not the paddle is already at a screen boundary.

Test

Testing this block was straightforward since it is the same as the *paddle* block with the addition of the simple *Move_paddle* process.

Bug(s)

Unfortunately, the code does have at least one bug that is known. It is in the *ball* block and it's effect is that if you narrowly miss the ball with the paddle, i.e. you are moving after it, the ball will go behind the paddle, hit the wall and bounce back rather than tell the *pongmain* state machine that the wall was hit and the game should be over.

This bug can be fixed by detecting whether the ball is between the paddle's plane and the edge of the screen rather than just detecting whether the ball has broken the plane. The ball seems to be traveling too fast to be detected under some circumstances.

VHDL Code

kb_main

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY kb_main IS
  PORT(
    clk      : IN   STD_LOGIC;
    reset    : IN   STD_LOGIC;
    keyboard_clk, keyboard_data : IN  STD_LOGIC;
    up, down, left, right      : OUT  STD_LOGIC);
END kb_main ;

ARCHITECTURE a OF kb_main IS
  SIGNAL scancode      : STD_LOGIC_VECTOR(7 downto 0);
  SIGNAL scan_ready    : STD_LOGIC;
  SIGNAL read_en       : STD_LOGIC;

  COMPONENT kb_control
    PORT(
      clk      : IN   STD_LOGIC;
      scancode : IN   STD_LOGIC_VECTOR(7 downto 0);
      reset    : IN   STD_LOGIC;
      scan_rdy : IN   STD_LOGIC;
      read_en  : OUT  STD_LOGIC;
      up, down, left, right : OUT  STD_LOGIC);
  END COMPONENT kb_control;

  COMPONENT keyboard
    PORT( keyboard_clk, keyboard_data, clock_25Mhz,
          reset, read      : IN   STD_LOGIC;
          scan_code       : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0);
          scan_ready      : OUT  STD_LOGIC);
  END COMPONENT keyboard;

BEGIN
  kb_control_u1 : kb_control port map
  (
    clk      => clk,
    scancode => scancode,
    reset    => reset,
```

```

    scan_rdy => scan_ready,
    up      => up,
    down    => down,
    left    => left,
    right   => right,
    read_en => read_en
);

keyboard_u1 : keyboard port map
(
    keyboard_clk => keyboard_clk,
    keyboard_data=> keyboard_data,
    clock_25Mhz  => clk,
    reset        => reset,
    read         => read_en,
    scan_code    => scancode,
    scan_ready   => scan_ready
);

END a;

```

Keyboard

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY keyboard IS
    PORT( keyboard_clk, keyboard_data, clock_25Mhz ,
          reset, read      : IN  STD_LOGIC;
          scan_code       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
          scan_ready      : OUT STD_LOGIC);
END keyboard;

ARCHITECTURE a OF keyboard IS
    SIGNAL INCNT          : std_logic_vector(3 downto 0);
    SIGNAL SHIFTR         : std_logic_vector(8 downto 0);
    SIGNAL READ_CHAR      : std_logic;
    SIGNAL INFLAG, ready_set : std_logic;
    SIGNAL keyboard_clk_filtered : std_logic;
    SIGNAL filter          : std_logic_vector(7 downto 0);
BEGIN

PROCESS (read, ready_set)
BEGIN
    IF read = '1' THEN scan_ready <= '0';
    ELSIF ready_set'EVENT and ready_set = '1' THEN
        scan_ready <= '1';
    END IF;
END PROCESS;

--This process filters the raw clock signal coming from the keyboard using a shift
register and two AND gates
Clock_filter: PROCESS
BEGIN
    WAIT UNTIL clock_25Mhz'EVENT AND clock_25Mhz= '1';
    filter (6 DOWNTO 0) <= filter(7 DOWNTO 1) ;
    filter(7) <= keyboard_clk;
    IF filter = "11111111" THEN keyboard_clk_filtered <= '1';
    ELSIF filter= "00000000" THEN keyboard_clk_filtered <= '0';

