ECE 5730 Final Project - Brick Breaker

Steve Brezenski, Brian Strickler

Procedure

A Quartus project, brick_breaker, was created using the system builder software. A top level vhdl file was created. Ports for the 50 MHz clock, two keys, VGA signals, LEDS, and ARDUINO headers were declared in the entity portion of code. Next a PLL was used to obtain a clock with a frequency of 25.170068 MHz by applying a multiplication factor of 74 and division factor of 147 to the 50 MHz clock, to be used for the VGA display. A PLL was also set up for the ADC. To set up the ADC, the code structure was taken from Lab 8.

The first step taken in creating the brick breaker game was the sound board. To do this, a vhdl file was written containing a state machine into which was fed a 3 bit signal representing the sound to be played. Within the state machine, logic to generate each sound was written. To produce unique noises, different square waves were generated. The block of code below shows the logic used to create one of the sounds.

```
when(b"001") => -- ball hit paddle
    if sound_count <= 9000000 then
        ready <= '0';
        sound_count <= sound_count + 1;
        if sound_count <= 4500000 then
            if sound_count mod 65536 = 0 then
                output <= not output;
            end if;
            else
                if sound_count mod 32768 = 0 then
                    output <= not output;
                end if;
            end if;
        else
            ready <= '1';
            output <= '0';
    end if;
```

A piezo-electric buzzer was used to produce the sounds. The buzzer was connected to IO7 and GND on the arduino headers. A 220 ohm resistor was placed in series with the buzzer to avoid over driving it.

Next, the logic for displaying the initial screen was written. An array of 640 12-bit values was created to hold the pixel data of brick layers that started with full bricks. Another array of the same size was also created to hold the pixel data of brick layers that started with half bricks. To keep track of whether a brick was broken or intact, an array of 1215 std_logic values was created. Bricks were numbered from left to right and top to bottom. Setting an element in this array to '1' meant the brick was intact whereas a '0' meant the brick was broken.

A process was then created to keep track of which brick was to be painted. This process determined whether or not to display horizontal mortar depending on the y_position on the display. If the current brick being painted was intact, it would choose whether to look at the pixel data from full_bricks or staggered array based

on the current brick row being painted. If the brick was broken, the pixel data to be painted would be set to black.

Next the paddle was given a position based on the upper left corner of the paddle and the reading from the potentiometer. A process would then paint the pixels brown for 40 pixels to the right and 5 pixels down from that position.The bricks and the paddle were fixed in what pixel rows they were displayed whereas the ball could move across pixel rows.

The ball was given a position referenced by the upper left corner. It then was given logic to display pixels from this position to ten pixels to the right and ten pixels below. Its color was set to be white. The initial ball position was set to be the middle of the display at the beginning of the game.

With the initial screen printed, the next step was to write a vhdl file to control ball movement. The ball_movement entity had four inputs: a clock, reset button, new_ball button, and a signal “collision” in which would be encoded the detected collisions. The outputs for this entity consisted of two signals to keep track of the ball position, a direction signal used in removing bricks, and a 10 bit signal used to control the LEDs on the DE10-Lite board.

To control ball movement a FSM was used, consisting of two processes. The first process was synchronous to the clock input. Within this process the ball position signals, ball_x and ball_y, were updated based on the signals next_ball_x and next_ball_y. The state of the FSM, current_ball_state, was also updated based on the signal next_ball_state. A counter was also implemented in this synchronous process to regulate how often these signals were updated. Each signal was updated at a rate of 60Hz to match the frame rate of the VGA display.

In the second process, next_ball_x, next_ball_y, and next_ball_state were defined. To do this, the signal “collision” was used to determine how the ball needed to move based on what the detection algorithm was finding. The “collision” value could take on 11 different values representing collision with different objects: five different areas of the paddle, the right wall or right side of a brick, the left wall or the left side of a brick, the top of the screen or the bottom of a brick, the top of a brick, the bottom of the screen, and no collision. A state was defined for each of these possible values as well as an IDLE state and an INITIAL state.

The IDLE state was used to wait for the game to begin as well as for a new ball to be dropped. Within this state the signals y_move and x_move were set to 0 so as to keep the ball stationary at the center of the screen. When the new_ball button was pressed, the FSM would then transition into the INITIAL state, in which x_move was set to 0 and y_move set to 2 so that the ball began dropping towards the bottom of the screen.

With the ball in motion, the continued motion of the ball was determined by the different collisions listed above encoded in the collision signal. The table below summarizes the different types of collisions detected and the resulting ball movements.

	Next Move in X Direction (x_move)	Next Move in Y Direction (y_move)
PAD_C	x_move	-y_move
PAD_R1	-2	-2
PAD_R2	-1	2
PAD_L1	-2	-2
PAD_L2	-1	-2
RIG	y_move	-x_move
LEF	y_move	-x_move
TOP	-y_move	x_move
DIE	y_move	x_move
BRICK_BELOW	-y_move	x_move

Table 1 - Summary of Collision Types and Resulting Ball Movement

To make the ball actually move, next_ball_x and next_ball_y were calculated by adding the current ball positions, defined by ball_x and ball_y, with the signals y_move and x_move, respectively.

To test this FSM, a test bench was written and the module was simulated in ModelSim. It took several iterations of updates to finally reach the desired outcome in ModelSim. Figure 1 shows the simulation of this testbench. When the ball movement was tested on the board, however, it was found that when the ball collided with the walls, that the ball would get stuck there, moving back and forth slightly. After further investigation, it was found that in order to avoid being caught in the walls that an additional 4 pixels of movement needed to be added to the ball position signals at each collision so as to avoid a second "collision" being detected.

