POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE

Digital Programmable Systems

Prof. Dr. Eng. Martino De Carlo

Documentation title:

Snake game on FPGA

Student:
Nicola Saltarelli

ANNO ACCADEMICO 2023-2024

Abstract

This documentation describes the implementation of the Snake game on the FPGA DE10-Lite platform using the VHDL hardware description language. The software used for the game's implementation is Quartus Prime version 20.1.1.

Snake is a classic game, originating in the 1970s, which gained enormous popularity with the spread of Nokia mobile phones in the late 1990s. The concept of the game is simple: the player controls a snake that moves within a game area. The objective is to eat the food that appears at random points on the screen, causing the snake to grow by one unit each time a piece of food is consumed. The game ends when the snake collides with itself or the edges of the game area and has no more lives.

In the following sections of the documentation, the main components of the project, their organization, and the logic underlying the game's operation will be illustrated.

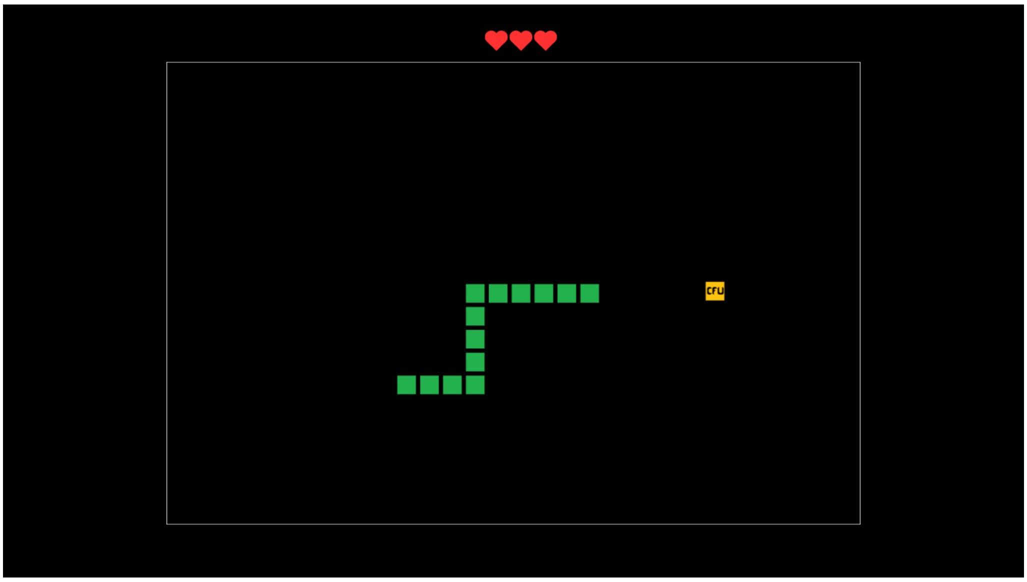

Figure 1: Game graphics.

Contents

1 Project Architecture

In this section, the schematic of the project will be presented (Figure 2), which is characterized by three distinct blocks.

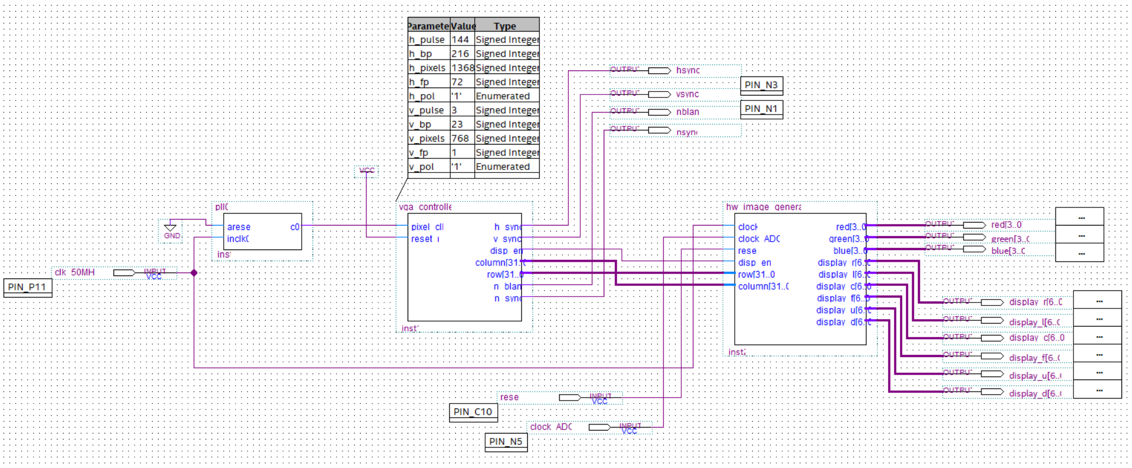

Figure 2: Schematic of the project.

The first block, named "pll0", the PLL0 (Phase-Locked Loop 0) in Quartus Prime is a component dedicated to generating a divided clock signal from an input clock. In this instance, the PLL0 receives as input the 50 MHz clock signal from the FPGA board and a reset signal constantly held at a logic low level.
The consistently low reset signal indicates that the PLL0 should not be initialized or reset at system startup. In other words, the PLL0 begins operation immediately without needing to go through a reset phase.
Essentially, Quartus Prime's PLL0 acts as a clock divider, allowing for the generation of clock signals at desired frequencies from a base input clock. In the project folder, there is a guide dedicated to the generation of PLL0, titled "Using the VGA.pdf", provided by Prof. Dr. Eng. Martino De Carlo.

The vga_controller block is responsible for managing the synchronization and control signals necessary for correct visual output. This module interfaces directly with the FPGA's clock signal provided by the "pll0" block and with a constant logic level reset input, ensuring uninterrupted operation without the need for initialization procedures upon system startup.
This module provides several outputs, each with a defined purpose in coordinating the display process:

- Horizontal and Vertical Sync Signals (hsync, vsync): These signals are assigned to specific pins on the FPGA, as outlined in the board's manual. They facilitate precise timing for horizontal and vertical synchronization, ensuring accurate image rendering.
- Display Enable Signal (disp_ena): This signal determines the activation status of the display. By controlling the display enablement, it allows dynamic control over when images are presented on the screen.
- Column and Row Position Signals (column[31..0], row[31..0]): These signals determine the exact positioning of pixels within the display. They serve as inputs to the subsequent

"hw_image_generator," guiding the generation of graphics and game logic with pixel-level accuracy.

- Blanking and Synchronization Signals (nblank, nsync): These signals provide additional information about the display's status. They indicate whether the current pixel is within the active display area or part of the synchronization intervals, though they are not utilized in this project.

It is important to specify that the parameters used in the vga_controller are specific to the Samsung LT19B300 monitor used in the project. Therefore, these parameters must be modified if the code in the project folder is to be reused with a different monitor.

Finally, the third block that characterizes the schematic is the "hw_image_generator," which constitutes the core of the project as it handles the generation of images and the game logic. This block receives several fundamental inputs necessary for the game's operation:

- 50MHz Clock from the board: Used for general timing and game operation.
- 10MHz Clock from the ADC: Used for specific functions related to the ADC (Analog-to-Digital Converter).
- Reset: Used to initialize the game to its default conditions.

Additionally, the "hw_image_generator" receives three inputs from the "vga_controller":

- disp_enable.
- row[31..0].
- column[31..0].

The outputs of the "hw_image_generator" include:

- red[3..0], blue[3..0], green[3..0]: These signals represent the red, blue, and green color channels for the display. Each channel consists of 4 bits, allowing the definition of color intensity and their combination to achieve full color representation on the screen.
- Other outputs related to the 7-segment display: These outputs are used to display additional information on the 7-segment display present on the FPGA board.