```

```

    END IF;
END PROCESS Clock_filter;

--This process reads in serial data coming from the terminal
PROCESS
BEGIN
WAIT UNTIL (KEYBOARD_CLK_filtered'EVENT AND KEYBOARD_CLK_filtered='1');
IF RESET='1' THEN
    INCNT <= "0000";
    READ_CHAR <= '0';
ELSE
    IF KEYBOARD_DATA='0' AND READ_CHAR='0' THEN
        READ_CHAR<= '1';
        ready_set<= '0';
    ELSE
        -- Shift in next 8 data bits to assemble a scan code
        IF READ_CHAR = '1' THEN
            IF INCNT < "1001" THEN
                INCNT <= INCNT + 1;
                SHIFTIN(7 DOWNT0 0) <= SHIFTIN(8 DOWNT0 1);
                SHIFTIN(8) <= KEYBOARD_DATA;
            ready_set <= '0';
            -- End of scan code character, so set flags and exit loop
            ELSE
                scan_code <= SHIFTIN(7 DOWNT0 0);
                READ_CHAR <='0';
                ready_set <= '1';
                INCNT <= "0000";
            END IF;
        END IF;
    END IF;
END IF;
END PROCESS;
END a;

```

kb_control

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY kb_control IS
    PORT(
        clk      : IN   STD_LOGIC;
        scancode  : IN   STD_LOGIC_VECTOR(7 downto 0);
        reset    : IN   STD_LOGIC;
        scan_rdy  : IN   STD_LOGIC;
        up, down, left, right : OUT STD_LOGIC;
        read_en   : OUT  STD_LOGIC);
END kb_control ;

ARCHITECTURE a OF kb_control IS
    TYPE STATE_TYPE IS (idle, readcode);
    TYPE STATUS_TYPE IS (None, E0code, F0code);
    SIGNAL state : STATE_TYPE;
    SIGNAL status : STATUS_TYPE;
BEGIN
    PROCESS (clk)
    BEGIN
        IF reset = '1' THEN
            state <= idle;
            status <= None;

```

```

    up <= '0';
    down <= '0';
    left <= '0';
    right <= '0';
ELSIF (clk'EVENT AND clk='1') THEN
    CASE state IS
        WHEN idle =>
            read_en <= '0';
            IF (scan_rdy = '0') THEN
                state <= idle;
            ELSE
                state <= readcode;
            END IF;
        WHEN readcode =>
            read_en <= '1';
            CASE scancode IS
                WHEN "11100000" =>
                    if ((status=None) OR (status=E0code)) THEN
                        status <= E0code;
                    ELSE
                        status <= None;
                    END IF;
                WHEN "11110000" =>
                    if ((status=E0code) OR (status=F0code)) THEN
                        status <= F0code;
                    ELSE
                        status <= None;
                    END IF;
                WHEN OTHERS =>
                    if (status=E0code) THEN
                        CASE scancode IS
                            WHEN "01110101" => up <= '1';
                            WHEN "01110010" => down <= '1';
                            WHEN "01101011" => left <= '1';
                            WHEN "01110100" => right <= '1';
                            WHEN OTHERS => status <= None;
                        END CASE;
                    ELSIF (status=F0code) THEN
                        CASE scancode IS
                            WHEN "01110101" => up <= '0';
                            WHEN "01110010" => down <= '0';
                            WHEN "01101011" => left <= '0';
                            WHEN "01110100" => right <= '0';
                            WHEN OTHERS => status <= None;
                        END CASE;
                    ELSE
                        status <= None;
                    END IF;
                END CASE;
            WHEN OTHERS =>
                state <= idle;
            END CASE;
        END IF;
    END PROCESS;

END a;

```

VGAsync

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;

