University of California, Los Angeles

School of Engineering and Applied Science

Department of Electrical and Computer Engineering

Professor M. Potkonjak

TA: Rohan Surve

COM SCI M152A

Final Project – Snake

Joani Bajlozi - 804 981 541

Aaron Lim - 004 999 347

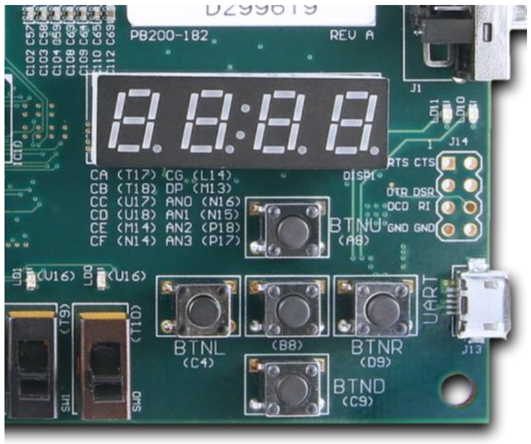
Project Statement and Design Description

1.0. Introduction

The goal of this project was to replicate the classic game of Snake, which made its debut in 1997. The game was first introduced on the Nokia 6110 and was programmed by software engineer Taneli Armanto. The game starts off with a snake that has a length of one block and it gets progressively longer the more food it is able to consume. The "food" is simply a randomly generated block that spawns anywhere on the grid, and the player is responsible of getting the snake to eat it, all while avoiding hitting the walls that surround the grid, or itself.

1.1. Overall Design Implementation

The main objective of this project was to implement a similar version of this nostalgic game on the Nexys3 Spartan-6 FPGA Board. The user would use a series of four pushbuttons to control the snake's movement, and an additional button to reset the game. The score, which is simply the length of our snake, is also displayed on the seven-segment displays, coupled with various words that flash periodically when the player either wins or loses a game. The pushbuttons, and the seven-segment displays used on the FPGA board are shown below:



In order to win the game, the snake must achieve a length of 100 blocks, which was arbitrarily chosen due to a mapping error (presumably due to limited memory) on the Nexys3 board. If the user does win the game, their score will flash along with the word **DUBS** on the seven-segment displays at a rate of 0.5 Hz. If the user loses the game by colliding with the borders, or with itself, then the score will flash along with the word **FAIL** on the seven-segment displays. There are a total of four borders, one for each side, which are two blocks thick (i.e. 20 pixels).

1.2. Top Module

The overall implementation of the game was headed by the `top_snake.v` module, which was responsible for instantiating the button-debouncing modules for each of the four pushbuttons responsible for the snake's movement, and for instantiating the gameplay module, `snake.v`, which was responsible for the game logic implementation, as well as instantiating all of the helper modules created specifically for this project.

1.3. Random Food Generator

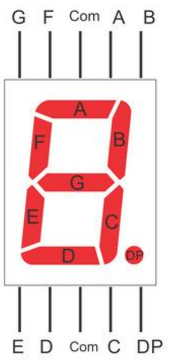
One of the most significant helper modules used by our snake gameplay module was the `random_food` module, which was responsible for generating two random coordinates, one for the **x** (horizontal), and one for the **y** (vertical) direction, where the food will be placed when the snake consumes the previous food block. The module utilizes a pseudo random technique which relies on the very fast nature of the 100 MHz clock in order to behave in a random fashion when it is called to use. The module takes in the Nexys3 built-in clock (denoted by `clk`) as well as reset (denoted by `rst`) as inputs, and outputs two variables `rand_x` and `rand_y`, that represent the coordinates. First, there are two step registers (i.e. `step_x` and `step_y`) and two temporary registers (i.e. `temp_x` and `temp_y`) declared, which will be altered to behave in a random fashion. If the game is reset, the coordinates are assigned to the center of the VGA display, while the horizontal step is given a value of 30, and the vertical step is given a step of 70. At every positive clock edge, each step is incremented by 10 blocks and then MOD-ed by 100, which are then added to the coordinate registers, where they are subsequently MOD-ed by the coordinate of the last block that the snake can touch on either border. Since this module runs at every positive edge of the 100 MHz clock of the Nexys3 board, this operation will be executed quite a number of times before the module is called again. Therefore, due to its repetitive nature, it will operate in a pseudo random fashion.

