Documentation for Module “Collision Detection“ in “Ping-Pong Game Project“

Subject name:

Entwurf digitaler Systeme mit VHDL - AI1398 (SoSe24)

Group Members:

Agha Muhammad Aslam

Matrikel Nr.: 1515659

Atharva Kishor Naik

Matrikel Nr: 1441440

1. Table of Contents

2. Introduction

This document describes the implementation and functionality of the Collision_Detection module in VHDL. This module is designed to detect collisions between a moving ball, two rackets, and the walls of the screen, as described in the Ping-Pong game task.

3. Approach

For this project, we first do a mockup simulation of the situation written on the One Note and Python program. We use the Python program to check that our collision checker logic behaves as it should. We also made another mockup to show whether the whole process logic represents the required state changes.

4. Entity Declaration

The “Collision_Detection” entity defines the interface of the module, including generic parameters for configuration and input/output ports.

```
entity Collision_Detection is
        generic (
                ball_length : integer := 6;
                racket_length : integer := 10;
                racket_height : integer := 30;
                racket_left_space : integer := 20;
                racket_right_space : integer := 610;
                screen_height : integer := 480);
        port (
                clock_i : in std_logic;
                reset_i : in std_logic;
                racket_y_pos1_i : in std_logic_vector (9 downto 0);
                racket_y_pos2_i : in std_logic_vector (9 downto 0);
                ball_x_i : in std_logic_vector (9 downto 0);
                ball_y_i : in std_logic_vector (9 downto 0);
                hit_wall_o : out std_logic_vector (2 downto 0);
                hit_racket_l_o : out std_logic_vector (1 downto 0);
                hit_racket_r_o : out std_logic_vector (1 downto 0));
end Collision_Detection;
```

Generic Parameters: Configure the dimensions and positions of the ball and rackets.
Ports: Define inputs for clock, reset, and positions of the ball and rackets, as well as outputs for collision signals.

5. Architecture Overview

```
architecture Behavioral of Collision_Detection is
	constant screen_width : integer := 640;
	constant segment_height : integer := racket_height / 5; -- TODO: test, if this way can be implemented! because of the divider, it might not work properly on the FPGA

	signal hit_wall : std_logic_vector(2 downto 0);
	signal hit_racket_l, hit_racket_r : std_logic_vector(1 downto 0);
```

The Behavioral architecture implements the collision detection logic, including constants, signals, and processes.

Constants: "screen_width" and "segment_height" define the screen width and segment height of rackets.

Signals: "hit_wall" stores the wall collision status. "hit_racket_l" and "hit_racket_r" store the collision status with the left and right rackets respectively.

```
-- Define a record to represent a rectangle
type rectangle is record
        x : integer;
        y : integer;
        width : integer;
        height : integer;
end record;
```

```
function check_collision(
        rect1 : rectangle;
        rect2 : rectangle
) return boolean is
begin
        return ((rect1.x + rect1.width) >= rect2.x) and
               (rect1.x <= (rect2.x + rect2.width)) and
               ((rect1.y + rect1.height) >= rect2.y) and
               (rect1.y <= (rect2.y + rect2.height));
end function;
```

A record type "rectangle" is defined to represent rectangular object. A function "check_collision" is implemented to check for collisions between two rectangles. The logic is based on Algorithm Practice: Intersection between Rectangle Intersection (Gillan, 2016). The inner logical expression only returns true if two rectangles collide, else false.

6. Collision Detection Process

```
collision_proc : process (reset_i, ball_x_i, ball_y_i, racket_y_pos1_i, racket_y_pos2_i)
	variable ball, racket_l, racket_r, segment : rectangle;
begin
	-- Initialize the variables

	ball := (x => to_integer(unsigned(ball_x_i)),
		 y => to_integer(unsigned(ball_y_i)),
		 width => ball_length,
		 height => ball_length);

	racket_l := (x => racket_left_space,
		     y => to_integer(unsigned(racket_y_pos1_i)),
		     width => racket_length,
		     height => racket_height);

	racket_r := (x => racket_right_space,
		     y => to_integer(unsigned(racket_y_pos2_i)),
		     width => racket_length,
		     height => racket_height);
```

The main process “collision_proc“ handles collision detection between the ball and the rackets, as well as the ball and the walls. It takes reset and positions of rackets and ball as input, then creates new variables for utilizing these values within the process.

```
if check_collision(racket_l, ball) then  -- left racket collides with ball.

        hit_racket_l <= (others => '0'); -- this line to remove an unwanted latch

        -- check which segment of the racket was hit
        for i in 0 to 4 loop
                segment := (x => racket_left_space,
                           y => racket_l.y + (i * segment_height),
                           width => racket_length,
                           height => segment_height);

                if check_collision(segment, ball) then
                        case i is
                                when 0 => hit_racket_l <= "01"; -- 1. Segment hit
                                when 1 => hit_racket_l <= "10"; -- 2. Segment hit
                                when 2 => hit_racket_l <= "11"; -- 3. Segment hit
                                when 3 => hit_racket_l <= "10"; -- 2. Segment hit
                                when 4 => hit_racket_l <= "01"; -- 1. Segment hit
                                when others => null;
                        end case;
                        exit; -- break the loop
                end if;
        end loop;

        hit_wall <= (others => '0');     -- no collision with wall
        hit_racket_r <= (others => '0'); -- no collision with right racket

elsif check_collision(ball, racket_r) then  -- right racket collides with ball
```

