University of Liege

faculty of applied sciences

ELEN0040-1 Digital Electronics VHDL project

Space Shooter Game : Group 19

Academic Year : 2023-2024

EL FEZZAOUI Hiba : s215439
BOUSTANI Mehdi : s221594
ALBASHITYALSHAIER Abdelkader : s211757
GANOUBONG TINGUE LESLIE Lucynda : s218152
GIOT Lucas : s212131

Professor : J-M. Redouté
Assistant professor : A. Fyon

Table des matières

1 Introduction **2**

2 Hardware **2**
2.1 LED Matrix . . . 2
2.2 Microswitches . . . 2
2.3 Decoders and 7-Segment Displays . . . 3
2.4 Breadboards and wiring . . . 4
2.5 RGB LED . . . 4

3 Electronic schematic **5**

4 Code Structure and Description **6**
4.1 **State machine diagram** . . . 6
4.2 Inputs : . . . 6
4.3 Outputs : . . . 7
4.4 Key signals, variables, and constants : . . . 7
4.5 Code . . . 8
4.5.1 Entity declaration . . . 8
4.5.2 Architecture . . . 8
4.5.3 Optimization . . . 9
4.5.4 Conclusion . . . 10

Complete code **11**

1 Introduction

This project aims to create a simplified version of a space shooter game, employing a tricolor LED matrix for visual output. The player controls a spaceship represented by a single LED. Similarly, obstacles are also depicted by single descending LEDs. The colors of the player will be dictated by how many lives are left : if all three lives are intact, the color of the LED representing the player will be green, if only two are left then it will be orange, and if the player has only one life left, then the color becomes red.

The objective is to achieve a certain score by shooting down descending obstacles. The spaceship's movement is limited to horizontal motions — either left or right. If more than two obstacles pass the spaceship without being destroyed, the player loses, prompting a game over. If the player however reaches a maximum score of 50, then the game is won.

To heighten the challenge, obstacles will vary in speed with every tenth hit, progressively increasing the difficulty of achieving the maximum score.

2 Hardware

2.1 LED Matrix

The game interface is presented on a 5 × 7 tricolor LED matrix. In order to light up a single LED, the row containing that LED will be set to 1 while the corresponding column will be set to 0.

In this LED matrix, the rows have their anodes connected, and the columns have their cathodes connected, which means that lighting up different LEDS in specific rows and columns can be tricky. For example, lighting up a single LED or the entire matrix at once wouldn't be hard, but switching on two different LEDS in two adjacent columns and rows would lead to more LEDS being switched on than just ones we wanted.

It is, however, not impossible to do this, by choosing to display each wanted row separately at a very fast rate, it looks like all the LEDS we need are switched on at the same time to the unassisted sight.

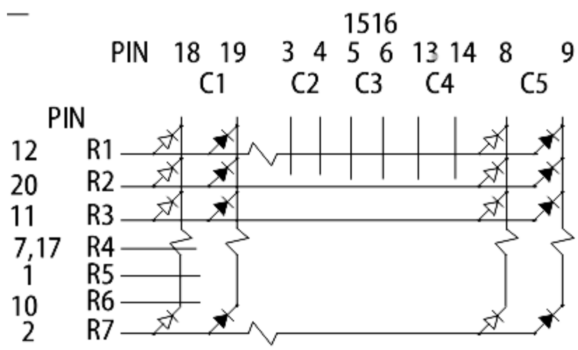


Figure 1 – LED matrix Pins

2.2 Microswitches

We were initially considering using buttons but we discovered that microswitches had better performance and were able to easily and quickly detect the player's input, either right, left, shoot or control, especially in cases requiring quick and repetitive switching, which the buttons couldn't handle properly.

These microswitches serve the same purpose as buttons, they allow the passage of the current when they are pressed, enabling thus the detection of the player's input based on which switch was pressed : Left, right, shoot or control. A hardware filter was also not necessary as there was no significant rebound observed.

And finally to ensure stability, each microswitch had a pull down resistor of around 10k Ohm, to make sure that the signal holds close to zero and is not floating when the switch is not pressed.

FIGURE 2 – An alcoswitch

2.3 Decoders and 7-Segment Displays

The 7-segment displays are used for two things in this game :

- The first use is that the right 7-segment display shows the difficulties that the player can choose from, ranging from 1 : easy mode to 3 : extra hard mode.
- The second use is that they indicate the player's score, incremented with each successful obstacle hit. The highest possible score in this game is 50, which is the winning score.

