CE339 - High Level Digital Design

Assignment 2 – "Snake" Video Game

Akshay Gopinath
Registration Number: 2005614
Word Count: 3944

A report presented for the degree of
Electronic Engineering

School of Computer Science and Electronic Engineering
University of Essex
England
March 27, 2024

CE339 - High Level Digital Design

Assignment 2 – "Snake" Video Game

Akshay Gopinath

ABSTRACT

This report aims to showcase and explain the design as well the process of developing a complex digital system. The digital system designed in this experiment is a simplified version of the class 'snake' game. The main aim is to design a playable snake game with hardware sprites (static and animated), whilst showing the player score. The target board is the Digilent Basys3, hence the VGA port is used to display graphics and the 7-segment display is used to show the score.

CE339 - High Level Digital Design

Assignment 2 – "Snake" Video Game

Akshay Gopinath

ACKNOWLEDGEMENTS

I would like to give my deepest appreciation to those who provided me the possibility to complete my this design challenge. A special gratitude I give to the module supervisor of the CE339 module, Dr. Wenqiang Yi for their guidance and encouragement, and providing me with the necessary material to learn how to complete this project.

I would also like to acknowledge with appreciation, the help of the technicians from Laboratory 8, who provided me access to the necessary equipment to complete the project, both during and outside lab sessions hours. A very special thanks goes to my peer, PhD student and GLA, Michal Borowski who provided me with tips and guidance to complete my design, especially when it came to rendering hardware sprites on the VGA display.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This report documents an experiment to design a "Snake" Video Game on hardware using a Hardware Description Language (HDL) called VHDL (Very High Speed Integrated Circuit Hardware Design Language). The target platform is the Digilent Basys3 Board which houses an Artix-7 based FPGA[1]. The VHDL code is synthesised using Xilinx Vivado. The VGA (Video Graphics Array) port on the Basys3 board is used to display the game on a compatible monitor and the player score is shown on the 7-segment display. This report will explain the design in a top-down approach, whilst going into detail on every sub-components. The top level schematic will generate the necessary signals required to correctly display the game and the score, such as the VGA synchronisation signals, RGB (red, green, blue) colour signals and the 7-segment display cathode and anode signals.

2. THE DESIGN

2.1. High Level Overview

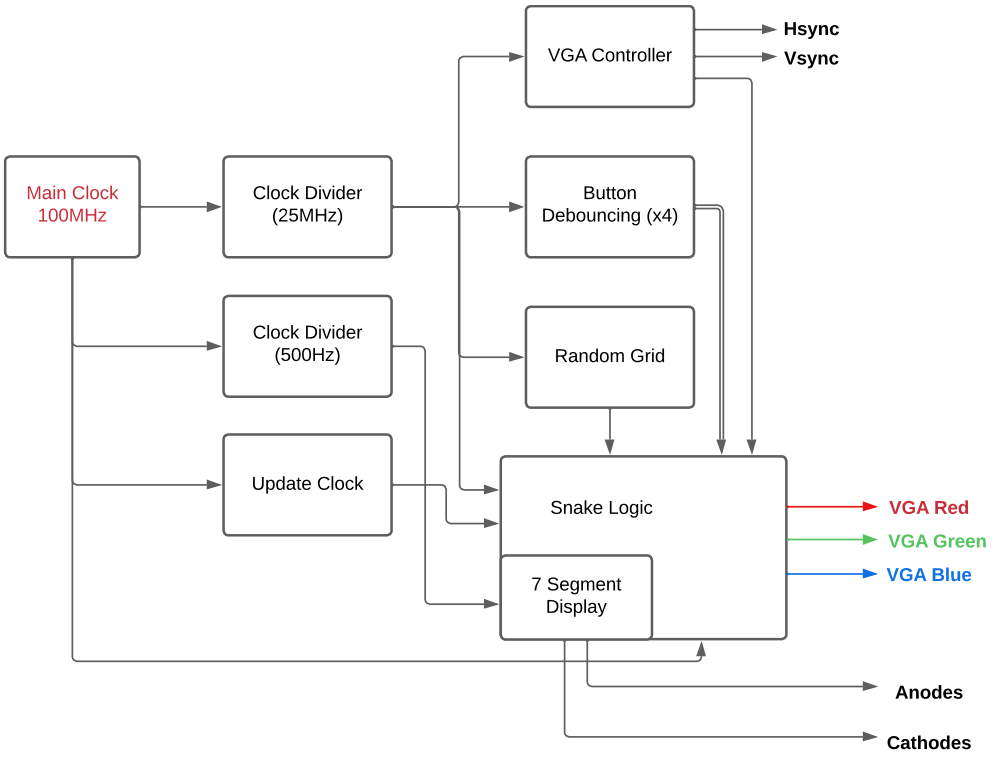


FIG. 1: *High level architecture of the system*

Figure 1 above shows the high level block diagram of the system. The main clock is from the Basys3 board which is at 100MHz frequency. The targer resolution is 640x480. The pixel clock is 25MHz, with a refresh rate of 60Hz. In order to generate the required synchronisation signals (by the VGA Controller module) for these specifications, the VGA controller module needs a 25MHz a clock. Hence the 100MHz master clock is given as input to a clock divider to generate this frequency. The 25MHz clock is also used for button debouncing (to keep it in synchronisation with the graphics being rendered to the screen), and also for the Random Grid module. Another clock divider was instantiated to generate a 500Hz clock. The 500Hz clock is used for the time multiplexed 7-segment display driver, which resides inside the Snake Logic module. The update clock is a module used to generate a pulse at a desired frequency (in this case 25Hz), which is high only for one clock cycle of the master clock. The purpose of this module is to set the update frequency of the game logic, hence the output of the update clock is given as input to the Snake Logic

module. The Random Grid module is used to generate pseudo random locations for the positions of the food in the game. And the output is given to the Snake Module as input. The VGA Controller module generates the horizontal and vertical synchronisation signals for the VGA Port. It also outputs the current x and y count co-ordinate as well as the blanking signal for the Snake Logic module to keep track of the screen position. The Snake Logic block is the main heart of the system. The Snake Logic module is the heart of the entire system, and contains the game main logic, as well as the rendering signals. This module contains many processes to control game elements such as the snake location direction, snake direction, snake size, game state, game levels etc. The module also contains the 7-segment display driver and BCD (Binary Coded Decimal) counters to display the score whilst playing the game. This module outputs the anode and cathode signals for the 7-segment display as well as the red, green and blue signals for the VGA port. The high level diagram from Figure 1 represents the main top level file shown in Source Code 1.

2.2. Detailed Overview

2.2.1. snake

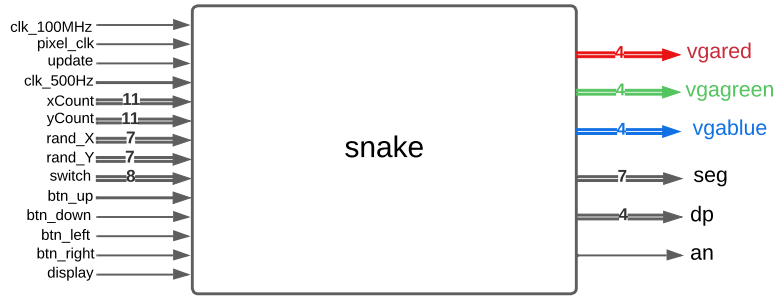


FIG. 2: *Inputs/outputs of snake module*

Figure 2 on the left depicts the inputs and outputs of the `snake` module. This module contains the main logic that controls the snake game, as well as the Read Only Memories (ROMs) that store the bitmapped sprites/graphics. The inputs are master 100MHz clock (for synchronisation), pixel clock (25MHz), update clock (25Hz), x and y counters, random food location, and the debounced buttons. It outputs the red, green, blue signals for VGA and the cathode and anode signals for the 7-segment display.