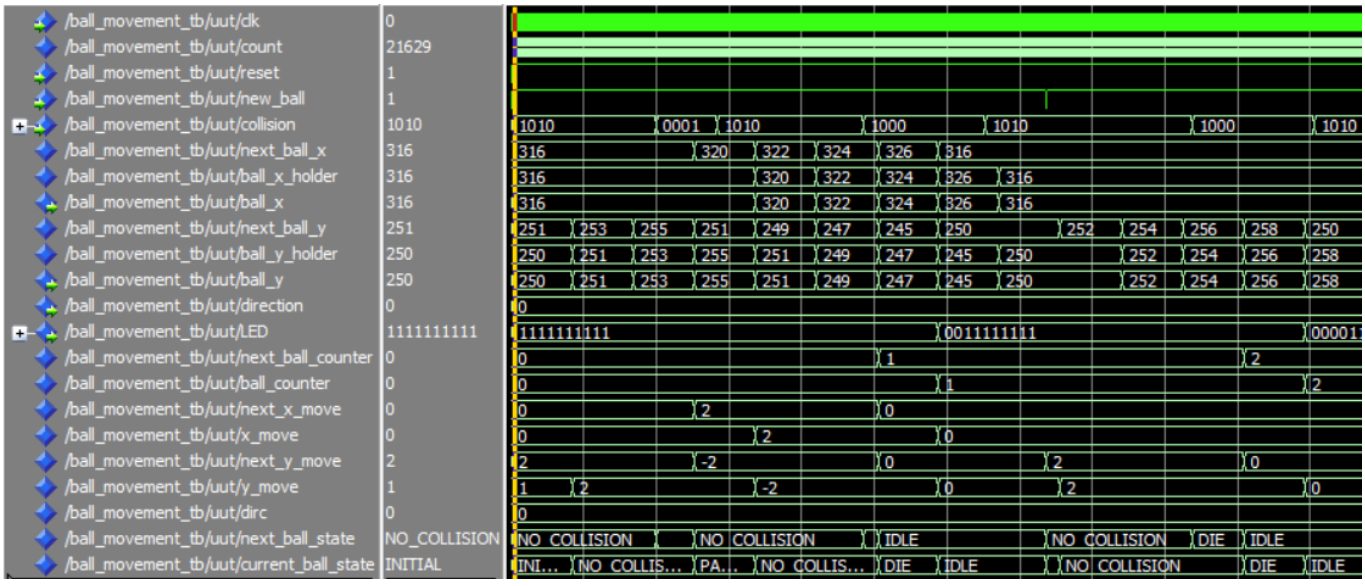

Figure 1 - ModelSim Waveforms :: Ball Movement Simulation

With the ball moving as desired, the next step was to implement a ball counter to keep track of the number of balls used. This was also implemented in the ball_movement file. This was done simply by using a variable

ball_counter to keep track of how many balls had been used. Each time the ball fell off the screen this variable was incremented and the new_ball button would have to be pressed to get a new ball. When the ball_counter reached 5 the game ended and a new ball could only be retrieved by hitting the reset button and starting a new game. To indicate to the user the number of balls left, the LEDs were lit up corresponding to the number of lives remaining: all 10 LEDs when 5 lives remained, 8 LEDs when 4 lives remained, and so on.

Next a process was written to detect any collision the ball might incur. The first step in creating this was to determine what the possible collisions were. The first collisions considered were those resulting from the ball bouncing off the left, top, and right of screen. We also needed to consider when the ball hit the paddle and how it would react. We also needed to consider how the ball would fall off the screen. Finally, we considered how the ball would interact with hitting bricks from the top, bottom, left, and right of them.

Due to the complexity of there being so many possible collisions, several separate processes were written to check whether or not a collision occurred each time the ball changed position. If a collision was detected, a flag was set in each of these processes that was then fed to the main collision process to determine which collision had precedence. When a collision was detected, the corresponding sound was played, then the ball's direction was adjusted, and, if needed, the corresponding brick would be broken.

The most simple collisions to check for were whether or not the ball collided with a wall. To check this, the ball's x and y positions were compared to the x and y bounds of the screen. If the ball tried to go beyond the limits, the ball's direction would be reversed, and a short tone would play. If the ball's y value exceeded the bottom of the screen, the ball would essentially "die", play a 1 second melody, and reset to the center of the screen as long as there were remaining lives.

For the collisions of the ball with the paddle, the position of paddle was compared to the position of the ball. This difference corresponded to five positions on the paddle: PAD_L2, PAD_L1, PAD_C, PAD_R1, and PAD_R2.

The most difficult collisions to account for were the bricks. Timing was crucial to ensure the brick disappeared after the ball had reversed direction, otherwise, the ball plowed through the bricks only changing direction on paddle and wall collisions.

Four separate processes were used to check for collisions on top, left, right, and bottom of bricks. Logic was implemented to figure out whether or not it collided with two bricks or just one. These processes calculated what the theoretical brick above, below, to the left, and to the right were. Next it checked if the brick at this position was still intact. If so, a corresponding flag would be set and fed into the main collision control process. From this control process, signals were set accordingly to be fed into the ball movement to reverse the necessary direction of the ball.

The figure below shows the hardware setup used for the brick breaker game.