Two decoders accompany these 7-segment displays to lower the number of pins and allow for an easier way to display the numbers needed by converting binary-coded inputs into corresponding outputs.

Four pins, labeled from A to D, of the decoder represent the four bits of the binary number that is transmitted from the CPLD.

The pins a to g of the other side of the decoder are connected to the corresponding pins on the 7 segment display via 1k resistors. This ensure that we stay within the specified current limitations of both the decoders and the displays. The dot pin of the seven segment remains untouched as it is not used and the middle pins of the displays are connected to the ground.

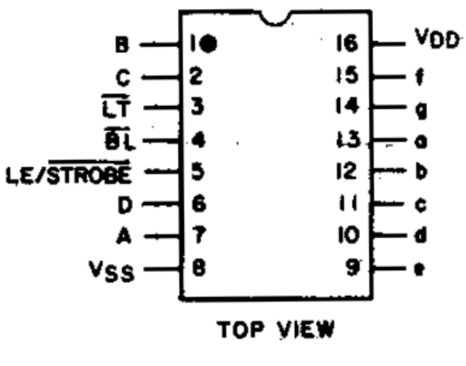


FIGURE 3 – Decoder pins

2.4 Breadboards and wiring

Three breadboards have been used to make everything in this game, the wiring was made to be as unobtrusive and close to the breadboard as possible so as not to cause any problems.