```

```

USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;

-- VGA Video Sync generation

ENTITY VGA_SYNC IS

    PORT(clock_25Mhz, red, green, blue      : IN  STD_LOGIC;
         red_out, green_out, blue_out,
         horiz_sync_out, vert_sync_out     : OUT STD_LOGIC;
         pixel_row, pixel_column          : OUT STD_LOGIC_VECTOR(9 DOWNT0 0));

END VGA_SYNC;

ARCHITECTURE a OF VGA_SYNC IS
    SIGNAL horiz_sync, vert_sync           : STD_LOGIC;
    SIGNAL video_on, video_on_v, video_on_h : STD_LOGIC;
    SIGNAL h_count, v_count                 : STD_LOGIC_VECTOR(9 DOWNT0 0);

BEGIN

    -- video_on is high only when RGB data is being displayed
    video_on <= video_on_H AND video_on_V;

--Generate Horizontal and Vertical Timing Signals for Video Signal

PROCESS
BEGIN
    WAIT UNTIL(clock_25Mhz'EVENT) AND (clock_25Mhz='1');

    -- H_count counts pixels (640 + extra time for sync signals)
    --
    -- Horiz_sync -----
    -- H_count      0          640          659          755          799
    --
    IF (h_count = 799) THEN
        h_count <= "0000000000";
    ELSE
        h_count <= h_count + 1;
    END IF;

    --Generate Horizontal Sync Signal using H_count
    IF (h_count <= 755) AND (h_count >= 659) THEN
        horiz_sync <= '0';
    ELSE
        horiz_sync <= '1';
    END IF;

    --V_count counts rows of pixels (480 + extra time for sync signals)
    --
    -- Vert_sync -----
    -- V_count      0          480          493-494          524
    --
    IF (v_count >= 524) AND (h_count >= 699) THEN
        v_count <= "0000000000";
    ELSIF (h_count = 699) THEN
        v_count <= v_count + 1;
    END IF;

    -- Generate Vertical Sync Signal using V_count
    IF (v_count <= 494) AND (v_count >= 493) THEN
        vert_sync <= '0';
    END IF;
END PROCESS

```

```

ELSE
    vert_sync <= '1';
END IF;

    -- Generate Video on Screen Signals for Pixel Data
IF (h_count <= 639) THEN
    video_on_h <= '1';
    pixel_column <= h_count;
ELSE
    video_on_h <= '0';
END IF;

IF (v_count <= 479) THEN
    video_on_v <= '1';
    pixel_row <= v_count;
ELSE
    video_on_v <= '0';
END IF;

    -- Put all video signals through DFFs to eliminate
    -- any logic delays that can cause a blurry image
red_out    <= red AND video_on;
green_out  <= green AND video_on;
blue_out   <= blue AND video_on;
horiz_sync_out <= horiz_sync;
vert_sync_out <= vert_sync;

END PROCESS;

END a;

```

pongmain

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY pongmain IS
    PORT(
        clk                : IN  STD_LOGIC;
        reset_n            : IN  STD_LOGIC;
        start_n            : IN  STD_LOGIC;
        vert_sync          : IN  STD_LOGIC;
        move_up, move_down : IN  STD_LOGIC;
        pixel_row, pixel_column : IN  STD_LOGIC_VECTOR(9 downto 0);
        Red, Green, Blue    : OUT  STD_LOGIC);
END pongmain ;

ARCHITECTURE a OF pongmain IS

    TYPE STATE_TYPE IS (S1, S2, S3);
    SIGNAL state      : STATE_TYPE;
    SIGNAL counter    : STD_LOGIC_VECTOR(1 downto 0);
    SIGNAL player1win, player2win : STD_LOGIC;
    SIGNAL backstop1, backstop2   : STD_LOGIC;
    SIGNAL Red_ball, Grn_ball, Blu_ball : STD_LOGIC;
    SIGNAL Red_pad, Grn_pad, Blu_pad : STD_LOGIC;
    SIGNAL Red_pad2, Grn_pad2, Blu_pad2 : STD_LOGIC;
    SIGNAL Paddle1_plane, Paddle2_plane : STD_LOGIC;
    SIGNAL ball_en : STD_LOGIC;
    SIGNAL Paddle_Y_pos, Paddle2_Y_pos : STD_LOGIC_VECTOR(9 downto 0);
    SIGNAL Ball_Y_pos_ext : STD_LOGIC_VECTOR(9 downto 0);

```