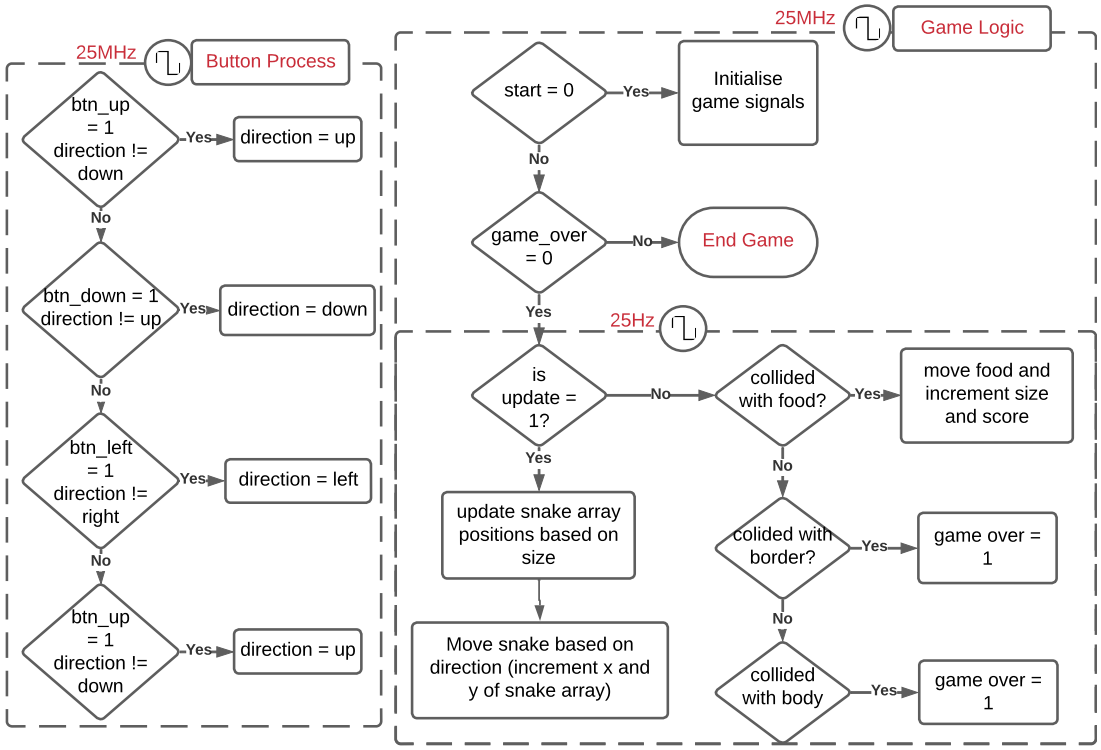


FIG. 3: *Button and game logic processes of the snake module*

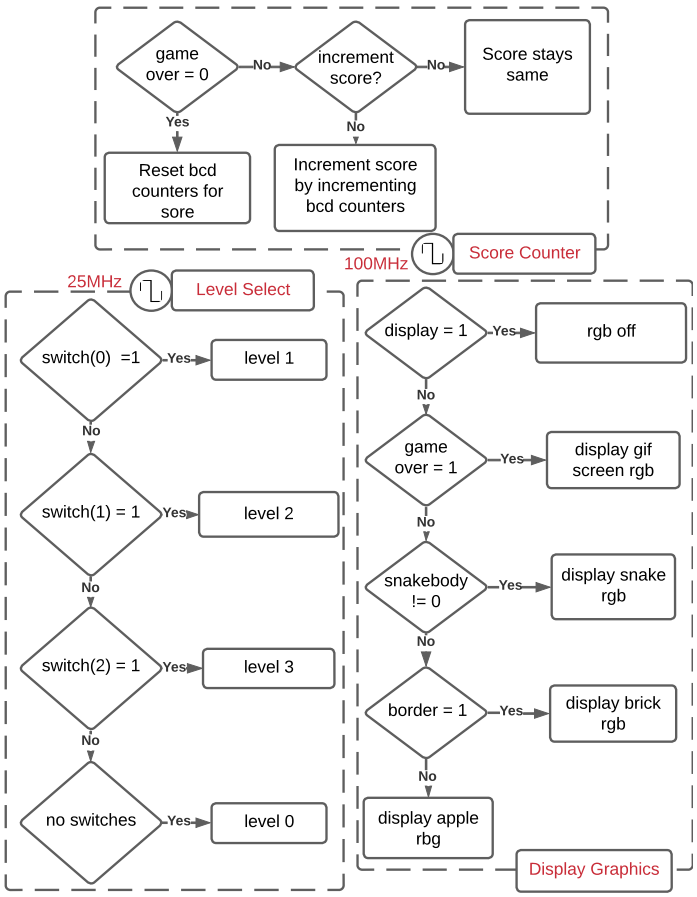


FIG. 4: *Level select and score count process and colour rendering*

The `snake` module is composed of many processes to construct the game logic and behaviour. Figure 3 and Figure 4 shows flow diagrams to represent this. Figure 3 above shows the button/input and the main game logic process. The button process sets the value of the `direction` register based on the button input. The button input is received from the `Debounce` module (which debounces the button presses for reliability). This process updated with the pixel clock of 25MHz. The game logic process handles updating the snake for the movement as well as checking for collisions. If the start signal is zero, the game is idle and all the starting game conditions are set to defaults (such as initial food location). If the game over signal is high, the game will end until the game is reset, and once reset, the game is reset back to default values. This part of this process is updated at 25MHz. The other parts of the process is updated at 25Hz, which is the games update clock. When the update clock is high, the snake position array (for both x and y locations) are updated in a for loop depending on the current snake size. The snake position at the first index (snake head) is updated depending on the current direction (set by the buttons). When the update clock is low, the collision between the border, food and the snake body is checked. If a collision with the food is detected, the snake size is incremented (unless max snake length is reached). Next if the snake head collides with a border or itself, then the game over signal will be set, thus ending the game, until the game is reset. Figure 4 shows three other important processes in this module. The score counting process increments the score during the game when food is collected and when the game ends, the game is reset. BCD counters and a 7-segment display driver is used to achieve this. The level select process is very simple, it selects the level based on the switch input. And the levels are border signals that are generated based on certain conditions that decide where the borders are placed on the screen. Lastly, the colour displaying/rendering logic is a combinational process, the other processes we have seen so far are sequential logic. If the display signal is high (same as blanking, which is active low logic), then the RBG colours are turned off. The rgb colours are selected based on which graphics is to be rendered, in this case, the border graphics, the game over GIF and the snake. Graphics in this game are all bitmapped ROMs, this will be explained later in the report.

2.2.2. **updateClk**

(a) *Flow chart for updateClk*

(b) *Entity block of updateClk*

FIG. 5: *updateClk module diagrams*

Figure 5 above contains two sub figures, Figure 5a and Figure 5b which are the architecture and block diagram of this entity respectively. This module has one input, the master clock and one output, the update clock. This is a generic value where the max count can be configured for a different update clock. The update clock used in this game is set to 25Hz. The architecture counts till the set max value and sets the output high for one clock cycle of the master clock, and then becomes low after.

2.2.3. randomGrid

FIG. 6: *randomGrid entity diagram*

Figure 6 on the left shows the inputs and outputs of the randomGrid module. This module is used to generate the pseudo random x and y locations for the food. The pseudo random generation implementation in this project is very simple, as a simple mathematical operation is done on the current random location to move the food 'unpredictably'.

Code Snippet 1: randomGrid architecture implementation

```
if rising_edge(pixel_clk) then
    rand_X <= ((rand_X + 3) mod 37) + 1; -- set random x and y position
    rand_Y <= ((rand_Y + 3) mod 27) + 1;
end if;
```

Since the implementation is simple, it is shown in Code Snippet 1. A simple arithmetic oprtation is computed on the value read back from the current random x and y value, at the rising egde of the pixel clock. The output of this module is given as input to the `snake` module.

2.2.4. 7-Segment Display

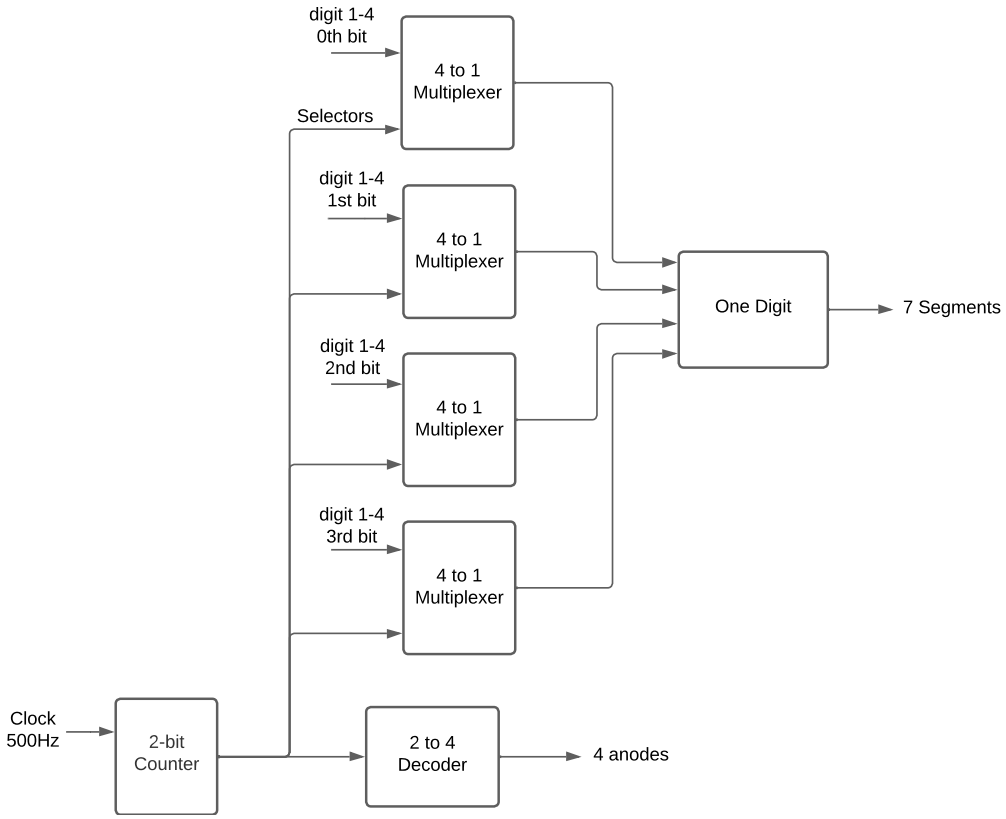


FIG. 7: *Architectural block diagram of the four_digits module*

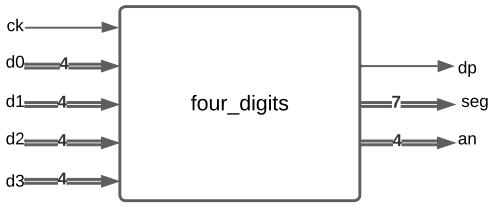


FIG. 8: *four_digits entity diagram*

This section explains the 7-segment display driver (and it's subcomponents) circuit used to display the score. Figure 8 on the left shows the entity inputs and outputs and Figure 7 above shows the high level connection diagram. This module takes in a 500Hz clock, and four inputs which are four bits wide, and generates the required anode and cathode displays. Time division multiplexing is used to display the four digits on the four seven segment displays available on the Basys3 board. The 500Hz clock given to a two bit counter which counts from 0→3, and it's output is fed into a 4-t0-1 multiplexer to select the corresponding bit for that digit. The four multiplexer output is given to a module called `one_digit` which is a 7-segment decoder. The 2-bit counter output is also used by a 2-to-4 decoder to select the correct anode which is currently being displayed. The anodes are swapped at a rate of 500Hz, a refresh rate human eyes cannot visibly see. The 2-bit counter being very simple is implemented using behavioural modelling. Figure 9, 10 and 11 below shows the entity block diagrams of `mux4_1`, `decoder2_4` and `one_digit` respectively, which is used in the circuit from Figure 7.

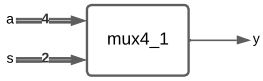


FIG. 9: *mux4_1 entity*

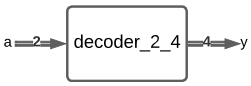


FIG. 10: *decoder_2_4 entity*

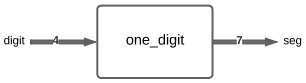
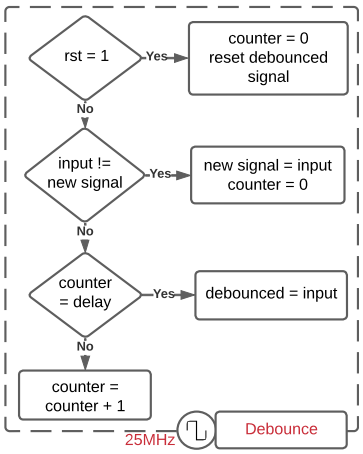


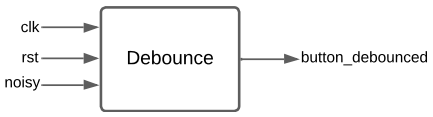
FIG. 11: one_digit entity

All the three modules are very simple. `one_digit` takes in a 4 bit number as input and outputs the corresponding 7 bit output for the 7 segments on the 7-segment display. The `decoder` entity is a simple two-to-four decoder, a certain bit of the 4 bit output is low depending on the input. For example, if the input is 01_2, the output is 1101_2, the 2^{nd} bit is low. And lastly, the 4-to-1 mux routes the corresponding bit of the input `a` depending on the select input `s`, to the output. The `four_digit` module is instantiated inside the `snake` module from Figure 2 and its outputs are propagated outside the `snake` module.

2.2.5. **Debounce**



(a) *Flow chart for Debounce*



(b) *Entity block of Debounce*

FIG. 12: *Debounce module diagrams*

Figure 12a and 12b shows the flow chart and entity diagram for the Debounce module respectively. There are three inputs, clk, rst and noisy (the button input) and one output which is thd debounced press. This module has a configurable generic parameter, `DELAY`. This module is clocked at the pixel clock, 25MHz. The module has a counter that counts till a delay value is reached, and once the count is same as the delay, the button press is debounced, and the counter is reset. The debounced output is input to the `snake` module.