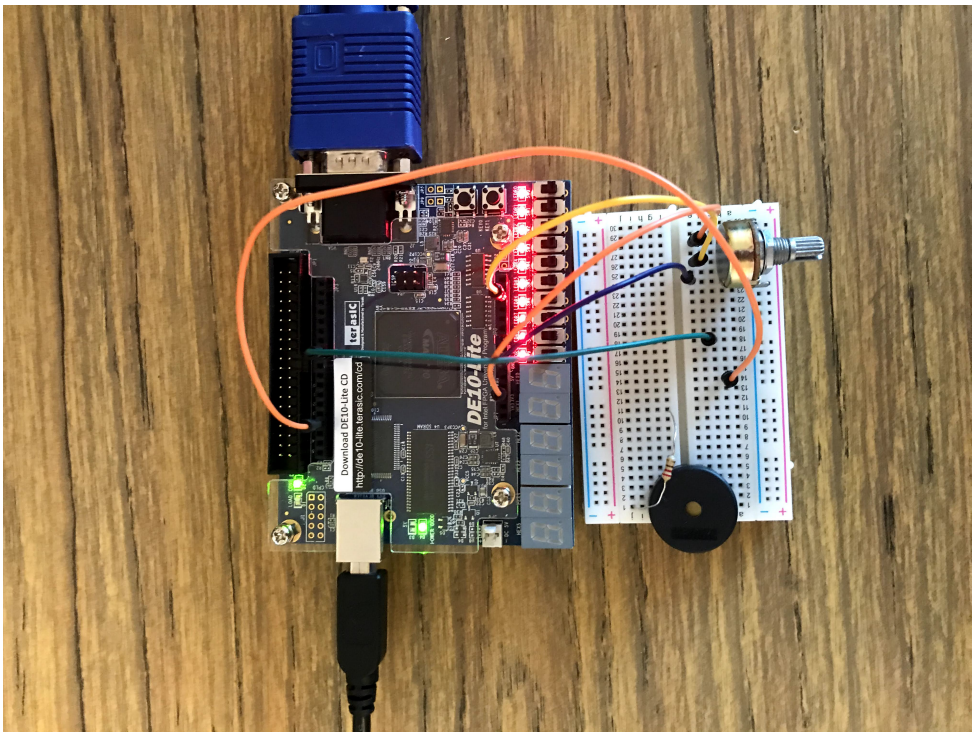

Figure 2 - Hardware Setup for Brick Breaker

Results

The final product worked mostly as expected. When the new_ball button was pressed the ball would be begin dropping from the middle of the screen. When the ball reached the paddle, the ball would rebound depending on where on the paddle it hit and a sound would be generated. From the center of the paddle, it bounced back at the same speed at the opposite angle. If the ball hit the paddle to the right or left of center it rebounded off to the right or left, respectively. When the ball hit a brick, the brick disappeared and the brick breaking sound was generated. As expected, when the ball hit two bricks both bricks were eliminated. If the ball hit any of the walls, it rebounded as specified in Table 1 above.

Figure 3 - Brick Breaker :: Initial Screen

After the ball died five times, the game was ended and a new ball could only be generated by pressing the reset button and starting a new game. When the reset button was pressed in the middle of the game, all bricks were restored and all five balls replenished.

An odd behavior that occasionally occured in the game is that when it strikes perfectly in a corner between two bricks it will strike the side of the brick, rebound off of the brick above it, and continue in like fashion clearing out long sections of brick. This result was due to the way in which we detected bricks but was welcome as defeating this game took nigh unto an hour to accomplish.

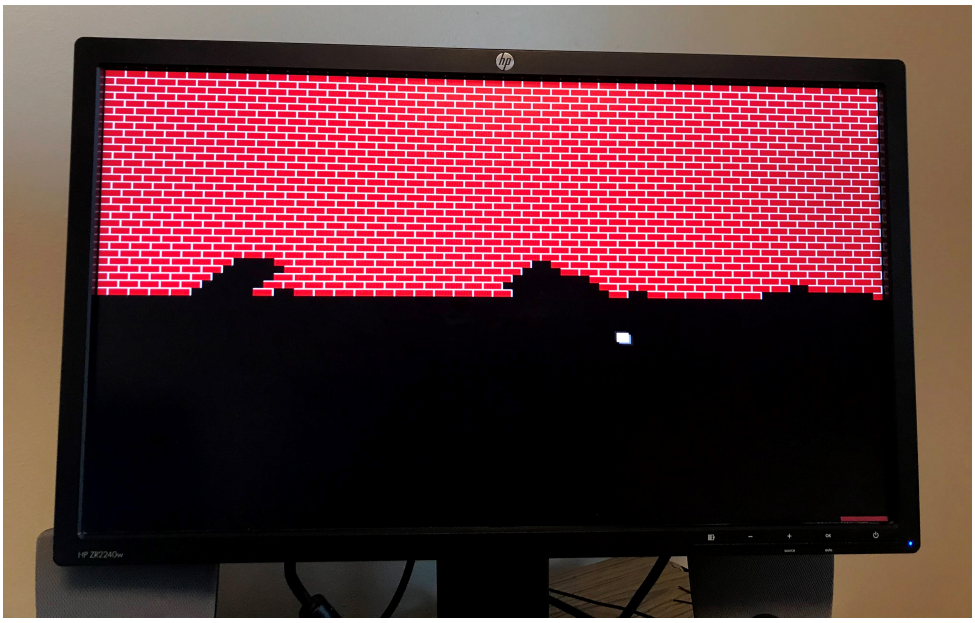

Figure 4 - Brick Breaker :: Mid-Game

Conclusion

This lab provided an opportunity to really solidify the use of VHDL and FSMs. It also provided greater insight into timing and the necessity to count clock cycles correctly in order to acheive the desired behavior. Overall, building brick breaker was an enjoyable experience and immensely satisfying to finish.