```

COMPONENT ball
  PORT(
    Red,Green,Blue           : OUT STD_LOGIC;
    Paddle1_plane, Paddle2_plane : OUT STD_LOGIC;
    Backstop1, Backstop2      : OUT STD_LOGIC;
    reset_n, ball_en, vert_sync : IN STD_LOGIC;
    pixel_row, pixel_column    : IN STD_LOGIC_VECTOR(9 downto 0);
    Paddle_Y_pos, Paddle2_Y_pos : IN STD_LOGIC_VECTOR(9 downto 0);
    Ball_Y_pos_ext            : OUT STD_LOGIC_VECTOR(9 downto 0));

END COMPONENT ball;

COMPONENT paddle
  PORT(
    Red,Green,Blue           : OUT STD_LOGIC;
    Paddle_Y_pos_out         : OUT STD_LOGIC_VECTOR(9 DOWNTO 0);
    reset_n, vert_sync       : IN STD_LOGIC;
    move_up, move_down       : IN STD_LOGIC;
    pixel_row, pixel_column  : IN STD_LOGIC_VECTOR(9 downto 0));
END COMPONENT paddle;

COMPONENT paddle2
  PORT(
    Red,Green,Blue           : OUT STD_LOGIC;
    Paddle_Y_pos_out         : OUT STD_LOGIC_VECTOR(9 DOWNTO 0);
    reset_n, vert_sync       : IN STD_LOGIC;
    ball_Y_pos               : IN STD_LOGIC_VECTOR(9 downto 0);
    pixel_row, pixel_column  : IN STD_LOGIC_VECTOR(9 downto 0));
END COMPONENT paddle2;

BEGIN

ball_u1 : ball port map
(
  Red           => Red_ball,
  Green         => Grn_ball,
  Blue          => Blu_ball,
  Paddle1_plane => Paddle1_plane,
  Paddle2_plane => Paddle2_plane,
  Backstop1     => backstop1,
  Backstop2     => backstop2,
  reset_n       => reset_n,
  ball_en       => ball_en,
  vert_sync     => vert_sync,
  pixel_row     => pixel_row,
  pixel_column  => pixel_column,
  Paddle_Y_pos => Paddle_Y_pos,
  Paddle2_Y_pos => Paddle2_Y_pos,
  Ball_Y_pos_ext => Ball_Y_pos_ext
);

paddle1_u1 : paddle port map
(
  Red           => Red_pad,
  Green         => Grn_pad,
  Blue          => Blu_pad,
  Paddle_Y_pos_out => Paddle_Y_pos,
  reset_n       => reset_n,
  vert_sync     => vert_sync,
  move_up       => move_up,
  move_down     => move_down,
  pixel_row     => pixel_row,
  pixel_column  => pixel_column

```

```

);

paddle2_u1 : paddle2 port map
(
  Red           => Red_pad2,
  Green         => Grn_pad2,
  Blue          => Blu_pad2,
  Paddle_Y_pos_out => Paddle2_Y_pos,
  reset_n       => reset_n,
  vert_sync     => vert_sync,
  ball_Y_pos    => Ball_Y_pos_ext,
  pixel_row     => pixel_row,
  pixel_column  => pixel_column
);

Red   <= Red_ball OR Red_pad OR Red_pad2 OR player2win;
Green <= Grn_ball OR Grn_pad OR Grn_pad2 OR player1win;
Blue  <= Blu_ball OR Blu_pad OR Blu_pad2;

PROCESS (clk)
BEGIN
  IF reset_n = '0' THEN
    state <= S1;
    player1win <= '0';
    player2win <= '0';
    ball_en <= '0';
  ELSIF (clk'EVENT AND clk='1') THEN
    CASE state IS
      WHEN S1 =>
        IF (start_n = '0') THEN
          state <= S2;
        ELSE
          state <= S1;
        END IF;
      WHEN S2 =>
        ball_en <= '1';
        IF backstop2 = '1' THEN
          state <= S3;
          player1win <= '1';
        ELSIF backstop1 = '1' THEN
          state <= S3;
          player2win <= '1';
        ELSE
          state <= S2;
        END IF;
      WHEN S3 =>
        ball_en <= '0';
        state <= S3;
      WHEN OTHERS =>
        state <= S1;
    END CASE;
  END IF;
END PROCESS;

