POLITECNICO DI BARI

ELECTRICAL AND INFORMATION ENGINEERING DEPARTMENT

MASTER'S DEGREE COURSE IN

AUTOMATION ENGINEERING - ROBOTICS

SUBJECT

DIGITAL PROGRAMMABLE SYSTEMS

AVOID CLASH GAME ON FPGA BOARD

PROFESSOR:

Martino De Carlo

STUDENTS:

Mariapaola Germinario

Gianluca Latanza

Academic Year 2023 −2024

Table of Contents

Abstract

The purpose of the project is to create a game using the Terasic DE10-Lite board. To this end, the game “Avoid Clash” was developed, with the objective of avoiding obstacles falling from above. It was used VHDL language in order to create the game logic and to control the hardware resources.

The project demonstrated the feasibility of developing arcade games on FPGA and provides a foundation for future improvements.

This report will analyze the hardware and software tools used, as well as the game logic and structures implemented in VHDL.

1 Introduction

“Avoid Clash” comes from a project which has the aim of developing a game in VHDL language using ad FPGA board and some additional hardware.

The player must move a paddle, located at the bottom of the screen, to avoid obstacles coming from above. The movements are limited to horizontal directions, controlled by a joystick.

The screen display is managed through the VGA port of the FPGA board.

The game begins with the paddle positioned in the center of the play area and the first obstacle descending. The player has 3 lives, indicated by three LEDs on the board, which are lost each time the paddle hits an obstacle. To win a game, the player must reach 100 points; the score is time-based, with one point added for each second of gameplay, and is displayed on three 7-segment displays on the board. If all lives are lost before reaching 100 points, a red screen is shown, the score is frozen, and the game can be restarted using a switch on the board.

When the reset is on, the game is stopped, displaying a black screen, and both lives and score are reset to their initial values.

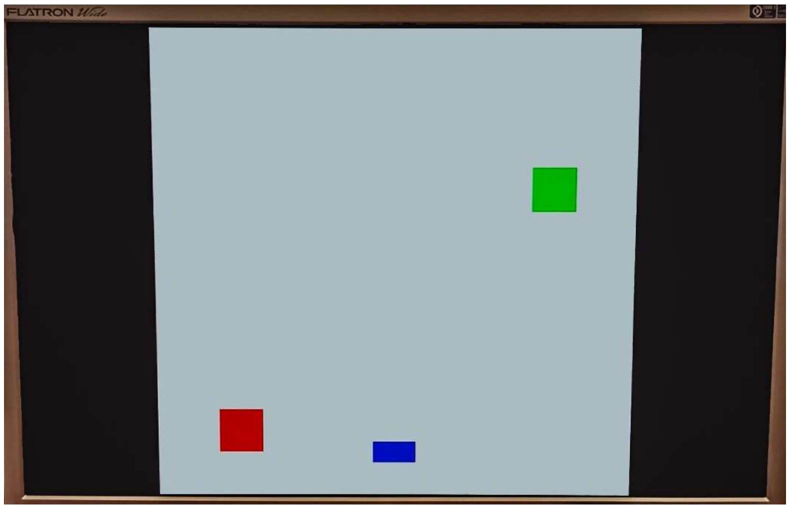


Figure 1: Avoid Clash

2 Tools

2.1 Hardware

The main component is the Terasic DE10-Lite FPGA board. Additionally, the following hardware components are used for game development:

1. A-to-B USB cable
2. Analog joystick module
3. Female-to-Male Dupont wires
4. VGA-to-VGA cable
5. VGA Screen.

2.1.1 FPGA board

The Intel DE10-Lite Altera MAX 10 board is a development platform designed for learning, prototyping, and creating FPGA-based projects. It is built around the Intel MAX 10 FPGA, a low-cost, low-power programmable device that features approximately 50,000 logic elements (LE) and an integrated analog-to-digital converter (ADC), high-quality integrated USB-Blaster, SDRAM, accelerometer, VGA output, and a 2x20 GPIO expansion connector, which provides a means to connect external devices and sensors for extended functionality.

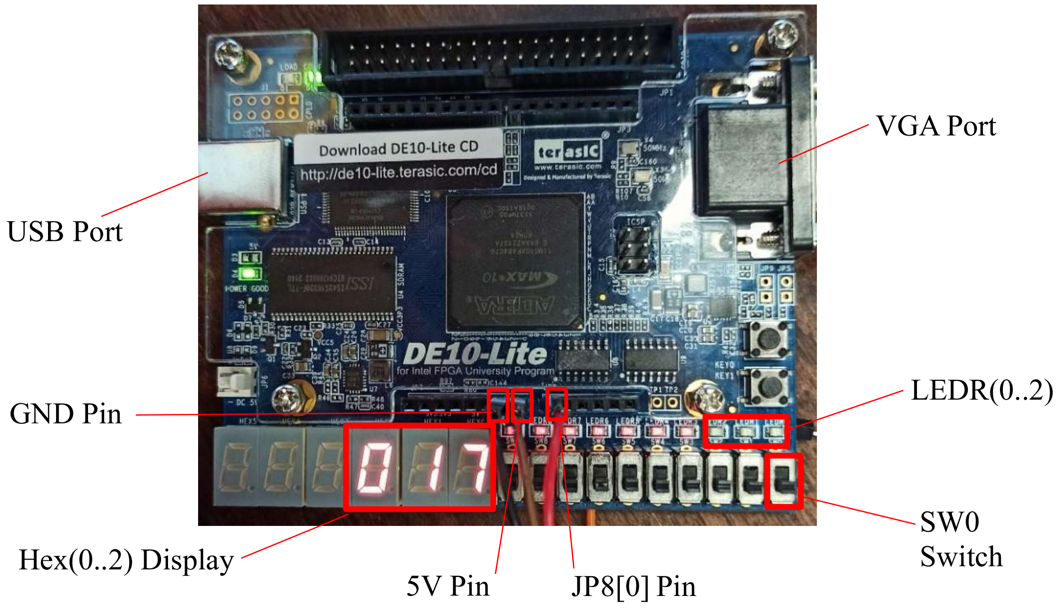


Figure 2: DE10-Lite Board

For display functionalities, the DE10-Lite board includes a VGA output port that allows users to connect it to standard monitors and display graphic content. The board is fully compatible with Intel Quartus Prime design software, used to develop this project.

The DE10-Lite board is connected to the computer using an A-to-B USB cable. It has 10 user-controllable LEDs and 10 switches; for this project, the LEDs LEDR0, LEDR1, and LEDR2 are used to display the player's lives. The reset is implemented using the board's "SW0" switch. The score is shown on 3 of the 6 seven-segment displays, specifically Hex(0..2), with Hex0 as the least significant digit of the score.

In Figure 2 are highlighted the board components used in the project.

2.1.2 Joystick

The joystick is connected to the FPGA board via three female-to-male Dupont wires: one connected to the 5V power supply, another connected to the ground, and the last one connects the "VRx" pin of the joystick to the board's "JP8[0]" pin, allowing horizontal paddle movements through its readings.

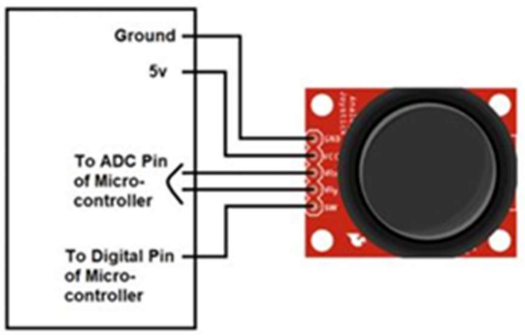
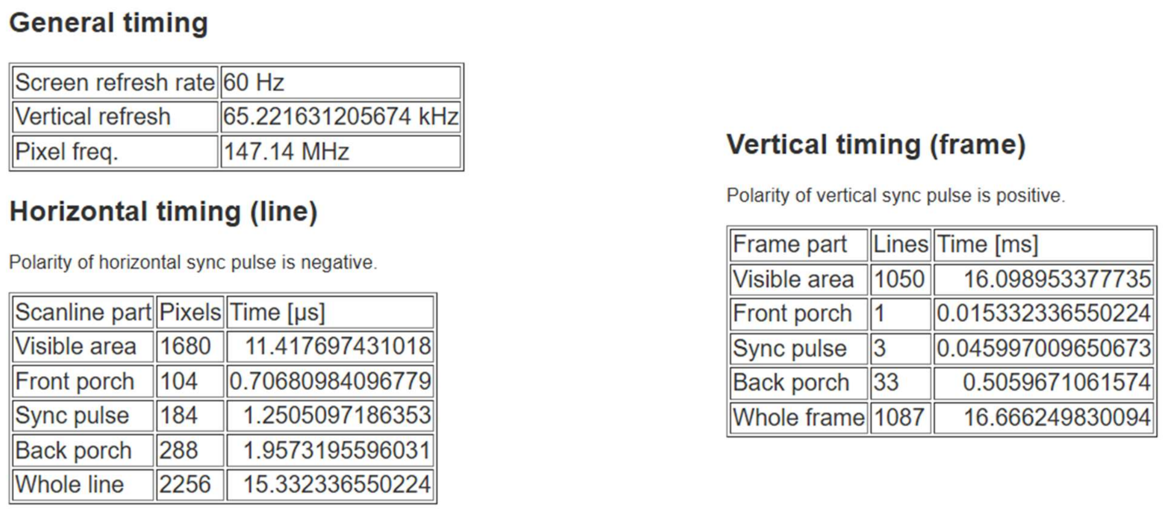

Ground
5v
To ADC Pin of Micro-controller
To Digital Pin of Micro-controller

Figure 3: Schematic of the Joystick

2.1.3 VGA Screen

For this project, an LG monitor is used, with a resolution of 1680x1050 pixels, directly connected to the FPGA board via a VGA-to-VGA cable. The monitor specifications are in Figure 4.



General timing

Screen refresh rate	60 Hz
Vertical refresh	65.221631205674 kHz
Pixel freq.	147.14 MHz

Horizontal timing (line)

Polarity of horizontal sync pulse is negative.

Scanline part	Pixels	Time [µs]
Visible area	1680	11.417697431018
Front porch	104	0.70680984096779
Sync pulse	184	1.2505097186353
Back porch	288	1.9573195596031
Whole line	2256	15.332336550224

Vertical timing (frame)

Polarity of vertical sync pulse is positive.

Frame part	Lines	Time [ms]
Visible area	1050	16.098953377735
Front porch	1	0.015332336550224
Sync pulse	3	0.045997009650673
Back porch	33	0.5059671061574
Whole frame	1087	16.666249830094

Figure 4:Monitor specifications

2.2 Software

Quartus Prime Lite Edition is used to implement the code, it is a software tool developed by Intel for designing and programming FPGA devices. It is part of the Quartus Prime design software suite, which provides a comprehensive environment for designing, verifying, synthesizing, and implementing digital logic.

Quartus Prime Lite Edition supports multiple design input methods, including schematic capture, Verilog, VHDL, and SystemVerilog. The tool includes a timing analyzer that verifies the performance of the designed circuit by calculating and analyzing critical paths and ensuring timing requirements are met.