2.2.6. vga_controller_640_60

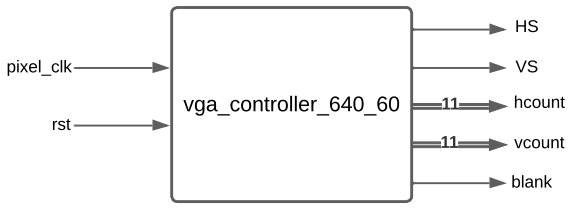
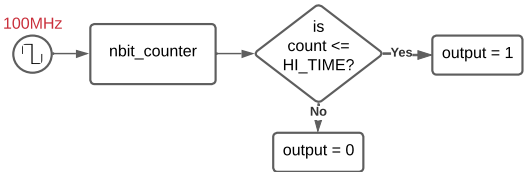


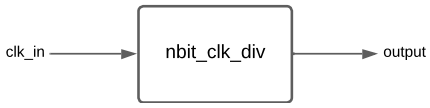
FIG. 13: *VGA Controller module*

Figure 13 on the left shows the entity diagram of the `vga_controller_60_40` module. This module wasn't written by the author and was given by the supervisor. This module has 2 inputs, the 25MHz pixel clock (`pixel_clock`) and reset (`rst`). And outputs the horizontal sync (`HS`), vertical sync `VS`, current horizontal counter value (`hcount`), current vertical counter value (`vcount`) and the blanking signal (`blank`). The horizontal and vertical synchronisation signals are outputted to the VGA port in the top level hierarchy. The vertical counter, horizontal counter and the blanking signals are all given as inputs to the `snake` module from 2 to control the game logic.

2.2.7. nbit_clk_div



(a) *Flow chart for nbit_clk_div*



(b) *Entity block of nbit_clk_div*

FIG. 14: *nbit_clk_div module diagrams*

Figure 14 above shows the flow diagram and entity diagram of the `nbit_clk_div` module (Figure 14a and 14b respectively). There is one input, the clock and one output, the divided clock. This module is a configurable and generic which is configurable, such as the division factor and the duty cycle. This is possible because the module instantiates another module called `nbit_counter` inside it, which also a generic component. The counter is clocked at 100Mhz (the master clock) and the output is compared with the configured highest count. If the counter reaches a value within a certain threshold, it makes the output high, else makes it low. And the constant `HI_TIME` is calculated from a generic value `high_count`, to control the duty cycle as per the user's usage.

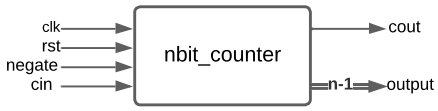


FIG. 15: *nbit_counter module*

Figure 15 on the left-hand side shows the generic counter used inside the clock divider. It has a configurable maximum count value. It is a very general purpose module as it has a `negate` input, and the `cin` input and `cout` output can be used to easily chain multiple counters. The output of this module is used in the clock divider module in order to generate the divided clock signal.

2.2.8. nbit_bcd_counter

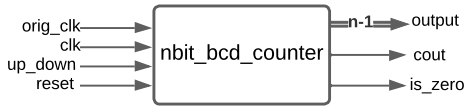


FIG. 16: *nbit_bcd_counter module*

Figure 16 on the left shows the entity block diagram of the `nbit_bcd_counter` module, which is a generic BCD counter which is configurable from a maximum count between 0→99. This module is used to keep track of the score as the snake eats the food objects. The implementation of this module is very simple, inside it are two modulo-10 counters that are cascaded. When one modulo-10 counter reaches it's 1001_2 (in up count) or 0000_2 (in down count), the second counter is incremented or decremented respectively. There are 3 inputs, the original master clock (`orig_clk`, to help with synchronisation), the clock (`clk`) at which the counter is incremented, `up_down` to swap between up and down count mode, and `reset`. The `is_zero` output is not used. This module is instantiated inside the `snake` module, and two are instantiated (as there are 4 7-segment displays) and are cascaded using the `cout` output. The output of the 2 BCD counters are inputs to the `four_digits` module.

2.2.9. Bitmapped Sprites and GIFs

As mentioned in section 2.2.1 of this report, graphics used in this game are stored as bitmap ROMs. This section details how sprites as well as GIFs were rendered on the VGA monitor. Bitmaps were used to render sprites and GIFs on the screen. A bitmap is essentially an array that holds information for each pixel in the image. In the case of this design, each element in the array holds RGB colour value for each pixel.

Code Snippet 2: *ROM used for the snake sprite*

```
type color_sprite_8 is array (0 to 7, 0 to 7) of std_logic_vector(0 to 11);
constant SNAKE_ROM : color_sprite_8 := (
  ("010101100010","011010000010","011010000010","011010000010","011010000010","011010000010","011010000010","001001010001"),
  ("011010000010","010011000001","001110110001","001110110001","001110110001","001110110001","001110110001","001001100001"),
  ("011010000010","001110110001","000110100001","000110100000","000110100000","000110100001","000110100000","001001100001"),
  ("011010000010","001110110001","000101110001","001010100001","001010100001","001001110001","001010100001","001001100001"),
  ("011010000010","001110110001","000110100001","001010010001","001010010001","001010100001","001010110000","001001100001"),
  ("011010000010","001110110001","000110100001","001010100001","001010100001","001010100001","001010110000","001001100001"),
  ("011010000010","001110110001","000110110000","001010110000","001010110000","001010110000","001010110000","001001100001"),
  ("001001010001","001001100001","001001100001","001001100001","001001100001","001001100001","001001100001","000101010001")
);
```

Code Snippet 2 above shows the ROM implemented to display the snake sprite. A sprite ROM is essentially a two dimensional array where the width and height represents the size of the sprite. In this case it is eight by eight. And each element is an RGB colour value. The ROM is indexed using the x and y counter values, and then is rendered at the desired location. Rendering static sprites is very simple, but it gets slightly tricky when trying to render animated sprites. An animated sprite contains multiple images, hence the bitmap array is an array of two-dimensional arrays.

Code Snippet 3: *Type definition for a GIF with 2 frames*

```
type apple_gif_sprite is array (0 to 1, 0 to 15, 0 to 15) of std_logic_vector(0 to 11);
```

Due to page space constraints, only the type definition is shown in Code Snippet 3 above. The above syntax defines a 16 by 16 sprite with 2 frames. And in this case, both the frame and the individual pixel colour in the bitmap needs to be indexed. A simple counter signal is used to index the current frame of the animated sprite. Within the current frame, the x and y counter values are used to get the RGB colour value.
These bit maps are generated by a Python script which takes in a GIF or PNG and generates VHDL ROM syntax. Which is then included in the code. All the ROMS are stored in the file `snake.vhd` which is shown in Source Code 2 in the Appendix.. And the python scripts are in Source Code 15 and Source Code 16 in the Appendix as well.

2.3. Development Process

This section describes the steps taken to develop the hardware design as well as what issues were faced and how they were rectified. First, the given VGA driver source code and simple colour rendering was tested, by attempting to display a solid red colour. This can be seen in the GitLab commit linked here.

After this, the main snake game logic was developed. As seen from this GitLab commit link, the colour rendering was not working initially, and some attempts were made to try and rectify this problem. The first attempt was to try making the sequential logic more reliable by following the discussion shown in reference [7]. Apparently in VHDL, it is better to synchronise most logic to the master clock of the board (thus rising edge of the 100MHz clock), and use the output of the clock divider as an enabler signal. This is because the synthesiser interprets the output of a clock divider as a normal 'wire' and not as a 'clock'. Although this didn't fix the issue after this commit, this is a good VHDL coding practice regardless. This issue was later rectified in this commit. Where in the render logic for the border, an `or` logic was used instead of `and` logic. Which was a silly mistake. Another fix was made in the same commit, where the movement logic was fixed because the x and y movement of the snake was inverted.

The next bug discovered was that the game would randomly end (game over) whilst the snake was in motion. This was initally beleived to be a synchronisation problem, but the input process was already clocked to the master clock (with the pixel clock as an enabler). To fix this issue, a button debouncing module was created in this commit. But not all the buttons were connected to this module. This bug was fixed in this commit when all four movement buttons were debounced using the module. This highlights the importance of debouncing button presses.

Next, rendering sprites and more complex graphics was attempted. First monochrome sprite rendering was attempted, and as seen from this commit, it wasn't working properly on the first try. The sprite appeared to be enlarged and had to be resized. This was later rectified in a later commit, when the rendering logic was given a context of the size of the image rather than using bit splicing. After this was fixed, coloured sprite rendering was added, as seen from this commit. The RGB rendering logic had to be modified, so that there is one signal for colour that can be spliced to separate signals. After a sprite was added for the borders, one more fixed had to be added for the first level of the game, as shown in this commit. Where the first level border generation border signal had to have a context of the sprite size. Lastly, the apple and game over screen GIFs were added. Since the logic for this was exactly the same, this worked as expected on the first attempt.

To understand how to render complex on an FPGA, the website from reference [6] was used. This website teaches the design approach in SystemVerilog rather than VHDL, but all the concepts depicted in the website carried over nicely, and the logic and theory was easily translatable to VHDL.

The sprite texture used for the food is shown in reference [2], for the brick/border in reference [3] and for the snake body in reference [4].

2.4. Possible Design Improvements

There are many places where the current hardware design can be improved. For example the current pseudo random location generation uses a very simple arithmetic operation to 'randomise' the location of the food. A smarter approach would have been to use a Linear Feedback Shift Register (LFSR) design as shown in reference [7], which is a very popular approach in both hardware and software to generate random numbers. The current apprach was chosen due to simplicity, and works well. And also due to time contraints for the project.

The next improvement that could be made with more time is the visuals of the game. The snake sprite could have been animated and also add a textured background to the game, rather than a solid black background. This was not done again due to time constraints but this shouldn't be hard to adapt into the current code base, as rendering sprites and even animated GIFs has been achieved in this project.

The snake source code shown in Source Code 2 could also be improved. As all the ROMs are stored in this one file, it makes the file rather large in terms of lines of code. The ROMs could have possibly been stored elsewhere, maybe a separate entity. This would improve code reliability.