Appendix

```
-- brick_breaker.vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity brick_breaker is
    port(
        ADC_CLK_10 : in std_logic;
        MAX10_CLK1_50 : in std_logic;
        MAX10_CLK2_50 : in std_logic;
        KEY : std_logic_vector(1 downto 0); -- [0] reset and [1] next
        LEDR : out std_logic_vector(9 downto 0);
        SW : in std_logic_vector(9 downto 0);
        VGA_R   : out std_logic_vector(3 downto 0);
        VGA_G   : out std_logic_vector(3 downto 0);
        VGA_B   : out std_logic_vector(3 downto 0);
        VGA_HS  : out std_logic;
        VGA_VS  : out std_logic;
        ARDUINO_IO : out std_logic_vector(15 downto 0);
        ARDUINO_RESET_N : inout std_logic
    );
end entity brick_breaker;

architecture behavioral of brick_breaker is

signal pot : std_logic_vector(11 downto 0) := X"000";
signal rst : std_logic := '0';
signal sound_fx : std_logic_vector(2 downto 0); -- b"000"/no sound, b"001"/ball
paddle, b"010"/walls+ceiling, b"011"/dead ball, b"100"/brick break

signal vga_clk : std_logic; -- maps to c0 of myPLL
signal vga_rst : std_logic := '1';
signal color : std_logic_vector(11 downto 0) := X"000";

signal adc_clk : std_logic := '0';

    component sound_board is
        port (
            CLK : in std_logic := 'X';
            SOUND_EFFECT : in std_logic_vector (2 downto 0);
            SOUND_OUT : out std_logic
        );
    end component sound_board;

    component my_adc is                                    -- creates digital
value from potentiometer
        port (
            CLOCK : in  std_logic                    := 'X';   -- clk
            RESET : in  std_logic                    := 'X';   -- reset
```

```
            CH3    : out std_logic_vector(11 downto 0)           -- CH0
        );
    end component my_adc;

    component myPLL is
        port
        (
            inclk0  : in std_logic; -- MAX10_CLK1_50
            c0          : out std_logic -- 25.17 MHz
        );
    end component myPLL;

    component sync is
        port (
            clk                 : in std_logic;
            reset               : in std_logic;
            reset_l         : in std_logic;
            new_ball            : in std_logic;
            vs_sig          : out std_logic;
            hs_sig          : out std_logic;
            pixel_data  : out std_logic_vector(11 downto 0);
            pot : in std_logic_vector(11 downto 0);
            LEDR : out std_logic_Vector(9 downto 0);
            sound_fx : out std_logic_Vector(2 downto 0)
        );
    end component sync;

    component adcPLL is
        port (
            inclk0      : IN STD_LOGIC  := '0';
            c0      : OUT STD_LOGIC
        );
    end component adcPLL;

begin

    u0 : component sound_board
        port map (
            CLK => MAX10_CLK1_50,
            SOUND_EFFECT => sound_fx,
            SOUND_OUT => ARDUINO_IO(7)
        );

    u1 : component my_adc
        port map (
            CLOCK => ADC_CLK_10, --      clk.clk
            RESET => rst, --    reset.reset
            CH3   => pot   --         .CH1
        );

    u2 : component myPLL
        port map
        (
            inclk0 => MAX10_CLK1_50,
```

```
            c0 => vga_clk
        );

    u3 : sync
        port map(
            clk     => vga_clk,
            reset   => vga_rst,
            reset_l => KEY(0),
            new_ball => KEY(1),
            vs_sig => VGA_VS,
            hs_sig => VGA_HS,
            pixel_data  => color,
            pot => pot,
            LEDR => LEDR,
            sound_fx => sound_fx
        );

    u4 : adcPLL
        port map(
            inclk0 => ADC_CLK_10,
            c0 => adc_clk
        );

    VGA_R <= color(11 downto 8);
    VGA_G <= color(7 downto 4);
    VGA_B <= color(3 downto 0);

end architecture behavioral;
```

```
-- sync.vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity sync is
    port (
        clk                 : in std_logic;
        reset               : in std_logic;
        reset_l         : in std_logic;
        new_ball            : in std_logic;
        vs_sig          : out std_logic;
        hs_sig          : out std_logic;
        pixel_data  : out std_logic_vector(11 downto 0);
        pot : in std_logic_vector(11 downto 0);
        LEDR : out std_logic_vector(9 downto 0);
        sound_fx : out std_logic_vector(2 downto 0)
    );
end entity sync;

architecture behavioral of sync is
```

```
    -- Note / R: bits 11:8 / G: bits 7:4 / B: bits 3:0 /  MSB:LSB ->
    constant red : std_logic_vector(11 downto 0) := X"B00";
    constant black : std_logic_vector(11 downto 0) := X"000";
    constant brown : std_logic_vector(11 downto 0) := X"722";
    constant grey : std_logic_vector(11 downto 0) := X"FFF";

    type BRICK_STATE is array (0 to 1214) of std_logic;
    signal brick : BRICK_STATE  := (others => '1');

    signal cur_brick : integer range 0 to 1214 := 0;

    type MY_MEM is array (0 to 639) of std_logic_vector(11 downto 0);
    constant full_bricks : MY_MEM := (red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey, red, red, red, red, red, red, red, red, red,
red, red, red, red, red, red, grey);
    constant staggered : MY_MEM := (red, red, red, red, red, red, red, grey, red,
```

```
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red, red, red, red, red, red, red, red, grey, red,
red, red, red, red, red, red, red);

    type vs_states is (V_FRONT_PORCH, V_SYNC, V_BACK_PORCH, DATA);
    signal current_vs_state, next_vs_state : vs_states;

    type hs_states is (H_FRONT_PORCH, H_SYNC, H_BACK_PORCH, P_DATA);
    signal current_hs_state, next_hs_state : hs_states;

    type my_horiz is range 0 to 799;

    signal x_pos : integer := 0;
    signal y_pos : integer := 0;
    signal layer : integer := 0;
    signal next_x_pos : integer := 0;
    signal next_y_pos : integer := 0;
```

```
    signal count1 : integer := 0;
    signal count2 : integer := 0;
    signal count1_mod : integer := 0;
    signal count2_mod : integer := 0;
    signal m : std_logic := '0';
    signal pad_pos : integer := 0;
    signal pot_sig : integer := 0;
    signal ball_x : integer := 316;
    signal ball_y : integer := 250;
    signal brick_x : integer := 0;
    signal difference : integer := 0; -- difference between pad_pos and ball_x
    signal collision : std_logic_Vector(3 downto 0) := "1010"; -- 0000 PAD_C /
0001 PAD_R1 / 0010 PAD_R2 / 0011 PAD_L1 / 0100 PAD_L2 / 0101 RIG / 0110 LEF / 0111
TOP / 1000 DIE / 1001 BRICK_BELOW / 1010 NONE

    signal brick_above : std_logic := '0';
    signal brick_above1 : std_logic := '0';
    signal brick_left : std_logic := '0';
    signal brick_right : std_logic := '0';
    signal brick_below : std_logic := '0';
    signal brick_below1 : std_logic := '0';

    signal layer_above : integer := 0;
    signal layer_left : integer := 0;
    signal layer_right : integer := 0;
    signal layer_below : integer := 0;

    signal cur_brick_above : integer := 0;
    signal cur_brick_above1 : integer := 0;
    signal cur_brick_left : integer := 0;
    signal cur_brick_right : integer := 0;
    signal cur_brick_below : integer := 0;
    signal cur_brick_below1 : integer := 0;

    signal brick_above_x : integer := 0;
    signal brick_above_x1 : integer := 0;
    signal brick_left_x : integer := 0;
    signal brick_right_x : integer := 0;
    signal brick_below_x : integer := 0;
    signal brick_below_x1 : integer := 0;
    signal direction : integer := 0;
    signal LED : std_logic_vector(9 downto 0);

    component ball_movement is
        port (
            clk             : in std_logic;
            reset               : in std_logic; -- KEY(0)
            new_ball            : in std_logic; -- KEY(1)
            collision       : in std_logic_vector(3 downto 0); -- from detection
process
            ball_x          : out integer;
            direction           : out integer;
            LED             : out std_logic_vector(9 downto 0);
            ball_y          : out integer
```

```
        );
    end component ball_movement;