2.2.1 VHDL

VHDL (VHSIC Hardware Description Language) is a hardware description language used to model, simulate, and synthesize digital systems. It is particularly used in designing integrated circuits and FPGA (Field Programmable Gate Array); it allows designers to describe the behaviour and structure of electronic circuits at various levels of abstraction.

The VHDL language can be used to simulate circuit behaviour before physical construction, allowing functional verification of the design.

A fundamental feature of the language is the ability to use constructs for both sequential and concurrent programming.

A typical VHDL project consists of several key components:

- Entity: Defines the component's interface, including inputs, outputs, and ports, and specifies what the component should do without describing how;
- Architecture: Describes the internal implementation of the entity. It can include processes, signals, components, and variables;
- Process: Blocks of sequential code within the architecture that describes the circuit's behaviour. They can be triggered by events on specific signals present in the sensitivity list.

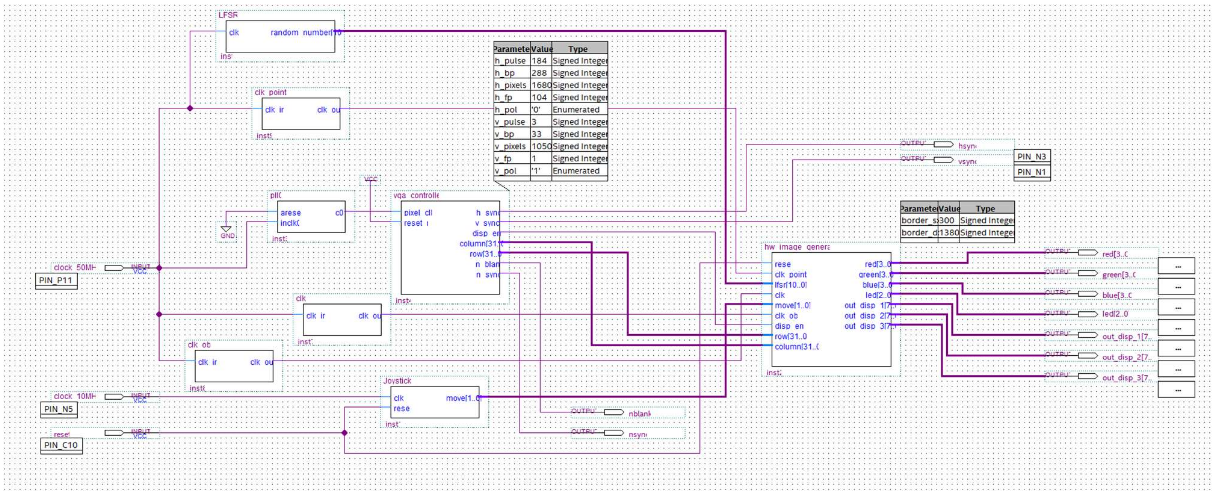
3 Code sections

3.1 Top level entity

The project is divided into several files to facilitate code management and maintenance. Each component of the game is developed within a single file, which is placed in the project directory. The complete list of files that make up the project is:

- TopLevel.bdf
- Vga_controller.vhd
- Hw_image_generator.vhd
- Clk.vhd
- Joystick.vhd
- LFSR.vhd
- Clk_obj.vhd
- Clock_points.vhd
- Seven_segment.vhd

Every VHDL project requires a Top Level Entity, the main file, which in this specific case is "TopLevel.bdf." This is a "Block Diagram" file, a tool that allows you to design digital circuits using functional blocks rather than HDL (Hardware Description Language) coding, enabling the visualization of the project structure. Within this file, individual project elements can be placed as symbolic components, after converting them into Symbolic Files, and interconnected with each other or with input/output signals.



Paramete	Value	Type
h_pulse	184	Signed Integer
h_bp	288	Signed Integer
h_pixels	1680	Signed Integer
h_fp	104	Signed Integer
h_pol	'0'	Enumerated
v_pulse	3	Signed Integer
v_bp	33	Signed Integer
v_pixels	1050	Signed Integer
v_fp	1	Signed Integer
v_pol	'1'	Enumerated

Paramete	Value	Type
border_s	300	Signed Integer
border_d	1380	Signed Integer

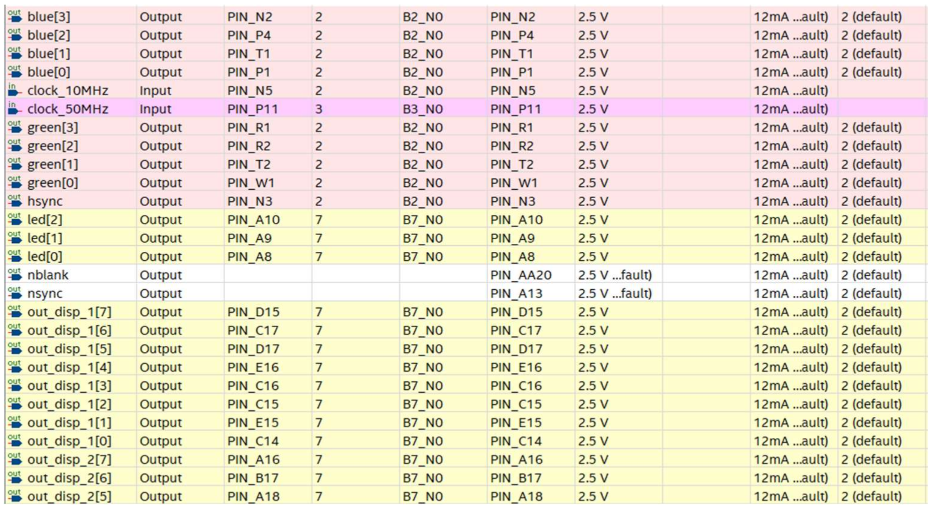
Figure 5:Top level entity

The Terasic DE10-Lite features a series of LEDs, switches, and 7-segment displays that can be associated with the input and output signals of the top level through a utility called "Pin Planner" (fig 6), by consulting the DE10-Lite user manual. To make the association, the signals to be mapped must be connected to a corresponding symbol in the schematic file. In this case study, only "n_blanc" and "n_sync" are not assigned.

The board provides several clock signals, including one at 50MHz and one at 10MHz, associated with pins PIN_P11 and PIN_N5 respectively, which are used within the project as the main clocks for the processes. A reset signal is also used, associated with PIN_C10, to reset the game.

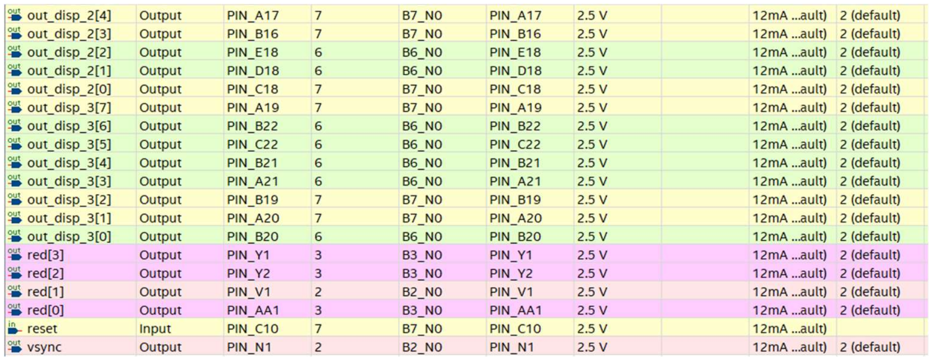
Regarding outputs, there are the VGA synchronization signals hsync and vsync, associated with pins PIN_N3 and PIN_N1 respectively; the colour display signals red[3..0], green[3..0], and blue[3..0] representing the red, green, and blue colours of the pixels; the LEDs associated with pins PIN_A8, PIN_A9, and PIN_A10 used to represent lives; and the out_disp_1[7..0], out_disp_2[7..0], and out_disp_3[7..0] signals associated with the 7-segment display.

After planning through the Pin Planner and compiling the code, it is possible to load the code onto the board using the Programmer utility. By selecting the correct hardware, the game can then be executed.



blue[3]	Output	PIN_N2	2	B2_N0	PIN_N2	2.5 V		12mA ...ault)	2 (default)
blue[2]	Output	PIN_P4	2	B2_N0	PIN_P4	2.5 V		12mA ...ault)	2 (default)
blue[1]	Output	PIN_T1	2	B2_N0	PIN_T1	2.5 V		12mA ...ault)	2 (default)
blue[0]	Output	PIN_P1	2	B2_N0	PIN_P1	2.5 V		12mA ...ault)	2 (default)
clock_10MHz	Input	PIN_N5	2	B2_N0	PIN_N5	2.5 V		12mA ...ault)	
clock_50MHz	Input	PIN_P11	3	B3_N0	PIN_P11	2.5 V		12mA ...ault)	
green[3]	Output	PIN_R1	2	B2_N0	PIN_R1	2.5 V		12mA ...ault)	2 (default)
green[2]	Output	PIN_R2	2	B2_N0	PIN_R2	2.5 V		12mA ...ault)	2 (default)
green[1]	Output	PIN_T2	2	B2_N0	PIN_T2	2.5 V		12mA ...ault)	2 (default)
green[0]	Output	PIN_W1	2	B2_N0	PIN_W1	2.5 V		12mA ...ault)	2 (default)
hsync	Output	PIN_N3	2	B2_N0	PIN_N3	2.5 V		12mA ...ault)	2 (default)
led[2]	Output	PIN_A10	7	B7_N0	PIN_A10	2.5 V		12mA ...ault)	2 (default)
led[1]	Output	PIN_A9	7	B7_N0	PIN_A9	2.5 V		12mA ...ault)	2 (default)
led[0]	Output	PIN_A8	7	B7_N0	PIN_A8	2.5 V		12mA ...ault)	2 (default)
nblank	Output				PIN_AA20	2.5 V ...fault)		12mA ...ault)	2 (default)
nsync	Output				PIN_A13	2.5 V ...fault)		12mA ...ault)	2 (default)
out_disp_1[7]	Output	PIN_D15	7	B7_N0	PIN_D15	2.5 V		12mA ...ault)	2 (default)
out_disp_1[6]	Output	PIN_C17	7	B7_N0	PIN_C17	2.5 V		12mA ...ault)	2 (default)
out_disp_1[5]	Output	PIN_D17	7	B7_N0	PIN_D17	2.5 V		12mA ...ault)	2 (default)
out_disp_1[4]	Output	PIN_E16	7	B7_N0	PIN_E16	2.5 V		12mA ...ault)	2 (default)
out_disp_1[3]	Output	PIN_C16	7	B7_N0	PIN_C16	2.5 V		12mA ...ault)	2 (default)
out_disp_1[2]	Output	PIN_C15	7	B7_N0	PIN_C15	2.5 V		12mA ...ault)	2 (default)
out_disp_1[1]	Output	PIN_E15	7	B7_N0	PIN_E15	2.5 V		12mA ...ault)	2 (default)
out_disp_1[0]	Output	PIN_C14	7	B7_N0	PIN_C14	2.5 V		12mA ...ault)	2 (default)
out_disp_2[7]	Output	PIN_A16	7	B7_N0	PIN_A16	2.5 V		12mA ...ault)	2 (default)
out_disp_2[6]	Output	PIN_B17	7	B7_N0	PIN_B17	2.5 V		12mA ...ault)	2 (default)
out_disp_2[5]	Output	PIN_A18	7	B7_N0	PIN_A18	2.5 V		12mA ...ault)	2 (default)