Firstly the “if-statement” checks if the left racket and right racket collide, if they do, then we need to find exactly which segment do they collide. In order to find segment number, we check along the racket with respect to its segments with the help of increment in segment height per iteration through “i”. Since segments are vertically divided, we move along y axis. Then we call “check_collision” again with ball and the respective segment per iteration in for loop and if the function returns true, i.e. the ball collides with the particular segment, then we set our states accordingly. The same logic is implemented in case of “right racket”, which is represented by the “elsif” part. In the end both of “hit_racket_l” and “hit_racket_r” are nullified cause if there ‘s a collision with one of the rackets, then of course there is no collision with other parts.

If there is no collision with both the rackets, we move in “else” condition, where we check for wall collisions.

```
        else
                -- the ball does not collide with any rackets
                hit_racket_l <= (others => '0');
                hit_racket_r <= (others => '0');
                hit_wall <= (others => '0');

                -- check if the ball collides with wall
                if (ball.x < (racket_l.x + 10)) then
                        hit_wall <= "110"; -- collides with left wall
                elsif (ball.x + ball.width) > (racket_r.x) then
                        hit_wall <= "101"; -- collides with right wall
                elsif ball.y <= 0 then
                        hit_wall <= "010"; -- collides with top wall
                elsif (ball.y + ball.height) >= (screen_height - 1) then
                        hit_wall <= "001"; -- collides with bottom wall
                end if;
        end if;
end process collision_proc;
```

The inner logic inside “if” condition checks if ball collided with the respective wall or not. For e.g. “ball.x” is the x coordinate of ball’s top left corner, “racket_l.x + 10” can be

thought of as a buffer zone near the left racket, if the ball's x-coordinate is less than this buffer zone, it means the ball has collided with the left wall.

Using buffer, we check the respective collisions similarly in all the other cases of wall collisions. As soon as a condition is satisfied, i.e collision detected, we update our states accordingly. And all this takes place per iteration in the “for loop”.

7. Registering Collision Signals

```
        DFF_N_Wall : entity work.D_FlipFlop_NBits
                generic map(N => 3)
                port map(
                        clk => clock_i,
                        rst => reset_i,
                        D => hit_wall,
                        Q => hit_wall_o);

        -- Flipflop for storing left racket collision
        DFF_N_Racket_L : entity work.D_FlipFlop_NBits
                generic map(N => 2)
                port map(
                        clk => clock_i,
                        rst => reset_i,
                        D => hit_racket_l,
                        Q => hit_racket_l_o);

        -- Flipflop for storing right racket collision
        DFF_N_Racket_R : entity work.D_FlipFlop_NBits
                generic map(N => 2)
                port map(
                        clk => clock_i,
                        rst => reset_i,
                        D => hit_racket_r,
                        Q => hit_racket_r_o);
end Behavioral;
```

The final part registers the collision signals using D flip-flops to ensure the outputs are assigned and synchronized with the clock**.**

8. Testbench

“tb_Collision_Detection” is our Testbench. The purpose of the testbench is to validate the functionality of the Collision_Detection module by simulating various scenarios and ensuring that the module's output matches the expected results. The testbench will generate clock signals, apply test vectors to the inputs, and check the outputs using assertions.

9. Testbench Implementation

The entity “tb_Collision_Detection” has no ports because it is solely for simulation. The architecture Behavioral includes constants for the game dimensions, signals to connect to the DUT, a clock generation process, and a test process. Constants define dimensions for the ball, rackets, and screen. Signals are used to connect the testbench to the Collision_Detection module.

```
entity tb_Collision_Detection is
--  Port ( );
end tb_Collision_Detection;

architecture Behavioral of tb_Collision_Detection is
    -- Constants for the testbench
    constant ball_length : integer := 6;
    constant racket_length : integer := 10;
    constant racket_height : integer := 30;
    constant racket_left_space : integer := 20;
    constant racket_right_space : integer := 610;
    constant screen_height : integer := 480;
    constant screen_width : integer := 640;

    -- Signals for the DUT (Device Under Test)
    signal reset_i : std_logic := '0';
    signal racket_y_pos1_i : std_logic_vector(9 downto 0) := (others => '0');
    signal racket_y_pos2_i : std_logic_vector(9 downto 0) := (others => '0');
    signal ball_x_i : std_logic_vector(9 downto 0) := (others => '0');
    signal ball_y_i : std_logic_vector(9 downto 0) := (others => '0');
    signal hit_wall_o : std_logic_vector(2 downto 0);
    signal hit_racket_l_o : std_logic_vector(1 downto 0);
    signal hit_racket_r_o : std_logic_vector(1 downto 0);

    -- Clock generation
    constant clock_period : time := 10 ns;
    signal clock : std_logic := '0';
begin
```