begin

    u0 : component ball_movement
        port map (
            clk => clk,
            reset => reset_1,
            new_ball => new_ball,
            collision => collision,
            direction => direction,
            ball_x => ball_x,
            LED => LED,
            ball_y => ball_y
        );


    process(clk, reset, current_vs_state, count1, count2, current_hs_state, x_pos,
y_pos)
    begin
        if rising_edge(clk) then
            if reset = '0' then
                current_vs_state <= V_FRONT_PORCH;
                current_hs_state <= H_FRONT_PORCH;
                count1 <= 0;
                count2 <= 0;
                x_pos <= 0;
                y_pos <= 0;
            else
                x_pos <= next_x_pos;
                y_pos <= next_y_pos;
                if m = '0' then
                    count1 <= count1 + 1;
                    count2 <= count2 + 1;
                else
                    count1 <= 0;
                    count2 <= 0;
                end if;
                current_vs_state <= next_vs_state;
                current_hs_state <= next_hs_state;
            end if;
        end if;
    end process;

    process(count1, current_vs_state, count1_mod)
    begin
        count1_mod <= count1 mod 420000;
        case current_vs_state is
            when V_FRONT_PORCH =>
                m <= '0';
                vs_sig <= '1';
                if count1_mod < 8000 then
                    next_vs_state <= V_FRONT_PORCH;
```

```
                else
                    next_vs_state <= V_SYNC;
                end if;

            when V_SYNC =>
                m <= '0';
                vs_sig <= '0';
                if count1_mod < 9600 then
                    next_vs_state <= V_SYNC;
                else
                    next_vs_state <= V_BACK_PORCH;
                end if;

            when V_BACK_PORCH =>
                m <= '0';
                vs_sig <= '1';
                if count1_mod < 36000 then
                    next_vs_state <= V_BACK_PORCH;
                else
                    next_vs_state <= DATA;
                end if;

            when DATA =>
                vs_sig <= '1';
                if count1_mod < 420000 and count1_mod >= 36000 then
                    next_vs_state <= DATA;
                    m <= '0';
                else
                    next_vs_state <= V_FRONT_PORCH;
                    m <= '1';
                end if;
        end case;
    end process;

    process(layer, count2,count2_mod, x_pos, y_pos, current_hs_state,
current_vs_state, pad_pos, ball_x, ball_y, brick, cur_brick, brick_above)
    begin
        count2_mod <= count2 mod 800;
        case current_hs_state is
            when H_FRONT_PORCH =>
                hs_sig <= '1';
                pixel_data <= black;
                next_x_pos <= 0;
                next_y_pos <= y_pos;
                if count2_mod < 16 then
                    next_hs_state <= H_FRONT_PORCH;
                else
                    next_hs_state <= H_SYNC;
                end if;

            when H_SYNC =>
                hs_sig <= '0';
                pixel_data <= black;
                next_x_pos <= 0;
```

```
                next_y_pos <= y_pos;
                if count2_mod < 112 then
                    next_hs_state <= H_SYNC;
                else
                    next_hs_state <= H_BACK_PORCH;
                end if;

            when H_BACK_PORCH =>
                hs_sig <= '1';
                pixel_data <= black;
                next_x_pos <= 0;
                if count2_mod < 160 then
                    next_hs_state <= H_BACK_PORCH;
                    next_y_pos <= y_pos;
                else
                    next_hs_state <= P_DATA;
                    next_y_pos <= y_pos + 1;
                end if;

            when P_DATA =>
                hs_sig <= '1';
                if count2_mod < 800 and count2_mod >= 160 then
                    next_hs_state <= P_DATA;
                    next_y_pos <= y_pos;
                    next_x_pos <= x_pos + 1;
                else
                    next_hs_state <= H_FRONT_PORCH;
                    next_y_pos <= y_pos;
                    next_x_pos <= x_pos;
                end if;
                if current_vs_state = V_FRONT_PORCH or current_vs_state = V_SYNC
or current_vs_state = V_BACK_PORCH then
                    pixel_data <= black;
                    next_x_pos <= 0;
                    next_y_pos <= 0;
                else
                    next_y_pos <= y_pos;
                    next_x_pos <= x_pos + 1;

                    if (y_pos < 240) then
                        if x_pos >= ball_x and x_pos < (ball_x + 10) and y_pos >=
ball_y and y_pos < (ball_y + 10) then
                                pixel_data <= grey;
                        elsif brick(cur_brick) = '0' then
                                pixel_data <= black;
                        else
                            if (y_pos mod 8 = 0) then
                                pixel_data <= grey;
                            elsif layer mod 2 = 0 then
                                pixel_data <= full_bricks(x_pos);
                            else
                                pixel_data <= staggered(x_pos);
                            end if;
                        end if;
```

```
                    end if;

                    if y_pos >= 240 and y_pos <= 475 then
                        if x_pos >= ball_x and x_pos < ball_x + 10 then
                            if y_pos >= ball_y and y_pos < ball_y + 10 then
                                pixel_data <= grey;
                            else
                                pixel_data <= black;
                            end if;
                        else
                            pixel_data <= black;
                        end if;
                    end if;

                    if y_pos < 481 and y_pos > 475 then  -- not sure the y_pos and
x_pos perfectly match the displays x and y coordinates
                        if x_pos >= ball_x and x_pos < (ball_x + 10) and y_pos >=
ball_y and y_pos < (ball_y + 10) then
                            pixel_data <= grey;
                        elsif x_pos >= pad_pos and x_pos < pad_pos + 40 then
                            pixel_data <= brown;
                        else
                            pixel_data <= black;
                        end if;
                    end if;

                end if;
        end case;
    end process;

    CURRENT_BRICK : process(x_pos, layer, brick_x) begin
        if layer mod 2 = 1 then
            brick_x <= (x_pos+8)/16;
            cur_brick <= (layer*40) + layer/2 + brick_x;
        else
            brick_x <= x_pos/16;
            cur_brick <= (layer*40) + layer/2 + brick_x;
        end if;
    end process;

    BRICK_LAYER : process(y_pos) begin
        if(y_pos < 240) then
            layer <= y_pos/8;
        else
            layer <= 0;
        end if;
    end process;

    PADDLE_POSITION : process(pot_sig) begin
            if pot_sig > 2400 then
                pad_pos <= 600;
            else
                pad_pos <= pot_sig/4;
            end if;
```

```
    end process;

    -- 0000 PAD_C / 0001 PAD_R1 / 0010 PAD_R2 / 0011 PAD_L1 / 0100 PAD_L2 / 0101
RIG / 0110 LEF / 0111 TOP / 1000 DIE / 1001 BOT / 1010 NONE
    -- b"000"/no sound, b"001"/ball paddle, b"010"/walls+ceiling, b"011"/dead
ball, b"100"/brick break
    COLLISION_DETECTION : process(clk, ball_x, ball_y, pad_pos, brick,
current_vs_state, difference) begin
        if(rising_edge(clk)) then
            if ball_x <= 0 then
                collision <= "0110"; --hit left wall
                sound_fx <= "010";
            elsif ball_x >= 630 then
                collision <= "0101"; -- hit right wall
                sound_fx <= "010";
            elsif ball_y <= 1 then
                collision <= "0111"; -- hit top wall
                sound_fx <= "010";
            elsif ball_y >= 479 then
                collision <= "1000"; -- ball dead