Figure 6: Pin planner (first part)



out_disp_2[4]	Output	PIN_A17	7	B7_N0	PIN_A17	2.5 V		12mA ...ault)	2 (default)
out_disp_2[3]	Output	PIN_B16	7	B7_N0	PIN_B16	2.5 V		12mA ...ault)	2 (default)
out_disp_2[2]	Output	PIN_E18	6	B6_N0	PIN_E18	2.5 V		12mA ...ault)	2 (default)
out_disp_2[1]	Output	PIN_D18	6	B6_N0	PIN_D18	2.5 V		12mA ...ault)	2 (default)
out_disp_2[0]	Output	PIN_C18	7	B7_N0	PIN_C18	2.5 V		12mA ...ault)	2 (default)
out_disp_3[7]	Output	PIN_A19	7	B7_N0	PIN_A19	2.5 V		12mA ...ault)	2 (default)
out_disp_3[6]	Output	PIN_B22	6	B6_N0	PIN_B22	2.5 V		12mA ...ault)	2 (default)
out_disp_3[5]	Output	PIN_C22	6	B6_N0	PIN_C22	2.5 V		12mA ...ault)	2 (default)
out_disp_3[4]	Output	PIN_B21	6	B6_N0	PIN_B21	2.5 V		12mA ...ault)	2 (default)
out_disp_3[3]	Output	PIN_A21	6	B6_N0	PIN_A21	2.5 V		12mA ...ault)	2 (default)
out_disp_3[2]	Output	PIN_B19	7	B7_N0	PIN_B19	2.5 V		12mA ...ault)	2 (default)
out_disp_3[1]	Output	PIN_A20	7	B7_N0	PIN_A20	2.5 V		12mA ...ault)	2 (default)
out_disp_3[0]	Output	PIN_B20	6	B6_N0	PIN_B20	2.5 V		12mA ...ault)	2 (default)
red[3]	Output	PIN_Y1	3	B3_N0	PIN_Y1	2.5 V		12mA ...ault)	2 (default)
red[2]	Output	PIN_Y2	3	B3_N0	PIN_Y2	2.5 V		12mA ...ault)	2 (default)
red[1]	Output	PIN_V1	2	B2_N0	PIN_V1	2.5 V		12mA ...ault)	2 (default)
red[0]	Output	PIN_AA1	3	B3_N0	PIN_AA1	2.5 V		12mA ...ault)	2 (default)
reset	Input	PIN_C10	7	B7_N0	PIN_C10	2.5 V		12mA ...ault)	
vsync	Output	PIN_N1	2	B2_N0	PIN_N1	2.5 V		12mA ...ault)	2 (default)

Figure 7: Pin planner (second part)

3.2 Hw_image_generator

The file “hw_image_generator.vhd” manages the game logic and on-screen display.

In the first lines of the file, there are several libraries used in other project files, and these will be explained in this section, but they are applicable to all other files. These lines include the “ieee” library, which is a collection of packages including declarations of types, constants, functions, and standardized procedures. Specifically, the “std_logic_1164” package defines a standard for designers to describe interconnection data types used in VHDL modeling. The “std_logic_unsigned” package provides a set of unsigned arithmetic, conversion, and comparison functions for std_logic_vector. The “std_logic_arith” package offers arithmetic, conversion, and comparison functions for signed, unsigned, std_ulogic, std_logic, and std_logic_vector.

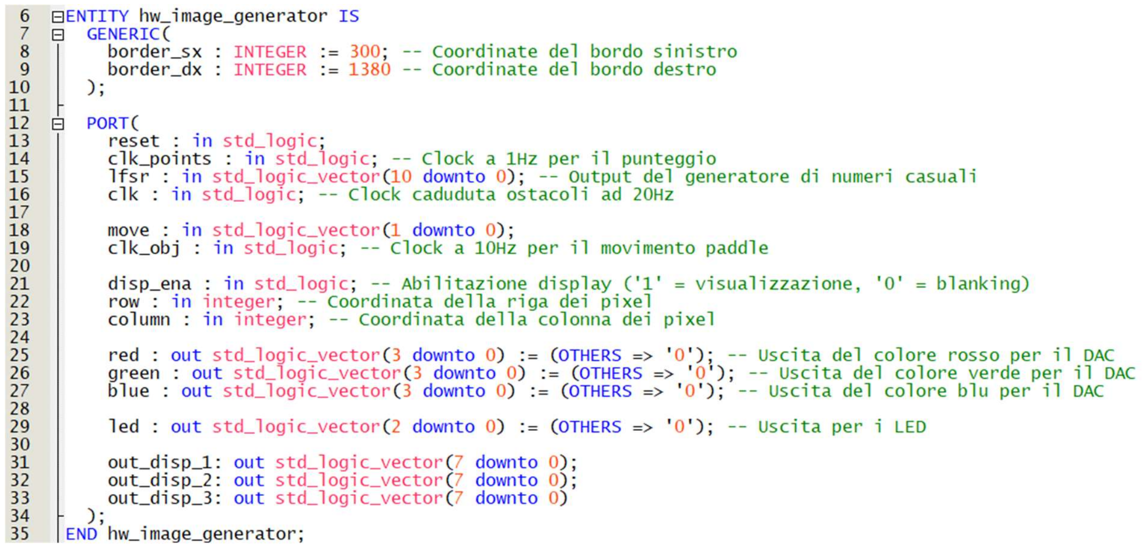
In the entity definition of hw_image_generator, within the generic section containing static parameters, two integers are declared to specify the dimensions of the game area borders. In the port section, the inputs and outputs of the entity are defined.

The inputs are:

- Reset, associated with the board switch;
- Clk_points, linked to the 1Hz clock;
- Lfsr, number returned by the random number generator;
- Clk, linked to the 20Hz clock;
- Move, a two-bit vector output from the joystick;
- Clk_obj, linked to the 10Hz clock;
- Disp_ena, display enable input ('1' = display, '0' = blanking);
- Row and column, pixel coordinates on the display.

The outputs are:

- Red, green, and blue, std_logic_vector outputs that print the desired colour shade on the screen;
- Led, a 3-bit std_logic_vector output associated with 3 LEDs on the board;
- Out_disp_ (1,2,3), 8-bit std_logic_vector outputs linked to the seven-segment displays on the board.

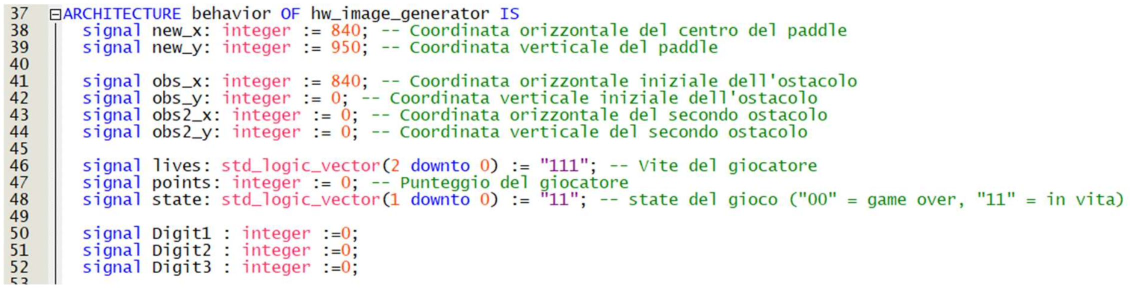


```vhdl
ENTITY hw_image_generator IS
  GENERIC(
    border_sx : INTEGER := 300; -- Coordinate del bordo sinistro
    border_dx : INTEGER := 1380 -- Coordinate del bordo destro
  );

  PORT(
    reset : in std_logic;
    clk_points : in std_logic; -- Clock a 1Hz per il punteggio
    lfsr : in std_logic_vector(10 downto 0); -- Output del generatore di numeri casuali
    clk : in std_logic; -- Clock caduduta ostacoli ad 20Hz

    move : in std_logic_vector(1 downto 0);
    clk_obj : in std_logic; -- Clock a 10Hz per il movimento paddle

    disp_ena : in std_logic; -- Abilitazione display ('1' = visualizzazione, '0' = blanking)
    row : in integer; -- Coordinata della riga dei pixel
    column : in integer; -- Coordinata della colonna dei pixel

    red : out std_logic_vector(3 downto 0) := (OTHERS => '0'); -- Uscita del colore rosso per il DAC
    green : out std_logic_vector(3 downto 0) := (OTHERS => '0'); -- Uscita del colore verde per il DAC
    blue : out std_logic_vector(3 downto 0) := (OTHERS => '0'); -- Uscita del colore blu per il DAC

    led : out std_logic_vector(2 downto 0) := (OTHERS => '0'); -- Uscita per i LED

    out_disp_1: out std_logic_vector(7 downto 0);
    out_disp_2: out std_logic_vector(7 downto 0);
    out_disp_3: out std_logic_vector(7 downto 0)
  );
END hw_image_generator;
```

Figure 8: Hw_image_generator entity

Within the architecture, several signals are declared. These are data objects used for communication between processes that can assume different values over time and can be read and written within concurrent processes. They include:

- new_x, new_y, represent the coordinates of the center of the paddle, initialized with the central values of the movement range;
- obs_x, obs_y, represent the coordinates of the center of the first generated obstacle, intentionally placed in the center of the screen concerning the x-coordinate;
- obs2_x, obs2_y, represent the coordinates of the center of the next obstacle;
- lives, is a 3-bit std_logic_vector representing the player's lives ("111" = 3 lives, "110" = 2 lives, "100" = 1 life, "000" = 0 lives);
- points, is an integer representing the player's score, initialized to 0;
- state, is a 2-bit std_logic_vector representing the game state ("00" = game over, "11" = alive);
- Digit1, Digit2, Digit3, associated with the individual digits displayed on the seven-segment displays, represent units, tens, and hundreds, respectively.