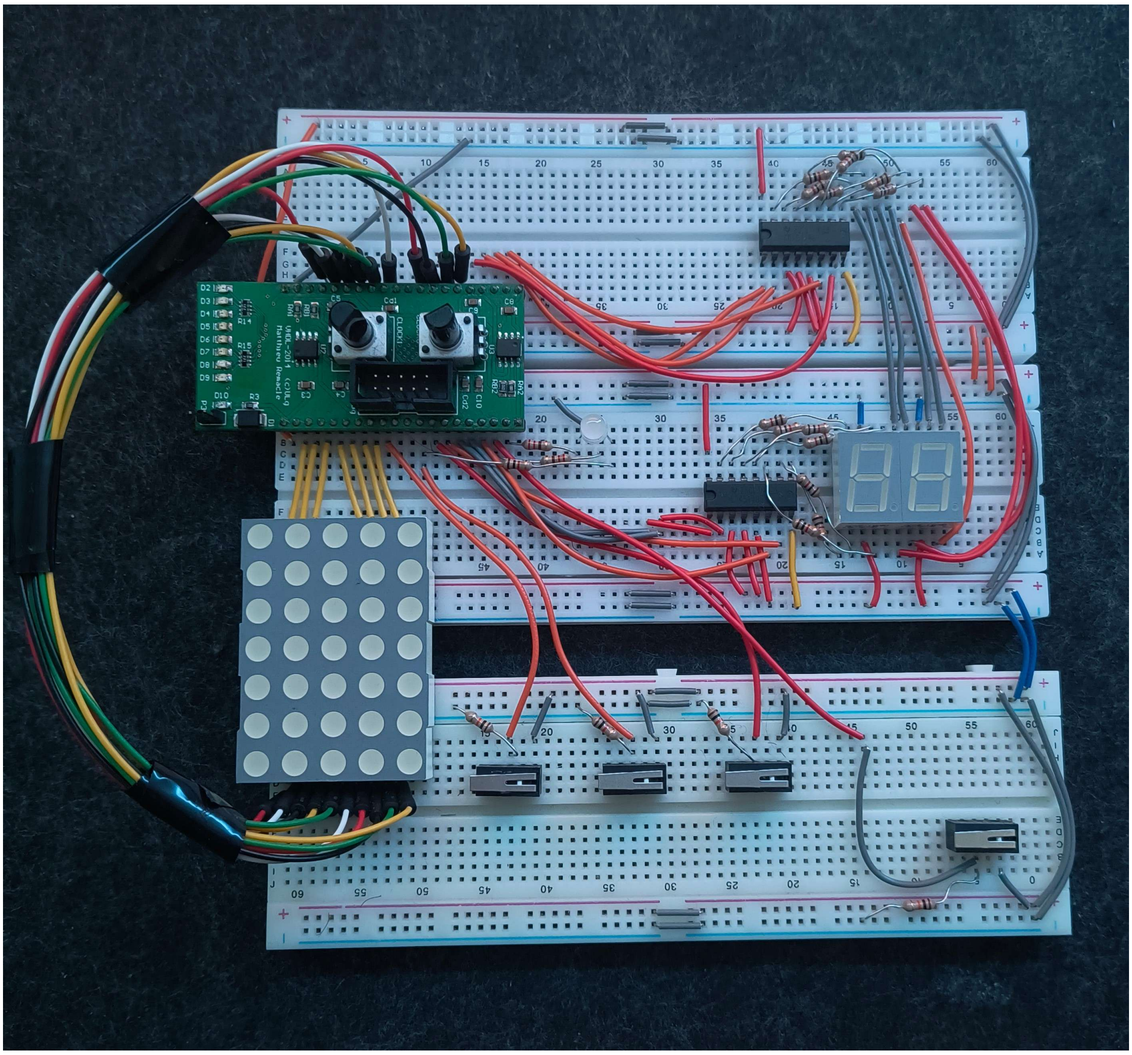


FIGURE 4 – The breadboards, the components and the wiring

2.5 RGB LED

This common cathode RGB LED has four leads, R, G, B or ground, that can be distinguished by their different lengths.

The ground led is, naturally, connected to the ground. The red led, on the other hand, is connected to the CPLD via a 220 Ohm resistor, this led blinks red when the game over state is reached. The blue led is triggered whenever the reset switch is pressed, and is regulated with a 330 Ohm resistor. Similarly, the last led, which is the green one is triggered with each press of the shoot switch is pressed and is connected to a 1 k resistor.

The different resistors are chosen in an attempt to keep the brightness of the different colors on the same level.

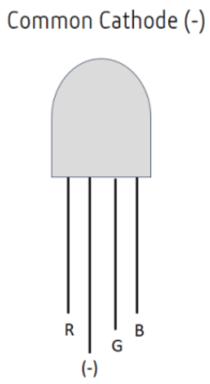
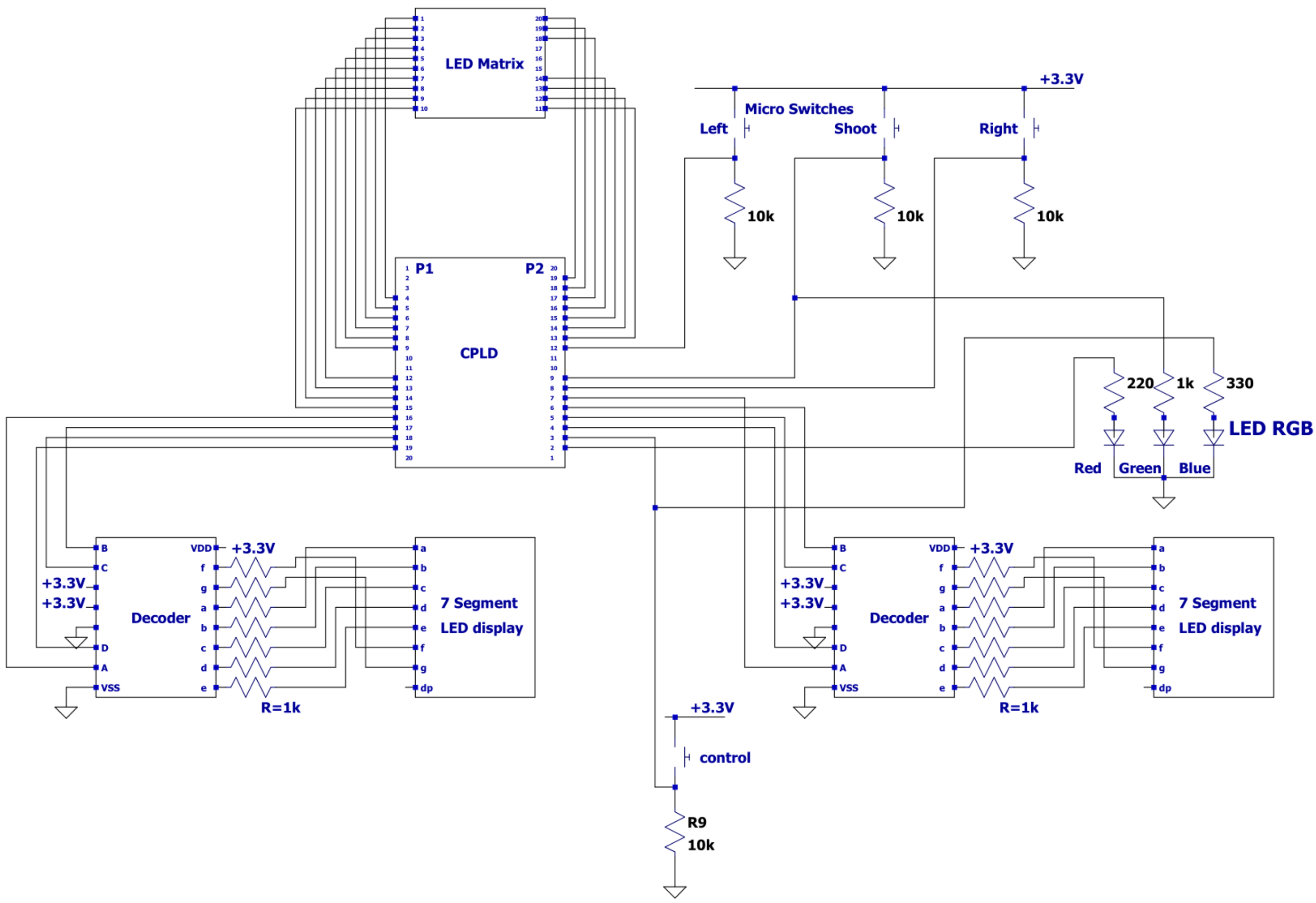