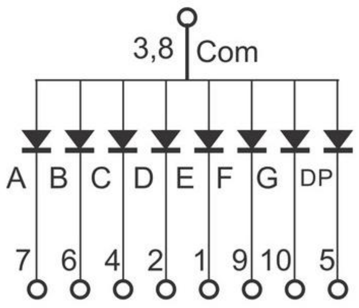
1.4. Score Display

The `score_display` module was responsible for converting the size of the snake into three different 4-bit registers in order for them to be displayed on all four seven-segment displays on the Nexys3 board. Since the maximum size the snake can have is 100, only the last 3 digits of the display would be necessary. Hence, the module only converts the score into three different registers, while a zero is constantly displayed on the outer-left segment display.

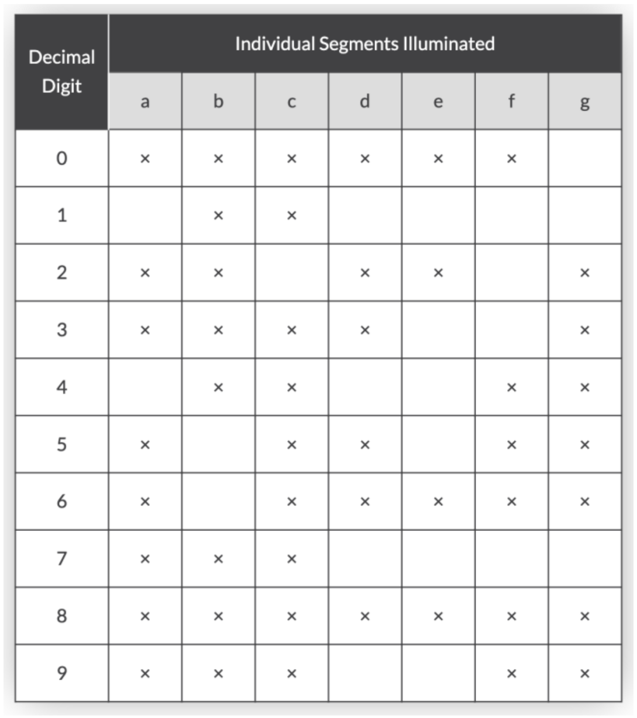
1.5. Segment Display

The `seg_display` module would individually take each of the 4-bit registers outputted from the `score_display` module and return a 7-bit register `seg`, where each bit corresponds to a segment on the display. The anatomy, along with a top-level circuit schematic of the seven-segment display is shown below:





We only used seven of the eight bits available on the seven-segment display as the DP bit was unnecessary for our implementation of the snake game. It should be noted that the seven-segment display is an active low device, where each segment of the display is illuminated when it is set to logic low. The following chart depicts the truth table of the seven-segment display which was used to encode each decimal digit:



Decimal Digit	Individual Segments Illuminated						
	a	b	c	d	e	f	g
0	×	×	×	×	×	×	
1		×	×				
2	×	×		×	×		×
3	×	×	×	×			×
4		×	×			×	×
5	×		×	×		×	×
6	×		×	×	×	×	×
7	×	×	×				
8	×	×	×	×	×	×	×
9	×	×	×			×	×

1.6. Clock Divider

The seven-segment displays also needed two different clocks frequencies to function as desired. First, the display required a 500 Hz clock in order to display the score as it can only select one anode at a time, therefore it is necessary to have a clock that operates at a speed much faster than the human flicker fusion threshold, so that it gives the illusion that all of the digits were displayed at the same time, when in fact, it does not. During the `win` or `lose` states, the display also alternates between a word and the player's score, and thus requires a 0.5 Hz clock for better visibility and readability. The clock divider uses the same set of logic as the other clock divider modules that were implemented for previous lab assignments.

1.7. VGA Display

The `vga640x480` module was created in order to display the game on a computer display via the VGA output of the Nexys3 board. The actual implementation of the VGA module was outside the scope of the class. Hence, it was taken directly from an online source (referenced at the end of the report) and modified it for the implementation of our snake game.