The Collision_Detection module is instantiated with its generics and ports mapped to the testbench signals.

```
  -- Instantiate the Unit Under Test (UUT)
  uut: entity work.Collision_Detection
      generic map (
          ball_length => ball_length,
          racket_length => racket_length,
          racket_height => racket_height,
          racket_left_space => racket_left_space,
          racket_right_space => racket_right_space,
          screen_height => screen_height
      )
      port map (
          clock_i => clock,
          reset_i => reset_i,
          racket_y_pos1_i => racket_y_pos1_i,
          racket_y_pos2_i => racket_y_pos2_i,
          ball_x_i => ball_x_i,
          ball_y_i => ball_y_i,
          hit_wall_o => hit_wall_o,
          hit_racket_l_o => hit_racket_l_o,
          hit_racket_r_o => hit_racket_r_o
      );
```

A simple clock generation process that toggles the clock signal every half-period.

```
-- Clock process
clock_process :process
begin
    clock <= '0';
    wait for clock_period/2;
    clock <= '1';
    wait for clock_period/2;
end process clock_process;
```

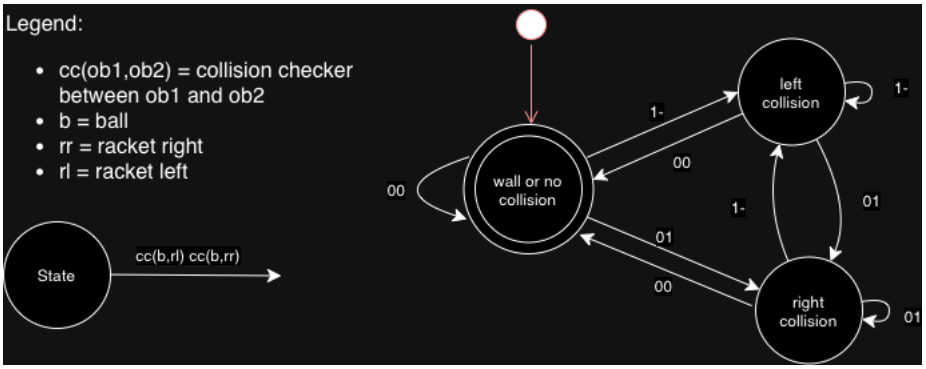
The test process initializes the inputs, applies test vectors, and checks the outputs using assertions.

```
-- Test process
test_process : process
begin
    -- Initialize Inputs
    reset_i <= '1';
    wait for clock_period;
    reset_i <= '0';
    wait for clock_period;

    -- Test collision with left racket
    ball_x_i <= std_logic_vector(to_unsigned(racket_left_space, 10));
    ball_y_i <= std_logic_vector(to_unsigned(15, 10));
    racket_y_pos1_i <= std_logic_vector(to_unsigned(10, 10));
    wait for clock_period;
    assert (hit_racket_l_o = "01") report "Collision with left racket top segment not detected" severity error;
```

For. e.g. This Case focuses on whether the ball hits the top segment of the left racket, i.e 1. Segment. We firstly set Ball Position and Left Racket Position, then wait for some clock period and in assertion we check if the output signal “hit_racket_l_o” correctly indicates a collision with the top segment of the left racket. Similarly, the rest of the cases are implemented.

10. State Diagram



Our implementation does not use any "type-record state", instead we only use if-else statement to determine the output of hit_wall and hit_racket. Each state represents a group of outputs. This means that within the state there is further logic that checks which output should be assigned.

The process starts with the state "no_collision or no collisions" and every output is set to value 0. If the function "collision checker" sees a collision between "left racket and ball", no matter the value of the function between "right racket and ball", it always goes into the

left collision. Only if the function detects a false collision between the "left racket and the ball" will it either switch to the "right collision" or stay in the "wall or no collision". Within the "wall or no collision" state, there is another logic to determine what the "hit_wall" should be

11. Challenges

We had a big problem to understanding “double counting” and the glitch with collision between left and right wall and the ball. We actually thought that the whole synchronous circuit might be the problem. As we try to analyze it, it is not actually the synchronous circuit concept that is the problem, but the double counting inside the right and left wall. We see that the ball always tries to move 5-10 pixels to the right and left, depending on the collision position with the wall. Also, the code in the ball movement has an "unusual" approach when the right and left wall collide with the ball, so we thought that increasing the boundary might help to get the game to work.

12. Conclusion

This documentation provides a comprehensive overview of the Collision_Detection module and its testbench, detailing the design and functionality of the collision detection logic. The systematic verification process ensures accurate detection of Ball-collisions with the rackets and walls.

13. Bibliography

Gillan, J. (16. January 2016). *Algorithm Practice: Rectangle Intersection*. Von Medium: https://medium.com/@jessgillan/algorithm-practice-rectangle-intersection-7821411fd114 abgerufen