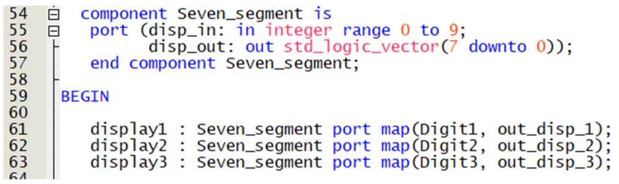


```
37  ARCHITECTURE behavior OF hw_image_generator IS
38      signal new_x: integer := 840; -- Coordinata orizzontale del centro del paddle
39      signal new_y: integer := 950; -- Coordinata verticale del paddle
40
41      signal obs_x: integer := 840; -- Coordinata orizzontale iniziale dell'ostacolo
42      signal obs_y: integer := 0; -- Coordinata verticale iniziale dell'ostacolo
43      signal obs2_x: integer := 0; -- Coordinata orizzontale del secondo ostacolo
44      signal obs2_y: integer := 0; -- Coordinata verticale del secondo ostacolo
45
46      signal lives: std_logic_vector(2 downto 0) := "111"; -- Vite del giocatore
47      signal points: integer := 0; -- Punteggio del giocatore
48      signal state: std_logic_vector(1 downto 0) := "11"; -- state del gioco ("00" = game over, "11" = in vita)
49
50      signal Digit1 : integer :=0;
51      signal Digit2 : integer :=0;
52      signal Digit3 : integer :=0;
```

Figure 9: Signals of hw_image_generator

Starting from line 54, the declaration of the "Seven_segment" component is made, which represents the interface of the component to the architecture; in this way a structural level of abstraction is applied. Within the component, an input "disp_in" and an output "disp_out" are defined, which respectively represent the integer value to be displayed and the corresponding segments to be shown lit.

The component requires the instantiation statement, which maps the interface of the component to other objects in the architecture. In this case, the port map is performed three times, once for each seven-segment display used, and in each instance, the input is mapped to the corresponding Digit(*) and the output to the corresponding out_disp(*).



```
54    component Seven_segment is
55     port (disp_in: in integer range 0 to 9;
56           disp_out: out std_logic_vector(7 downto 0));
57     end component Seven_segment;
58
59  BEGIN
60
61     display1 : Seven_segment port map(Digit1, out_disp_1);
62     display2 : Seven_segment port map(Digit2, out_disp_2);
63     display3 : Seven_segment port map(Digit3, out_disp_3);
```

Figure 10: Seven segment declaration and port map

3.2.1 Game logic

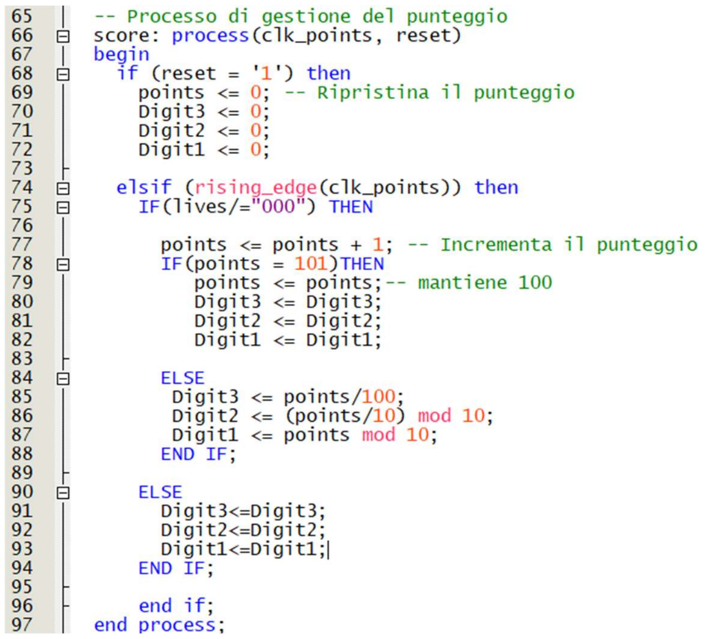
Within "hw_image_generator", the game logic is defined through several processes, which are fundamental units in VHDL used to model sequential behavior in digital systems. These processes are executed concurrently, while instructions within them are executed sequentially. The presence of these processes ensures that the abstraction level is behavioral.

The "score" process manages the score update, which is incremented every second using a 1Hz clock, and updates the display digits in response to signals clk_points and reset. When reset is high (equal to 1) the process resets the score and the digit values to zero. Otherwise, at every rising edge of the clock, it checks for remaining lives and, if lives are present, increments the score by one and distributes it among the various digits.

Digit1 contains the units digit of the score, obtained using the operation of module 10 on the score; Digit2 contains the tens digit, obtained by performing an integer division of the score by 10 and applying the operation of module 10; Digit3 contains the hundreds digit, obtained by performing an integer division of the score by 100.

Once the score reaches "101," which corresponds to winning, the process freezes the update and displays "100" on the board due to a delay.

In case of defeat, the achieved score remains displayed.

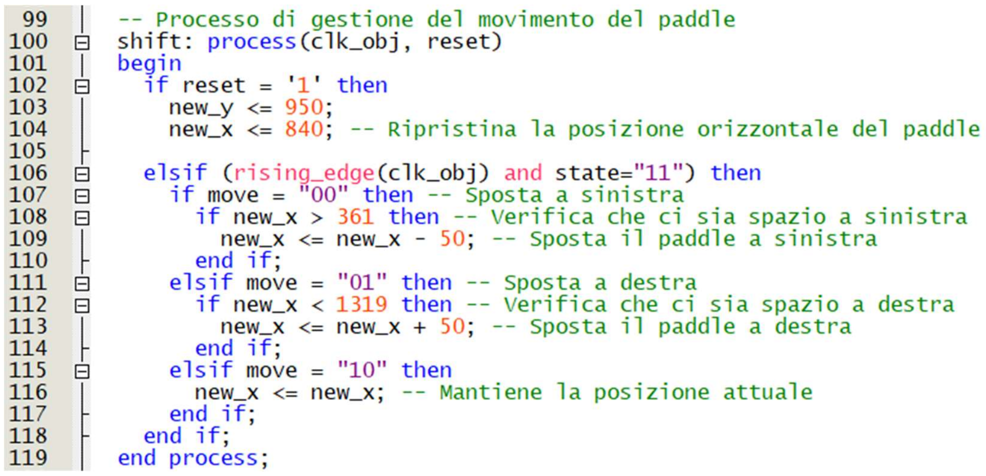


```vhdl
-- Processo di gestione del punteggio
score: process(clk_points, reset)
begin
  if (reset = '1') then
    points <= 0; -- Ripristina il punteggio
    Digit3 <= 0;
    Digit2 <= 0;
    Digit1 <= 0;

  elsif (rising_edge(clk_points)) then
    IF(lives/="000") THEN

      points <= points + 1; -- Incrementa il punteggio
      IF(points = 101)THEN
         points <= points;-- mantiene 100
         Digit3 <= Digit3;
         Digit2 <= Digit2;
         Digit1 <= Digit1;

      ELSE
       Digit3 <= points/100;
       Digit2 <= (points/10) mod 10;
       Digit1 <= points mod 10;
      END IF;

    ELSE
      Digit3<=Digit3;
      Digit2<=Digit2;
      Digit1<=Digit1;
    END IF;

    end if;
end process;
```

Figure 11: Points management

The "shift" process handles the movement of the paddle in response to clk_obj and reset signals. When reset is high, equal to 1, it resets the paddle to its initial coordinates. Otherwise, on a rising edge of the 10Hz clock and when the state is "11" (alive), it checks the input from the joystick, "move."

If "move" is "00" or "01" and there is still enough space, the paddle will move left or right by 50 pixels, respectively. If the input is "10," the paddle will maintain its current position.



```
-- Processo di gestione del movimento del paddle
shift: process(clk_obj, reset)
begin
  if reset = '1' then
    new_y <= 950;
    new_x <= 840; -- Ripristina la posizione orizzontale del paddle

  elsif (rising_edge(clk_obj) and state="11") then
    if move = "00" then -- Sposta a sinistra
      if new_x > 361 then -- Verifica che ci sia spazio a sinistra
        new_x <= new_x - 50; -- Sposta il paddle a sinistra
      end if;
    elsif move = "01" then -- Sposta a destra
      if new_x < 1319 then -- Verifica che ci sia spazio a destra
        new_x <= new_x + 50; -- Sposta il paddle a destra
      end if;
    elsif move = "10" then
      new_x <= new_x; -- Mantiene la posizione attuale
    end if;
  end if;
end process;
```

Figure 12: Movements management

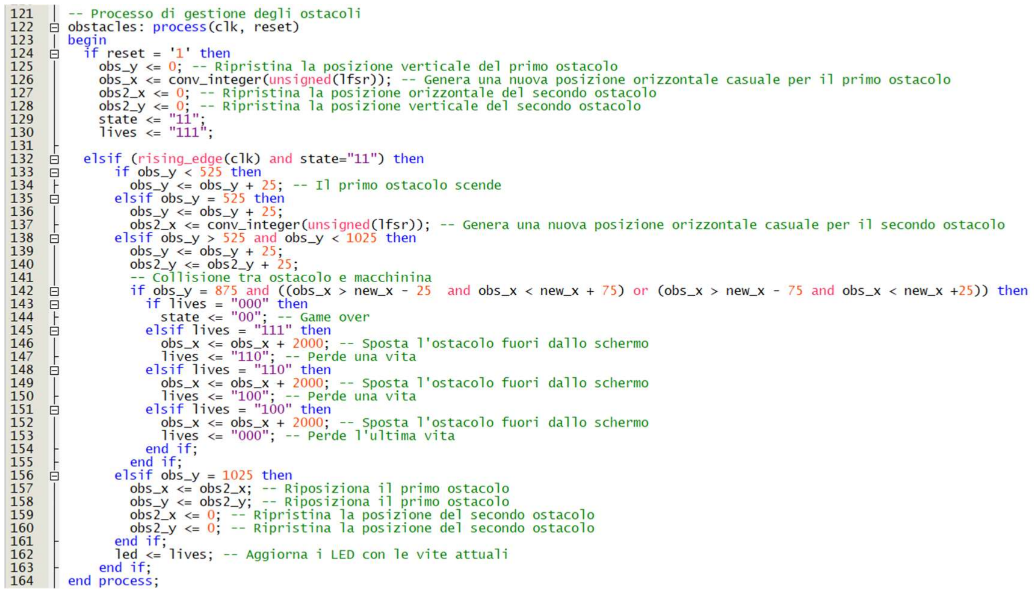
The "obstacle" process manages the generation and descent of obstacles, and the collision handling in response to clk and reset signals. When reset is high, state and lives are reset to their initial values, a new x coordinate is generated for the new obstacle, and the other coordinates are reset. Otherwise, on a rising edge of the clock at 20Hz, and when the state is "11," the following operations are executed.

At each rising edge of the clock, the obstacle moves downward by 25 pixels until the y coordinate reaches "525," which represents the exact midpoint of the play area along the y-axis. At this point, the first obstacle continues moving downward at the same speed, and a second obstacle is generated with an x coordinate corresponding to a pseudo-random value obtained using the LFSR and y = 0. Subsequently, both obstacles continue moving downward at the same speed until the first obstacle either reaches the end of the play area or collides with the paddle.

Collision occurs if the y coordinate of the obstacle is equal to "875" and at least one of its side edges is within those of the paddle. In this situation, if there are at least two lives remaining, one life is decremented, and the obstacle is moved out of the play area (using an offset on the x coordinate). If only one life remains, after the collision, the life is decremented and on the next rising edge, the state transitions to "00," indicating game over.

When the y coordinate of the obstacle reaches the boundary of the play area, at 1025 pixels, the obstacle disappears by taking on the coordinates of the second obstacle, which in turn has reached the midpoint of the play area, and the coordinates of the second obstacle are reset for the generation of the next obstacle.