Further details on the operation of this block will be explained in the following section, where the VHDL code will be discussed in more detail.

2 Codes description

In this section of the documentation, the VHDL code implemented will be described. This will provide a better understanding of the design choices made and the underlying logic of the game. The code has been structured following modular principles, with "hw_image_generator.vhd" serving as the main component. However, within this code, only the graphical aspects are managed, while the game logic is delegated to separate entities such as "snake_entity.vhd," "food.vhd," "score.vhd," and "joystick.vhd."
These modules have been carefully designed to handle specific functionalities of the game, ranging from snake movement and food interaction to scorekeeping and joystick input processing.
This structured approach to coding offers several advantages. Firstly, it enhances readability and comprehensibility, as each module is responsible for a well-defined aspect of the game's functionality. This facilitates problem-solving, maintenance, and future modifications. Additionally, the modular approach promotes reusability, allowing individual components to be used in other projects or scenarios requiring similar functionality.

2.1 Snake movement and logic

The "snake_entity.vhd" code is responsible for managing the snake's movement logic, collision detection, and the increase in the snake's length when it eats food. The flowchart in Figure 3 provides a simplified representation of the "snake_logic" process within the architecture of the "snake_entity" entity.

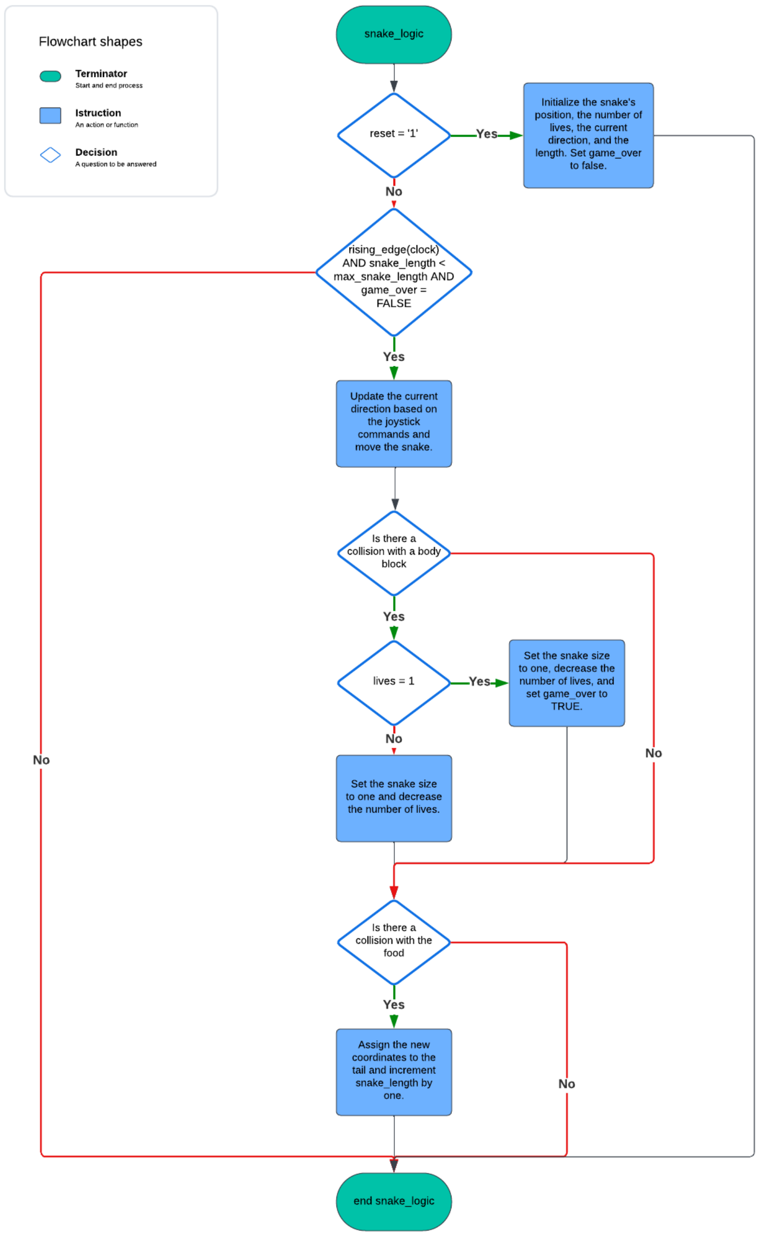

Figure 3: Snake logic flowchart.

We will now examine specific portions of the code, focusing particularly on how snake movement occurs and how collisions with the game boundaries, itself, and food are managed.

The following code (Code 1) is responsible for the snake's movement within the game, employing a mechanism similar to a shift register.
A shift register is a type of register where data is shifted from one flip-flop to the next. In an n-bit shift register, each bit takes the value of the previous bit, creating a propagation effect of values through the register. In the case of the snake, the coordinates of each segment are moved to the position of the previous segment, similar to how data is shifted in a register. The head of the snake receives new coordinates (managed separately in the code), and these coordinates propagate along the snake's body, updating the position of each segment.

Code 1: Move snake's body

```
FOR i IN max_snake_length - 1 DOWNTO 1 LOOP
    IF i < snake_length THEN
       internal_snake(i) ≤ internal_snake(i - 1);
    END IF;
END LOOP;
```

The ultimate effect of this mechanism is that the snake appears to move smoothly across the game grid. When the snake's head changes position (e.g., in response to a direction command), this new position is transmitted along the snake's body, creating the illusion of continuous and coherent movement. Each segment of the body follows the preceding segment, maintaining the shape and integrity of the snake during its motion.

Another code segment worth commenting on is the one concerning the movement of the snake's head based on the current direction stored in the variable "current_direction". (Code 2)

Code 2: Move snake's head

```
CASE current_direction IS
    WHEN "00" => IF internal_snake(0).y - rect_height > game_top THEN ...
        internal_snake(0).y ≤ internal_snake(0).y - rect_height; END IF;  -- Up
    WHEN "10" => IF internal_snake(0).x - rect_width > game_left THEN ...
        internal_snake(0).x ≤ internal_snake(0).x - rect_width; END IF;   -- Left
    WHEN "11" => IF internal_snake(0).y + rect_height < game_bottom THEN ...
        internal_snake(0).y ≤ internal_snake(0).y + rect_height; END IF;-- Down
    WHEN "01" => IF internal_snake(0).x + rect_width < game_right THEN ...
        internal_snake(0).x ≤ internal_snake(0).x + rect_width; END IF;   -- Right
    WHEN OTHERS => NULL;
END CASE;
```

For each possible value of "current_direction", conditional checks are performed to determine if the snake's head can move in that direction, ensuring that such movement adheres to the game's boundaries. This approach ensures that the snake's movement is controlled and consistent with the rules and boundaries defined by "game_top", "game_left", "game_bottom", and "game_right" variables, along with the dimensions of the snake's rectangle ("rect_width" and "rect_height").

The final code section we will examine in this subsection concerns collisions, which can occur either with the snake's body or with food. Collisions with the game boundary are not explicitly handled because the snake is constrained within the boundaries by first checking the head coordinates (Code 2). Therefore, when the snake reaches the edge of the game field, on the next clock cycle, the coordinates of two snake blocks align, triggering a collision with its own body. This explains why the snake, when it has a length of one, can collide with the boundaries without losing lives.