FIGURE 5 – Common cathode RGB LED

3 Electronic schematic

The following picture depicts the circuit topology of our project, it contains all of the connections used and the correct pins for each component.

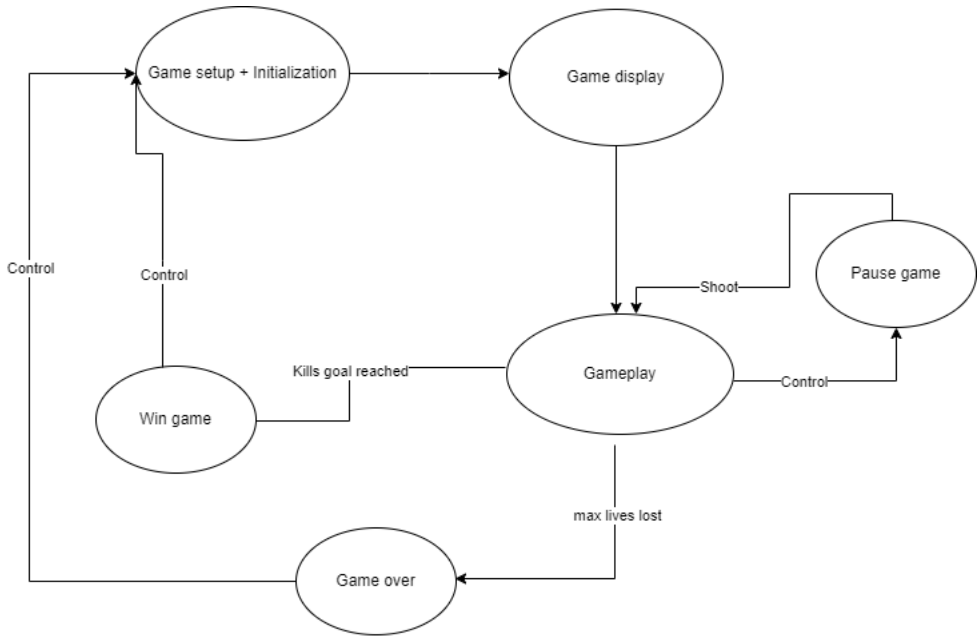


4 Code Structure and Description

Our code structure involves several key steps, each corresponding to different states and functionalities within the game :

- **Game Setup :** Initializes the game based on a difficulty level, positioning the player at the bottom center of the screen and placing the first obstacle randomly on top. We begin by defining the entity, including declaring inputs, outputs, and variables.
- **Game Initialization :** At the start of the game, we set the score to zero and initialize the number of lives to 3. This phase also involves setting up the game board and placing the player's spaceship and initial obstacles.
- **Game Display :** Renders various game elements on the LED matrix according to their positions on the game board.
- **Gameplay :** Continuously updates states by :
  - Moving obstacles and adjusting their speed as necessary.
  - Monitoring switch presses to control spaceship movement (left/right) or shooting, control as well in case the player chooses to pause the game.
  - Checking if obstacles were successfully hit before reaching the end ; decrement lives if not.
  - Updating the score based on successful hits.
- **Pause game :** Pauses the game when the player presses the control switch and resumes it when the shoot switch is pressed.
- **Game end :** This state has two possibilities :
  - **Win game :** Declares the player as the winner upon reaching the target score.
  - **Game Over :** Concludes the game if the player exhausts all lives, reaching the maximum lives.