1.8. Update Clock

Finally, the last helper module that was used by the gameplay module is the `update_clk` module, which was responsible for creating an 8 Hz strobe, giving the effect that the snake moves block-by-block, instead of moving continuously. This is the default method in snake movement in most remakes since the game's creation. Therefore, it was only fitting to implement the appropriate retro aesthetic, instead of having continuous movement.

1.9. Gameplay Logic

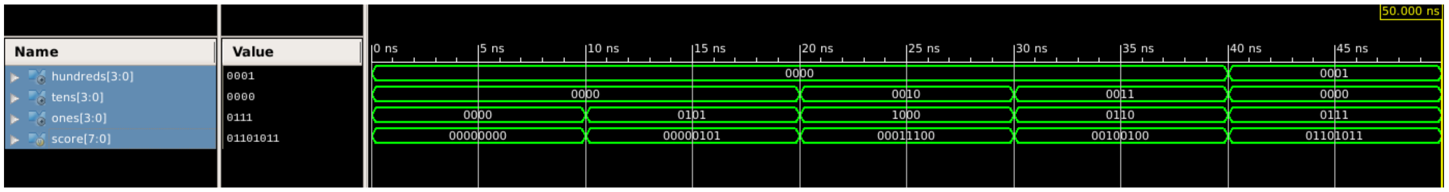
The gameplay was carried out in the `snake` module, where each of the previously mentioned helper modules were utilized and incorporated. After instantiating the `clk_div` module, the direction of the snake was determined from the button inputs. If the direction pressed by the player was the opposite of the current direction (e.g. snake moving up and player presses down), the direction of the snake would simply remain the same as it was before. A 25 MHz pixel strobe was then generated for VGA purposes.

The snake itself is just composed of two arrays that hold the horizontal and vertical coordinates, respectively. As the snake length becomes larger, more and more of the array elements start to hold coordinates that appear of the VGA display. This logic makes it possible to have each block essentially move to the location of the next block, which will implement the classic snake turning functionality. The gameplay logic is dictated by the reset, win, or lose states, as it behaves differently in each scenario. At every positive clock edge, if the reset is true, the snake is placed at the top right corner, the food is placed in the center, and the snake coordinates are initialized to values that are outside of the VGA display. If the game has not been won nor lost, we will first check if we need to update our coordinates, which is dictated by the 8 Hz strobe. If we do need to update our coordinates, we check direction and update accordingly. If we do not need to update, we first check if the head of the snake has hit the apple, to which then we would generate a new location accordingly and check if the game has been won, which is when the length is 100. If it hasn't hit the apple, then we check if we have hit either the wall or the snake body itself, to which we would assign a value of 1 to the lose state. The snake module then basically displays the game to the VGA screen and the score to the seven-segment display.

Simulation Documentation

2.0. Score Display Testbench

The following screenshot of the simulation waveforms below demonstrate that the `score_display` module is functioning normally under the game's guidelines.

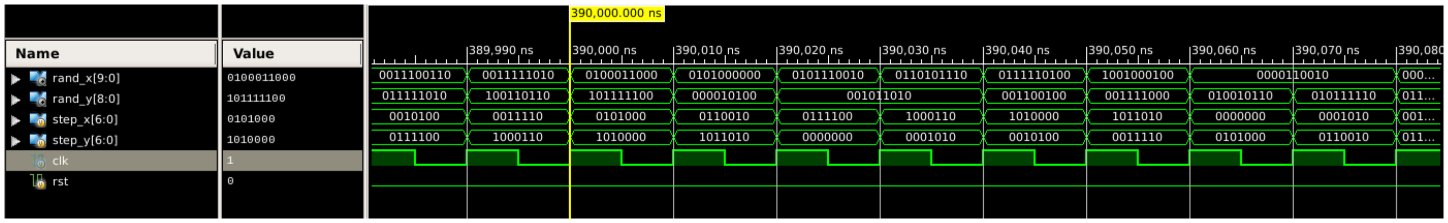


It can be observed that 4-bit registers, `hundreds`, `tens`, and `ones` are outputting the appropriate digits corresponding to various values of score, which is listed in the table below:

Number	8-Bit Binary Representation	Hundreds	Tens	Ones
5	00000101	0000	0000	0101
28	00011100	0000	0010	1000
36	00100100	0000	0011	0110
107	01101011	0001	0000	0111

