Snake Game on VGA using FPGA

Nolan Nguyen
College of Engineering
California State University, Long Beach
Long Beach, USA
nolan.nguyen01@student.csulb.edu

Tri Nguyen
College of Engineering
California State University, Long Beach
Long Beach, USA
tri.nguyen03@student.csulb.edu

Jordan Havelaar
College of Engineering
California State University, Long Beach
Long Beach, USA
Jordan.Havelaar01@student.csulb.edu

***Abstract*—The purpose behind our final project is to implement the "Snake" game, which features a snake that gets longer as it eats apples, on our FPGA board through VGA (Video Graphics Array). You can move left, right, up and down. There will be two different types of collisions in the game, which are colliding with apples to increase length, or colliding into yourself, which will result in the game ending. Using the push buttons to move the snake, collect the maximum number of apples to win the game. The score will be kept through the seven-segment display on the board as well.**

I. Introduction

The project focuses on creating an FPGA-based snake game. We will be using the Nexys A7 100T Trainer Board, which has an Artix 7 FPGA and a variety of peripherals. We will be using buttons, displays, and VGA output. The maximum score for the snake game will be 15. There will be a certain size to the background that aligns with our VGA display (640 x 480). There will be random apples spawned as you collect apples. The score will be displayed on the 7-segment display. There will be two types of collisions: fatal and nonfatal collisions. The fatal collisions will be a collision into yourself or a collision into the walls of the background. The non-fatal collisions are with the apples, which increase the length of the snake.

II. Background

A. VGA Controller

To implement the project, we must study the concepts of VGA (Video Graphics Array). This is an interfacing for CRT (cathode ray tube) monitors. There are five main signals in the VGA connector and the module. They are hsync (horizontal sync), vsync (vertical sync), red, green, and blue. Depending on the FPGA, the color signals can vary in size. In our case, the Artrix-7 has 4 bits for each color signal. VGA can handle 25MHz as the clock speed for the pixel generation. Since the base clock speed of our FPGA is 100MHz, we need to generate a pixel every 4 cycles.

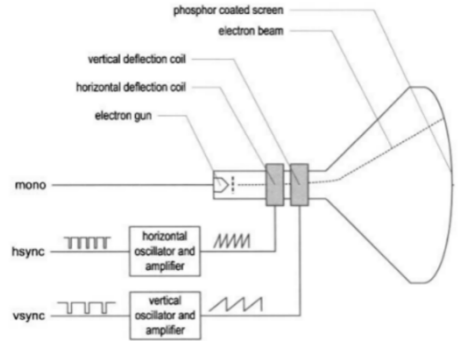

Figure 1. Cathode Ray Tube (CRT) with VGA Monitor

B. Seven Segment Display

The score will be displayed via the onboard seven segment LED display. The display has, as the name suggests, seven segments that are LEDs. In a configured pattern, each display can represent numbers zero through 15 in one-bit hexadecimal form, so zero through nine are represented as 0, 1, 2, etc. while ten through fifteen are represented with letters A, B, C, D, E, and F. In our case, however, ten through fifteen will not be represented as a single-bit hex number since we are using two LED displays. Each segment in the LED is activated if the control signal for that signal is 0, as every LED is configured as active low.

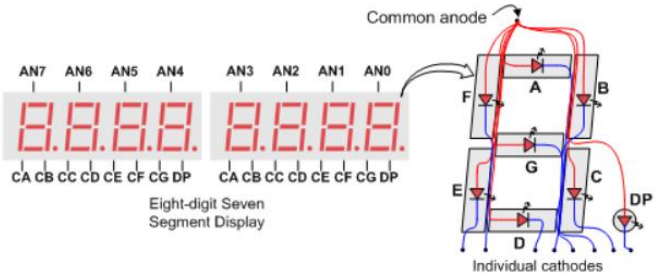

Figure 2. Diagram of Seven Segment Display

For our project, we will be utilizing the seven segment displays provided on our Nexys A7- 100T Board. There are eight displays, which are all tied to a common anode, which means to turn these on, we must send a logic level low to the pins. The individual segments themselves have individual cathodes, which means they use a logic high to light up the segment. A more complicated multiplexing scheme is needed to implement two-digit scores on two displays.