Code 3: Snake collisions

```
-- Check for collisions
FOR i IN 1 TO max_snake_length - 1 LOOP
    IF i < snake_length AND internal_snake(0).x = internal_snake(i).x AND ...
        internal_snake(0).y = internal_snake(i).y THEN
        IF lives = 1 THEN
            game_over ≤ TRUE;
            lives ≤ lives - 1;
            snake_length ≤ 1;  -- Reset game
        ELSE
            lives ≤ lives - 1;
            snake_length ≤ 1;  -- Reset game
        END IF;
    END IF;
END LOOP;
-- Check for food
IF internal_snake(0).x = food_x AND internal_snake(0).y = food_y THEN
    internal_snake(snake_length).x ≤ internal_snake(snake_length - 1).x;
    internal_snake(snake_length).y ≤ internal_snake(snake_length - 1).y;
    snake_length ≤ snake_length + 1;
END IF;
```

Code 3 manages two critical aspects of the snake game: collision detection and food consumption handling.

The first FOR loop iterates through all segments of the snake (excluding the head) to check if the head ("internal_snake(0)") collides with any body segment ("internal_snake(i)"). If the coordinates "x" and "y" of the head match those of a body segment and the snake has more than one segment ("i $<$ snake_length"), a collision is detected. If the number of lives is one, the game sets "game_over" to TRUE, decreases the remaining lives by one, and resets the snake's length to one. If lives are greater than one, only the number of lives is decremented and the snake's length is reset to one to restart the game.

After collision checks, the code verifies if the snake's head ("internal_snake(0)") overlaps with the food position ("food_x" and "food_y"). If so, the snake eats the food. The coordinates of the new tail segment ("internal_snake(snake_length)") are set equal to the coordinates of the previous tail segment ("internal_snake(snake_length - 1)"). The snake's length is incremented by one to add a new segment to the snake's tail.

This mechanism ensures proper functionality of the snake game by managing collisions with the snake's own body and food consumption, dynamically updating the snake's length and available lives.

2.2 Food logic

The food logic has been implemented in a dedicated VHDL file named "food.vhd" within the project folder. This file manages the random positioning of food items. The operational logic is illustrated in the flowchart depicted in Figure 4.

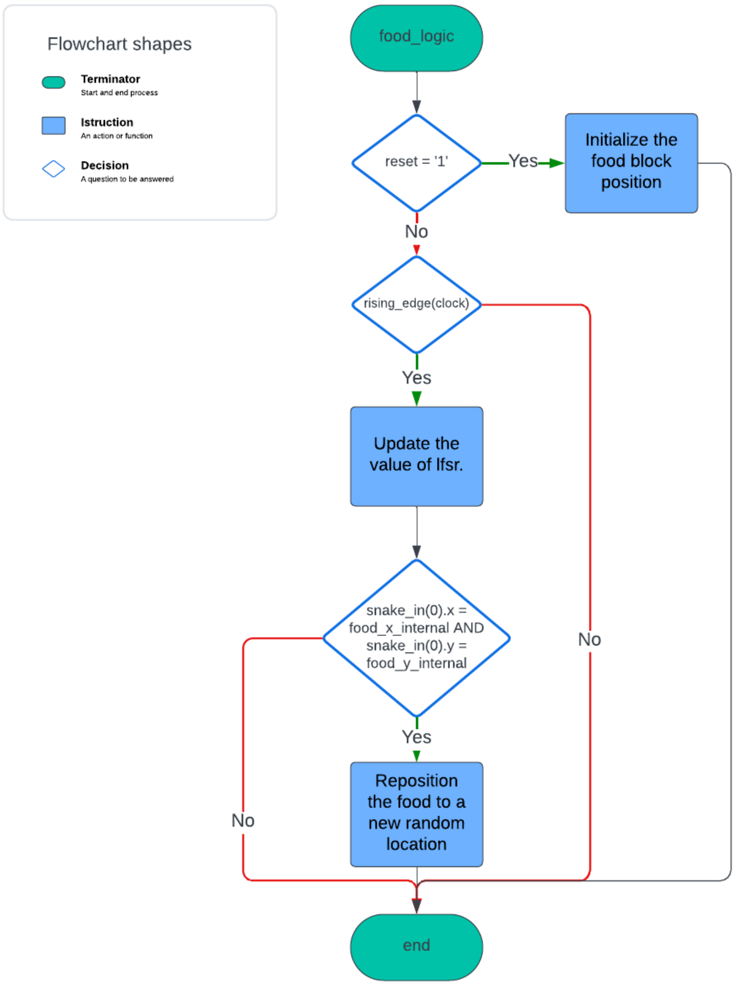

Figure 4: Food logic flowchart.

To facilitate random positioning of food, a dedicated function named "random_number" has been implemented in the code (Code 4), as VHDL lacks built-in functions for generating random numbers. This function utilizes a Linear Feedback Shift Register (LFSR) to generate random numbers. An LFSR is a type of register where bits are cyclically shifted, and the least significant bit (LSB) is generated based on previous bits using XOR operations. This method ensures a sequence of numbers that appear random but are deterministic, meaning that initializing the register with the same initial state will produce the same sequence of numbers every time. The LFSR implementation within the "random_number" function enables the generation of random numbers necessary for dynamic food positioning in the game.

Code 4: Random number generator

```
1  FUNCTION random_number(min_val, max_val : INTEGER; lfsr_val : STD_LOGIC_VECTOR(15 ...
       DOWNTO 0)) RETURN INTEGER IS
2      VARIABLE lfsr_temp : STD_LOGIC_VECTOR(15 DOWNTO 0);
3      VARIABLE random_val : INTEGER;
4  BEGIN
5      -- Copy the initial LSFR value to a temporary variable
6      lfsr_temp := lfsr_val;
7
8      -- Perform one iteration of the LSFR
9      lfsr_temp := lfsr_temp(14 DOWNTO 0) & (lfsr_temp(15) XOR lfsr_temp(13) XOR ...
           lfsr_temp(12) XOR lfsr_temp(10));
10
11     -- Convert the LSFR value to an integer
12     random_val := to_integer(unsigned(lfsr_temp));
13
14     -- Limit the value within the specified range
15     RETURN (random_val MOD (max_val - min_val + 1)) + min_val;
16 END FUNCTION;
```

The parameters of the "random_number" function are "min_val" and "max_val", which define the range within which we want to generate the random number, and "lfsr_val", which is the initial value of the LFSR represented as a 16-bit vector (from 15 to 0). The local variables are "lfsr_temp", a temporary variable that represents the current state of the LFSR during the function's execution, and "random_val", the variable where we store the value calculated as the random number.
The main instruction is the one in line 9. This instruction represents a feedback loop of the LFSR. We shift all the bits of the LFSR one position to the left (discarding the MSB) and calculate the new bit to insert into the register. The new bit is obtained through a combination of XOR operations between specific bits of the LFSR. In this specific case, bits in positions 15, 13, 12, and 10 are used to calculate the new bit. This step is illustrated more clearly in Figure 5.

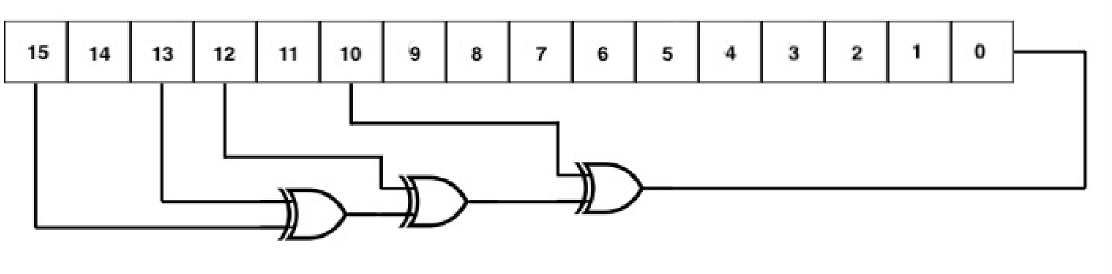

Figure 5: LFSR working principle.

