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 26, 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

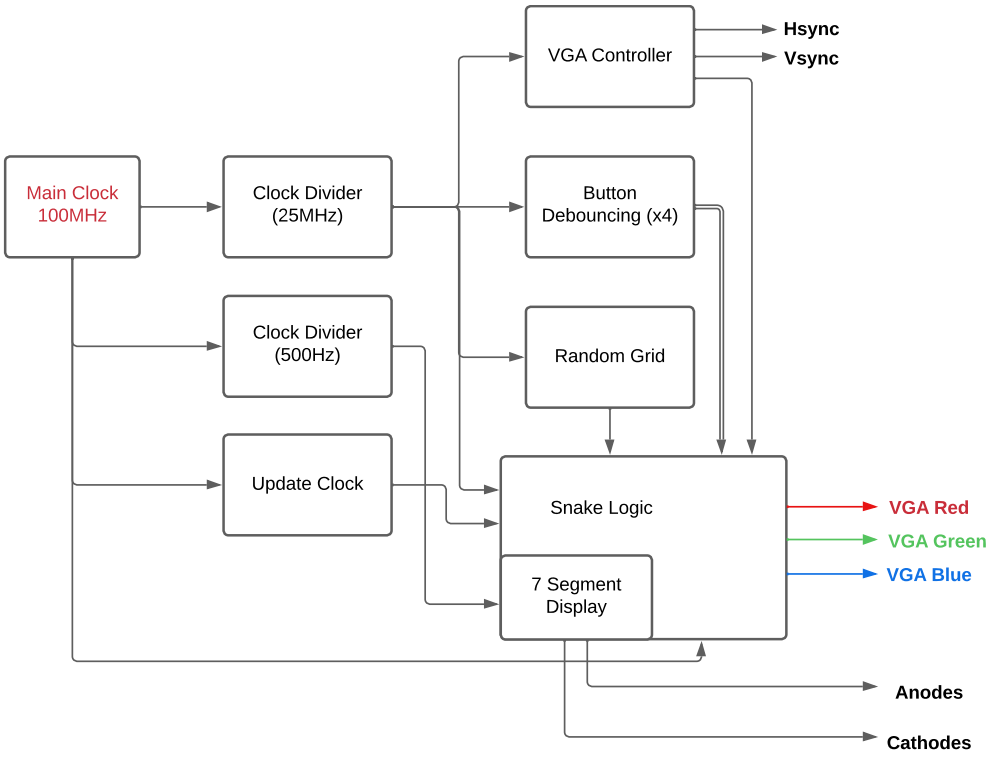


FIG. 1: *High level architecture of the system*

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

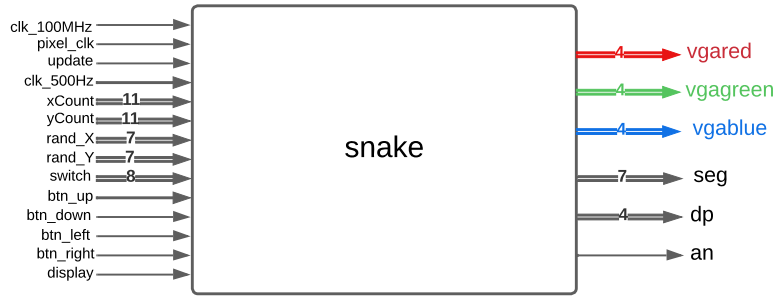


FIG. 2: *Inputs/outputs of snake module*

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.