3. TESTING AND FINAL RESULTS

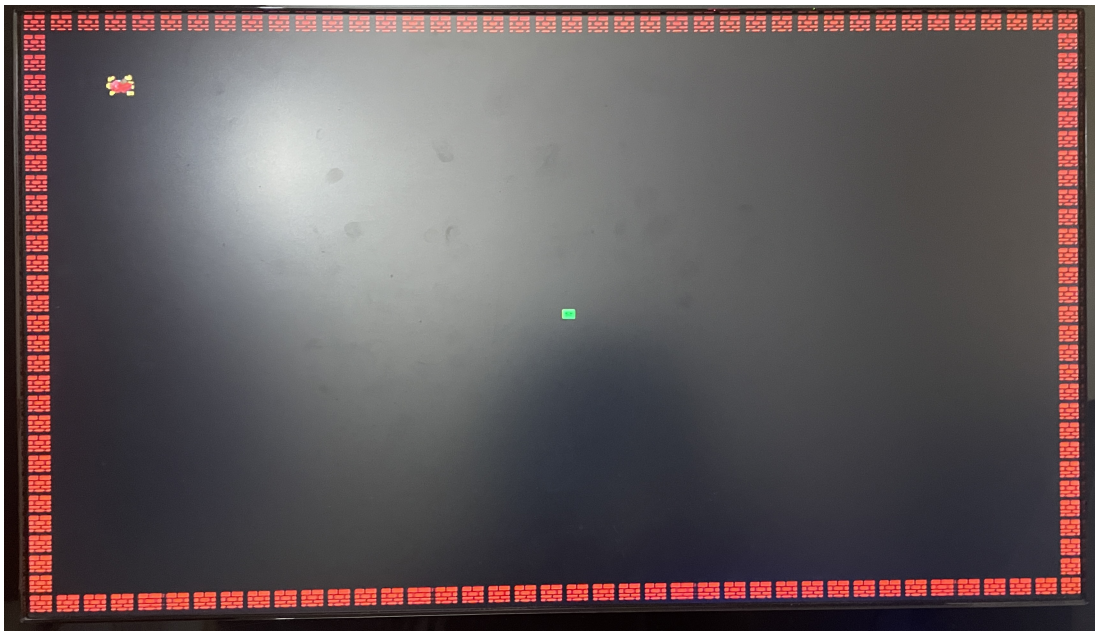


FIG. 17: *When game is stationary*

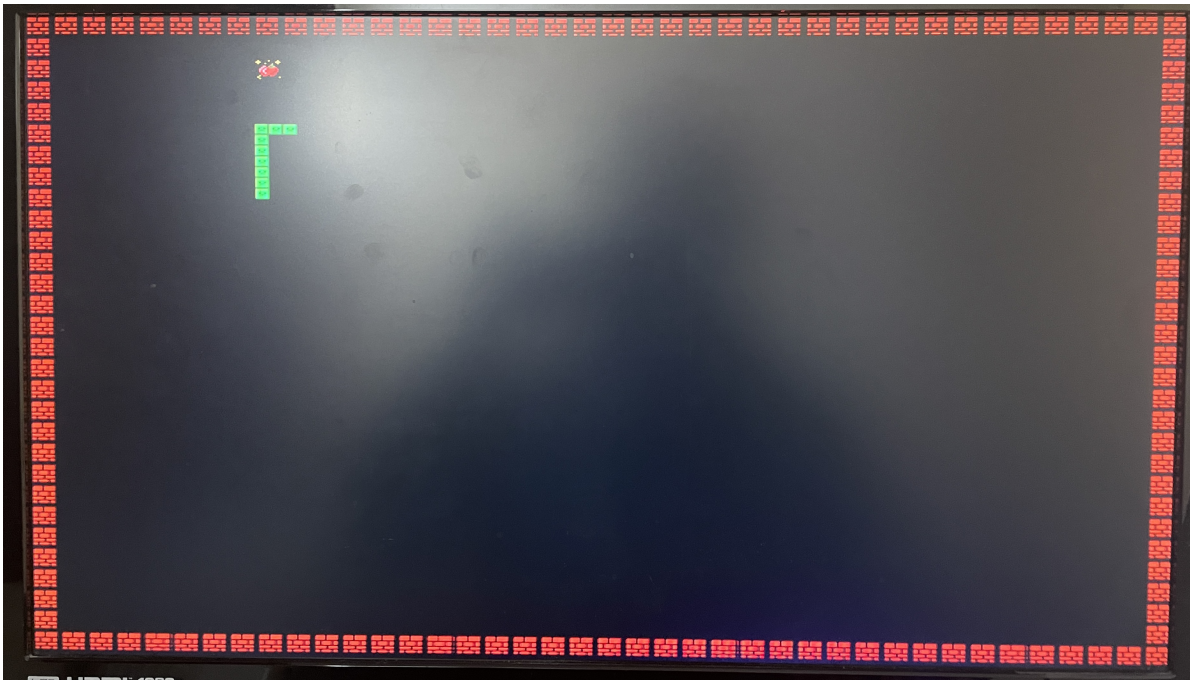


FIG. 18: *When game is moving*

FIG. 19: *One of the game levels*

FIG. 20: *Game over screen*

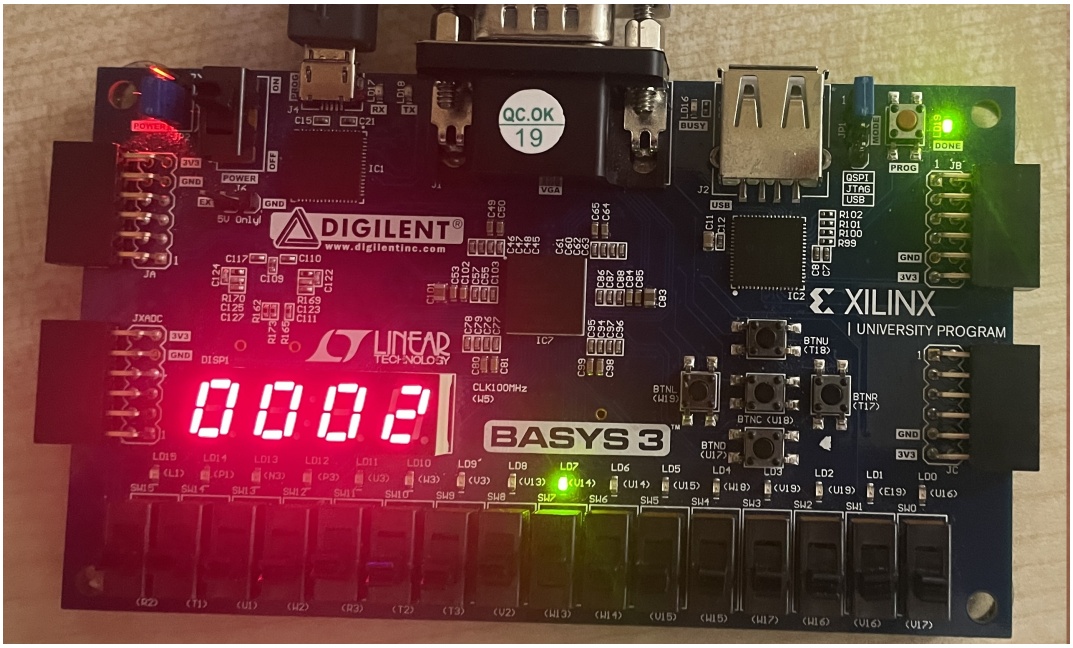


FIG. 21: *Displaying the score on the 7-segment display*

A video/gif showcase can be seen in the GitLab repo README page linked here

4. CONCLUSION

By the end of the project, a playable and functional snake game was implemented with hardware static and animated sprites as well as multiple levels. Both the VGA graphics (to display the game) and the 7-segment display (to show the player score) worked successfully and as expected. This can be seen from the pictures from Section 3 of the report.

There is a lot to learn from this project. First is observing how the VGA standards work, such as the synchronisation signals, its timings and how the RGB colours raster is used to produce graphics. With the use of memory systems, in this case ROMs (Read Only Memory), bitmaps can be stored so that they can be indexed to display both static and animated sprite based graphics. The usage and the workings of 7-segment displays and how time division multiplexing can be used to control multiple displays with one set of pins, was also observed.

Another important lesson learnt is how hierarchical and modular based design can be used to design a digital system. In this case, every distinct logic has been attempted to be split into separate entities to allow for a modular design with an easy-to-follow hierarchy from top to low level. For example, using separate entities for circuits such as button debouncing, clock divider, snake game logic etc.

Often in engineering, first iterations are not always bug free or even functional. And a design will need to go through different patches in order to reach a functional/playable state. And this was the case for this project as well, as showcased in 2.3 of this report. And this highlights the importance of using a version control system, in this case Git in order to keep track of changes within the source code that is being developed.

Overall, this project was successfully implemented with great results, just like every other engineering project, has scope for future extensions and improvements.

5. REFERENCES

[1] S. K, "Basys 3," Basys 3 - Digilent Reference, https://digilent.com/reference/programmable-logic/basys-3/start (accessed Mar. 26, 2024).
[2] RndmWeirdo, "A shiny apple? by rndmweirdo on DeviantArt," by RndmWeirdo on DeviantArt, https://www.deviantart.com/rndmweirdo/art/A-shiny-Apple-840730109 (accessed Mar. 26, 2024).
[3] BloodyYoshi, "Super Mario Bros HD - brick block by bloodyyoshi on DeviantArt," by BloodyYoshi on DeviantArt, https://www.deviantart.com/bloodyyoshi/art/Super-Mario-Bros-HD-Brick-Block-740859067 (accessed Mar. 26, 2024).
[4] Super Mario Wiki, "Snake block," Super Mario Wiki, https://www.mariowiki.com/Snake_Block (accessed Mar. 26, 2024).
[5] "Linear-feedback shift register," Wikipedia, https://en.wikipedia.org/wiki/Linear-feedback_shift_register (accessed Mar. 26, 2024).
[6] W. Green, "FPGA & RISC-V Tutorials," Project F, https://projectf.io/tutorials/ (accessed Mar. 26, 2024).
[7] user1175889user1175889 14133 gold badges44 silver badges1313 bronze badges, Martin ThompsonMartin Thompson 16.6k11 gold badge3838 silver badges5757 bronze badges, and sonicwavesonicwave 6, "Making a clock divider," Stack Overflow, https://stackoverflow.com/questions/19708301/making-a-clock-divider (accessed Mar. 26, 2024).

6. APPENDIX

Note: For the file snake.vhd, the contents of the ROMs have been replaced with a comment: `-- rom value goes here`. This is done to save space and to make the report look cleaner. Please refer to the full VHDL code submitted with the assignment for the full contents.

Source Code 1: *main.vhd, main top level file*

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

entity main is
    Port ( clk_100mhz : in STD_LOGIC;          --  master clock 100MHz
           switch : in STD_LOGIC_VECTOR(7 downto 0);  -- switches 0-7
           btn_up : in STD_LOGIC;             -- up button
           btn_left : in STD_LOGIC;            -- left button
           btn_right : in STD_LOGIC;           -- right button
           btn_down : in STD_LOGIC;            -- doum button
           led : out STD_LOGIC_VECTOR(7 downto 0);     -- leds 0-7
           vgared : out STD_LOGIC_VECTOR(3 downto 0);  -- vga red
           vgagreen : out STD_LOGIC_VECTOR(3 downto 0);  -- vga green
           vgablue : out STD_LOGIC_VECTOR(3 downto 0);  -- vga blue
           hsync : out STD_LOGIC;             -- horizontal sync
           vsync : out STD_LOGIC;             -- vertical sync
           seg   : out std_logic_vector (6 downto 0);    -- 7-segment display
           dp    : out std_logic;             -- 7-segment display decimal point
           an    : out std_logic_vector (3 downto 0)     -- 7-segment display anodes
         );
end main;

architecture Behavioral of main is
    signal pixel_clk : std_logic;
    signal clk_500hz : std_logic;
    signal xCount : unsigned(10 downto 0);  -- x position from horizontal counter of vga driver
    signal yCount : unsigned(10 downto 0);  -- y position from vertical counter of vga driver
    signal rand_X : unsigned(6 downto 0);   -- x random position for the food
    signal rand_Y : unsigned(6 downto 0);  -- y random position for the foodq
    signal update : std_logic;         -- signal to update the game
    signal up : std_logic;           -- debounced up button
    signal down : std_logic;         -- debounced down button
    signal left : std_logic;         -- debounced left button
    signal right : std_logic;         -- debounced right button
    signal display : std_logic;         -- signal to display the game
begin
    -- connect the signals to the VGA controller
    vga_controller : entity work.vga_controller_640_60(Behavioral)
        Port map (rst => '0', pixel_clk => pixel_clk, HS => hsync, VS => vsync, hcount => xCount, vcount =>
↪  yCount, blank => display);

  -- instantiate clock divider for 100MHz to 25MHz
    clk_div_unit_25Mhz : entity work.nbit_clk_div(Behavioral)
        Generic map (div_factor => 4,
                     high_count => 2,
                     num_of_bits => 3)
        Port map (clk_in => clk_100mhz, output => pixel_clk);