2.3 Score module

The VHDL code named "score.vhd" is designed to manage the score display on 7-segment displays. Specifically, it uses all six 7-segment displays provided by the board to show constants represented by the letters C, F, U, and the dash, in addition to two displays dedicated to showing the score, which can range from 0 to 31. This is why two displays are required for the score.
The code features a single process named "display" that activates whenever the value of point changes. If point is between 0 and 32, it calculates the unit digit and the tens digit. It uses the "get_display_code" function to obtain the corresponding 7-segment codes. This approach ensures that the display management for each number is handled automatically, minimizing the lines of code and eliminating the need to manage each of the 32 displayed numbers individually.

2.4 Joystick module

The VHDL code "joystick.vhd" utilizes the integrated ADC in the FPGA device to acquire analog values from the X and Y axes of a joystick. These analog signals are converted into digital values and interpreted to determine the direction of joystick movement. The "unnamed" component acts as an interface between the main VHDL code and the ADC hardware, enabling monitoring of the joystick position. To accurately replicate the game's functionality, it is crucial to create this component following the detailed instructions provided in the document "ADC with DE-series boards.pdf" by Professor Dr. Eng. Martino De Carlo, available in the project folder.

The VHDL file "joystick.vhd" contains a single process that reads the analog values of the X and Y axes of the joystick ("joystickReadingX" and "joystickReadingY") and converts them into integer values. Using these data, it determines the direction of the joystick and sets the direction signal accordingly. The chosen values for "homeX" and "homeY" were carefully measured using dedicated code (Code 12) to represent the resting values on the X and Y axes. The tolerance constant indicates the sensitivity of the joystick; a lower value corresponds to higher sensitivity. Through testing, it was found that a tolerance of 400 allows smooth and precise execution of commands, as demonstrated in the video provided in the project folder.

Code 12 presents the same operational logic as the codes named "joystick.vhd" and "score.vhd" which can be viewed in the appendix of the documentation. This code is designed to read input from a joystick and display the corresponding values on seven-segment displays. Specifically, four seven-segment displays are used because a 12-bit ADC is employed, and thus the binary value read from the ADC, once converted to a decimal number, has a maximum of four digits.

The code outputs are described as follows:

- display_r: 7-bit logic vector for displaying the rightmost digit.
- display_mr: 7-bit logic vector for displaying the second digit from the right.
- display_ml: 7-bit logic vector for displaying the third digit from the right.
- display_l: 7-bit logic vector for displaying the leftmost digit.

Unlike the "joystick.vhd" code, in this case, only one channel of the ADC is used. To replicate the code's functionality, in addition to following the guide for creating the "unnamed" component, the correct pin assignments must be made by referring to the FPGA board manual. It is crucial to properly connect the joystick pins on the board, noting that the pin for analog reading is ADC_IN0, as indicated on page 32 of the manual.

2.5 Display logic

The "hw_image_generator.vhd" code assumes a fundamental role in managing the graphical representation of the game. Within this code, in addition to the fundamental logic for image generation, several components are integrated that define the entire gameplay dynamics. These components include "snake_entity" for snake management, "food" for handling food items, "score" for scoring, and "joystick" for user interaction. This structural design not only facilitates clear and modular management of the game logic but also utilizes the "hw_image_generator.vhd" file as a central point for effectively assigning input and output parameters to each component.

The two processes that characterize the architecture are "my_clockDivider" and "display_logic". The "my_clockDivider" process plays a crucial role in the game's timing, determining, for instance, the speed of the snake's movement. The FPGA clock runs at 50 MHz and is divided to produce a 10 Hz clock. It is important to specify that the effective result is 10 Hz because the period of "clk_out_signal" is defined by both a rising and falling edge.

Code 5: my_clockDivider

```
my_clockDivider: PROCESS(clock)
BEGIN
    IF rising_edge(clock) THEN
        IF count = "1001100010010110011111" THEN
            clk_out_signal <= not clk_out_signal;
            count <= (OTHERS => '0');
        ELSE
            count <= count + '1';
        END IF;
    END IF;
END PROCESS my_clockDivider;
```

The "display_logic" process is responsible for managing the display of various components on the screen. Depending on the current situation, the appropriate components will be shown. If no lives are remaining and thus it is a game over condition, the game over graphic will be displayed. If the maximum score is reached, the victory graphic will be shown. In the absence of these two conditions, the playing field with the snake and the food will be displayed.

All the graphics created were converted using a dedicated Python script, found in the appendix, into matrices of RGB values. These matrices represent each individual pixel of the image and are commonly referred to in VHDL as "Sprites". Sprites related to the snake, food, game over screen, victory screen, and life indicators have been saved within the project package, which contains all the constants and data types used in the code.
To accurately represent the snake and food sprites, the X and Y coordinates indicate the center point of a 31x31 pixel block. Specifically, the playing field of 930x620 pixels has been divided into a grid of 20 rows and 30 columns of 31x31 pixel blocks. This division allows the snake to move across all areas of the field and ensures proper interaction with the food items.

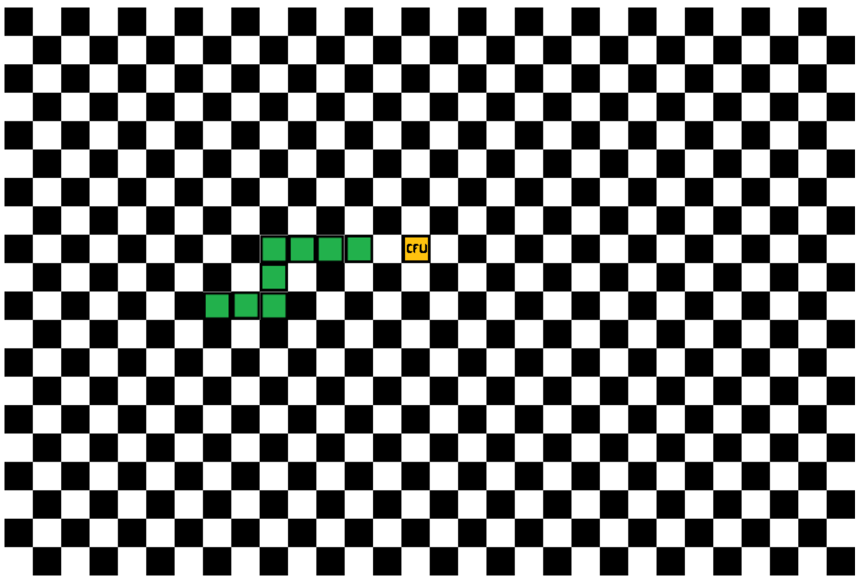

Figure 6: Game grid.

Now we will comment on the code related to the generation of the snake's graphics. The same logic applies to other components displayed on the screen, which are omitted from this documentation for simplicity to avoid redundancy.

Code 6: Draw snake

```
FOR i IN 0 TO max_snake_length - 1 LOOP
    IF i < snake_length THEN
        IF row ≥ internal_snake(i).y - ((rect_height-1) / 2) AND row ≤ ...
            internal_snake(i).y + ((rect_height-1) / 2) AND
        column ≥ internal_snake(i).x - ((rect_width-1) / 2) AND column ≤ ...
            internal_snake(i).x + ((rect_width-1) / 2) THEN
            -- Calculate the relative coordinates within the 31x31 block
            snake_row := row - (internal_snake(i).y - ((rect_height-1) / 2));
            snake_col := column - (internal_snake(i).x - ((rect_width-1) / 2));
            -- Retrieve the pixel color from the array
            red   ≤ snake_image(snake_row, snake_col)(11 DOWNTO 8);
            green ≤ snake_image(snake_row, snake_col)(7 DOWNTO 4);
            blue  ≤ snake_image(snake_row, snake_col)(3 DOWNTO 0);
        END IF;
    END IF;
END LOOP;
```

As shown in Code 6, to display the snake on the monitor, each segment of the snake is iterated through with a FOR loop. It checks if the current row and column fall within the coordinates of the snake segment, specified by the segment's X and Y coordinates and the rectangle's dimensions ("rect_width" and "rect_height"). Then, the relative coordinates within the segment block are calculated. Finally, the values for the red, green, and blue components of the current pixel are set, retrieving them from the "snake_image" array that contains the snake's image data.