III. Methodology

A. VGA and Pixel Generation

To generate images on the monitor through VGA, we need to implement the VGA synchronization module and the pixel generation module. Pong Chu's "*FPGA Prototyping by Verilog Examples"* provides a very comprehensive method creating the VGA controller. The two components are horizontal sync (*hsync*) and vertical sync (*vsync*). The *hsync* and *vsync* signals will be generated with a timing of 25MHz, which also spans the entire screen.

To generate this 25MHz timing, we need to use a mod-4 counter. Essentially, we are dividing the 100MHz base clock of the board by four to achieve proper VGA synchronization. This was integrated into the VGA controller module to keep the number of files to a minimum.

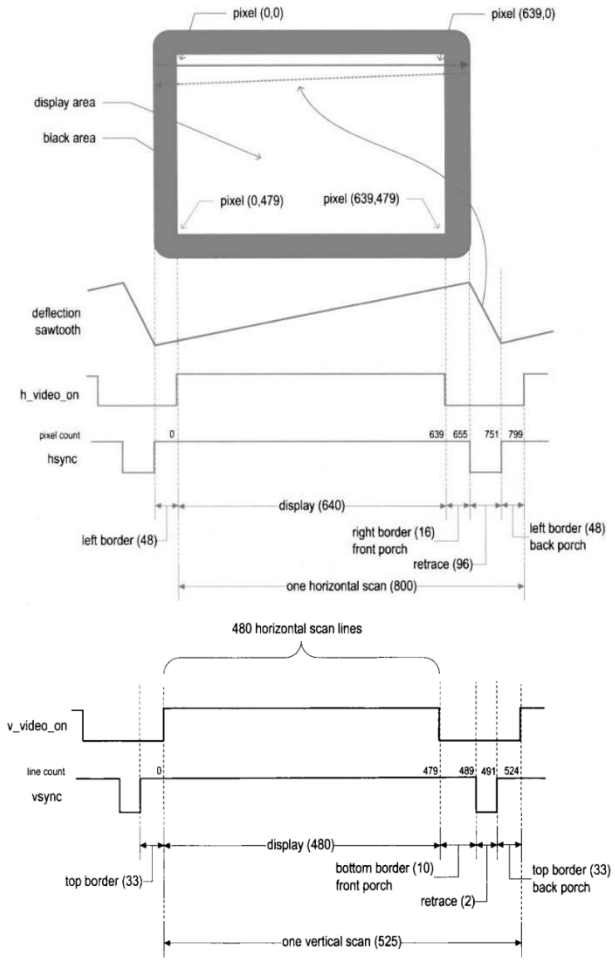

Figure 3. Of Horizontal and Vertical Scan Diagrams

We will instantiate these two modules in the *top* module, which is where all our modules will be connected. To create the background of our game, we will initialize the colors in our top module using the instantiation of the pixel generation module.

B. Button and Game Logic

To implement input and the snake movement, we need to determine the direction based on the button press to make the snake move up, down, left, and right. In a certain amount of time, the food will spawn randomly and let the users try to control the snake to eat the food or collide with the food. Then, we want to calculate and update the snake position. We also need to check for collisions, whether the snake itself or the map's border. Next, we check for the food collision. Every time the snake hits a food collision, the snake will increase or become extended. The users control the snake to hit the border or itself, and then the game will end.

C. Random Food Generation

The food position would be updated as the snake eats and the clock signal refreshed. The module uses a linear feedback shift register to generate pseudo-random values, so the food's position is based on the random values. It used 10 bits to match the x and y coordinates of the snake. We also ensured the position of the food aligned with the grid by using modulo with the height and width divided by the snake's size to guarantee it was within the screen. As a result, the food is generated randomly with the same specific pattern within the screen. We wanted the food to be generated in random places every time we reset the snake to its initial position, but it was enough for testing and observing how the random generation worked.