  -- instantiate clock divider for 100MHz to 500Hz
  clk_div_unit_500hz : entity work.nbit_clk_div(Behavioral)
        Generic map (div_factor => 200000,
                     high_count => 200000/2,
                     num_of_bits => 18)
        Port map (clk_in => clk_100mhz, output => clk_500hz);

    -- instatiate random grid
    random_grid : entity work.randomGrid(Behavioral)
        Port map (pixel_clk => pixel_clk, rand_X => rand_X, rand_Y => rand_Y);

  -- insitiate update clock
    update_clk : entity work.updateClk(Behavioral)
        Generic map (max_value => 4000000)
        Port map (clk_100mhz => clk_100mhz, update => update);

  -- instantiate debounce for buttons
    up_sig : entity work.Debounce(Behavioral)
        Port map (clk => pixel_clk, rst => '0', noisy => btn_up, button_debounced => up);

    down_sig : entity work.Debounce(Behavioral)
        Port map (clk => pixel_clk, rst => '0', noisy => btn_down, button_debounced => down);
```

```
    left_sig : entity work.Debounce(Behavioral)
        Port map (clk => pixel_clk, rst => '0', noisy => btn_left, button_debounced => left);

    right_sig : entity work.Debounce(Behavioral)
        Port map (clk => pixel_clk, rst => '0', noisy => btn_right, button_debounced => right);

    snake_logic: entity work.snake(Behavioral)
        Port map (
            clk_100mhz => clk_100mhz,
            pixel_clk => pixel_clk,
            update => update,
            clk_500hz => clk_500hz,
            xCount => xCount,
            yCount => yCount,
            rand_X => rand_X,
            rand_Y => rand_Y,
            switch => switch,
            btn_up => up,
            btn_left => left,
            btn_right => right,
            btn_down => down,
            display => display,
            led => led,
            vgared => vgared,
            vgagreen => vgagreen,
            vgablue => vgablue,
            seg => seg,
            dp => dp,
            an => an
        );


end Behavioral;
```

Source Code 2: *debounce.vhd, button debouncing module*

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

entity Debounce is
    Generic (DELAY : integer := 400000); --  configurable delay
    Port ( clk : in STD_LOGIC; -- clock
           rst : in STD_LOGIC; -- reset
           noisy : in STD_LOGIC; -- unfiltered signal
           button_debounced : out STD_LOGIC); -- debounced signal
end Debounce;

architecture Behavioral of Debounce is
    signal counter : unsigned(19 downto 0);
    signal new_sig : std_logic;
begin
    process(clk, rst)
    begin
        if rising_edge(clk) then
            if rst = '1' then -- reset
                counter <= (others => '0');
                new_sig <= noisy;
                button_debounced <= noisy;
            elsif noisy /= new_sig then -- if signal changes
                new_sig <= noisy;
                counter <= (others => '0');
            elsif counter = DELAY then
                button_debounced <= new_sig; -- debounced signal
            else
                counter <= counter + 1; -- increment counter
            end if;
        end if;
    end process;
end Behavioral;
```

Source Code 3: *decoder_2_4.vhd, 2 to 4 decoder*

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

entity decoder_2_4 is
    Port ( a : in std_logic_vector (1 downto 0);   -- input to the decoder
           y : out std_logic_vector (3 downto 0)); -- output of the decoder
end decoder_2_4;

architecture Behavioral of decoder_2_4 is

begin
    process(a)
    begin
        case a is
            when "00" => y <= "1110";  -- output is active low
            when "01" => y <= "1101";  -- Make the corresponding bit low
            when "10" => y <= "1011";
            when "11" => y <= "0111";
            when others => y <= "1111";
        end case;
    end process;

end Behavioral;
```

Source Code 4: *four_digits.vhd, 7-segment display circuit*

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

entity four_digits is
    Port ( d3 : in std_logic_vector (3 downto 0);   -- 4 4-bit inputs
           d2 : in std_logic_vector (3 downto 0);
           d1 : in std_logic_vector (3 downto 0);
           d0 : in std_logic_vector (3 downto 0);
           ck : in std_logic;                       -- 500Hz clock
           seg : out std_logic_vector (6 downto 0); -- 7-segment display
           an : out std_logic_vector (3 downto 0);  -- anodes
           dp : out std_logic);                     -- decimal point on 7-segment display
end four_digits;

architecture Behavioral of four_digits is
    component decoder_2_4                       -- 2 to 4 decoder component
    Port (
        a : in std_logic_vector (1 downto 0);
        y : out std_logic_vector (3 downto 0)
    );
    end component;

    component mux4_1                            -- 4 to 1 mux component
    Port (
        a : in std_logic_vector (3 downto 0);
        s : in std_logic_vector (1 downto 0);
        y : out std_logic
    );
    end component;

    component one_digit                         -- 7-segment display decoder component
    Port (
        digit : in std_logic_vector (3 downto 0);
        seg : out std_logic_vector (6 downto 0)
    );
    end component;

    signal count: unsigned (1 downto 0);        -- counter to multiplex between 4 inputs

    signal mux_out_1: std_logic;
    signal mux_out_2: std_logic;            -- output of the 4 muxes
    signal mux_out_3: std_logic;
    signal mux_out_4: std_logic;

begin
```

```
    process(ck)
    begin
        if rising_edge(ck) then            -- counter increments at rising edge of 500Hz clock
            if count = 3 then               -- reset counter if count = 3 (3+1 = 4 inputs)
                count <= "00";
            else
                count <= count + 1;
            end if;
        end if;
    end process;

    U1: decoder_2_4 Port map (          -- 2 to 4 decoder to select anodes
        a => std_logic_vector(count),
        y => an
    );

    -- the first mux takes input d0(0), d1(0), d2(0), d3(0) and selects one of them based on count
    U2: mux4_1 Port map (
        a(0) => d0(0),
        a(1) => d1(0),
        a(2) => d2(0),
        a(3) => d3(0),
        s => std_logic_vector(count),
        y => mux_out_1
    );

    -- the second mux takes input d0(1), d1(1), d2(1), d3(1)
    U3: mux4_1 Port map (
        a(0) => d0(1),
        a(1) => d1(1),
        a(2) => d2(1),
        a(3) => d3(1),
        s => std_logic_vector(count),
        y => mux_out_2
    );

    -- the third mux takes input d0(2), d1(2), d2(2), d3(2)
    U4: mux4_1 Port map (
        a(0) => d0(2),
        a(1) => d1(2),
        a(2) => d2(2),
        a(3) => d3(2),
        s => std_logic_vector(count),
        y => mux_out_3
    );

    -- the fourth mux takes input d0(3), d1(3), d2(3), d3(3)
    U5: mux4_1 Port map (
        a(0) => d0(3),
        a(1) => d1(3),
        a(2) => d2(3),
        a(3) => d3(3),
        s => std_logic_vector(count),
        y => mux_out_4
    );

    -- the output of the 4 muxes is sent to the 7-segment display decoder to display the 4 inputs
    U6: one_digit Port map (
        digit(0) => mux_out_1,
        digit(1) => mux_out_2,
        digit(2) => mux_out_3,
        digit(3) => mux_out_4,
        seg => seg
    );

    -- disable decimal point
    dp <= '1';

end Behavioral;
```

Source Code 5: *mux4_1.vhd, 4 to 1 multiplexer*

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

entity mux4_1 is
    Port ( a : in std_logic_vector (3 downto 0);  -- 4 bit input
           s : in std_logic_vector (1 downto 0);  -- 2 bit select
           y : out std_logic);                    -- 1 bit output
end mux4_1;

architecture Behavioral of mux4_1 is

begin
    process(a,s)
    begin
        case s is
            when "00" => y <= a(0);  -- select the corresponding bit based on the select
            when "01" => y <= a(1);
            when "10" => y <= a(2);
            when "11" => y <= a(3);
            when others => y <= 'X';
        end case;
    end process;

end Behavioral;
```

Source Code 6: *nbit_bcd_counter.vhd, generic BCD counter*

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

entity nbit_bcd_counter is
    Generic (bcd_width : natural := 8;   -- 4 or 8
             max_value : natural := 99   -- 0 to 99
    );
    Port ( orig_clk : in std_logic;      -- 100MHz clock for synchronisation
           clk : in std_logic;           -- a divided clock for the counter (used as an enabler for the counter)
           up_down : in std_logic;       -- 0 for up, 1 for down
           reset : in std_logic;         -- reset the counter
           cout : out std_logic;         -- carry out signal
           is_zero : out std_logic;      -- 1 when the counter is at 0, 0 otherwise
           output : out std_logic_vector ((bcd_width-1) downto 0));  -- output of the counter
end nbit_bcd_counter;

architecture Behavioral of nbit_bcd_counter is
    signal count_1 : unsigned (3 downto 0) := (others => '0');  -- 4 bit counter for the 1's place
    signal count_2 : unsigned (3 downto 0) := (others => '0');  -- 4 bit counter for the 10's place
    signal cout_int : std_logic := '0';                         -- internal carry out signal
    signal last_clk : std_logic := '0';                         -- last clock signal for synchronisation
begin
    process(orig_clk, clk , up_down, reset)
    begin
        if reset = '1' then                 -- reset the counter
            count_1 <= (others => '0');
            count_2 <= (others => '0');
            last_clk <= '0';                -- reset the last clock signal
        elsif rising_edge(orig_clk) then
            last_clk <= clk;                -- synchronise the last clock signal
            if (clk = '1' and last_clk = '0') then
                if up_down = '0' then
                    count_1 <= count_1 + 1;     -- count up and reset the 1's place counter if it reaches 9
                    if count_1 = "1001" then
                        count_1 <= "0000";
                        count_2 <= count_2 + 1; -- count up and reset the 10's place counter if it reaches 9
                        if count_2 = "1001" then
                            count_2 <= "0000";
                        end if;
                    end if;
                else
                    if count_1 = "0000" then   -- count down and reset the 1's place counter if it reaches 0
                        count_1 <= "1001";
                        if count_2 = "0000" then
```

```
                            count_2 <= "1001";
                        else
                            count_2 <= count_2 - 1;  -- count down and reset the 10's place counter if it reaches
↪  0
                        end if;
                    else
                        count_1 <= count_1 - 1;
                    end if;
                end if;

                if count_2 = (to_unsigned(max_value/10, 4)) and count_1 = (to_unsigned(max_value mod 10, 4))
↪  then  -- set the carry out signal if the counter (syncronised to the clock) is at 0
                    cout_int <= '1';
                else
                    cout_int <= '0';
                end if;
                -- reset the counter if the max value is reached
                if (count_2 = (to_unsigned(max_value/10, 4)) and count_1 = (to_unsigned(max_value mod 10, 4)))
↪  and up_down = '0' then
                    count_2 <= "0000";  -- reset the both counters
                    count_1 <= "0000";
                elsif (count_2 = "0000" and count_1 = "0000") and up_down = '1' then
                    count_2 <= (to_unsigned(max_value/10, 4));      -- set the counters to the max value if
↪  counting doun and the counter is at 0
                    count_1 <= (to_unsigned(max_value mod 10, 4));
                end if;
            end if;
        end if;
    end process;

    cout <= cout_int;  -- set the carry out signal

    is_zero <= '1' when (count_2 = "0000" and count_1 = "0000") else '0';   -- set the is_zero signal if output
↪  is 0 (combinateional logic)

    -- output is the concatenation of count_2 and count_1 if bcd_width = 8, if bcd_width = 4 then output is
↪  count_1
    output <= std_logic_vector(count_2 & count_1) when bcd_width = 8 else std_logic_vector(count_1);

end Behavioral;
```