A VHDL codes

Code 7: hw_image_generator.vhd

```
-- Import IEEE standard libraries
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_arith.all;

-- Import custom package for the snake game
USE work.snake_pkg.ALL;

ENTITY hw_image_generator IS
    PORT(
      clock       : IN  STD_LOGIC;      -- Clock signal
      clock_ADC   : IN  STD_LOGIC;      -- ADC clock signal
      reset       : IN  STD_LOGIC;      -- Reset signal
      disp_ena    : IN  STD_LOGIC;      -- Display enable ('1' = display time, ...
          '0' = blanking time)
      row         : IN  INTEGER;        -- Pixel row coordinate
      column      : IN  INTEGER;        -- Pixel column coordinate
      red         : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');  -- Red ...
          color output
      green       : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');  -- ...
          Green color output
      blue        : OUT STD_LOGIC_VECTOR(3 DOWNTO 0) := (OTHERS => '0');  -- Blue ...
          color output
```

```
        display_r    : OUT STD_LOGIC_VECTOR (6 downto 0);
        display_l    : OUT STD_LOGIC_VECTOR (6 downto 0);
        display_c    : OUT STD_LOGIC_VECTOR (6 downto 0);
        display_f    : OUT STD_LOGIC_VECTOR (6 downto 0);
        display_u    : OUT STD_LOGIC_VECTOR (6 downto 0);
        display_d    : OUT STD_LOGIC_VECTOR (6 downto 0)
    );
END hw_image_generator;

-- Architecture definition for the hw_image_generator entity
ARCHITECTURE behavior OF hw_image_generator IS

    -- Internal signals for clock divider process
    SIGNAL clk_out_signal    : STD_LOGIC := '0';
    SIGNAL count                 : STD_LOGIC_VECTOR (21 downto 0) := (OTHERS => '0');

    -- Internal signals for components
    SIGNAL snake_length          : INTEGER;
    SIGNAL lives                 : INTEGER;
    SIGNAL internal_snake    : snake_array;
    SIGNAL food_x                : INTEGER;
    SIGNAL food_y                : INTEGER;
    SIGNAL game_over             : BOOLEAN;
    SIGNAL direction         : STD_LOGIC_VECTOR(1 DOWNTO 0);

    -- Snake entity component declaration
    COMPONENT snake_entity
        PORT (
            clock            : IN STD_LOGIC;
            reset            : IN STD_LOGIC;
            food_x           : IN INTEGER;
            food_y           : IN INTEGER;
            direction        : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
            snake_out       : OUT snake_array;
            snake_len_out   : OUT INTEGER;
            gameOver_out    : OUT BOOLEAN;
            lives_out       : OUT INTEGER
        );
    END COMPONENT;

    -- Food entity component declaration
    COMPONENT food
        PORT (
            clock                : IN STD_LOGIC;
            reset            : IN STD_LOGIC;
            food_x           : OUT INTEGER;
            food_y           : OUT INTEGER;
            snake_in         : IN snake_array
        );
    END COMPONENT;

    -- Score entity component declaration
    COMPONENT score
        PORT (
            display_r    : OUT STD_LOGIC_VECTOR (6 downto 0);
            display_l    : OUT STD_LOGIC_VECTOR (6 downto 0);
            display_c    : OUT STD_LOGIC_VECTOR (6 downto 0);
            display_f    : OUT STD_LOGIC_VECTOR (6 downto 0);
            display_u    : OUT STD_LOGIC_VECTOR (6 downto 0);
            display_d    : OUT STD_LOGIC_VECTOR (6 downto 0);
            point            : IN INTEGER
        );
    END COMPONENT;

    -- Joystick entity component declaration
```

```
    COMPONENT joystick
        PORT (
            clock       : IN STD_LOGIC;
            reset           : IN STD_LOGIC;
            direction   : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
        );
    END COMPONENT;


BEGIN

     -- Instantiate snake entity
    S1 : snake_entity PORT MAP (
        clock           => clk_out_signal,
        reset           => reset,
        food_x          => food_x,
        food_y          => food_y,
        direction       => direction,
        snake_out       => internal_snake,
        snake_len_out   => snake_length,
        gameOver_out    => game_over,
        lives_out       => lives
    );

    -- Instantiate food entity
    S2 : food PORT MAP (
        clock    => clk_out_signal,
        reset    => reset,
        food_x   => food_x,
        food_y   => food_y,
        snake_in => internal_snake
    );

    -- Instantiate score entity
    S3 : score PORT MAP (
        display_r    => display_r,
        display_l    => display_l,
        display_c    => display_c,
        display_f    => display_f,
        display_u    => display_u,
        display_d    => display_d,
        point        => snake_length
    );

    -- Instantiate joystick entity
    S4 : joystick PORT MAP (
        clock        => clock_ADC,
        reset        => reset,
        direction    => direction
    );

    -- Clock divider process
    my_clockDivider: PROCESS(clock)
    BEGIN
        IF rising_edge(clock) THEN
            IF count = "1001100010010110011111" THEN
                clk_out_signal <= not clk_out_signal;
                count <= (OTHERS => '0');
            ELSE
                count <= count + '1';
            END IF;
        END IF;
    END PROCESS my_clockDivider;

    -- Display logic process
```

```
    display_logic: PROCESS(disp_ena, row, column)
        VARIABLE i                    : INTEGER;
        VARIABLE snake_row       : INTEGER;
        VARIABLE snake_col       : INTEGER;
        VARIABLE food_row        : INTEGER;
        VARIABLE food_col        : INTEGER;
        VARIABLE gameOver_row    : INTEGER;
        VARIABLE gameOver_col    : INTEGER;
        VARIABLE live_row        : INTEGER;
        VARIABLE live_col        : INTEGER;
        VARIABLE victory_row     : INTEGER;
        VARIABLE victory_col     : INTEGER;
    BEGIN
        IF disp_ena = '1' THEN   -- Display time
            red ≤ (OTHERS => '0');
            green ≤ (OTHERS => '0');
            blue ≤ (OTHERS => '0');

            IF game_over = FALSE THEN

                IF snake_length = max_snake_length THEN

                    -- Draw victory screen
                    IF column ≥ game_left + (game_width/2) - 156 AND column < ...
                        game_left + (game_width/2) + 156 AND
                        row ≥ game_top + (game_height/2) - 20 AND row < game_top ...
                            + (game_height/2) + 20 THEN
                        -- Calculate relative coordinates within the block
                        victory_row := row - (game_top + (game_height/2) - 20);
                        victory_col := column - (game_left + (game_width/2) - 156);
                        -- Get the pixel color from the array
                        red   ≤ victory_image(victory_row, victory_col)(11 DOWNTO 8);
                        green ≤ victory_image(victory_row, victory_col)(7 DOWNTO 4);
                        blue  ≤ victory_image(victory_row, victory_col)(3 DOWNTO 0);
                    END IF;

                ELSE

                    -- Draw snake
                    FOR i IN 0 TO max_snake_length - 1 LOOP
                        IF i < snake_length THEN
                            IF row ≥ internal_snake(i).y - ((rect_height-1) / 2) ...
                                AND row ≤ internal_snake(i).y + ((rect_height-1) ...
                                / 2) AND
                                column ≥ internal_snake(i).x - ((rect_width-1) / ...
                                    2) AND column ≤ internal_snake(i).x + ...
                                    ((rect_width-1) / 2) THEN
                                 -- Calculate relative coordinates within the ...
                                     31x31 block
                                snake_row := row - (internal_snake(i).y - ...
                                    ((rect_height-1) / 2));
                                snake_col := column - (internal_snake(i).x - ...
                                    ((rect_width-1) / 2));
                                -- Get the pixel color from the array
                                red   ≤ snake_image(snake_row, snake_col)(11 ...
                                    DOWNTO 8);
                                green ≤ snake_image(snake_row, snake_col)(7 ...
                                    DOWNTO 4);
                                blue  ≤ snake_image(snake_row, snake_col)(3 ...
                                    DOWNTO 0);
                            END IF;
                        END IF;
                    END LOOP;

                    -- Draw food
```

```
                    IF row ≥ food_y - ((rect_height-1) / 2) AND row ≤ food_y + ...
                        ((rect_height-1) / 2) AND
                        column ≥ food_x - ((rect_width-1) / 2) AND column ≤ ...
                            food_x + ((rect_width-1) / 2) THEN
                        -- Calculate relative coordinates within the 31x31 block
                        food_row := row - (food_y - ((rect_height-1) / 2));
                        food_col := column - (food_x - ((rect_width-1) / 2));
                        -- Get the pixel color from the array
                        red   ≤ food_image(food_row, food_col)(11 DOWNTO 8);
                        green ≤ food_image(food_row, food_col)(7 DOWNTO 4);
                        blue  ≤ food_image(food_row, food_col)(3 DOWNTO 0);
                    END IF;

                    -- Draw gray lines for game field border
                    IF (row ≥ game_top AND row ≤ game_bottom AND (column = ...
                        game_left OR column = game_right)) OR
                        (column ≥ game_left AND column ≤ game_right AND (row = ...
                            game_top OR row = game_bottom)) THEN
                        red ≤ "1000";
                        green ≤ "1000";
                        blue ≤ "1000";
                    END IF;

                     -- Draw game lives
                    IF lives = 1 THEN
                        IF column ≥ game_left+(game_width/2)-15 AND column ≤ ...
                            game_left+(game_width/2)+15 AND
                            row ≥ game_top-41 AND row ≤ game_top-10 THEN
                            -- Calculate relative coordinates within the block
                            live_row := row - (game_top-41);
                            live_col := column - (game_left+(game_width/2)-15);
                            -- Get the pixel color from the array
                            red   ≤ live_image(live_row, live_col)(11 DOWNTO 8);
                            green ≤ live_image(live_row, live_col)(7 DOWNTO 4);
                            blue  ≤ live_image(live_row, live_col)(3 DOWNTO 0);
                        END IF;
                    ELSIF lives = 2 THEN
                        IF column ≥ game_left+(game_width/2)-30 AND column ≤ ...
                            game_left+(game_width/2)+31 AND
                            row ≥ game_top-41 AND row ≤ game_top-10 THEN
                            -- Calculate relative coordinates within the block
                            live_row := row - (game_top-41);
                            live_col := column - (game_left+(game_width/2)-30);
                            -- Get the pixel color from the array
                            red   ≤ live_image(live_row, live_col)(11 DOWNTO 8);
                            green ≤ live_image(live_row, live_col)(7 DOWNTO 4);
                            blue  ≤ live_image(live_row, live_col)(3 DOWNTO 0);
                        END IF;
                    ELSE
                        IF column ≥ game_left+(game_width/2)-46 AND column ≤ ...
                            game_left+(game_width/2)+46 AND
                            row ≥ game_top-rect_width-10 AND row ≤ game_top-10 THEN
                            -- Calculate relative coordinates within the block
                            live_row := row - (game_top-41);
                            live_col := column - (game_left+(game_width/2)-46);
                            -- Get the pixel color from the array
                            red   ≤ live_image(live_row, live_col)(11 DOWNTO 8);
                            green ≤ live_image(live_row, live_col)(7 DOWNTO 4);
                            blue  ≤ live_image(live_row, live_col)(3 DOWNTO 0);
                        END IF;
                    END IF;
                END IF;

            ELSE