2.1. Random Food Generator Testbench

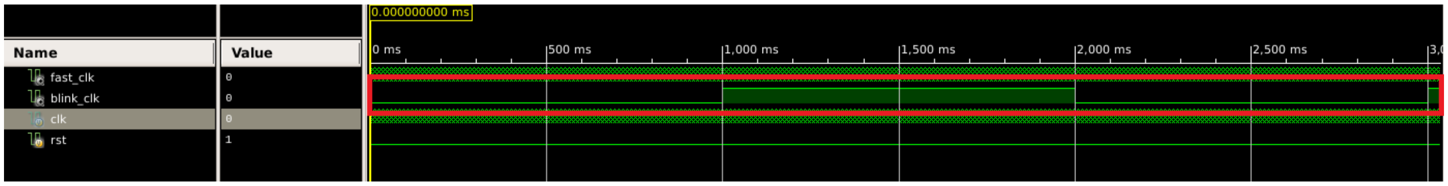
The following screenshot of the simulation waveforms below demonstrate that the `random_food` module is functioning normally under the game's guidelines.

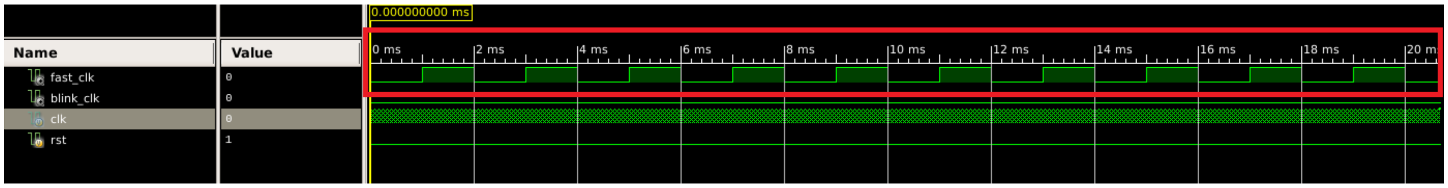


It can be observed that both coordinate registers, `rand_x` and `rand_y` are coordinates which are multiples of 10 that are within the borders of our snake game.

2.2. Clock Divider Testbench

The following screenshot of the simulation waveforms below demonstrate that the `clk_div` module is functioning normally under the game's guidelines. As highlighted in red, the four different clocks possess their respective frequencies.





Contribution

3.0. Individual Contribution

Both partners contributed equally to designing the snake game. Aaron focused more on the game logic, while Joani focused more on the actual functionality of the game. Aaron specifically worked on implementing the rules of the game, while Joani specifically worked on the helper modules that controlled the scoreboard, the pseudo random number generator, and button-debouncing.

Contribution

4.0. Conclusion

In conclusion, designing the classic game Snake was the most beneficial learning experience that we could have had in this class as it required us to construct a working game from complete scratch, which included developing the game logic as well as its implementation.

We first started off by examining existing implementations and seeing which features of the snake we wanted to use in our version of the game. We then created a plan and started by getting the VGA to work. Once we learned how to control the display of the VGA, we then created the walls and the space that the snake would be able to move in. This led us to figuring out how to move a block in the classic snake fashion, which is block by block instead of continuous movement, while also having the moving block detect the walls and bounce off. The main challenge of getting the block to move this way is realizing that the clock needed to perform this movement needed to have a strobe, which means that its duty cycle was very low. Once we were able to calibrate the walls and the block movement, it was simply time to code the game logic.

We simply made a list of all of the rules we needed to follow, then began implementing the logic in the main module for this game. We added an extra functionality, which was to display the length of the snake on the seven-segment display, so we incorporated this functionality within the game implementation itself.

One problem we did not get to fix was the case where an apple would be generated on the snake body itself. Since our pseudo random generator worked very well, this problem was very rare and we only encountered it once during the time we played the game, but it would have still been beneficial to implement this functionality if we had more time. After we thought we had everything solidly pieced together, it was finally time to try running the code and seeing if it worked, which brought up the final challenge.

The original implementation had too much logic for the FPGA board to handle, which meant we had to cut out some of the functionalities. We went with making the snake have a maximum length of 100 blocks, which meant the game would theoretically end earlier than the original game, but this was a price we had to pay due to the limited functionalities of an FPGA board.

In the end, the game actually worked very well with the only flaw being the apple generation, but overall, it was a complete success.