4.1 State machine diagram



4.2 Inputs :

- `clk_fast` : The faster clock : the frequency used for this clock is 266 hz.

- `clk_slow` : The slower clock : the frequency used for this clock is 90 hz.
- `move_left` : This is the input of the left switch and it lets the player move to the left.
- `move_right` : This is the input of the right switch and it lets the player move to the right.
- `shoot` : This refers to the input of the shoot switch, it can shoot obstacles and resume the game after it has been paused.
- `control` : This is the input coming from the control switch, that can start the game, pause when it's playing and reset it at the end.

4.3 Outputs :

- `rows` : This represents the output that determines whether the rows are going to be on 0 or on 1.
- `cols_green` : This represents the output that determines whether the columns that control the green lighting of the LED matrix are going to be on 0 or on 1.
- `cols_red` : This represents the output that determines whether the columns that control the red lighting of the LED matrix are going to be on 0 or on 1.
- `tens` : This output controls the tens part of the score.
- `unit` : This output controls the units part of the score.
- `led_end` : This output controls whether the RGB LED will be switched on or not.

4.4 Key signals, variables, and constants :

The **types** used in this game, the one that holds all the different states the game could be :

- `GameState` : This type has three different states : MENU, PLAY, PAUSE and GAMEOVER.

The **signals** used in this game are :

1. The general game logic signals are :
   - `state` : This signal represents the current game state.
   - `not_action` : This signal is a boolean that shows whether any switch has been pressed, it's true by default and false if a switch is pressed.
   - `lives` : Integer range 0 to 3, this signal represents the current amount of lives that the player has.
   - `end_led_state` : This signal shows whether the led should be turned off or not.
   - `win` : This signal is a boolean that indicates whether a player has reached the winning score : 50 or not.
   - `difficulty` : Integer range 0 to 3, this signal represents the difficulty chosen for the game.

2. The signals used for the LED matrix display :
   - `bullet_row` : Integer range 0 to 6, this signal represents the row that the bullet is currently in.
   - `obstacle_row` : Integer range 1 to 7, this signal represents the row that the obstacle is currently on.
   - `obstacle_col` : Integer range 1 to 5, this signal represents the column that the obstacle is currently on.
   - `rand_col` : Integer range 1 to 5, this signal gives a random column number for the obstacles to spawn in.

- `row_counter` : natural range 1 to 7, This signal is used to facilitate the display on the menu.
- `switch` : This signal indicates what is going to be displayed at each rising edge of the clock, the player, the obstacles or the bullets.

3. The signals used to manage the bullets, obstacles and collision are :
   - `bullet` : This signal is a boolean that indicates if a bullet has been shot or not.
   - `collision` : This signal is a boolean that indicates if there is a collision between the bullet and the obstacle.
   - `counter` : This signal is an integer that serves the purpose of managing the speed of both the RGB LED as well as the obstacles.
   - `obstacle_speed` : Integer 1 to OBSTACLE_SPEED_MAX, this signal represents the speed of the obstacles.
   - `delete_obstacle` : This signal is a boolean that indicates if an obstacle should be deleted.

4. The signals used for the seven segment display are :
   - `tens_score` : This signal is an integer that shows the current tens part of the number to be displayed on the seven segment display.
   - `units_score` : This signal is an integer that shows the current units part of the number to be displayed on the seven segment display.

The **variables and constants** used in this game are :

- `player_row` : This constant shows that player is only able to move horizontally.
- `GRID_WIDTH` : This constant represents the width of the LED matrix.
- `GRID_HEIGHT` : This constant represents the height of the LED matrix.
- `OBSTACLE_SPEED_MAX` : This constant represents the lowest possible speed.
- `LIVES_MAX` : This constant represents the maximum number of lives which is 3.
- `MAX_SCORE` : This constant represents the highest number possible for the tens part of the score which is 5.

4.5 Code

4.5.1 Entity declaration

During this phase, the entity **SpaceShooter** is declared, along with the different inputs and outputs mentioned above and the ones needed for the game to work properly, from the player's movement to the clock's input.