Finally, the display of lives on the LEDs is updated.



```vhdl
-- Processo di gestione degli ostacoli
obstacles: process(clk, reset)
begin
  if reset = '1' then
    obs_y <= 0; -- Ripristina la posizione verticale del primo ostacolo
    obs_x <= conv_integer(unsigned(lfsr)); -- Genera una nuova posizione orizzontale casuale per il primo ostacolo
    obs2_x <= 0; -- Ripristina la posizione orizzontale del secondo ostacolo
    obs2_y <= 0; -- Ripristina la posizione verticale del secondo ostacolo
    state <= "11";
    lives <= "111";

  elsif (rising_edge(clk) and state="11") then
      if obs_y < 525 then
        obs_y <= obs_y + 25; -- Il primo ostacolo scende
      elsif obs_y = 525 then
        obs_y <= obs_y + 25;
        obs2_x <= conv_integer(unsigned(lfsr)); -- Genera una nuova posizione orizzontale casuale per il secondo ostacolo
      elsif obs_y > 525 and obs_y < 1025 then
        obs_y <= obs_y + 25;
        obs2_y <= obs2_y + 25;
        -- Collisione tra ostacolo e macchinina
        if obs_y = 875 and ((obs_x > new_x - 25  and obs_x < new_x + 75) or (obs_x > new_x - 75 and obs_x < new_x +25)) then
          if lives = "000" then
            state <= "00"; -- Game over
          elsif lives = "111" then
            obs_x <= obs_x + 2000; -- Sposta l'ostacolo fuori dallo schermo
            lives <= "110"; -- Perde una vita
          elsif lives = "110" then
            obs_x <= obs_x + 2000; -- Sposta l'ostacolo fuori dallo schermo
            lives <= "100"; -- Perde una vita
          elsif lives = "100" then
            obs_x <= obs_x + 2000; -- Sposta l'ostacolo fuori dallo schermo
            lives <= "000"; -- Perde l'ultima vita
          end if;
        end if;
      elsif obs_y = 1025 then
        obs_x <= obs2_x; -- Riposiziona il primo ostacolo
        obs_y <= obs2_y; -- Riposiziona il primo ostacolo
        obs2_x <= 0; -- Ripristina la posizione del secondo ostacolo
        obs2_y <= 0; -- Ripristina la posizione del secondo ostacolo
      end if;
      led <= lives; -- Aggiorna i LED con le vite attuali
    end if;
end process;
```

Figure 13: Obstacle management

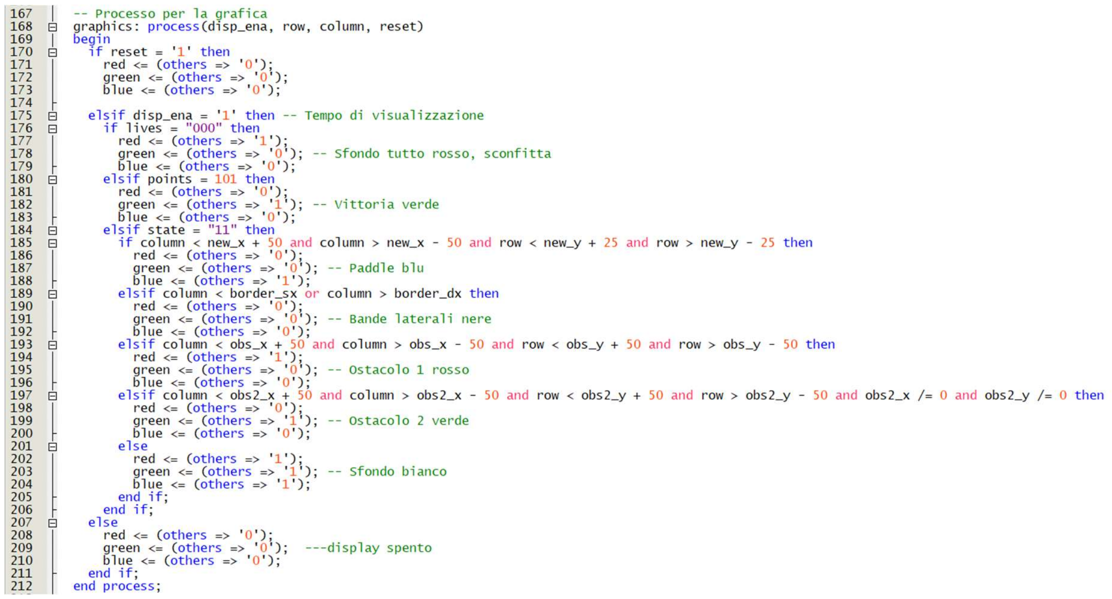
3.2.2 Graphics

The "graphics" process is responsible for rendering the graphical elements of the game based on disp_ena, row, column, and reset declared in the sensitivity list. These determine the colours to be displayed for each pixel. When a high reset signal is present, the colours are set to 0 to display a black screen. Otherwise, if the display enable input (disp_ena) is logical 1, the display is activated; while if it is 0, the display is turned off again.

Whenever the display is active, various scenarios can occur. When the state is "11", the game area is displayed based on the values read in "row" and "column", as follows:

- A blue paddle, 100 pixels wide and 50 pixels high;
- Black side bands, with a total width of 600 pixels (300 per side), delimiting the game area;
- Two obstacles, one red and one green, each 100x100 pixels in size;
- A white background.

The objects are displayed starting from their center coordinates x and y, with offsets added to determine their dimensions. Upon all lives are lost (lives = "000"), the display turns red to indicate defeat. Conversely, in the case of a victory (i.e., points = 101), the display turns green.



```vhdl
-- Processo per la grafica
graphics: process(disp_ena, row, column, reset)
begin
  if reset = '1' then
    red <= (others => '0');
    green <= (others => '0');
    blue <= (others => '0');

  elsif disp_ena = '1' then -- Tempo di visualizzazione
    if lives = "000" then
      red <= (others => '1');
      green <= (others => '0'); -- Sfondo tutto rosso, sconfitta
      blue <= (others => '0');
    elsif points = 101 then
      red <= (others => '0');
      green <= (others => '1'); -- Vittoria verde
      blue <= (others => '0');
    elsif state = "11" then
      if column < new_x + 50 and column > new_x - 50 and row < new_y + 25 and row > new_y - 25 then
        red <= (others => '0');
        green <= (others => '0'); -- Paddle blu
        blue <= (others => '1');
      elsif column < border_sx or column > border_dx then
        red <= (others => '0');
        green <= (others => '0'); -- Bande laterali nere
        blue <= (others => '0');
      elsif column < obs_x + 50 and column > obs_x - 50 and row < obs_y + 50 and row > obs_y - 50 then
        red <= (others => '1');
        green <= (others => '0'); -- Ostacolo 1 rosso
        blue <= (others => '0');
      elsif column < obs2_x + 50 and column > obs2_x - 50 and row < obs2_y + 50 and row > obs2_y - 50 and obs2_x /= 0 and obs2_y /= 0 then
        red <= (others => '0');
        green <= (others => '1'); -- Ostacolo 2 verde
        blue <= (others => '0');
      else
        red <= (others => '1');
        green <= (others => '1'); -- Sfondo bianco
        blue <= (others => '1');
      end if;
    end if;
  else
    red <= (others => '0');
    green <= (others => '0');  ---display spento
    blue <= (others => '0');
  end if;
end process;
```

Figure 14: Graphics management

3.3 Clock dividers

In the game, various clocks are used, all derived from the 50MHz clock through clock dividers, except for the clock associated with the joystick, which uses the 10MHz clock of the board.

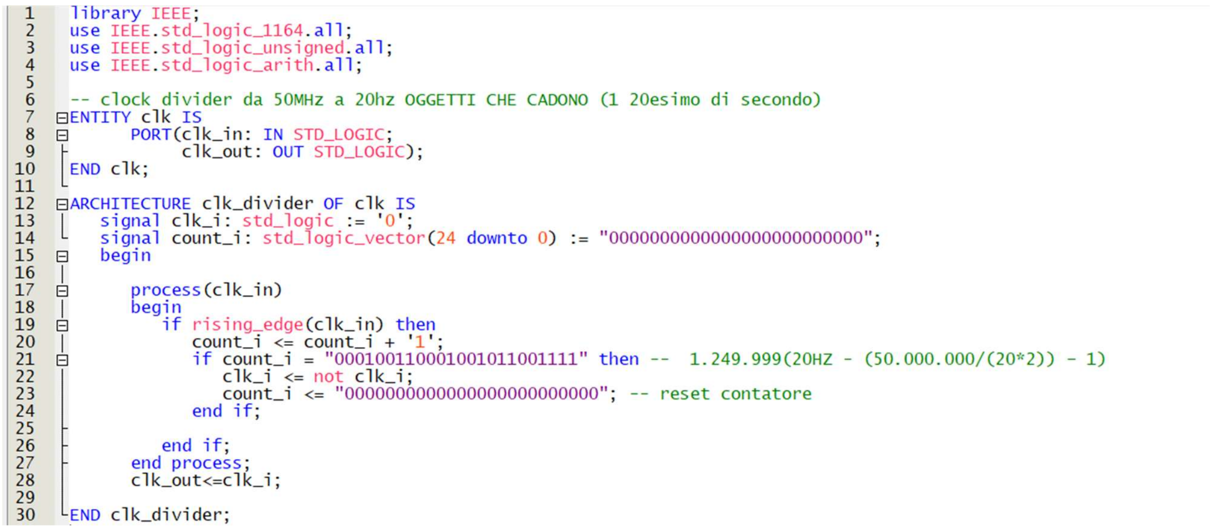
The three clock dividers created have frequencies of 1Hz, 10Hz, and 20Hz, and are called "clock_points," "clock_obj," and "clk," respectively. These frequencies were appropriately chosen based on the game's logic requirements.

The clock divider entity includes an input of type STD_LOGIC, corresponding to the input clock, and an output, also of type STD_LOGIC, corresponding to the desired clock.

In its architecture, the behaviour of the clock divider is described. Specifically, in lines 13-14, two signals are declared: clk_i, an internal clock that will be used to store the new clock to be provided as output, and count_i, a counter necessary to keep track of the clock cycles of the original clock.

The process within the clock divider is triggered at each rising edge of the clock_in, which is included in the sensitivity list of the process. Whenever a rising edge occurs, the counter is incremented by one until it reaches the value obtained using the formula $(50\,000\,000/(2*n))-1$, corresponding to half the period of the input clock minus computational delay. At this point, the clock signal is toggled, and the counter is reset to zero. Finally, the value of the clock in the signal clk_i is assigned to the output variable (line 28).