END a;

```

ball

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;

```

```

LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY ball IS

    PORT(
        SIGNAL Red,Green,Blue           : OUT std_logic;
        SIGNAL Paddle1_plane, Paddle2_plane : OUT std_logic;
        SIGNAL Backstop1, Backstop2       : OUT std_logic;
        SIGNAL reset_n, ball_en, vert_sync : IN std_logic;
        SIGNAL pixel_row, pixel_column     : IN std_logic_vector(9 downto 0);
        SIGNAL Paddle_Y_pos, Paddle2_Y_pos : IN std_logic_vector(9 downto 0);
        SIGNAL Ball_Y_pos_ext              : OUT std_logic_vector(9 downto 0));

END ball;

architecture behavior of ball is

    SIGNAL Ball_on                : std_logic;
    SIGNAL Ball_Y_motion, Ball_X_motion : std_logic_vector(9 DOWNTO 0);
    SIGNAL Ball_Y_pos, Ball_X_pos   : std_logic_vector(9 DOWNTO 0);
    signal i1, i2                  : integer;
    CONSTANT Size: std_logic_vector(9 DOWNTO 0) := CONV_STD_LOGIC_VECTOR(4,10);
    CONSTANT Paddle_SizeY: std_logic_vector(9 DOWNTO 0) := CONV_STD_LOGIC_VECTOR(32,10);
    CONSTANT Plane_dist: std_logic_vector(9 DOWNTO 0) := CONV_STD_LOGIC_VECTOR(40,10);
    CONSTANT speedX: integer := 3;
    CONSTANT speedY: integer := 3;

BEGIN

    Red    <= Ball_on;
    Green  <= Ball_on;
    Blue   <= Ball_on;

    RGB_Display: Process (pixel_column, pixel_row)
    BEGIN
        -- Set Ball_on = '1' to display ball
        IF ('0' & Ball_X_pos <= pixel_column + Size) AND
            -- compare positive numbers only
            (Ball_X_pos + Size >= '0' & pixel_column) AND
            ('0' & Ball_Y_pos <= pixel_row + Size) AND
            (Ball_Y_pos + Size >= '0' & pixel_row ) THEN
            Ball_on <= '1';
        ELSE
            Ball_on <= '0';
        END IF;
    END process RGB_Display;

    Move_Ball: PROCESS (vert_sync)
    BEGIN
        -- Move ball once every vertical sync
        IF vert_sync = '1' THEN
            -- Bounce off top or bottom of screen
            IF reset_n = '0' Then
                Ball_X_pos <= CONV_STD_LOGIC_VECTOR(320, 10);
                Ball_Y_pos <= CONV_STD_LOGIC_VECTOR(240, 10);
                Paddle1_plane <= '0';
                Paddle2_plane <= '0';
                Backstop1 <= '0';
                Backstop2 <= '0';
                i1 <= -speedY;
            END IF;
        END IF;
    END process Move_Ball;