4.5.2 Architecture

The architecture **space_arch** of the entity SpaceShooter is declared alongside all the signals needed to manage the internal workings of the game.
This architecture defines three processes that refresh at each rising edge of the slower clock :

- `Logic` : This process handles the different states of the game :

  The **start menu** allows the player to choose between three difficulties : the easy difficulty with three lives and obstacles that start with a manageable speed, the hard difficulty, in which the player still has three lives, but the speed of the obstacles is higher and the extra hard difficulty with fast obstacles and only one life. The player can achieve this by pressing the left and right switches to choose a difficulty between 1 and 3 on the right 7 segment display.

The **play** state starts when the player presses the shoot switch to start the game, the process checks for the player's input, if either the right or left switch is pressed, then the corresponding move is immediately executed. When the shoot switch is pressed, then the signal bullet becomes true, and its ascending trajectory is displayed and treated. The obstacles are then scanned to see if there is a collision happening, if there is, then the obstacle is deleted and the score is incremented, otherwise it continues its descent. If the obstacle reaches the end without being shot, the player loses a life, and the obstacle's position is reset.

The **pause** state is triggered by pressing the control switch in the **play** state, then it detects if shoot is pressed after, in which case the game is resumed once more.

In the **game end** state, the RGB LED will start blinking in red to specify that the player has won the game otherwise the RGB LED will just turn red and stay that way without blinking, which indicates that the player has lost the game. The player can then press the control switch and proceed to the menu to choose a new difficulty.

- `Random` : This process handles the random number generator for the column numbers, it's executed with each rising edge of the faster clock.

- `Display` : This is the process that handles the display of each different LEDs in the matrix. This is done with the use of the variable : **switch** and is executed with each rising edge of the faster clock.

  When switch is at 1, and so the first rising edge of the faster clock, then the player's position is the one that's being displayed. The concerned row is at 1 while the others are 0 and vice versa for the columns. The defining color of the player is however determined by the number of lives left, and is done by putting the columns needed for the color at 0.

  When switch is at 2 so at the second rising edge of the clock, if the obstacle is present, then its position is the one that's being displayed by following the same logic as before.

  And finally when switch is at 3, if a bullet has been fired then the bullet's position is the one displayed by following the previous steps.

  If the player has won the game then we display the matrix as fully green, if the game is lost, however, then we display it as fully red instead.

4.5.3 Optimization

There are a number of things that we have done to optimize the code and stay within the limit of the logic gates. One significant optimization is the utilization of a single counter to manage the clock throughout the entire game to manage the clock, so for both the obstacles as well as the LED RGB.

The second thing we have optimized is the way that the score is handled. We could have looked at the score as a single number, in which case we would have to derive the tens and units to be able to separate them before displaying them on the seven segment displays, however this would require additional logic gates since we would have to use division and modulo operations.

Instead, we decided to increment the units individually and whenever they have to be incremented after they reach 9, the tens are immediately incremented and the units part goes back to 0.

The third important optimization that we did, is the fact that we used both the shoot as well as the control switch for different applications throughout the game. This ensures that we don't use more unnecessary switches, when the control and shoot switches can respectively also act as the pause and resume switches, as well as the start and reset switches.

And finally the last optimization was the use of a signal for the **win game** state, instead of adding it to the **GameState** type which would have used up a lot more logic gates than just adding both win and game over into the game end state and giving them different characteristics.

4.5.4 Conclusion

This project has taught us quite a few new things, from learning how to manipulate hardware components, to understanding how to implement the game logic.
The visual outputs from the LED matrix as well as the 7-segment displays have allowed us to create an immersive gameplay experience while still staying within the restraints of the CPLD.

We have finished the game with exactly 0 logic gates left. If we had more, we could have tried to make the game even more immersive but alas.
All in all, this project has allowed us to understand how electronics work on a deeper level and how we can use it to make different applications based on what is needed in the future.

Complete code

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

entity SpaceShooter is
    port (
    -- Inputs
    clk_fast, clk_slow, move_left, move_right, shoot, control : in std_logic;

    -- Outputs

    -- Tricolor Matrix
    rows : out bit_vector(1 to 7);
    cols_green, cols_red : out bit_vector(1 to 5) := (others => '1');

    -- 7-segment display
    tens, unit: out std_logic_vector(3 downto 0) := (others => '0');

    -- RGB LED
    led_end: out std_logic

    );
end SpaceShooter;

architecture space_arch of SpaceShooter is

  -- Constants
  constant GRID_WIDTH : integer := 5;
  constant GRID_HEIGHT : integer := 7;
  constant OBSTACLE_SPEED_MAX : integer := 18;
  constant LIVES_MAX : integer := 3;
  constant MAX_SCORE : integer := 5;