The 1Hz clock has a period of 1 second, the 10Hz clock has a period of 1/10 seconds, and the 20Hz clock has a period of 1/20 seconds.



```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

-- clock divider da 50MHz a 20hz OGGETTI CHE CADONO (1 20esimo di secondo)
ENTITY clk IS
      PORT(clk_in: IN STD_LOGIC;
             clk_out: OUT STD_LOGIC);
END clk;

ARCHITECTURE clk_divider OF clk IS
   signal clk_i: std_logic := '0';
   signal count_i: std_logic_vector(24 downto 0) := "0000000000000000000000000";
   begin

      process(clk_in)
      begin
         if rising_edge(clk_in) then
            count_i <= count_i + '1';
            if count_i = "000100110001001011001111" then --  1.249.999(20HZ - (50.000.000/(20*2)) - 1)
               clk_i <= not clk_i;
               count_i <= "0000000000000000000000000"; -- reset contatore
            end if;

         end if;
      end process;
      clk_out<=clk_i;

END clk_divider;
```

Figure 15: Clock divider sample (20Hz)

3.4 Joystick

The joystick file in the project reads the analog signals from the joystick and converts them into a logical vector sent to the "hw_image_generator" for managing the paddle movements.

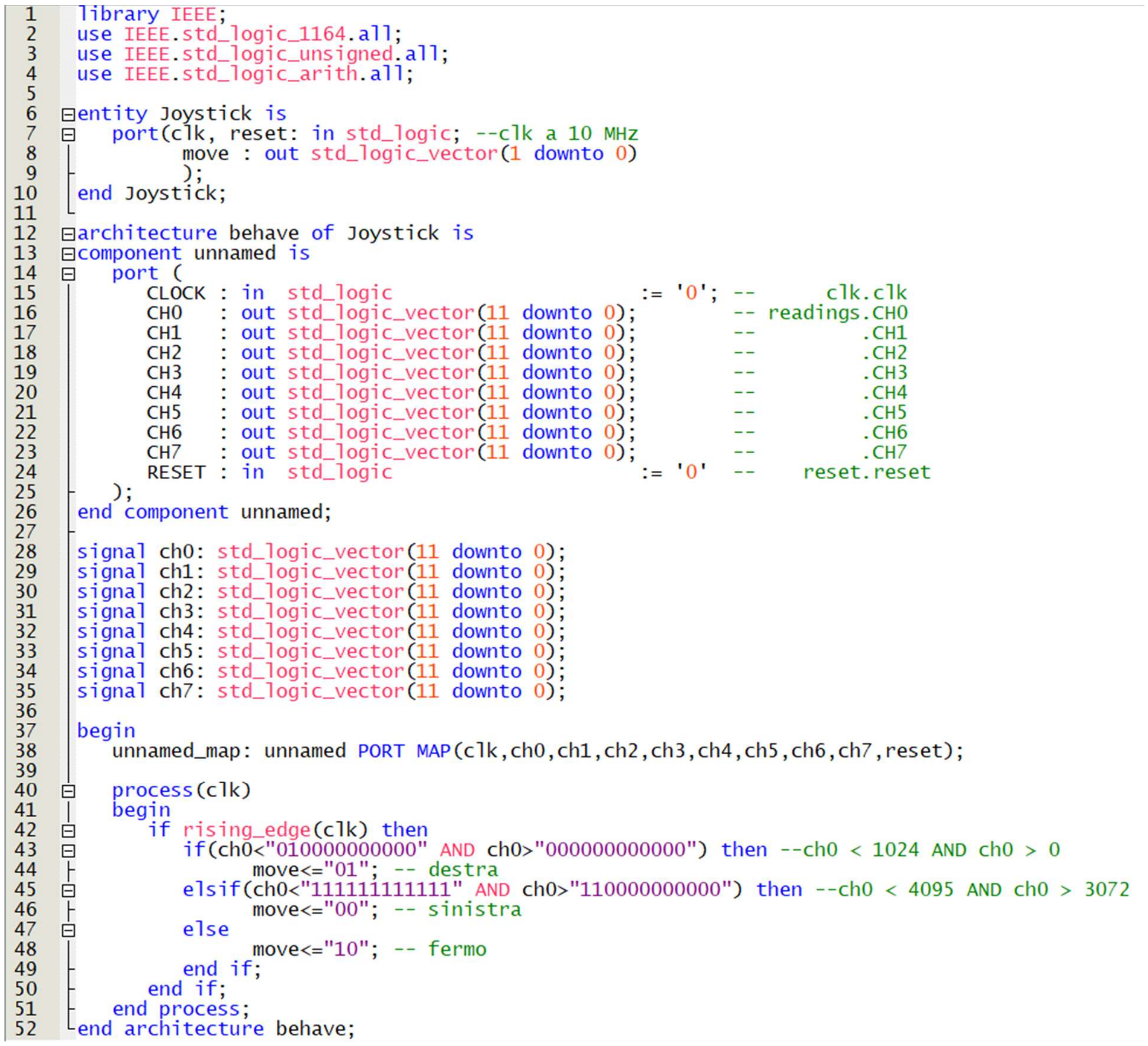
Lines 6 to 10 define the joystick’s entity. It has two inputs: a specific ADC 10MHz clock, provided by the board, and a reset which will be mapped as manual switch “SW0” on the board. The only output is a 2-bit logic vector, which indicates the movement’s direction.

In the architecture of the file, firstly, the unnamed component is recalled (line 13), which handles the main logic of the ADC components, following a structural logic; subsequently, eight 12-bit std_logic_vector signals are declared. Then, through a port map, the input clock, reset, and signals are mapped to the corresponding ones in the unnamed component. The only signal that will read the movements is “ch0,” associated with the x-axis of the joystick.

Starting from line 40, the process that manages the behavior of the joystick is described, driven by a 10MHz clock. On the rising edge of the clock, the voltage value from the joystick is read through the signal ch0, and the value of the output move is determined. The paddle will move to the right if the value read is between 0 (000000000000) and 1024 (010000000000), to the left if the value is between 3072 (110000000000) and 4095 (111111111111), otherwise, it will remain in the current position.

Each movement is encoded using a pair of bits: 01 corresponds to right, 00 to left, and 10 to stationary. The chosen value ranges were determined after a series of tests conducted on the joystick.

The reset has not been reprogrammed as it is managed in the unnamed component.



```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

entity Joystick is
    port(clk, reset: in std_logic; --clk a 10 MHz
            move : out std_logic_vector(1 downto 0)
            );
end Joystick;

architecture behave of Joystick is
component unnamed is
    port (
        CLOCK : in  std_logic                     := '0'; --       clk.clk
        CH0   : out std_logic_vector(11 downto 0);        -- readings.CH0
        CH1   : out std_logic_vector(11 downto 0);        --         .CH1
        CH2   : out std_logic_vector(11 downto 0);        --         .CH2
        CH3   : out std_logic_vector(11 downto 0);        --         .CH3
        CH4   : out std_logic_vector(11 downto 0);        --         .CH4
        CH5   : out std_logic_vector(11 downto 0);        --         .CH5
        CH6   : out std_logic_vector(11 downto 0);        --         .CH6
        CH7   : out std_logic_vector(11 downto 0);        --         .CH7
        RESET : in  std_logic                     := '0'  --    reset.reset
    );
end component unnamed;

signal ch0: std_logic_vector(11 downto 0);
signal ch1: std_logic_vector(11 downto 0);
signal ch2: std_logic_vector(11 downto 0);
signal ch3: std_logic_vector(11 downto 0);
signal ch4: std_logic_vector(11 downto 0);
signal ch5: std_logic_vector(11 downto 0);
signal ch6: std_logic_vector(11 downto 0);
signal ch7: std_logic_vector(11 downto 0);

begin
    unnamed_map: unnamed PORT MAP(clk,ch0,ch1,ch2,ch3,ch4,ch5,ch6,ch7,reset);

    process(clk)
    begin
        if rising_edge(clk) then
            if(ch0<"010000000000" AND ch0>"000000000000") then --ch0 < 1024 AND ch0 > 0
                    move<="01"; -- destra
            elsif(ch0<"111111111111" AND ch0>"110000000000") then --ch0 < 4095 AND ch0 > 3072
                    move<="00"; -- sinistra
            else
                    move<="10"; -- fermo
            end if;
        end if;
    end process;
end architecture behave;
```

Figure 16: Joystick code

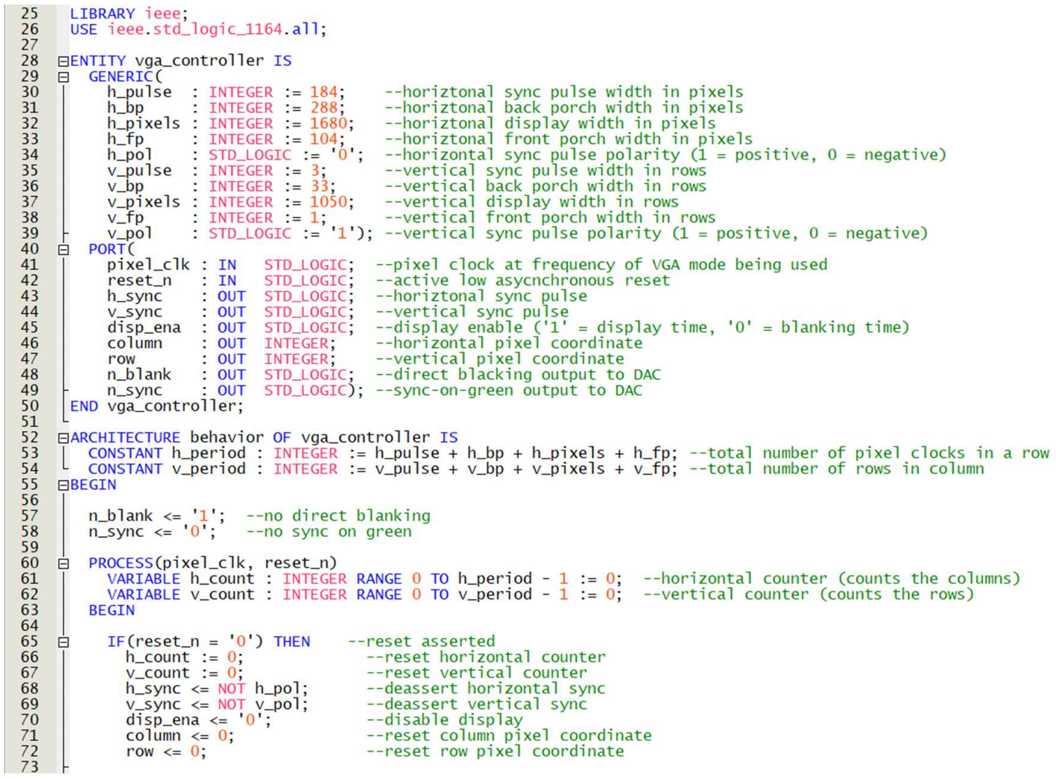
3.5 VGA

The display logic of the project is managed through two files: "pll0.vhd" and "vga_controller.vhd".

The "PLL0" is a utility files that takes as inputs "GND" and the 50MHz clock from the board and provides as output a specific clock for the type of monitor used.