                sound_fx <= "011";
            elsif brick_above = '1' then
                collision <= "0111"; -- ball hit bottom of brick
                sound_fx <= "100";
            elsif brick_below = '1' then
                collision <= "1001";    -- ball hit top of brick
                sound_fx <= "100";
            elsif brick_left = '1' then
                collision <= "0101"; -- ball hit left side of brick
                sound_fx <= "100";
            elsif brick_right = '1' then
                collision <= "0110";    -- ball hit right side of brick
                sound_fx <= "100";
            elsif ball_y >= 465 and ball_y <= 467 then
                if difference <= 9 and difference >= 0 then
                    collision <= "0100";    -- PAD_L2
                    sound_fx <= "001";
                elsif difference <= -1 and difference >= -10 then
                    collision <= "0011"; -- PAD_L1
                    sound_fx <= "001";
                elsif difference <= -11 and difference >= -20 then
                    collision <= "0000"; -- PAD_C
                    sound_fx <= "001";
                elsif difference <= -21 and difference >= -30 then
                    collision <= "0001"; -- PAD_R1
                    sound_fx <= "001";
                elsif difference <= -31 and difference >= -39 then
                    collision <= "0010"; -- PAD_R2
                    sound_fx <= "001";
                else
                    collision <= "1010"; --no collision
                    sound_fx <= "000";
                end if;
            else
```

```
                collision <= "1010";
                sound_fx <= "000";
            end if;
        end if;
    end process;

    DIFFERENCE_PAD_BALL : process(m, ball_x, pad_pos) begin
            difference <= pad_pos - ball_x;
    end process;

    BRICK_ABOVE_DETECTION : process(ball_x, ball_y, cur_brick_above,
cur_brick_above1, layer_above, brick, brick_above_x, brick_above_x1, brick_above,
brick_above1) begin
        if ball_y < 241 then
            layer_above <= (ball_y-1)/8;
            if layer_above mod 2 = 1 then
                brick_above_x <= (ball_x+8)/16;
                brick_above_x1 <= (ball_x+18)/16;
                cur_brick_above <= (layer_above*40) + layer_above/2 +
brick_above_x;
                cur_brick_above1 <= (layer_above*40) + layer_above/2 +
brick_above_x1;
            else
                brick_above_x <= (ball_x)/16;
                brick_above_x1 <= (ball_x+10)/16;
                cur_brick_above <= (layer_above*40) + layer_above/2 +
brick_above_x;
                cur_brick_above1 <= (layer_above*40) + layer_above/2 +
brick_above_x1;
            end if;

            if brick(cur_brick_above) = '1' then
                brick_above <= '1';
            else
                brick_above <= '0';
            end if;
            if brick(cur_brick_above1) = '1' then
                brick_above1 <= '1';
            else
                brick_above1 <= '0';
            end if;
        else
            layer_above <= 0;
            brick_above_x <= 0;
            brick_above <= '0';
            cur_brick_above <= 0;
            brick_above_x1 <= 0;
            brick_above1 <= '0';
            cur_brick_above1 <= 0;
        end if;
    end process;

    BRICK_LEFT_DETECTION : process(ball_x, ball_y, layer_left, brick_left_x,
brick, cur_brick_left) begin
```

```
        if ball_y < 241 then
            layer_left <= (ball_y)/8;
            if layer_left mod 2 = 1 then
                brick_left_x <= (ball_x+18)/16;
                cur_brick_left <= (layer_left*40) + layer_left/2 + brick_left_x;
            else
                brick_left_x <= (ball_x+10)/16;
                cur_brick_left <= (layer_left*40) + layer_left/2 + brick_left_x;
            end if;

            if brick(cur_brick_left) = '1' then
                brick_left <= '1';
            else
                brick_left <= '0';
            end if;
        else
            brick_left_x <= 0;
            layer_left <= 0;
            brick_left <= '0';
            cur_brick_left <= 0;
        end if;
    end process;

    BRICK_RIGHT_DETECTION : process(ball_x, ball_y, layer_right, brick_right_x,
brick, cur_brick_right) begin
        if ball_y < 241 then
            layer_right <= (ball_y+10)/8;
            if layer_right mod 2 = 1 then
                brick_right_x <= (ball_x+8)/16;
                cur_brick_right <= (layer_right*40) + layer_right/2 +
brick_right_x;
            else
                brick_right_x <= (ball_x)/16;
                cur_brick_right <= (layer_right*40) + layer_right/2 +
brick_right_x;
            end if;

            if brick(cur_brick_right) = '1' then
                brick_right <= '1';
            else
                brick_right <= '0';
            end if;
        else
            brick_right_x <= 0;
            layer_right <= 0;
            brick_right <= '0';
            cur_brick_right <= 0;
        end if;
    end process;

    BRICK_BELOW_DETECTION : process(ball_x, ball_y, cur_brick_below,
cur_brick_below1, layer_below, brick, brick_below_x, brick_below_x1, brick_below,
brick_below1) begin
        if ball_y < 241 then
```

```
            layer_below <= (ball_y+10)/8;
            if layer_below mod 2 = 1 then
                brick_below_x <= (ball_x+18)/16;
                brick_below_x1 <= (ball_x + 8)/16;
                cur_brick_below <= (layer_below*40) + layer_below/2 +
brick_below_x;
                cur_brick_below1 <= (layer_below*40) + layer_below/2 +
brick_below_x1;
            else
                brick_below_x <= (ball_x+10)/16;
                brick_below_x1 <= (ball_x)/16;
                cur_brick_below <= (layer_below*40) + layer_below/2 +
brick_below_x;
                cur_brick_below1 <= (layer_below*40) + layer_below/2 +
brick_below_x1;
            end if;

            if brick(cur_brick_below) = '1' then
                brick_below <= '1';
            else
                brick_below <= '0';
            end if;

            if brick(cur_brick_below1) = '1' then
                brick_below1 <= '1';
            else
                brick_below1 <= '0';
            end if;
        else
            layer_below <= 0;
            brick_below_x <= 0;
            brick_below <= '0';
            cur_brick_below <= 0;
            brick_below_x1 <= 0;
            brick_below1 <= '0';
            cur_brick_below1 <= 0;
        end if;
    end process;

    process(clk, count1_mod, brick_above, collision, brick) begin
        if rising_edge(clk) then
            if reset_l <= '0' then
                brick <= (others => '1');
            else
                if ((brick_above = '1' or brick_above1 = '1') and collision =
"0111" and direction = 1) then
                    brick(cur_brick_above) <= brick(cur_brick_above) and '0';
                    brick(cur_brick_above1) <= brick(cur_brick_above1) and '0';
                elsif ((brick_below = '1' or brick_below1 = '1') and collision =
"1001" and direction = 2) then
                    brick(cur_brick_below) <= brick(cur_brick_below) and '0';
                    brick(cur_brick_below1) <= brick(cur_brick_below1) and '0';
                elsif (brick_left = '1' and collision = "0101" and direction = 3)
then
```

```
                    brick(cur_brick_left) <= brick(cur_brick_left) and '0';
                elsif (brick_right = '1' and collision = "0110" and direction = 4)
then
                    brick(cur_brick_right) <= brick(cur_brick_right) and '0';
                else
                    brick(cur_brick_above) <= brick(cur_brick_above) and '1';
                    brick(cur_brick_below) <= brick(cur_brick_below) and '1';
                    brick(cur_brick_left) <= brick(cur_brick_left) and '1';
                    brick(cur_brick_right) <= brick(cur_brick_right) and '1';
                end if;
            end if;
        end if;
    end process;

    LEDR <= LED;
    pot_sig <= to_integer(unsigned(pot));
end architecture behavioral;
```