```

```

        i2 <= speedX;
    ELSIF ball_en = '1' THEN
        -- Take care of ball on top & bottom of screen
        IF '0' & ( Ball_Y_pos + CONV_STD_LOGIC_VECTOR(speedY,10) ) >=
CONV_STD_LOGIC_VECTOR(480,11) - Size THEN
            i1 <= -speedY;
        ELSIF '0' & Ball_Y_pos <= Size + CONV_STD_LOGIC_VECTOR(speedY,10) THEN
            i1 <= speedY;
        END IF;

        -- Take care of ball on right side of screen (+8 factor due to ball &
paddle size):
        -- Did we hit the wall?
        IF '0' & ( Ball_X_pos + CONV_STD_LOGIC_VECTOR(speedX,11) ) >=
CONV_STD_LOGIC_VECTOR(640,10) - Size THEN
            Backstop1 <= '1';
            -- did we hit paddle1?
            ELSIF '0' & ( Ball_X_pos + CONV_STD_LOGIC_VECTOR(speedX+8,11) ) >=
CONV_STD_LOGIC_VECTOR(640,10) - Plane_dist THEN
                IF ('0' & Paddle_Y_pos < Ball_Y_pos + Paddle_SizeY) AND (Paddle_Y_pos
+ Paddle_SizeY > '0' & Ball_Y_pos) THEN
                    i2 <= -speedX;
                ELSE
                    Paddle1_plane <= '1';
                END IF;
            -- Take care of ball on left side of screen (+8 factor due to ball &
paddle size):
            -- Did we hit the wall?
            ELSIF '0' & Ball_X_pos <= Size + CONV_STD_LOGIC_VECTOR(speedX,10) THEN
                Backstop2 <= '1';
                -- did we hit paddle2?
                ELSIF Ball_X_pos + CONV_STD_LOGIC_VECTOR(speedX,10) <= Plane_dist +
CONV_STD_LOGIC_VECTOR(8,10) THEN
                    IF ('0' & Paddle2_Y_pos < Ball_Y_pos + Paddle_SizeY) AND
(Paddle2_Y_pos + Paddle_SizeY > '0' & Ball_Y_pos) THEN
                        i2 <= speedX;
                    ELSE
                        Paddle2_plane <= '1';
                    END IF;
                END IF;
            -- Compute next ball Y position
            Ball_Y_motion <= CONV_STD_LOGIC_VECTOR(i1,10);
            Ball_X_motion <= CONV_STD_LOGIC_VECTOR(i2,10);
            Ball_Y_pos <= Ball_Y_pos + Ball_Y_motion;
            Ball_X_pos <= Ball_X_pos + Ball_X_motion;
            Ball_Y_pos_ext <= Ball_Y_pos;
        END IF;
    END IF;
END process Move_Ball;

END behavior;

```

paddle

```

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY paddle IS

```

```

PORT(
    SIGNAL Red,Green,Blue          : OUT std_logic;
    SIGNAL Paddle_Y_pos_out        : OUT std_logic_vector(9 DOWNTO 0);
    SIGNAL reset_n, vert_sync      : IN std_logic;
    SIGNAL move_up, move_down      : IN std_logic;
    signal pixel_row, pixel_column : IN std_logic_vector(9 downto 0));

END paddle;

architecture behavior of paddle is

    SIGNAL Paddle_on                : std_logic;
    SIGNAL Paddle_X_pos, Paddle_Y_pos : std_logic_vector(9 DOWNTO 0);
    signal i1, i2                   : integer;

    CONSTANT SizeX: std_logic_vector(9 DOWNTO 0) := CONV_STD_LOGIC_VECTOR(4,10);
    CONSTANT SizeY: std_logic_vector(9 DOWNTO 0) := CONV_STD_LOGIC_VECTOR(32,10);

BEGIN

    Paddle_X_pos <= CONV_STD_LOGIC_VECTOR(600,10); -- this could have been a CONSTANT

    -- Colors for pixel data on video signal
    Red <= Paddle_on;
    Green <= Paddle_on;
    Blue <= Paddle_on;

    Paddle_Y_pos_out <= Paddle_Y_pos;

    RGB_Display: Process (pixel_column, pixel_row)
    BEGIN
        -- Set Paddle_on = '1' to display paddle
        IF ('0' & Paddle_X_pos <= pixel_column + SizeX) AND
            (Paddle_X_pos + SizeX >= '0' & pixel_column) AND
            ('0' & Paddle_Y_pos <= pixel_row + SizeY) AND
            (Paddle_Y_pos + SizeY >= '0' & pixel_row ) THEN
            Paddle_on <= '1';
        ELSE
            Paddle_on <= '0';
        END IF;
    END process RGB_Display;

    Move_Paddle: process (vert_sync)
    BEGIN
        -- Move paddle once every vertical sync
        IF vert_sync = '1' THEN
            -- stop at the top or bottom of screen
            IF (reset_n = '0') Then
                Paddle_Y_pos <= CONV_STD_LOGIC_VECTOR(210, 10);
            ELSE
                IF (move_up='1') AND (Paddle_Y_pos > SizeY) THEN
                    Paddle_Y_pos <= Paddle_Y_pos - CONV_STD_LOGIC_VECTOR(2, 10);
                ELSIF (move_down='1') AND (('0' & Paddle_Y_pos) <=
                    CONV_STD_LOGIC_VECTOR(480,10) - SizeY) THEN
                    Paddle_Y_pos <= Paddle_Y_pos + CONV_STD_LOGIC_VECTOR(2, 10);
                END IF;
            END IF;
        END IF;
    END process Move_Paddle;

END behavior;