  -- Game states
  type GameState is (MENU, PLAY, END_GAME, PAUSE);
  signal state : GameState := MENU;
  signal win : boolean := false;

  -- Player positions
  signal player_col : integer range 1 to GRID_WIDTH := 3;
  constant player_row : integer := GRID_HEIGHT;

  -- Game logic
  signal not_action : boolean := true; -- This boolean is false if a button has been pressed (moves, shoot & control)
  signal lives : integer range 0 to LIVES_MAX := LIVES_MAX; -- Number of lives of the player: 3 initially
  signal difficulty : integer range 1 to 3 := 1; -- Difficulty of the game

  -- Player shots
  signal bullet_row : integer range 0 to 6 := 6; -- Current row of the bullet fired by the player
  signal bullet : boolean := false; -- Indicates if our player has fired or not
  signal collision : boolean := false; -- Indicates if an obstacle has been hit or not

  -- Obstacles
  signal obstacle_row : integer range 1 to GRID_HEIGHT := 1; -- Vertical position of the obstacle
  signal obstacle_col : integer range 1 to GRID_WIDTH; -- Horizontal position of the obstacle
  signal obstacle_speed : integer range 1 to OBSTACLE_SPEED_MAX := OBSTACLE_SPEED_MAX; -- Speed of movement of the obstacle
```

```
signal rand_col : integer range 1 to GRID_WIDTH := 1; -- Random column for the obstacle

-- Facilitate matrix display according to current frequency
signal counter : integer range 0 to 20; -- Index of the current line for the menu display
signal switch : natural range 1 to 3; -- Switch for alternating between obstacles and the player

-- 7 segments (0 to 50)
signal tens_score : integer range 0 to MAX_SCORE;
signal units_score : integer range 0 to 9;

-- Controls the state of the RGB LED
signal end_led_state : std_logic := '0';

begin

  -- Game logic process responsible for managing the game's logic based on user input
  logic : process(clk_slow, clk_fast, move_left, move_right, shoot, control)
  begin
      if rising_edge(clk_slow) then

      case state is

        when MENU =>

          -- RESET
          led_end <= '0';
          tens_score <= 0;
          units_score <= 0;
          player_col <= 3;
          counter <= 0;
          win <= false;

          -- Cheking if the user pressed a button or not
          if(move_right = '0' and move_left = '0') then
            not_action <= true;

          -- Choosing the difficulty
          elsif(not_action) then
            not_action <= false;

            if move_left = '1' then
              if(difficulty > 1) then
                difficulty <= difficulty - 1;
              end if;

            elsif move_right = '1' then
              if(difficulty < 3) then
                difficulty <= difficulty + 1;
              end if;
            end if;

          end if;

          -- Initiating the game upon pressing the shoot button
          if(shoot = '1') then
            state <= PLAY;
          end if;

          -- During difficulty selection, only the units are utilized
          tens <= "0000";
          unit <= std_logic_vector(to_unsigned(difficulty, 4));

```

```
          -- Adjusting the game speed and number of lives based on the selected difficulty
          case difficulty is
            when 1 =>
              obstacle_speed <= OBSTACLE_SPEED_MAX;
              lives <= LIVES_MAX;
            when 2 =>
              obstacle_speed <= 12;
              lives <= LIVES_MAX;
            when 3 =>
              obstacle_speed <= 12;
              lives <= 1;
          end case;

        when PLAY =>

          -- Cheking if the user pressed a button or not
          if(move_right = '0' and move_left = '0' and shoot = '0') then
            not_action <= true;

          elsif(not_action) then
            not_action <= false;

            -- Move the player left
            if move_left = '1' then
              if player_col > 1 then
                player_col <= player_col - 1;
              end if;

            -- Move the player right
            elsif move_right = '1' then
              if player_col < GRID_WIDTH then
                  player_col <= player_col + 1;
              end if;

            -- Shoot
            elsif shoot = '1' then
              bullet <= true;
            end if;

          end if;

          -- Moves the bullet upward if the user fires
          if(bullet) then
            if(bullet_row > 1) then
              bullet_row <= bullet_row - 1;
            else
              bullet_row <= 6;
              bullet <= false;
            end if;
          end if;