Source Code 7: *nbit_bcd_counter.vhd, generic BCD counter*

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

entity nbit_bcd_counter is
    Generic (bcd_width : natural := 8;   -- 4 or 8
             max_value : natural := 99   -- 0 to 99
    );
    Port ( orig_clk : in std_logic;      -- 100MHz clock for synchronisation
           clk : in std_logic;           -- a divided clock for the counter (used as an enabler for the counter)
           up_down : in std_logic;       -- 0 for up, 1 for down
           reset : in std_logic;         -- reset the counter
           cout : out std_logic;         -- carry out signal
           is_zero : out std_logic;      -- 1 when the counter is at 0, 0 otherwise
           output : out std_logic_vector ((bcd_width-1) downto 0));  -- output of the counter
end nbit_bcd_counter;

architecture Behavioral of nbit_bcd_counter is
    signal count_1 : unsigned (3 downto 0) := (others => '0');  -- 4 bit counter for the 1's place
    signal count_2 : unsigned (3 downto 0) := (others => '0');  -- 4 bit counter for the 10's place
    signal cout_int : std_logic := '0';                         -- internal carry out signal
    signal last_clk : std_logic := '0';                         -- last clock signal for synchronisation
begin
    process(orig_clk, clk , up_down, reset)
    begin
        if reset = '1' then                 -- reset the counter
            count_1 <= (others => '0');
            count_2 <= (others => '0');
            last_clk <= '0';                -- reset the last clock signal
        elsif rising_edge(orig_clk) then
            last_clk <= clk;                -- synchronise the last clock signal
            if (clk = '1' and last_clk = '0') then
                if up_down = '0' then
                    count_1 <= count_1 + 1;     -- count up and reset the 1's place counter if it reaches 9
                    if count_1 = "1001" then
```

```
                        count_1 <= "0000";
                        count_2 <= count_2 + 1; -- count up and reset the 10's place counter if it reaches 9
                        if count_2 = "1001" then
                            count_2 <= "0000";
                        end if;
                    end if;
                else
                    if count_1 = "0000" then   -- count down and reset the 1's place counter if it reaches 0
                        count_1 <= "1001";
                        if count_2 = "0000" then
                            count_2 <= "1001";
                        else
                            count_2 <= count_2 - 1;  -- count down and reset the 10's place counter if it reaches
↪   0
                        end if;
                    else
                        count_1 <= count_1 - 1;
                    end if;
                end if;

                if count_2 = (to_unsigned(max_value/10, 4)) and count_1 = (to_unsigned(max_value mod 10, 4))
↪   then  -- set the carry out signal if the counter (syncronised to the clock) is at 0
                    cout_int <= '1';
                else
                    cout_int <= '0';
                end if;
                -- reset the counter if the max value is reached
                if (count_2 = (to_unsigned(max_value/10, 4)) and count_1 = (to_unsigned(max_value mod 10, 4)))
↪   and up_down = '0' then
                    count_2 <= "0000";  -- reset the both counters
                    count_1 <= "0000";
                elsif (count_2 = "0000" and count_1 = "0000") and up_down = '1' then
                    count_2 <= (to_unsigned(max_value/10, 4));      -- set the counters to the max value if
↪   counting down and the counter is at 0
                    count_1 <= (to_unsigned(max_value mod 10, 4));
                end if;
            end if;
        end if;
    end process;

    cout <= cout_int;  -- set the carry out signal

    is_zero <= '1' when (count_2 = "0000" and count_1 = "0000") else '0';   -- set the is_zero signal if output
↪   is 0 (combinateional logic)

    -- output is the concatenation of count_2 and count_1 if bcd_width = 8, if bcd_width = 4 then output is
↪   count_1
    output <= std_logic_vector(count_2 & count_1) when bcd_width = 8 else std_logic_vector(count_1);

end Behavioral;
```

Source Code 8: *nbit_clk_div.vhd, generic clock divider*

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

entity nbit_clk_div is
    Generic (
        div_factor : natural := 16; -- divide the clock by this number
        high_count : natural := 8; -- num of clk pulses until high; duty cycle = (div_factor - high_count) /
↪   div_factor
        num_of_bits : natural := 4  -- number of bits in the counter needed to divide the clock
    );
    Port ( clk_in : in std_logic;  -- input clock
           output : out std_logic  -- output divided clock
    );
end nbit_clk_div;

architecture Behavioral of nbit_clk_div is
    component nbit_counter is       -- component declaration for nbit_counter
        Generic (
            width : natural;
            modulo : natural
        );
        Port (
            clk: in std_logic;
            cin    : in  std_logic;
```

```
            negate: in std_logic;
            rst : in std_logic;
            cout   : out std_logic;
            output : out std_logic_vector((width-1) downto 0)
        );
    end component;
    signal ignore : std_logic;  -- signal to ignore the carry out of the counter
    constant HI_TIME : std_logic_vector := std_logic_vector(to_unsigned(high_count-1, num_of_bits)); -- constant
↪   to compare the counter output
    signal counter_output : std_logic_vector((num_of_bits-1) downto 0);  -- counter output
begin
    counter : nbit_counter
    Generic Map (
        width => num_of_bits,
        modulo => div_factor
    )
    Port Map (
        clk => clk_in,
        cin => '1',
        negate => '0',
        rst => '0',
        cout => ignore,
        output => counter_output  -- output of the counter is used to generat the output clock
    );

    process(clk_in)
        variable out_int : std_logic := '0';  -- internal signal to generate the output clock
    begin
        output <= out_int;
        if rising_edge(clk_in) then
            if counter_output <= HI_TIME then   -- compare the counter output with the high_count to set the duty
↪   cycle
                out_int := '1';
            else
                out_int := '0';
            end if;
        end if;
    end process;
--    output <= counter_output(num_of_bits - 1);
end Behavioral;
```

Source Code 9: *nbit_counter.vhd, generic counter*

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

entity nbit_counter is
    Generic (width  : natural := 4;   -- Number of bits in the counter needed to count upto the modulo
              modulo : natural := 16); -- Modulo upto which the counter should count to
    Port ( clk    : in  std_logic;    -- Clock signal
           cin    : in  std_logic;    -- Carry in signal
           negate : in  std_logic;    -- Negate the output
           rst    : in  std_logic;    -- Reset signal
           cout   : out std_logic;    -- Carry out signal
           output : out std_logic_vector ((width-1) downto 0));    -- Output of the counter
end nbit_counter;

architecture Behavioral of nbit_counter is
    signal count : unsigned (output'RANGE) := (others => '0');   -- Counter signal
begin
    process(clk, rst)
    begin
        if rising_edge(clk) then
            if rst = '1' then   -- synchronous reset
                count <= (others => '0');
            elsif (cin = '1') and (count = (modulo-1)) then   -- reset when modulo is reached
                count <= (others => '0');
            elsif (cin = '1') then  -- increment when carry in is high
                count <= count + 1;
            end if;
        end if;
    end process;

    cout <= '1' when (cin = '1') and (count = (modulo-1)) else '0';  -- Carry out signal is high when modulo is
↪   reached
    output <= std_logic_vector(count) when negate = '0' else std_logic_vector(not count);  -- Negate the output
↪   if negate is high
```

```
end Behavioral;
```

Source Code 10: *one_digit, 7-segment decoder*

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

entity one_digit is
    Port ( digit : in std_logic_vector (3 downto 0);   -- input digit
           seg : out std_logic_vector (6 downto 0) );  -- 7-segment display cathode output
end one_digit;

architecture Behavioral of one_digit is

begin
    process(digit)
    begin
    case digit is  -- outout the 7-segment display bit pattern for the input digit
        when "0000" => seg <= "1000000"; -- 0
        when "0001" => seg <= "1001111"; -- 1
        when "0010" => seg <= "0100100"; -- 2
        when "0011" => seg <= "0110000"; -- 3
        when "0100" => seg <= "0011001"; -- 4
        when "0101" => seg <= "0010010"; -- 5
        when "0110" => seg <= "0000010"; -- 6
        when "0111" => seg <= "1111000"; -- 7
        when "1000" => seg <= "0000000"; -- 8
        when "1001" => seg <= "0010000"; -- 9
        when others => seg <= "1111111";
    end case;
    end process;
end Behavioral;
```

Source Code 11: *random_grid, pseudo-random number generator*

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

entity randomGrid is
    Port ( pixel_clk : in STD_LOGIC;
           rand_X :inout unsigned(6 downto 0);
           rand_Y : inout unsigned(6 downto 0)
         );
end randomGrid;

architecture Behavioral of randomGrid is
    signal rand_X_reg : unsigned(6 downto 0);
    signal rand_Y_reg : unsigned(6 downto 0);
begin
    process(pixel_clk)
    begin
        if rising_edge(pixel_clk) then
            rand_X <= ((rand_X + 3) mod 37) + 1; -- set random x and y position
            rand_Y <= ((rand_Y + 3) mod 27) + 1;
        end if;
    end process;
end Behavioral;
```

Source Code 12: *update_clock.vhd, game update clock*

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

entity updateClk is
    Generic (
        max_value : NATURAL := 4000000
    );
    Port ( clk_100mhz : in STD_LOGIC;
           update : out STD_LOGIC);
end updateClk;

architecture Behavioral of updateClk is
    signal count : unsigned(64 downto 0);
begin
    process(clk_100mhz)
    begin
        if rising_edge(clk_100mhz) then
            if count = max_value then
                count <= (others => '0');
                update <= '1';
            else
                count <= count + 1;
                update <= '0';
            end if;
        end if;
    end process;
end Behavioral;
```

Source Code 13: *vga_controller_640_60.vhd, VGA driver*

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

-- the vga_controller_640_60 entity declaration
-- read above for behavioral description and port definitions.
entity vga_controller_640_60 is
port(
   rst         : in std_logic;
   pixel_clk   : in std_logic;

   HS          : out std_logic;
   VS          : out std_logic;
   hcount      : out unsigned(10 downto 0);
   vcount      : out unsigned(10 downto 0);
   blank       : out std_logic
);
end vga_controller_640_60;

architecture Behavioral of vga_controller_640_60 is

-------------------------------------------------------------------------
-- CONSTANTS
-------------------------------------------------------------------------
subtype HTYPE is unsigned(hcount'RANGE);
subtype VTYPE is unsigned(vcount'RANGE);

-- maximum value for the horizontal pixel counter
constant HMAX  : HTYPE := "01100100000"; -- 800
-- total number of visible columns
constant HLINES: HTYPE := "01010000000"; -- 640
-- value for the horizontal counter where front porch ends
constant HFP   : HTYPE := "01010001000"; -- 648
-- value for the horizontal counter where the synch pulse ends
constant HSP   : HTYPE := "01011101000"; -- 744
-- maximum value for the vertical pixel counter
constant VMAX  : VTYPE := "01000001101"; -- 525
-- total number of visible lines
constant VLINES: VTYPE := "00111100000"; -- 480
-- value for the vertical counter where the front porch ends
constant VFP   : VTYPE := "00111100010"; -- 482
```