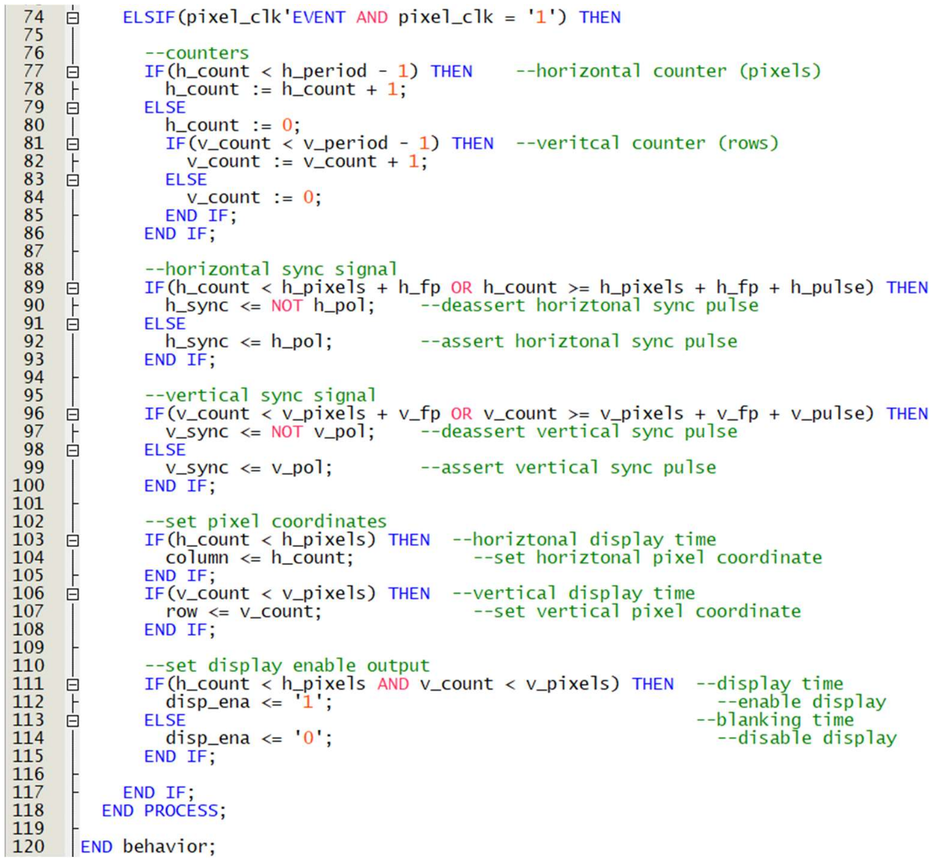
The "vga_controller.vhd" file describes a VGA (Video Graphics Array) controller responsible for horizontal and vertical synchronization, and it generates necessary signals for displaying images on a VGA monitor. Its inputs include the clock generated by "PLL0" and a voltage signal "Vcc". While, its outputs include "h_sync" and "v_sync" for horizontal and vertical synchronization signals, "disp_ena" for display enablement (1 = display on, 0 = blanking time), row and column integers representing pixel coordinates on the screen, "n_blank" for direct blacking output to DAC, and "n_sync" for sync-on-green output to DAC.

The basic structure of the code is based on existing code and has been appropriately configured for the characteristics of the LG monitor.



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

ENTITY vga_controller IS
  GENERIC(
    h_pulse  : INTEGER := 184;    --horiztonal sync pulse width in pixels
    h_bp     : INTEGER := 288;    --horiztonal back porch width in pixels
    h_pixels : INTEGER := 1680;   --horiztonal display width in pixels
    h_fp     : INTEGER := 104;    --horiztonal front porch width in pixels
    h_pol    : STD_LOGIC := '0';  --horizontal sync pulse polarity (1 = positive, 0 = negative)
    v_pulse  : INTEGER := 3;      --vertical sync pulse width in rows
    v_bp     : INTEGER := 33;     --vertical back porch width in rows
    v_pixels : INTEGER := 1050;   --vertical display width in rows
    v_fp     : INTEGER := 1;      --vertical front porch width in rows
    v_pol    : STD_LOGIC := '1'); --vertical sync pulse polarity (1 = positive, 0 = negative)
  PORT(
    pixel_clk : IN   STD_LOGIC;  --pixel clock at frequency of VGA mode being used
    reset_n   : IN   STD_LOGIC;  --active low asycnchronous reset
    h_sync    : OUT  STD_LOGIC;  --horiztonal sync pulse
    v_sync    : OUT  STD_LOGIC;  --vertical sync pulse
    disp_ena  : OUT  STD_LOGIC;  --display enable ('1' = display time, '0' = blanking time)
    column    : OUT  INTEGER;    --horizontal pixel coordinate
    row       : OUT  INTEGER;    --vertical pixel coordinate
    n_blank   : OUT  STD_LOGIC;  --direct blacking output to DAC
    n_sync    : OUT  STD_LOGIC); --sync-on-green output to DAC
END vga_controller;

ARCHITECTURE behavior OF vga_controller IS
  CONSTANT h_period : INTEGER := h_pulse + h_bp + h_pixels + h_fp; --total number of pixel clocks in a row
  CONSTANT v_period : INTEGER := v_pulse + v_bp + v_pixels + v_fp; --total number of rows in column
BEGIN

  n_blank <= '1';  --no direct blanking
  n_sync <= '0';   --no sync on green

  PROCESS(pixel_clk, reset_n)
    VARIABLE h_count : INTEGER RANGE 0 TO h_period - 1 := 0;  --horizontal counter (counts the columns)
    VARIABLE v_count : INTEGER RANGE 0 TO v_period - 1 := 0;  --vertical counter (counts the rows)
  BEGIN

    IF(reset_n = '0') THEN    --reset asserted
      h_count := 0;             --reset horizontal counter
      v_count := 0;             --reset vertical counter
      h_sync <= NOT h_pol;      --deassert horizontal sync
      v_sync <= NOT v_pol;      --deassert vertical sync
      disp_ena <= '0';          --disable display
      column <= 0;              --reset column pixel coordinate
      row <= 0;                 --reset row pixel coordinate

```

Figure 17: Vga_controller (first part)



```
    ELSIF(pixel_clk'EVENT AND pixel_clk = '1') THEN

      --counters
      IF(h_count < h_period - 1) THEN    --horizontal counter (pixels)
        h_count := h_count + 1;
      ELSE
        h_count := 0;
        IF(v_count < v_period - 1) THEN  --veritcal counter (rows)
          v_count := v_count + 1;
        ELSE
          v_count := 0;
        END IF;
      END IF;

      --horizontal sync signal
      IF(h_count < h_pixels + h_fp OR h_count >= h_pixels + h_fp + h_pulse) THEN
        h_sync <= NOT h_pol;    --deassert horiztonal sync pulse
      ELSE
        h_sync <= h_pol;        --assert horiztonal sync pulse
      END IF;

      --vertical sync signal
      IF(v_count < v_pixels + v_fp OR v_count >= v_pixels + v_fp + v_pulse) THEN
        v_sync <= NOT v_pol;    --deassert vertical sync pulse
      ELSE
        v_sync <= v_pol;        --assert vertical sync pulse
      END IF;

      --set pixel coordinates
      IF(h_count < h_pixels) THEN  --horiztonal display time
        column <= h_count;           --set horiztonal pixel coordinate
      END IF;
      IF(v_count < v_pixels) THEN  --vertical display time
        row <= v_count;              --set vertical pixel coordinate
      END IF;

      --set display enable output
      IF(h_count < h_pixels AND v_count < v_pixels) THEN  --display time
        disp_ena <= '1';                                      --enable display
      ELSE                                                  --blanking time
        disp_ena <= '0';                                      --disable display
      END IF;

    END IF;
  END PROCESS;

END behavior;
```

Figure 18:Vga_controller (second part)

3.6 Seven segments

The file "Seven_segment.vhd" describes a decoding unit for a 7-segment display. This module takes an integer input "disp_in" (ranging from 0 to 9), representing the number to be displayed on the 7-segment display. It produces an output of type STD_LOGIC_VECTOR with 8 bits, indicating which segments of the 7-segment display should be lit up to display the corresponding number.

Each segment in a display is indexed from 0 to 6 and DP (decimal point), with corresponding positions given in Figure 19. The segment can be turned on or off by applying a low logic level or high logic level from the FPGA, respectively.

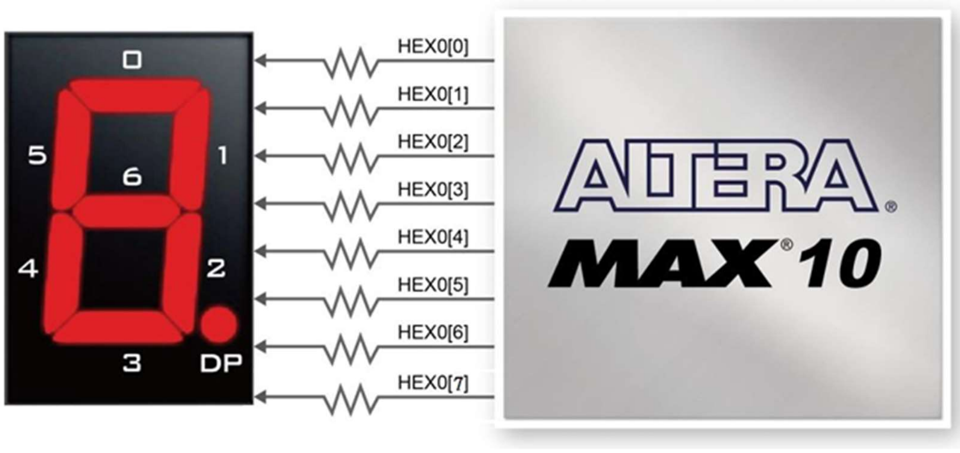
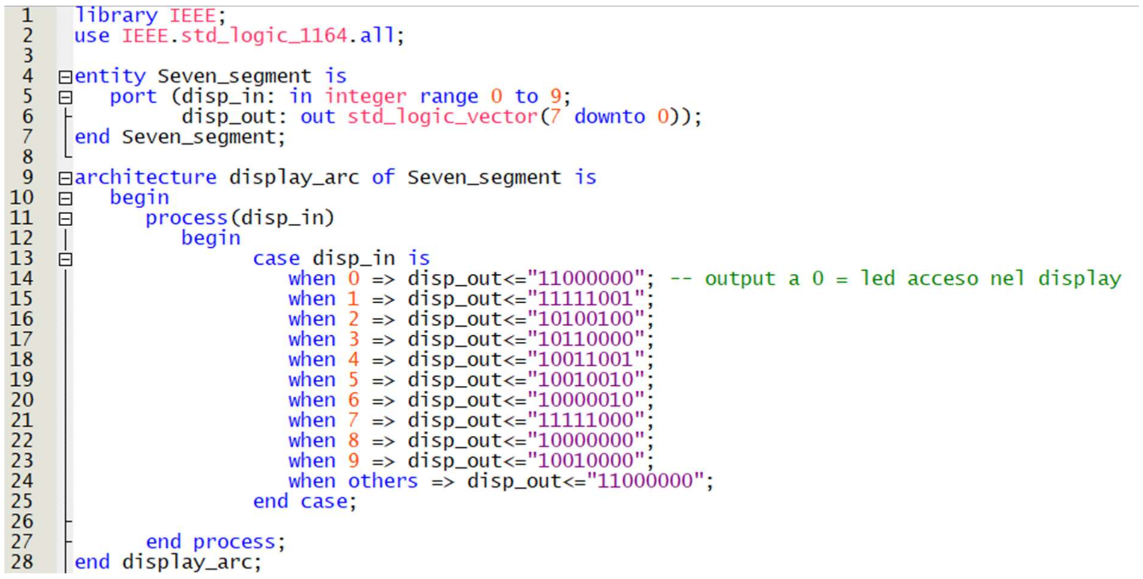


Figure 19: Connections between the 7-segment display HEX0 and the MAX 10 FPGA (Terasic, 2020)

Within the architecture of the file, there is a process that is activated whenever "disp_in" changes its value and using the case-when construct, it maps the integer input value to the corresponding binary output value. Additionally, a default case is included to handle input values outside the range of 0-9.



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

entity Seven_segment is
   port (disp_in: in integer range 0 to 9;
         disp_out: out std_logic_vector(7 downto 0));
end Seven_segment;

architecture display_arc of Seven_segment is
   begin
      process(disp_in)
         begin
            case disp_in is
               when 0 => disp_out<="11000000"; -- output a 0 = led acceso nel display
               when 1 => disp_out<="11111001";
               when 2 => disp_out<="10100100";
               when 3 => disp_out<="10110000";
               when 4 => disp_out<="10011001";
               when 5 => disp_out<="10010010";
               when 6 => disp_out<="10000010";
               when 7 => disp_out<="11111000";
               when 8 => disp_out<="10000000";
               when 9 => disp_out<="10010000";
               when others => disp_out<="11000000";
            end case;

      end process;
end display_arc;
```