          -- If the obstacle reaches the bottom of the matrix, reset its position and decrease a player's life
          if(obstacle_row = GRID_HEIGHT) then
            lives <= lives - 1;
            obstacle_row <= 1;
            obstacle_col <= rand_col;

          else
            -- If there is no collision, the obstacle can move downward
            if not collision then

              counter <= counter + 1;
              -- Move the obstacle downward according to the slowing down factor
```

```
              if counter = obstacle_speed then
                counter <= 0;
                obstacle_row <= obstacle_row + 1;
              end if;

            end if;
          end if;


          if(collision) then

            -- Manually increment the score to save logic gates (optimization)
            if units_score = 9 then
              tens_score <= tens_score + 1;
              obstacle_speed <= obstacle_speed - 2;
              units_score <= 0;
            else
              units_score <= units_score + 1;
            end if;

            -- Reset the obstacle and the bullet
            obstacle_row <= 1;
            obstacle_col <= rand_col;
            bullet <= false;
            bullet_row <= 6;

          end if;


          -- Update the 7-segment display based on the score
          tens <= std_logic_vector(to_unsigned(tens_score, 4));
          unit <= std_logic_vector(to_unsigned(units_score, 4));

          if(lives = 0) then
            state <= END_GAME;
          end if;

          if(control = '1') then
            state <= PAUSE;
          end if;

          -- The user reached the final score
          if(tens_score = MAX_SCORE) then
            win <= true;
            state <= END_GAME;
          end if;


        when END_GAME =>

          -- If the user won, the RGB LED blinks in red
          if(win) then
            counter <= counter + 1;
            if(counter = 20) then
              counter <= 0;
              end_led_state <= not end_led_state;
              led_end <= end_led_state;
            end if;
          -- If the user lost, the RGB LED is continuously red
          else
            led_end <= '1';
          end if;

          if(control = '1') then
            state <= MENU;
          end if;
```

```
        when PAUSE =>

          -- Press the shoot button to resume the game
          if(shoot = '1') then
            state <= PLAY;
          end if;

      end case;
    end if;
  end process logic;

-- Random process for the obstacle column
random : process(clk_fast)
begin
  if rising_edge(clk_fast) then
    if rand_col = GRID_WIDTH then
      rand_col <= 1;
    else
      rand_col <= rand_col + 1;
    end if;
  end if;
end process random;

-- Display process to show elements on the tricolor matrix
display : process(clk_fast, control)
begin

  if rising_edge(clk_fast) then
    case state is
      when PLAY | PAUSE =>
         rows <= (others => '0');
         cols_green <= (others => '1');
         cols_red <= (others => '1');

         -- If the bullet collides with an obstacle then it's true, otherwise it's false
         collision <= ((((bullet_row - 1 = obstacle_row) or bullet_row = obstacle_row) and player_col = obstacle_col) and bullet);

         case switch is

           -- On the first rising edge, display the player according to their state
          when 1 =>

            -- Determine the player's display color based on the number of lives (3 -> Green, 2 -> Orange, 1 -> Red)
            case lives is
              when LIVES_MAX =>
                 cols_green(player_col) <= '0';
              when 2 =>
                 cols_green(player_col) <= '0';
                 cols_red(player_col) <= '0';
              when 1 =>
                 cols_red(player_col) <= '0';
              when others =>
            end case;

            -- Displaying only the player's row
            rows(player_row) <= '1';

          -- On the second rising edge, display the obstacle
          when 2 =>

            cols_red <= (others => '1');
```

```
              rows <= (others => '0');
              rows(obstacle_row) <= '1';
              cols_red(obstacle_col) <= '0';

            -- On the third rising edge, display the player's shot if fired
            when 3 =>
              if bullet then
                rows(bullet_row) <= '1';
                cols_red(player_col) <= '0';
              end if;

          end case;

          switch <= switch + 1;

        when END_GAME =>

          -- Activate all lines of the tricolor matrix
          rows <= (others => '1');

          -- Suppress the display of the next obstacle
          cols_red(obstacle_col) <= '1';

          -- Display the tricolor matrix in green if the user wins, red if the game is lost
          if(win) then
            cols_green <= (others => '0');
          else
            cols_green <= (others => '1');
            cols_red <= (others => '0');
          end if;

        when others =>

          -- Reset the tricolor matrix when the user is in the menu
          rows <= (others => '0');
          cols_green <= (others => '1');
          cols_red <= (others => '1');

      end case;
    end if;
  end process display;

end architecture space_arch;
```

Listing 1 – VHDL code