```
-- value for the vertical counter where the synch pulse ends
constant VSP   : VTYPE := "00111100100"; -- 484
-- polarity of the horizontal and vertical synch pulse
-- only one polarity used, because for this resolution they coincide.
constant SPP   : std_logic := '0';

-------------------------------------------------------------------------
-- SIGNALS
-------------------------------------------------------------------------

-- horizontal and vertical counters
signal hcounter : HTYPE := (others => '0');
signal vcounter : VTYPE := (others => '0');


-- active when inside visible screen area.
signal video_enable: std_logic;

begin

   -- output horizontal and vertical counters
   hcount <= hcounter;
   vcount <= vcounter;

   -- blank is active when outside screen visible area
   -- color output should be blacked (put on 0) when blank in active
   -- blank is delayed one pixel clock period from the video_enable
   -- signal to account for the pixel pipeline delay.
   blank <= not video_enable when rising_edge(pixel_clk);

   -- increment horizontal counter at pixel_clk rate
   -- until HMAX is reached, then reset and keep counting
   h_count: process(pixel_clk)
   begin
      if(rising_edge(pixel_clk)) then
         if(rst = '1') then
            hcounter <= (others => '0');
         elsif(hcounter = HMAX) then
            hcounter <= (others => '0');
         else
            hcounter <= hcounter + 1;
         end if;
      end if;
   end process h_count;

   -- increment vertical counter when one line is finished
   -- (horizontal counter reached HMAX)
   -- until VMAX is reached, then reset and keep counting
   v_count: process(pixel_clk)
   begin
      if(rising_edge(pixel_clk)) then
         if(rst = '1') then
            vcounter <= (others => '0');
         elsif(hcounter = HMAX) then
            if(vcounter = VMAX) then
               vcounter <= (others => '0');
            else
               vcounter <= vcounter + 1;
            end if;
         end if;
      end if;
   end process v_count;

   -- generate horizontal synch pulse
   -- when horizontal counter is between where the
   -- front porch ends and the synch pulse ends.
   -- The HS is active (with polarity SPP) for a total of 96 pixels.
   do_hs: process(pixel_clk)
   begin
      if(rising_edge(pixel_clk)) then
         if(hcounter >= HFP and hcounter < HSP) then
            HS <= SPP;
         else
            HS <= not SPP;
         end if;
      end if;
   end process do_hs;

   -- generate vertical synch pulse
   -- when vertical counter is between where the
   -- front porch ends and the synch pulse ends.
   -- The VS is active (with polarity SPP) for a total of 2 video lines
   -- = 2*HMAX = 1600 pixels.
   do_vs: process(pixel_clk)
```

```
    begin
       if(rising_edge(pixel_clk)) then
          if(vcounter >= VFP and vcounter < VSP) then
             VS <= SPP;
          else
             VS <= not SPP;
          end if;
       end if;
    end process do_vs;

    -- enable video output when pixel is in visible area
    video_enable <= '1' when (hcounter < HLINES and vcounter < VLINES) else '0';

end Behavioral;
```

Source Code 14: *snake.vhd, snake game logic*

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

entity snake is
    Port ( clk_100mhz : in STD_LOGIC;          --  master clock 100MHz
           pixel_clk : in STD_LOGIC;          -- pixel clock
           update : in STD_LOGIC;            -- signal to update the food position
           clk_500hz : in STD_LOGIC;          -- 500Hz clock
           xCount : in unsigned(10 downto 0);      -- x position from horizontal counter of vga driver
           yCount : in unsigned(10 downto 0);      -- y position from vertical counter of vga driver
           rand_X : in unsigned(6 downto 0);      -- random x position for the food
           rand_Y : in unsigned(6 downto 0);      -- random y position for the food
           switch : in STD_LOGIC_VECTOR(7 downto 0);  -- switches 0-7
           btn_up : in STD_LOGIC;            -- up button
           btn_left : in STD_LOGIC;           -- left button
           btn_right : in STD_LOGIC;          -- right button
           btn_down : in STD_LOGIC;           -- down button
           display : in STD_LOGIC;            -- display signal to enable rgb when blanking is off
           led : out STD_LOGIC_VECTOR(7 downto 0);    -- leds 0-7
           vgared : out STD_LOGIC_VECTOR(3 downto 0);  -- vga red
           vgagreen : out STD_LOGIC_VECTOR(3 downto 0);  -- vga green
           vgablue : out STD_LOGIC_VECTOR(3 downto 0);  -- vga blue
           seg   : out std_logic_vector (6 downto 0);   -- 7-segment display
           dp    : out std_logic;             -- 7-segment display decimal point
           an    : out std_logic_vector (3 downto 0)    -- 7-segment display anodes
         );
end snake;

architecture Behavioral of snake is
    -- rick astley never gonna give you up GIF
    type color_gif_sprite is array (0 to 31, 0 to 47, 0 to 26) of std_logic_vector(0 to 11);
    constant COLOR_GIF_ROM : color_gif_sprite := (
        -- rom value goes here
    );

    -- brick sprite
    type color_sprite is array (0 to 15, 0 to 15) of std_logic_vector(0 to 11);

    constant BRICK_ROM : color_sprite := (
        -- rom value goes here
    );

    -- apple gif sprite
    type apple_gif_sprite is array (0 to 1, 0 to 15, 0 to 15) of std_logic_vector(0 to 11);
    constant APPLE_GIF_ROM : apple_gif_sprite := (
        -- rom value goes here
    );

    -- snake body sprite
    type color_sprite_8 is array (0 to 7, 0 to 7) of std_logic_vector(0 to 11);
    constant SNAKE_ROM : color_sprite_8 := (
        -- rom value goes here
    );

    constant SIZE_INCREMENT : integer := 4;   -- size increment for the snake body

    signal size : unsigned(6 downto 0);      -- keep track of the size of the snake
    signal game_over : std_logic;  -- signal to indicate game over
    signal border : std_logic; -- signal to draw border and food

    type snake_array is array (0 to 127) of unsigned(6 downto 0);  -- array to keep track of the snake body
```

```
    -- type snakeY_array is array (0 to 127) of unsigned(6 downto 0);

    signal snakeX : snake_array;  -- snake x positions
    signal snakeY : snake_array;  -- snake y positions

    signal snakeBody : unsigned(127 downto 0);  -- vector to render the snake body
    signal direction : std_logic_vector(3 downto 0) := "0001";

    signal count : integer;  -- counter to keep track of the snake body in the for loops

    signal start : std_logic;  -- signal to start the game

    constant img_size_x : natural := 16; -- size of the image sprites
    constant img_size_y : natural := 16;

    signal is_img_painted : std_logic;    -- is the rick gif painted
    signal img_clr : std_logic_vector(11 downto 0);

    signal img_x : unsigned(10 downto 0) := to_unsigned(50, 11); -- position of rick gif
    signal img_y : unsigned(10 downto 0) := to_unsigned(50, 11);

    signal rgb : std_logic_vector(11 downto 0); -- 12 bit rbg signal

    signal current_frame_gif : unsigned(11 downto 0) := to_unsigned(0, 12); -- the current gif frame from the
↪   array

    signal brick_clr : std_logic_vector(11 downto 0);  -- brick colour sognal

    signal snake_clr : std_logic_vector(11 downto 0); -- snake colour signal

    signal bcd_counter_out_1 : std_logic_vector(7 downto 0);  -- signals for the bcd counters
    signal bcd_counter_out_2 : std_logic_vector(7 downto 0);
    signal bcd_counter_1_cout : std_logic;
    signal bcd_counter_1_clk : std_logic;
    signal bcd_counter_2_clk : std_logic;
    signal increment_score : std_logic;
    signal bcd_counter_1_reset : std_logic;
    signal bcd_counter_2_reset : std_logic;

    signal is_gif_painted : std_logic;  -- is the gif painted
    signal gif_clr : std_logic_vector(11 downto 0); -- gif colour signal
    signal gif_x : unsigned(10 downto 0) := to_unsigned(212, 11);
    signal gif_y : unsigned(10 downto 0) := to_unsigned(50, 11);
begin
    led <= switch;  -- connect the leds to the switches
    start <= switch(7);

    process(pixel_clk) -- process to select the current frame of the GIF (can control the speed too)
    begin
        if rising_edge(pixel_clk)then
            if yCount = to_unsigned(1, yCount'length) and xCount = to_unsigned(1, xCount'length) then
               current_frame_gif <= current_frame_gif + 1;
            end if;
        end if;
    end process;

    -- paint the apple gif at the current x, y position
    is_img_painted <= '1' when (xCount >= img_x and xCount < img_x + 16 and yCount >= img_y and yCount < img_y +
↪   16) else '0';

    -- select the colours from the ROMs
    img_clr <= APPLE_GIF_ROM(to_integer(current_frame_gif(11 downto 3)), (to_integer(yCount - img_y) mod
↪   16),(to_integer(xCount - img_x)) mod 16) when is_img_painted = '1' else (others => '0');

    brick_clr <= BRICK_ROM((to_integer(yCount) mod 16),(to_integer(xCount) mod 16)) when border = '1' else
↪   (others => '0');

    snake_clr <= SNAKE_ROM((to_integer(yCount) mod 8),(to_integer(xCount) mod 8)) when snakeBody /= (127 downto 0
↪   => '0') else (others => '0');

    is_gif_painted <= '1' when (xCount >= gif_x and xCount < gif_x + 216 and yCount >= gif_y and yCount < gif_y +
↪   384) else '0';
    gif_clr <= COLOR_GIF_ROM(to_integer(current_frame_gif(11 downto 3)), (to_integer(yCount(10 downto 3) -
↪   gif_y(10 downto 3)) mod 48),(to_integer(xCount(10 downto 3) - gif_x(10 downto 3))) mod 27) when
↪   is_gif_painted = '1' else (others => '0');

    -- instantiate BCD counter for minutes
    bcd_counter_unit_1 : entity work.nbit_bcd_counter(Behavioral)
        Port map (orig_clk => clk_100mhz, clk => bcd_counter_1_clk, up_down => '0', reset => bcd_counter_1_reset,
↪   cout => bcd_counter_1_cout, is_zero => open, output => bcd_counter_out_1);

    -- instantiate BCD counter for seconds
    bcd_counter_unit_2 : entity work.nbit_bcd_counter(Behavioral)
```

```
        Port map (orig_clk => clk_100mhz, clk => bcd_counter_2_clk, up_down => '0', reset => bcd_counter_2_reset,
↪   cout => open, is_zero => open, output => bcd_counter_out_2);

    -- instantiate four digits display
    four_digits_unit : entity work.four_digits(Behavioral)
        Port map (d3 => bcd_counter_out_2(7 downto 4),
                  d2 => bcd_counter_out_2(3 downto 0),
                  d1 => bcd_counter_out_1(7 downto 4),
                  d0 => bcd_counter_out_1(3 downto 0),
                  ck => clk_500hz, seg => seg, an => an, dp => dp);

    -- process to set the state of the bcd counters for the score
    process(clk_100mhz)
    begin
        if rising_edge(clk_100mhz) then
            bcd_counter_2_clk <= bcd_counter_1_cout;
            if game_over = '0' then
                bcd_counter_1_reset <= '0';
                bcd_counter_2_reset <= '0';
                if increment_score = '1' then
                    bcd_counter_1_clk <= '1';
                else
                    bcd_counter_1_clk <= '0';
                end if;
            elsif game_over = '1' then
                bcd_counter_1_clk <= '1';
                bcd_counter_2_clk <= '1';
                bcd_counter_1_reset <= '1';
                bcd_counter_2_reset <= '1';
            end if;
        end if;
    end process;

    process(clk_100mhz)
    begin
        if rising_edge(clk_100mhz) then
        if pixel_clk = '1' then
            if start = '0' then -- intial start of the game conditions
                snakeX(0) <= to_unsigned(40, 7);
                snakeY(0) <= to_unsigned(30, 7);
                for count in 1 to 127 loop
                    snakeX(count) <= to_unsigned(127, 7);
                    snakeY(count) <= to_unsigned(127, 7);
                end loop;
                size <= to_unsigned(1, 7);
                game_over <= '0';
            elsif game_over = '0' then
                if update = '1' then
                    for count in 1 to 127 loop
                        if size > count then
                            snakeX(count) <= snakeX(count-1); -- update the snake body position
                            snakeY(count) <= snakeY(count-1);
                        end if;
                    end loop;
                    case direction is
                        when "0001" =>
                            snakeY(0) <= snakeY(0) - to_unsigned(1, 7); -- update snake position based on the
↪   direction
                        when "0010" =>
                            snakeY(0) <= snakeY(0) + to_unsigned(1, 7);
                        when "0100" =>
                            snakeX(0) <= snakeX(0) - to_unsigned(1, 7);
                        when "1000" =>
                            snakeX(0) <= snakeX(0) + to_unsigned(1, 7);
                        when others =>
                            null;
                    end case;
                else
                    if img_clr /= "000000000000" and (snakeBody /= (127 downto 0 => '0')) then
                        img_x <= rand_X & "0000";  -- if food is eaten, increment size and change food position
                        img_y <= rand_Y & "0000";
                        if size < (128 - SIZE_INCREMENT) then
                            size <= size + SIZE_INCREMENT; -- increment the size of snake
                        end if;
                        increment_score <= '1';

                    elsif brick_clr /= "000000000000" and snakeBody(0) = '1' then -- border collision
                         game_over <= '1';

                    elsif (snakeBody(127 downto 1) /= (127 downto 1 => '0') and snakeBody(0) = '1') then --snake
↪   collision
                        game_over <= '1';

                    else
```

```
                        increment_score <= '0';
                    end if;
                end if;
            end if;
        end if;
        end if;
    end process;

    -- process to the direction of the snake based on the buttons
    process(clk_100mhz)
    begin
        if rising_edge(clk_100mhz) then
            if pixel_clk = '1' then
            if (btn_up = '1' and direction /= "0010") then
                direction <= "0001";
            elsif (btn_down = '1' and direction /= "0001") then
                direction <= "0010";
            elsif (btn_left = '1' and  direction /= "1000") then
                direction <= "0100";
            elsif (btn_right  = '1' and direction /= "0100") then
                direction <= "1000";
            end if;
            end if;
        end if;
    end process;

    -- process to select level based on switch input
    process(clk_100mhz)
    begin
        if rising_edge(clk_100mhz) then
        if pixel_clk = '1' then
            if switch(0) = '1' then
                if ((xCount(9 downto 3) = 0) or (xCount(9 downto 3) = 79) or (yCount(9 downto 3) = 0) or
↪   (yCount(9 downto 3) = 59) or ((xCount(9 downto 3) = 10) and (yCount(9 downto 3) >= 10 and yCount(9 downto 3)
↪   <= 20)) or ((xCount(9 downto 3) = 69) and (yCount(9 downto 3) >= 39 and yCount(9 downto 3) <= 49)) or
↪   ((yCount(9 downto 3) = 10) and (xCount(9 downto 3) >= 10 and xCount(9 downto 3) <= 20)) or ((yCount(9 downto
↪   3) = 49) and (xCount(9 downto 3) >= 59 and xCount(9 downto 3) <= 69))) then
                    border <= '1';
                else
                    border <= '0';
                end if;
            elsif switch(1) = '1' then
                if ((xCount(9 downto 3) = 0) or (xCount(9 downto 3) = 79) or (yCount(9 downto 3) = 0) or
↪   (yCount(9 downto 3) = 59) or((yCount(9 downto 3) = 20) and (xCount(9 downto 3) >= 10 and xCount(9 downto 3)
↪   <= 69)) or ((yCount(9 downto 3) =40 ) and (xCount(9 downto 3) >= 10 and xCount(9 downto 3) <= 69))) then
                    border <= '1';
                else
                    border <= '0';
                end if;
            elsif switch(2) = '1' then
                if ((xCount(9 downto 3) = 0) or (xCount(9 downto 3) = 79) or (yCount(9 downto 3) = 0) or
↪   (yCount(9 downto 3) = 59) or ((xCount(9 downto 3) = 39) and (yCount(9 downto 3) >= 0 and yCount(9 downto 3)
↪   <=10)) or ((xCount(9 downto 3) = 39) and (yCount(9 downto 3) >= 49 and  yCount(9 downto 3)<=59))) then
                    border <= '1';
                else
                    border <= '0';
                end if;
            else
                if (xCount < img_size_x or xCount > 640 - img_size_x or yCount < img_size_y or yCount > 480 -
↪   img_size_y) then
                    border <= '1';
                else
                    border <= '0';
                end if;
            end if;
        end if;
        end if;
    end process;

    -- process to paint the snake body
    process(clk_100mhz)
    begin
        if rising_edge(clk_100mhz) then
        if pixel_clk = '1' then
            for count in 0 to 127 loop
                if (xCount(9 downto 3) = snakeX(count)) and (yCount(9 downto 3) = snakeY(count)) then
                    snakeBody(count) <= '1';
                else
                    snakeBody(count) <= '0';
                end if;
            end loop;
        end if;
        end if;
    end process;
```

```
    -- set vgared, vgagreen, vgablue by splitting the 12 bit rgb signal
    vgared <= rgb(11 downto 8);
    vgagreen <= rgb(7 downto 4);
    vgablue <= rgb(3 downto 0);

    -- set the rgb signal based on the game state and the colours from the ROMs
    rgb <= (others => '0') when display = '1' else
           gif_clr when game_over = '1' else
           snake_clr when (snakeBody /= (127 downto 0 => '0')) else
           brick_clr when border = '1' else
           img_clr;

end Behavioral;
```