```

```
                -- Draw game over screen
                IF column ≥ game_left + (game_width/2) - 184 AND column < ...
                    game_left + (game_width/2) + 184 AND
                    row ≥ game_top + (game_height/2) - 20 AND row < game_top + ...
                        (game_height/2) + 20 THEN
                    -- Calculate relative coordinates within the block
                    gameOver_row := row - (game_top + (game_height/2) - 20);
                    gameOver_col := column - (game_left + (game_width/2) - 184);
                    -- Get the pixel color from the array
                    red   ≤ gameOver_image(gameOver_row, gameOver_col)(11 DOWNTO 8);
                    green ≤ gameOver_image(gameOver_row, gameOver_col)(7 DOWNTO 4);
                    blue  ≤ gameOver_image(gameOver_row, gameOver_col)(3 DOWNTO 0);
                END IF;

            END IF;

        ELSE  -- Blanking time
            red ≤ (OTHERS => '0');
            green ≤ (OTHERS => '0');
            blue ≤ (OTHERS => '0');
        END IF;

    END PROCESS display_logic;

END behavior;
```

Code 8: snake_entity.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_arith.all;
USE work.snake_pkg.ALL;

ENTITY snake_entity IS
    PORT(
        clock           : IN STD_LOGIC;
        reset           : IN STD_LOGIC;
        food_x          : IN INTEGER;
        food_y          : IN INTEGER;
        direction       : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        snake_out      : OUT snake_array;
        snake_len_out  : OUT INTEGER;
        gameOver_out   : OUT BOOLEAN;
        lives_out      : OUT INTEGER
    );
END snake_entity;

ARCHITECTURE behavior OF snake_entity IS

    -- Internal signals
    SIGNAL internal_snake : snake_array := (
        OTHERS => (x => game_left + (rect_width*10)+((rect_width+1) / 2), y => ...
            game_top + (rect_width*10)+((rect_height+1) / 2))
    );  -- Initial position of the snake's head
    SIGNAL snake_length : INTEGER := 1; -- Initial snake length
    SIGNAL current_direction : STD_LOGIC_VECTOR(1 DOWNTO 0) := "00";  -- Initial ...
        direction: 00 = Up
    SIGNAL game_over : BOOLEAN;
    SIGNAL lives : INTEGER := 3; -- Initial number of lives

BEGIN
    -- Process for handling snake movement and game logic
    snake_logic: PROCESS(clock, reset)
```

```
        VARIABLE i : INTEGER;
    BEGIN
        IF reset = '1' THEN
            internal_snake(0).x ≤ game_left + (rect_width*10) + ((rect_width+1) / 2);
            internal_snake(0).y ≤ game_top + (rect_width*10) + ((rect_height+1) / 2);
            snake_length ≤ 1;
            current_direction ≤ "00";
            lives ≤ 3;
            game_over ≤ FALSE;

        ELSIF rising_edge(clock) AND snake_length < max_snake_length AND ...
            game_over = FALSE THEN