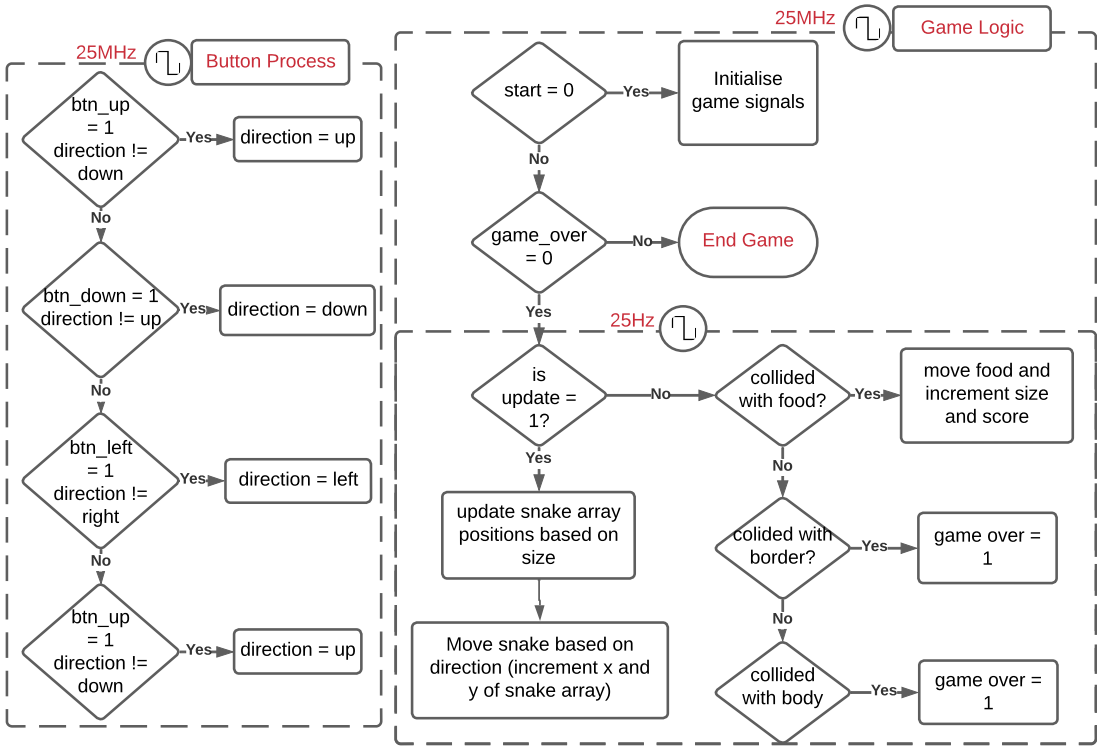


FIG. 3: *Button and game logic processes of the snake module*

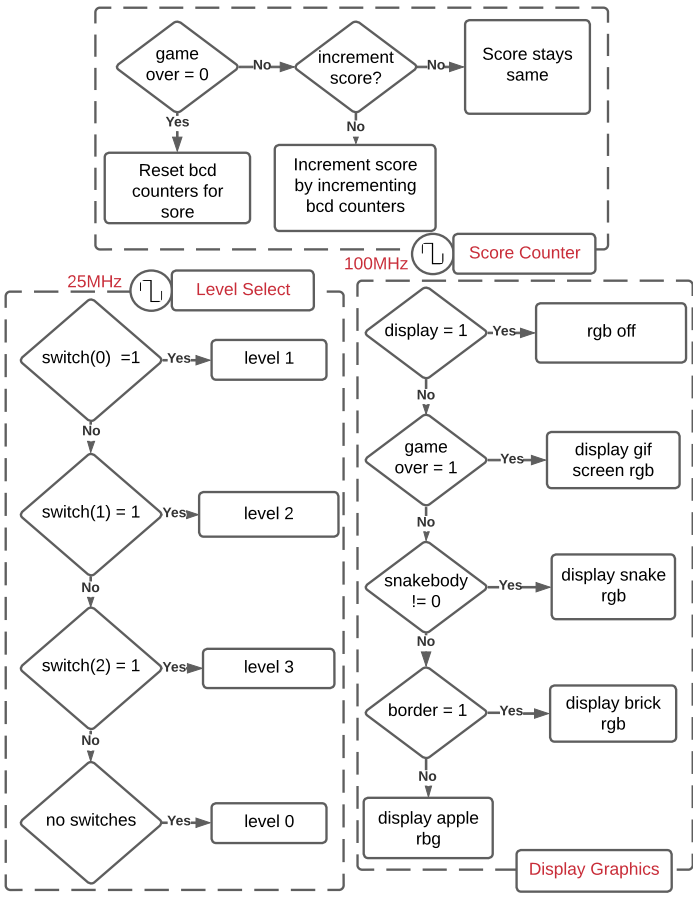


FIG. 4: *Level select and score count process and colour rendering*

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

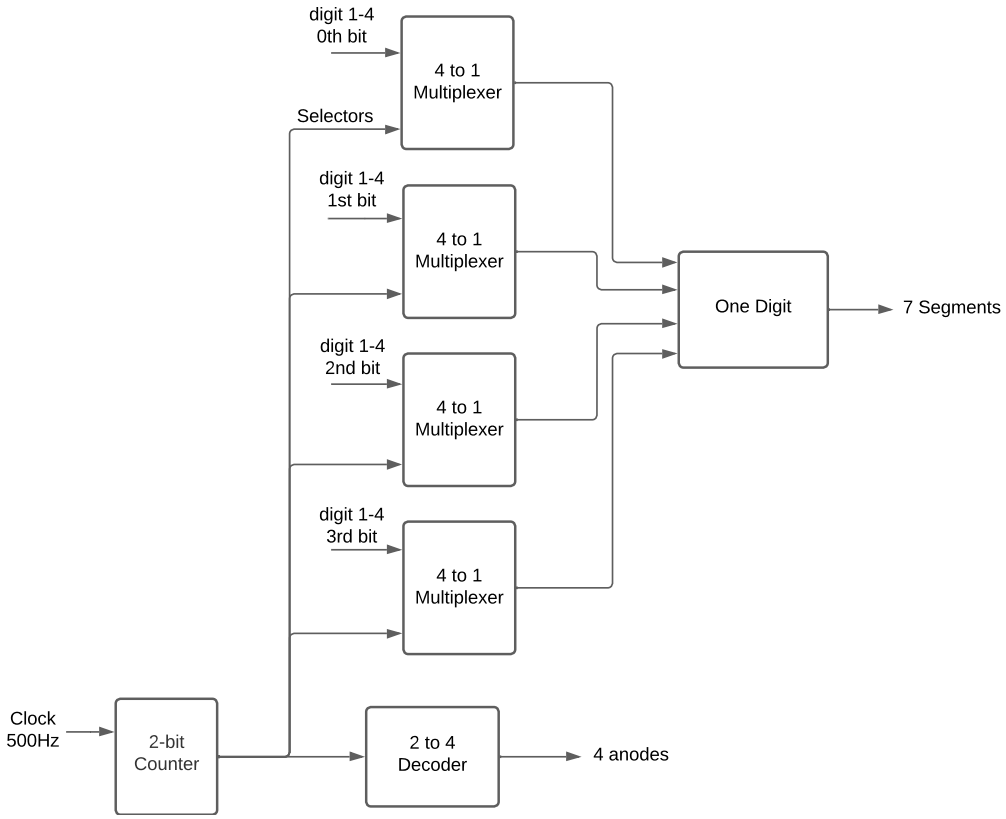


FIG. 7: *Architectural block diagram of the four_digits module*

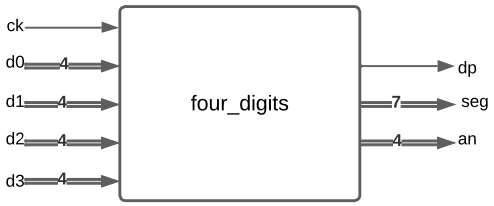


FIG. 8: *four_digits entity diagram*

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.

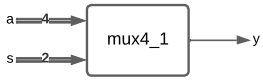


FIG. 9: *mux4_1 entity*

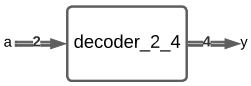


FIG. 10: *decoder_2_4 entity*

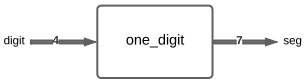
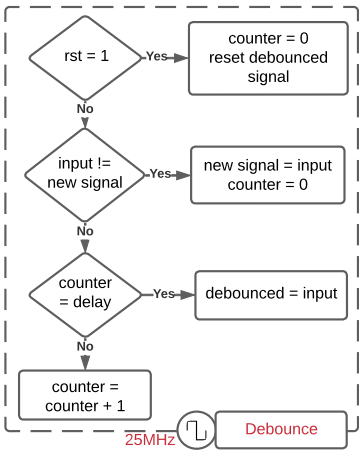