Source Code 15: *Python Script to convert PNGs to a VHDL ROM*

```
import os

from PIL import Image

img = Image.open('apple.png')

def resize_image(img, w, h):
    return img.resize((w, h))

def convert(img, output, w=211, h=91):
    ''' Convert image so it can be input into VHDL code like:


    type color_sprite is array (0 to 1, 0 to 1) of std_logic_vector(0 to 11);
    constant COLOR_ROM : color_sprite := (
        ("000000000000","000000000000"),
        ("000000000000", "000000000000")
    );

    Where 1 is w and 1 is h and 12 is the number of bits per pixel.
    '''
    img = resize_image(img, w, h)
    pixels = img.load()

    with open(output, 'w') as f:
        f.write('type color_sprite is array (0 to {0}, 0 to {1}) of std_logic_vector(0 to 11);\n'.format(w-1,
        ↪  h-1))
        f.write('constant COLOR_ROM : color_sprite := (\n')
        for i, y in enumerate(range(h)):
            f.write('\t(')
            for j, x in enumerate(range(w)):
                r, g, b, a = pixels[x, y]
                f.write('"{0:04b}{1:04b}{2:04b}"'.format(r >> 4, g >> 4, b >> 4))
                if j < w-1:
                    f.write(',')
            f.write(')')
            if i < h-1:
                f.write(',\n')
        f.write('\n);\n')

convert(img, 'snake-block.vhdl')
```

Source Code 16: *Python Script to convert GIFs to a VHDL ROM*

```
import os
from PIL import Image
import requests
import glob
import traceback

def resize_image(img, w, h):
    # resize to match the aspect ratio
    return img.resize((w, h))

def convert(img, w=16, h=16):
    ''' Convert image so it can be input into VHDL code like:
    type color_sprite is array (0 to 1, 0 to 1) of std_logic_vector(0 to 11);
    constant COLOR_ROM : color_sprite := (
        ("000000000000","000000000000"),
        ("000000000000", "000000000000")
    );

    Where 1 is w and 1 is h and 12 is the number of bits per pixel.
    '''
    pixels = img.load()

    output = ''
    for i, y in enumerate(range(h)):
        output += '\t('
        for j, x in enumerate(range(w)):
            # import pdb; pdb.set_trace()
            r, g, b, a = pixels[x, y]
            output += '"{0:04b}{1:04b}{2:04b}"'.format(r >> 4, g >> 4, b >> 4)
            if j < w-1:
                output += ','
        output += ')'
        if i < h-1:
            output += ',\n'
    return output

def save_image(output, f_name_out, dimensions):
    with open(f_name_out, 'w') as f:
        f.write('type color_sprite is array (0 to {0}, 0 to {1}) of std_logic_vector(0 to
        ↪  11);\n'.format(dimensions[0]-1, dimensions[1]-1))
        f.write('constant COLOR_ROM : color_sprite := (\n')
        f.write(output)
        f.write('\n);\n')

def convert_gif(f_name, w=None, h=None):
    ''' Convert gif so it can be input into VHDL code like:
    type color_sprite is array (0 to 1, 0 to 1) of std_logic_vector(0 to 11);
    constant COLOR_ROM : color_sprite := (
        ("000000000000","000000000000"),
        ("000000000000", "000000000000")
    );

    Where 1 is w and 1 is h and 12 is the number of bits per pixel.
    '''
    imgs = gif_to_pil_imgs(f_name)
    output_gif = ""
    # import pdb; pdb.set_trace()
    for i, img in enumerate(imgs):
        output_gif += '\n\t('
        img = resize_image(img, w, h)
        # convert to rgba
        img = img.convert('RGBA')
        output_gif += convert(img, w, h)
        output_gif += ')\n' + ' -- {0}\n'.format(i)
        if i < len(imgs)-1:
            output_gif += ',\n'
    return output_gif

def save_gif(output_gif, f_name_out, dimensions):
    with open(f_name_out, 'w') as f:
        f.write('type color_gif_sprite is array (0 to {2}, 0 to {1}, 0 to {0}) of std_logic_vector(0 to
        ↪  11);\n'.format(dimensions[0]-1, dimensions[1]-1, dimensions[2]-1))
        f.write('constant COLOR_GIF_ROM : color_gif_sprite := (\n')
        f.write(output_gif)
        f.write('\n);\n')

def gif_to_pil_imgs(f_name):
    gif = Image.open(f_name)
```

```
    imgs = []
    for i in range(gif.n_frames):
        gif.seek(i)
        imgs.append(gif.copy())
    return imgs

desired_height = 16
for f_name in glob.glob('gifs/*.gif'):
    # try:
    imgs = gif_to_pil_imgs(f_name)
    dimensions = imgs[0].size
    w = int(desired_height * dimensions[0] / dimensions[1])

    output_gif = convert_gif(f_name, w=w, h=desired_height)
    save_gif(output_gif, f_name.split('.')[0] + '.vhdl', (w, desired_height, len(imgs)))
    # except Exception as e:
    #     print(f"Error converting {f_name}: {e}")
```

END OF REPORT