```

paddle2

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;
USE IEEE.STD_LOGIC_ARITH.all;
USE IEEE.STD_LOGIC_UNSIGNED.all;
LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY paddle2 IS

    PORT(
        SIGNAL Red,Green,Blue           : OUT std_logic;
        SIGNAL Paddle_Y_pos_out         : OUT std_logic_vector(9 DOWNTO 0);
        SIGNAL reset_n, vert_sync       : IN std_logic;
        signal ball_Y_pos                : IN std_logic_vector(9 downto 0);
        signal pixel_row, pixel_column   : IN std_logic_vector(9 downto 0));

END paddle2;

architecture behavior of paddle2 is

    SIGNAL Paddle_on                    : std_logic;
    SIGNAL Paddle_X_pos, Paddle_Y_pos   : std_logic_vector(9 DOWNTO 0);
    signal i1, i2                       : integer;
    SIGNAL move_up, move_down           : std_logic;

    CONSTANT SizeX: std_logic_vector(9 DOWNTO 0) := CONV_STD_LOGIC_VECTOR(4,10);
    CONSTANT SizeY: std_logic_vector(9 DOWNTO 0) := CONV_STD_LOGIC_VECTOR(32,10);
    CONSTANT speedY: integer := 2;

BEGIN

    Paddle_X_pos <= CONV_STD_LOGIC_VECTOR(40,10); -- this could have been a CONSTANT

    -- Colors for pixel data on video signal
    Red <= Paddle_on;
    Green <= Paddle_on;
    Blue <= Paddle_on;

    Paddle_Y_pos_out <= Paddle_Y_pos;

    Track_Ball: PROCESS (ball_Y_pos)
    BEGIN
        IF ball_Y_pos > Paddle_Y_pos THEN
            move_up <= '0';
            move_down <= '1';
        ELSIF ball_Y_pos < Paddle_Y_pos THEN
            move_up <= '1';
            move_down <= '0';
        ELSE
            move_up <= '0';
            move_down <= '0';
        END IF;
    END process Track_Ball;

    RGB_Display: Process (pixel_column, pixel_row)
    BEGIN
        -- Set Paddle_on = '1' to display ball
        IF ('0' & Paddle_X_pos <= pixel_column + SizeX) AND
            (Paddle_X_pos + SizeX >= '0' & pixel_column) AND
            ('0' & Paddle_Y_pos <= pixel_row + SizeY) AND
            (Paddle_Y_pos + SizeY >= '0' & pixel_row ) THEN
```

```

        Paddle_on <= '1';
    ELSE
        Paddle_on <= '0';
    END IF;
END process RGB_Display;

Move_Paddle: process (vert_sync)
BEGIN
    -- Move paddle once every vertical sync
    IF vert_sync = '1' THEN
        -- stop at the top or bottom of screen
        IF (reset_n = '0') Then
            Paddle_Y_pos <= CONV_STD_LOGIC_VECTOR(200, 10);
        ELSE
            IF (move_up='1') AND (Paddle_Y_pos > SizeY) THEN
                Paddle_Y_pos <= Paddle_Y_pos - CONV_STD_LOGIC_VECTOR(speedY, 10);
            ELSIF (move_down='1') AND (('0' & Paddle_Y_pos) <=
CONV_STD_LOGIC_VECTOR(480,10) - SizeY) THEN
                Paddle_Y_pos <= Paddle_Y_pos + CONV_STD_LOGIC_VECTOR(speedY, 10);
            END IF;
        END IF;
    END IF;
END process Move_Paddle;

END behavior;

```