D. Game Tick

Despite the VGA clock being 25MHz, it is impossible to play a game at that speed. Therefore, we needed to move the snake at a more realistic and visible clock speed for the player. As such, we created a clock divider module that using the 100MHz FPGA clock, stepped down to a mere 5Hz. We simply set a counter variable to increase until it reached 19999999.

E. Score on Seven Segment Display

To represent the two-digit score on the FPGA board, four modules are needed: one to control when the score resets to zero or when one point is added to the score after an apple has been eaten, a seven segment LED decoder modules to represent the tenth and one's place of the score, and a top module to instantiate the other modules and verify functionality. Starting with the score counter, two 4-bit output vectors will be initialized: tens and ones. They will hold the values that we want to display on the corresponding seven-segment LED.

Our inputs will be *clk*, *reset*, and *add*. Counter will start on the initial rising edge of *clk* and *reset*. Reset represents when the player has a fatal collision and add represents when an apple has been eaten. At this time, a variable "score" will be initialized to zero. If the signal for reset is one, the score stays the same. If the signal for add is logic level high and the score is less than thirty, then one point is added to the score. Once these conditions are checked, tens and one's values of the score are separated and stored into their respective vectors. The use of arithmetic operations is required to store values in the correct vector.

Afterwards, the decoder module will be programmed. The module will read the 4-bit vector and display the corresponding segments on the LED's. The

LED control signals are represented with one 8-bit signal. For testing purposes, a button will be used to add one point to the score counter and the reset button will be used to reset the score counter to zero.

IV. Results and Evaluation

The overall snake logic and game works, along with the score on the seven-segment display. We verified the operation of the VGA controller using test benches that toggled the bits to signal the horizontal and vertical scans. The following Figure 4 shows the vertical sync signal being high after around 640 counts of the horizontal scan.

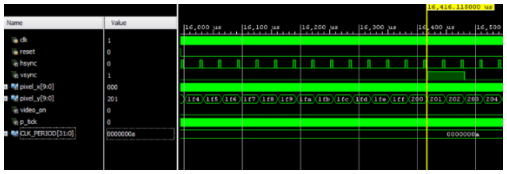

Figure 4. Waveform of VGA Controller: Vsync is toggled on every 1/60 second, indicating that a frame is complete.

The timing of the module was also 25MHz, which generates these signals that are toggled on and off accordingly.

In order to verify the movements and the snake logic, we employed an exhaustive self-checking test bench that simulated all possible movements of the snake, as well as the collisions into the border. The test bench also outputs to the console based on the cases checked in the for loop that makes up the test bench. The idea behind an exhaustive self-checking test bench is to be able to comprehensively point out any errors using the monitors in Verilog that output to the console. If an event happens, print out a certain message that indicates success or failure.

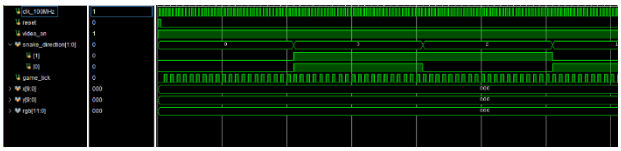

Figure 5. Waveform of Snake Movement (Zoomed In)

```
Simulating Snake Moving Right...
Game Tick           0, Snake Head Position: ( 330,  240)
Game Tick           5, Snake Head Position: ( 380,  240)
Game Tick          10, Snake Head Position: ( 430,  240)
Game Tick          15, Snake Head Position: ( 480,  240)
Simulating Snake Moving Down...
Game Tick           0, Snake Head Position: ( 530,  240)
Game Tick           5, Snake Head Position: ( 530,  290)
Game Tick          10, Snake Head Position: ( 530,  340)
Game Tick          15, Snake Head Position: ( 530,  390)
Simulating Snake Moving Left...
Game Tick           0, Snake Head Position: ( 530,  440)
Game Tick           5, Snake Head Position: ( 480,  440)
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:01 ; elapsed =
run 765 us
Game Tick          10, Snake Head Position: ( 430,  440)
Game Tick          15, Snake Head Position: ( 380,  440)
Simulating Snake Moving Up...
Game Tick           0, Snake Head Position: ( 330,  440)
Game Tick           5, Snake Head Position: ( 330,  390)
Game Tick          10, Snake Head Position: ( 330,  340)
Game Tick          15, Snake Head Position: ( 330,  290)
Simulating Snake Collision with Boundary...
Game Tick           0, Snake Head Position: ( 330,  240)
Game Tick           5, Snake Head Position: ( 380,  240)
Game Tick          10, Snake Head Position: ( 430,  240)
Game Tick          15, Snake Head Position: ( 480,  240)
```