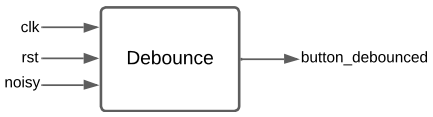
FIG. 11: one_digit entity

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

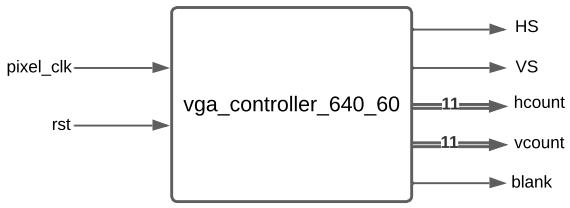
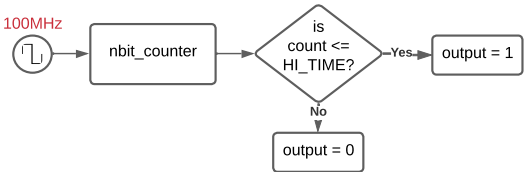


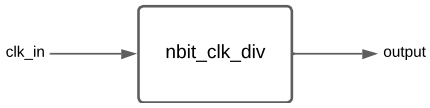
FIG. 13: *VGA Controller module*

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.

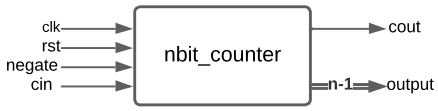


FIG. 15: *nbit_counter module*

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

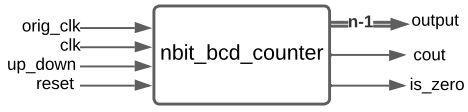


FIG. 16: *nbit_bcd_counter module*

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 script is in Source Code 13 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

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

6. APPENDIX