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

entity ball_movement is
    port (
        clk             : in std_logic;
        reset               : in std_logic; -- KEY(0)
        new_ball            : in std_logic; -- KEY(1)
        collision       : in std_logic_vector(3 downto 0); -- from detector
process
        ball_x          : out integer;
        ball_y          : out integer;
        direction       : out integer;
        LED             : out std_logic_vector(9 downto 0)
    );
end entity ball_movement;

architecture behavioral of ball_movement is

-- signals for ball movement
    signal next_ball_y : integer := 250;
    signal next_ball_x : integer := 316;
    signal ball_y_holder : integer := 250;
    signal ball_x_holder : integer := 316;
    signal ball_counter : integer := 0;
    signal next_ball_counter : integer := 0;
    signal x_move : integer := 0;
    signal y_move : integer := 1;
    signal next_x_move : integer := 0;
    signal next_y_move : integer := 1;
    signal dirc : integer := 0;
```

```
    signal count : natural := 0;

    type ball_states is (IDLE, DIE, INITIAL, NO_COLLISION, RIG, LEF, TOP, PAD_C,
PAD_R1, PAD_R2, PAD_L1, PAD_L2, BRICK_BELOW);
    signal current_ball_state, next_ball_state : ball_states;

begin
    -- synchronous process
    process(clk, reset, new_ball, ball_y_holder, ball_x_holder, next_ball_x,
next_ball_y, next_ball_state) is
    begin
        if rising_edge(clk) then
            ball_y <= ball_y_holder;
            ball_x <= ball_x_holder;
            if reset = '0' then
                ball_counter <= 0;
                current_ball_state <= IDLE;
            elsif new_ball = '0' then
                current_ball_state <= INITIAL;
            else
                if count >= 419999 then
                    ball_counter <= next_ball_counter;
                    count <= 0;
                    ball_x_holder <= next_ball_x;
                    ball_y_holder <= next_ball_y;
                    x_move <= next_x_move;
                    y_move <= next_y_move;
                    current_ball_state <= next_ball_state;
                else
                    count <= count + 1;
                end if;
            end if;
        end if;
    end process;

    -- FSM (IDLE, INITIAL, TOP, RIG, LEF, DIE, PAD_C, PAD_L1, PAD_L2, PAD_R1,
PAD_R2, BRICK_BELOW)
    process(clk,collision, ball_x_holder, ball_y_holder, x_move, y_move, new_ball,
reset, current_ball_state, ball_counter, reset, next_ball_counter) is
    begin
        if rising_edge(clk) then
        case current_ball_state is
            when IDLE =>
                dirc <= 0;
                next_ball_counter <= ball_counter;
                if reset = '1' then
                    next_ball_state <= current_ball_state;
                    next_y_move <= 0;
                    next_x_move <= 0;
                    next_ball_y <= 250;
                    next_ball_x <= 316;
                elsif next_ball_counter = 5 then
                    next_ball_state <= IDLE;
```