            -- Update direction based on joystick commands
            IF direction = "10" THEN
                CASE current_direction IS
                    WHEN "00" => current_direction ≤ "10";  -- Up -> Left
                    WHEN "10" => current_direction ≤ "10";  -- Left -> Left
                    WHEN "11" => current_direction ≤ "10";  -- Down -> Left
                    WHEN "01" => current_direction ≤ "01";  -- Right -> Right
                    WHEN OTHERS => NULL;
                END CASE;
            ELSIF direction = "01" THEN
                CASE current_direction IS
                    WHEN "00" => current_direction ≤ "01";  -- Up -> Right
                    WHEN "01" => current_direction ≤ "01";  -- Right -> Right
                    WHEN "11" => current_direction ≤ "01";  -- Down -> Right
                    WHEN "10" => current_direction ≤ "10";  -- Left -> Left
                    WHEN OTHERS => NULL;
                END CASE;
            ELSIF direction = "00" THEN
                CASE current_direction IS
                    WHEN "00" => current_direction ≤ "00";  -- Up -> Up
                    WHEN "01" => current_direction ≤ "00";  -- Right -> Up
                    WHEN "11" => current_direction ≤ "11";  -- Down -> Down
                    WHEN "10" => current_direction ≤ "00";  -- Left -> Up
                    WHEN OTHERS => NULL;
                END CASE;
            ELSIF direction = "11" THEN
                CASE current_direction IS
                    WHEN "00" => current_direction ≤ "00";  -- Up -> Up
                    WHEN "01" => current_direction ≤ "11";  -- Right -> Down
                    WHEN "11" => current_direction ≤ "11";  -- Down -> Down
                    WHEN "10" => current_direction ≤ "11";  -- Left -> Down
                    WHEN OTHERS => NULL;
                END CASE;
            END IF;

            -- Move the snake
            FOR i IN max_snake_length - 1 DOWNTO 1 LOOP
                IF i < snake_length THEN
                    internal_snake(i) ≤ internal_snake(i - 1);
                END IF;
            END LOOP;

            -- Use current_direction to move the snake
            CASE current_direction IS
                WHEN "00" => IF internal_snake(0).y - rect_height > game_top THEN ...
                    internal_snake(0).y ≤ internal_snake(0).y - rect_height; END ...
                    IF;  -- Up
                WHEN "10" => IF internal_snake(0).x - rect_width > game_left THEN ...
                    internal_snake(0).x ≤ internal_snake(0).x - rect_width; END ...
                    IF;   -- Left
                WHEN "11" => IF internal_snake(0).y + rect_height < game_bottom ...
                    THEN internal_snake(0).y ≤ internal_snake(0).y + rect_height; ...
```

```
                        END IF;-- Down
                    WHEN "01" => IF internal_snake(0).x + rect_width < game_right ...
                        THEN internal_snake(0).x <= internal_snake(0).x + rect_width; ...
                        END IF;   -- Right
                    WHEN OTHERS => NULL;
                END CASE;

                -- Check for collisions with the snake itself
                FOR i IN 1 TO max_snake_length - 1 LOOP
                    IF i < snake_length AND internal_snake(0).x = internal_snake(i).x ...
                        AND internal_snake(0).y = internal_snake(i).y THEN
                        IF lives = 1 THEN
                            game_over <= TRUE;
                            lives <= lives - 1;
                            snake_length <= 1;  -- Reset game
                        ELSE
                            lives <= lives - 1;
                            snake_length <= 1;  -- Reset game
                        END IF;
                    END IF;
                END LOOP;

                -- Check if the snake has eaten the food
                IF internal_snake(0).x = food_x AND internal_snake(0).y = food_y ...
                    THEN
                    internal_snake(snake_length).x <= internal_snake(snake_length - 1).x;
                    internal_snake(snake_length).y <= internal_snake(snake_length - 1).y;
                    snake_length <= snake_length + 1;
                END IF;

            END IF;
        END PROCESS snake_logic;

        -- Output assignments
        gameOver_out <= game_over;
        lives_out <= lives;
        snake_len_out <= snake_length;
        snake_out <= internal_snake;

    END behavior;
```

Code 9: food.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;
USE work.snake_pkg.ALL;

ENTITY food IS
    PORT(
        clock          : IN STD_LOGIC;
        reset           : IN STD_LOGIC;
        food_x          : OUT INTEGER;
        food_y          : OUT INTEGER;
        snake_in        : IN snake_array
    );
END food;

ARCHITECTURE behavior OF food IS

    -- Internal signals
    SIGNAL food_x_internal : INTEGER := game_left + (rect_width*10) + ...
        ((rect_width+1) / 2);
    SIGNAL food_y_internal : INTEGER := game_top + (rect_width*10) + ...
```

```
            ((rect_height+1) / 2);

    SIGNAL lfsr : STD_LOGIC_VECTOR(15 DOWNTO 0) := X"ACE1"; -- Initial value of ...
        the Linear Feedback Shift Register (LFSR)

    -- Function to generate random numbers using the LFSR
    FUNCTION random_number(min_val, max_val : INTEGER; lfsr_val : ...
        STD_LOGIC_VECTOR(15 DOWNTO 0)) RETURN INTEGER IS
        VARIABLE lfsr_temp : STD_LOGIC_VECTOR(15 DOWNTO 0);
        VARIABLE random_val : INTEGER;
    BEGIN
        lfsr_temp := lfsr_val;

        -- Perform one iteration of the LFSR
        lfsr_temp := lfsr_temp(14 DOWNTO 0) & (lfsr_temp(15) XOR lfsr_temp(13) ...
            XOR lfsr_temp(12) XOR lfsr_temp(10));

        -- Convert the LFSR value to an integer
        random_val := to_integer(unsigned(lfsr_temp));

        -- Limit the value within the specified range
        RETURN (random_val MOD (max_val - min_val + 1)) + min_val;
    END FUNCTION;

BEGIN

    -- Game logic process
    food_logic: PROCESS(reset, clock)
        VARIABLE new_food_col, new_food_row : INTEGER;
    BEGIN
        IF reset = '1' THEN
            -- Reset the food position to the initial coordinates
            food_x_internal <= game_left + (rect_width*10) + ((rect_width+1) / 2);
            food_y_internal <= game_top + (rect_width*10) + ((rect_height+1) / 2);

        ELSIF rising_edge(clock) THEN
            -- Update the LFSR
            lfsr <= lfsr(14 DOWNTO 0) & (lfsr(15) XOR lfsr(13) XOR lfsr(12) XOR ...
                lfsr(10));

            -- Check if the snake has eaten the food
            IF snake_in(0).x = food_x_internal AND snake_in(0).y = ...
                food_y_internal THEN
                -- Reposition the food to a new random location
                new_food_row := random_number(0, 19, lfsr);
                new_food_col := random_number(0, 29, lfsr);
                food_x_internal <= game_left + new_food_col * rect_width + ...
                    ((rect_width+1) / 2);
                food_y_internal <= game_top + new_food_row * rect_height + ...
                    ((rect_height+1) / 2);
            END IF;
        END IF;
    END PROCESS food_logic;

    -- Output assignments
    food_x <= food_x_internal;
    food_y <= food_y_internal;

END behavior;
```

Code 10: score.vhd

```
-- Import IEEE standard libraries
LIBRARY ieee;
```

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

-- Entity declaration for the score display
ENTITY score IS
    PORT(
        display_r   : OUT STD_LOGIC_VECTOR (6 downto 0);
        display_l   : OUT STD_LOGIC_VECTOR (6 downto 0);
        display_c   : OUT STD_LOGIC_VECTOR (6 downto 0);
        display_f   : OUT STD_LOGIC_VECTOR (6 downto 0);
        display_u   : OUT STD_LOGIC_VECTOR (6 downto 0);
        display_d   : OUT STD_LOGIC_VECTOR (6 downto 0);
        point       : IN INTEGER
    );
END score;

-- Architecture definition for the score entity
ARCHITECTURE behavior OF score IS

   -- Type declaration for the seven-segment display codes
    TYPE segments_type IS ARRAY (0 TO 13) OF STD_LOGIC_VECTOR(6 DOWNTO 0);