Figure 20: Seven segment code

3.7 LFSR

The "LFSR.vhd" file implements a pseudo-random number generator based on a Linear Feedback Shift Register (LFSR), used to generate the x-coordinate value for obstacles.

An LFSR is a type of shift register in which input data is generated by a linear function of its internal state, by the exclusive OR (XOR) of some stored bits within the registers. Because the register operation is deterministic, the sequence of values produced by the register is completely determined by its current or previous state. Similarly, because the register has a finite number of possible states, after a certain number of cycles, the output values repeat. [1]

Within the file, there is the declaration of the LFSR entity, which includes an input "clk" of type STD_LOGIC representing a clock signal, and an output "random_number" of type STD_LOGIC_VECTOR, an 11-bit vector containing the generated value.

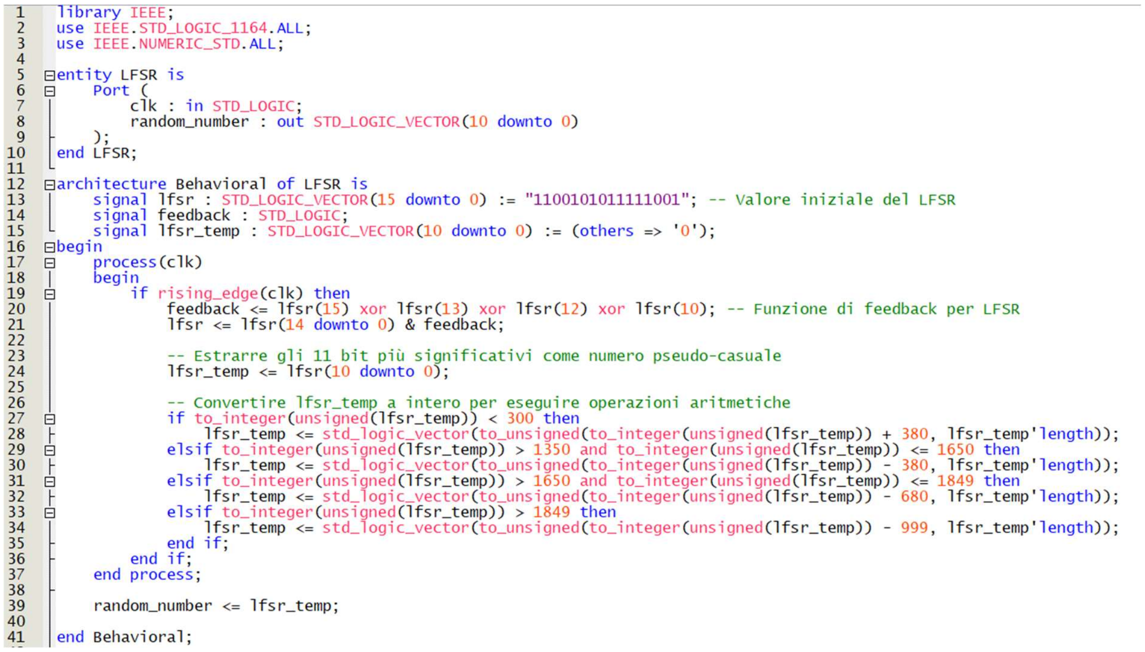
In the architecture of the file, three signals are declared: "lfsr," a 16-bit STD_LOGIC_VECTOR for storing the current state of the register; "feedback," a STD_LOGIC representing the calculated feedback bit; and "lfsr_temp," an 11-bit STD_LOGIC_VECTOR initialized to zero, which stores a value extracted from lfsr.

The "lfsr" signal is initialized with the value "1100101011111001," chosen to ensure a good distribution of 1s and 0s, facilitating a well-distributed pseudo-random sequence and avoiding an initial value of zero, which would halt generation of zeros.

Starting from line 17, there is a process where at each rising edge of the clock, the feedback bit is computed by performing an XOR operation between bits located at positions 15, 13, 12, and 10 of the lfsr. The lfsr is then updated by shifting all bits to the right by one position and inserting the feedback bit into the least significant position (LSB).

At line 24, the 11 most significant bits are extracted from the register and stored in the temporary variable "lfsr_temp," representing the generated binary value. To ensure the extracted number falls within the range of pixels corresponding to the game area, it is checked whether the value of lfsr_temp, properly converted to an integer, lies within that range. If not, it is adjusted by an offset to bring it within range.

Upon completion of the process execution, the temporary value is assigned to the output variable "random_number".



```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity LFSR is
    Port (
        clk : in STD_LOGIC;
        random_number : out STD_LOGIC_VECTOR(10 downto 0)
    );
end LFSR;

architecture Behavioral of LFSR is
    signal lfsr : STD_LOGIC_VECTOR(15 downto 0) := "1100101011111001"; -- Valore iniziale del LFSR
    signal feedback : STD_LOGIC;
    signal lfsr_temp : STD_LOGIC_VECTOR(10 downto 0) := (others => '0');
begin
    process(clk)
    begin
        if rising_edge(clk) then
            feedback <= lfsr(15) xor lfsr(13) xor lfsr(12) xor lfsr(10); -- Funzione di feedback per LFSR
            lfsr <= lfsr(14 downto 0) & feedback;

            -- Estrarre gli 11 bit più significativi come numero pseudo-casuale
            lfsr_temp <= lfsr(10 downto 0);

            -- Convertire lfsr_temp a intero per eseguire operazioni aritmetiche
            if to_integer(unsigned(lfsr_temp)) < 300 then
                lfsr_temp <= std_logic_vector(to_unsigned(to_integer(unsigned(lfsr_temp)) + 380, lfsr_temp'length));
            elsif to_integer(unsigned(lfsr_temp)) > 1350 and to_integer(unsigned(lfsr_temp)) <= 1650 then
                lfsr_temp <= std_logic_vector(to_unsigned(to_integer(unsigned(lfsr_temp)) - 380, lfsr_temp'length));
            elsif to_integer(unsigned(lfsr_temp)) > 1650 and to_integer(unsigned(lfsr_temp)) <= 1849 then
                lfsr_temp <= std_logic_vector(to_unsigned(to_integer(unsigned(lfsr_temp)) - 680, lfsr_temp'length));
            elsif to_integer(unsigned(lfsr_temp)) > 1849 then
                lfsr_temp <= std_logic_vector(to_unsigned(to_integer(unsigned(lfsr_temp)) - 999, lfsr_temp'length));
            end if;
        end if;
    end process;

    random_number <= lfsr_temp;

end Behavioral;
```

Figure 21: LFSR code

4 Conclusions

After writing the code and connecting the necessary hardware components, the FPGA board is connected to the computer to run the game.

During project development, a game for FPGA was developed using VHDL. The main objectives included implementing the game logic, managing graphics for visual feedback, and implementing movements using external hardware.

"Avoid Clash" offered us the chance to explore various aspects of FPGA programming, utilizing different architectures and software-hardware interconnections, debugging, simulation, and dividing the project into multiple components.

The project presented challenges and difficulties, including synchronization issues due to the need for appropriate clocks for individual elements, managing reset to ensure game restoration at any time, and implementing code to generate pseudo-random numbers within a predefined range. However, a game with semi-complex logic was successfully realized.

There are several areas where future improvements could be made, such as implementing different difficulty levels by increasing obstacle speed, adding blocks to collect for life recovery, optimizing VHDL code, and incorporating additional features to enhance the gaming experience.

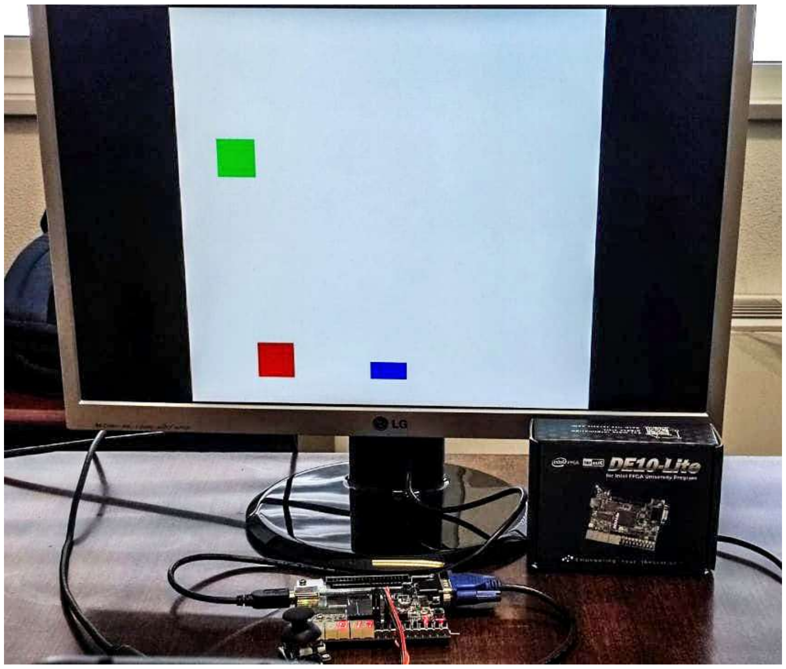


Figure 22: Setup of the game

Index of images

References

Linear Feedback Shift Register. (n.d.). Retrieved from https://it.wikipedia.org/wiki/Registro_a_scorrimento_a_retroazione_lineare

Martino, D. C. (2024). “Laboratory and VHDL Programming Slides.

Terasic. (2020). *DE10-Lite User Manual.*