```
                next_y_move <= 0;
                next_x_move <= 0;
                next_ball_y <= 500;
                next_ball_x <= 316;
            else
                next_ball_state <= IDLE;
                next_y_move <= y_move;
                next_x_move <= x_move;
                next_ball_y <= 250;
                next_ball_x <= 316;
            end if;

        when INITIAL =>
            dirc <= 0;
            next_ball_counter <= ball_counter;
            if new_ball = '1' then
                next_y_move <= 2;
                next_x_move <= 0;
                next_ball_y <= ball_y_holder + y_move;
                next_ball_x <= ball_x_holder + x_move;
                next_ball_state <= NO_COLLISION;
            else
                next_ball_state <= INITIAL;
                next_y_move <= y_move;
                next_x_move <= x_move;
                next_ball_y <= ball_y_holder;
                next_ball_x <= ball_x_holder;
            end if;

        when NO_COLLISION =>
            dirc <= 0;
            next_y_move <= y_move;
            next_x_move <= x_move;
            next_ball_y <= ball_y_holder + y_move;
            next_ball_x <= ball_x_holder + x_move;
            next_ball_counter <= ball_counter;
            case collision is
                when "0000" => -- PAD_C
                    next_ball_state <= PAD_C;
                when "0001" => -- PAD_R1
                    next_ball_state <= PAD_R1;
                when "0010" => -- PAD_R2
                    next_ball_state <= PAD_R2;
                when "0011" => -- PAD_L1
                    next_ball_state <= PAD_L1;
                when "0100" => -- PAD_L2
                    next_ball_state <= PAD_L2;
                when "0101" => -- RIG
                    next_ball_state <= RIG;
                when "0110" => -- LEF
                    next_ball_state <= LEF;
                when "0111" => -- TOP
                    next_ball_state <= TOP;
                when "1000" => -- DIE
```

```
                    next_ball_state <= DIE;
                when "1001" => -- BRICK_BELOW
                    next_ball_state <= BRICK_BELOW;
                when "1010" => -- NO_COLLISION
                    next_ball_state <= NO_COLLISION;
                when others =>
                    next_ball_state <= NO_COLLISION;
            end case;

        when PAD_C =>
            dirc <= 0;
            next_y_move <= -y_move;
            next_x_move <= x_move;
            next_ball_y <= ball_y_holder - 4;
            next_ball_x <= ball_x_holder;
            next_ball_state <= NO_COLLISION;
            next_ball_counter <= ball_counter;

        when PAD_R1 =>
            dirc <= 0;
            next_y_move <= -2;
            next_x_move <= 2;
            next_ball_y <= ball_y_holder - 4;
            next_ball_x <= ball_x_holder + 4;
            next_ball_state <= NO_COLLISION;
            next_ball_counter <= ball_counter;

        when PAD_R2 =>
            dirc <= 0;
            next_y_move <= -1;
            next_x_move <= 2;
            next_ball_y <= ball_y_holder - 4;
            next_ball_x <= ball_x_holder + 4;
            next_ball_state <= NO_COLLISION;
            next_ball_counter <= ball_counter;

        when PAD_L1 =>
            dirc <= 0;
            next_y_move <= -2;
            next_x_move <= -2;
            next_ball_y <= ball_y_holder - 4;
            next_ball_x <= ball_x_holder - 4;
            next_ball_state <= NO_COLLISION;
            next_ball_counter <= ball_counter;

        when PAD_L2=>
            dirc <= 0;
            next_y_move <= -1;
            next_x_move <= -2;
            next_ball_y <= ball_y_holder - 4;
            next_ball_x <= ball_x_holder - 4;
            next_ball_state <= NO_COLLISION;
            next_ball_counter <= ball_counter;
```

```
            when RIG =>
                dirc <= 3;
                next_y_move <= y_move;
                next_x_move <= -x_move;
                next_ball_y <= ball_y_holder;
                next_ball_x <= ball_x_holder - 4;
                next_ball_state <= NO_COLLISION;
                next_ball_counter <= ball_counter;

            when LEF =>
                dirc <= 4;
                next_y_move <= y_move;
                next_x_move <= -x_move;
                next_ball_y <= ball_y_holder;
                next_ball_x <= ball_x_holder + 4;
                next_ball_state <= NO_COLLISION;
                next_ball_counter <= ball_counter;

            when TOP =>
                dirc <= 1;
                next_y_move <= -y_move;
                next_x_move <= x_move;
                next_ball_y <= ball_y_holder + 4;
                next_ball_x <= ball_x_holder;
                next_ball_state <= NO_COLLISION;
                next_ball_counter <= ball_counter;

            when BRICK_BELOW =>
                dirc <= 2;
                next_y_move <= -y_move;
                next_x_move <= x_move;
                next_ball_y <= ball_y_holder - 4;
                next_ball_x <= ball_x_holder;
                next_ball_state <= NO_COLLISION;
                next_ball_counter <= ball_counter;

            when DIE =>
                dirc <= 0;
                next_y_move <= 0;
                next_x_move <= 0;
                next_ball_y <= ball_y_holder + y_move;
                next_ball_x <= ball_x_holder + x_move;
                next_ball_state <= IDLE;
                if next_ball_counter <= 4 then
                    next_ball_counter <= ball_counter + 1;
                    next_ball_state <= IDLE;
                else
                    next_ball_counter <= 5;
                    next_ball_state <= DIE;
                end if;
        end case;
        end if;
    end process;
```

```
    process(ball_counter) begin
        case(ball_counter) is
            when 0 =>
                LED <= "1111111111";
            when 1 =>
                LED <= "0011111111";
            when 2 =>
                LED <= "0000111111";
            when 3 =>
                LED <= "0000001111";
            when 4 =>
                LED <= "0000000011";
            when 5 =>
                LED <= "0000000000";
            when others =>
                LED <= "1010101010";
        end case;
    end process;

    direction <= dirc;
end architecture behavioral;
```

References