    -- Constant array holding the seven-segment display codes for digits 0-9, ...
        'C', 'F', 'U', and '-'
    CONSTANT segments : segments_type := (
        "1000000", -- 0
        "1111001", -- 1
        "0100100", -- 2
        "0110000", -- 3
        "0011001", -- 4
        "0010010", -- 5
        "0000010", -- 6
        "1111000", -- 7
        "0000000", -- 8
        "0010000", -- 9
        "1000110", -- C
        "0001110", -- F
        "1000001", -- U
        "0111111" -- dash
    );

    -- Function to get the display code for a given value
    FUNCTION get_display_code(value : INTEGER) RETURN STD_LOGIC_VECTOR IS
        BEGIN
            IF value ≥ 0 AND value ≤ 9 THEN
                RETURN segments(value);
            ELSE
                RETURN "1111111"; -- Display blank
            END IF;
   END get_display_code;

BEGIN
   -- Process to update the right and left display segments based on the score ...
       (point)
    display : PROCESS(point)
    BEGIN
        IF point ≥ 0 AND point ≤ 32 THEN
            display_r ≤ get_display_code((point-1) MOD 10);
         display_l ≤ get_display_code((point-1) / 10);
        ELSE
         display_r ≤ "1111111";
         display_l ≤ "1111111"; -- Display blank or error
      END IF;
   END PROCESS display;
```

```
    -- Constantly display 'C', 'F', 'U', and '-'
    display_c ≤ segments(10);
    display_f ≤ segments(11);
    display_u ≤ segments(12);
    display_d ≤ segments(13);
END behavior;
```

Code 11: joystick.vhd

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

ENTITY joystick IS

    PORT(
        clock       : IN STD_LOGIC;
        reset           : IN STD_LOGIC;
        direction   : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
    );

END joystick;

ARCHITECTURE behave OF joystick IS

    -- Declaration of component 'unnamed'
    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;

    -- Signals for joystick readings and internal processing
    SIGNAL joystickReadingX,joystickReadingY,c2,c3,c4,c5,c6,c7 : ...
        STD_LOGIC_VECTOR(11 downto 0);
    SIGNAL readX : INTEGER;
    SIGNAL readY : INTEGER;

    -- Constants defining joystick behavior
    CONSTANT tolerance : INTEGER := 400; -- Tolerance value for joystick movement ...
        detection
    CONSTANT homeX : INTEGER := 2008; -- Center X position of joystick
    CONSTANT homey : INTEGER := 1978; -- Center Y position of joystick

    BEGIN

        -- Instantiate the unnamed component
        A0 : unnamed PORT MAP (
            CLOCK => clock,
            CH0 => joystickReadingX,
            CH1 => joystickReadingY,
            CH2 => c2,
            CH3 => c3,
            CH4 => c4,
            CH5 => c5,
```

```
            CH6 => c6,
            CH7 => c7,
            RESET => reset
        );

        -- Process for joystick direction detection
        my_process: PROCESS(clock)
        BEGIN
            IF rising_edge(clock) THEN
                -- Convert joystick analog readings to integers
                readX <= to_integer(unsigned(joystickReadingX));
                readY <= to_integer(unsigned(joystickReadingY));

                -- Determine joystick direction based on analog readings relative to home position
            IF readX > homeX+tolerance THEN  -- Right
                    direction <= "01";
            ELSIF readX < homeX-tolerance THEN  -- Left
               direction <= "10";
            ELSIF readY > homeY+tolerance THEN  -- Down
               direction <= "11";
            ELSIF readY < homeY-tolerance THEN  -- Up
               direction <= "00";
            END IF;

        END IF;
    END PROCESS my_process;
    END behave;
```

Code 12: Joystick input to seven segment display converter

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

ENTITY joystick IS
    PORT(
        clock, reset: IN STD_LOGIC;
        display_r : OUT STD_LOGIC_VECTOR(6 downto 0);
        display_mr : OUT STD_LOGIC_VECTOR(6 downto 0);
        display_ml : OUT STD_LOGIC_VECTOR(6 downto 0);
        display_l : OUT STD_LOGIC_VECTOR(6 downto 0)
    );
END joystick;

ARCHITECTURE behave OF joystick IS
    TYPE segments_type IS ARRAY (0 TO 9) OF STD_LOGIC_VECTOR(6 DOWNTO 0);

    CONSTANT segments : segments_type := (
        "1000000", -- 0
        "1111001", -- 1
        "0100100", -- 2
        "0110000", -- 3
        "0011001", -- 4
        "0010010", -- 5
        "0000010", -- 6
        "1111000", -- 7
        "0000000", -- 8
        "0010000"  -- 9
    );

    FUNCTION get_display_code(value : INTEGER) RETURN STD_LOGIC_VECTOR IS
    BEGIN
        IF value >= 0 AND value <= 9 THEN
```

```
            RETURN segments(value);
        ELSE
            RETURN "1111111";
        END IF;
    END get_display_code;

    COMPONENT unnamed IS
        PORT (
            CLOCK : in  std_logic := '0';
            CH0   : out std_logic_vector(11 downto 0);
            CH1   : out std_logic_vector(11 downto 0);
            CH2   : out std_logic_vector(11 downto 0);
            CH3   : out std_logic_vector(11 downto 0);
            CH4   : out std_logic_vector(11 downto 0);
            CH5   : out std_logic_vector(11 downto 0);
            CH6   : out std_logic_vector(11 downto 0);
            CH7   : out std_logic_vector(11 downto 0);
            RESET : in  std_logic := '0'
        );
    END COMPONENT;

    SIGNAL joystickReading,c1,c2,c3,c4,c5,c6,c7 : STD_LOGIC_VECTOR(11 downto 0);
    SIGNAL point : INTEGER;

BEGIN
    A0 : unnamed PORT MAP (
        CLOCK => clock,
        CH0 => joystickReading,
        CH1 => c1,
        CH2 => c2,
        CH3 => c3,
        CH4 => c4,
        CH5 => c5,
        CH6 => c6,
        CH7 => c7,
        RESET => reset
    );

    display : PROCESS(clock)
    BEGIN
        point ≤ to_integer(unsigned(joystickReading));
        display_r ≤ get_display_code((point MOD 10));
        display_mr ≤ get_display_code((point / 10) MOD 10);
        display_ml ≤ get_display_code((point / 100) MOD 10);
        display_l ≤ get_display_code((point / 1000) MOD 10);
    END PROCESS display;
END behave;
```

B Python Code

Code 13: Code used for creating matrices of RGB values.

```
from PIL import Image  # Importing the required module from the Python Imaging ...
    Library (PIL)
import numpy as np     # Importing NumPy for array manipulation capabilities

# Load the image
img = Image.open('./image.png').convert('RGBA')

# Resize the image to 31x31 pixels
img = img.resize((31, 31))
```

```python

# Extract pixel data
pixel_data = np.array(img)

# Create the VHDL color array
vhdl_array = []
for row in pixel_data:
    vhdl_row = []
    for pixel in row:
        r, g, b, a = pixel
        # Use only fully opaque pixels (a == 255)
        if a == 255:
            vhdl_pixel = (r >> 4, g >> 4, b >> 4)  # Convert to 4-bit per color channel
        else:
            vhdl_pixel = (0, 0, 0)  # Black color for transparent pixels
        vhdl_row.append(vhdl_pixel)
    vhdl_array.append(vhdl_row)

# Generate VHDL code for the array
vhdl_code = "CONSTANT image : image_type := (\n"
for row in vhdl_array:
    vhdl_code += "    ("
    vhdl_code += ", ".join(f"X\"{r:01X}{g:01X}{b:01X}\"" for r, g, b in row)
    vhdl_code += "),\n"
vhdl_code = vhdl_code.rstrip(",\n") + "\n);"

# Save the VHDL code to a file
with open("./image.vhdl", "w") as f:
    f.write(vhdl_code)
```