Figure 6. Output to Console of the Test Bench

In addition, we also needed to verify the functionality of our linear shift registers that created the random food generation. Given the pseudo-random nature of the module, we utilized the same test bench in verifying the snake game logic. We simulated fifteen instances where food would spawn and displayed the coordinates of each food after it was eaten. The module worked as intended, and the results are in the figures below.

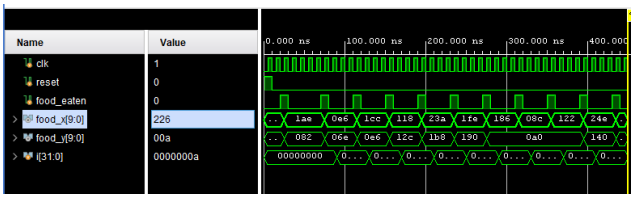

Figure 7. Random food generation waveform, toggling food_eaten

```
# run 1000ns
Food position: ( 430,  130)
Food position: ( 230,  110)
Food position: ( 460,  230)
Food position: ( 280,  300)
Food position: ( 570,  440)
Food position: ( 510,  400)
Food position: ( 390,  160)
Food position: ( 140,  160)
Food position: ( 290,  160)
Food position: ( 590,  320)
Food position: ( 550,   10)
```

Figure 8. Output of food positions to console (pseudo-random)

For the food positions, they are pseudo-random because that is the nature of the linear shift registers. If we were to implement true random food sequences, we would have had to import a library and configure our Vivado and Verilog compiler for that purpose.

The Binary Coded Decimal (BCD) to the Seven Segment decoder module was verified through a self-checking exhaustive test bench as well. If the input was less than 9 (decimal) it would flag for an invalid BCD by sending a logic level high on 8 bits or decimal 255. Otherwise, it would output a modulo 10 of the input. We verified the behavior and output it to the consoler in the following figure.

Figure 9. BCD Waveform

```
PASS: For ins =   0, counter = 0, got expected oneDigit =   0
PASS: For ins =   0, counter = 1, got expected oneDigit = 255
PASS: For ins =   1, counter = 0, got expected oneDigit =   1
PASS: For ins =   1, counter = 1, got expected oneDigit = 255
PASS: For ins =   2, counter = 0, got expected oneDigit =   2
PASS: For ins =   2, counter = 1, got expected oneDigit = 255
PASS: For ins =   3, counter = 0, got expected oneDigit =   3
PASS: For ins =   3, counter = 1, got expected oneDigit = 255
PASS: For ins =   4, counter = 0, got expected oneDigit =   4
PASS: For ins =   4, counter = 1, got expected oneDigit = 255
PASS: For ins =   5, counter = 0, got expected oneDigit =   5
PASS: For ins =   5, counter = 1, got expected oneDigit = 255
PASS: For ins =   6, counter = 0, got expected oneDigit =   6
PASS: For ins =   6, counter = 1, got expected oneDigit = 255
PASS: For ins =   7, counter = 0, got expected oneDigit =   7
PASS: For ins =   7, counter = 1, got expected oneDigit = 255
PASS: For ins =   8, counter = 0, got expected oneDigit =   8
PASS: For ins =   8, counter = 1, got expected oneDigit = 255
PASS: For ins =   9, counter = 0, got expected oneDigit =   9
PASS: For ins =   9, counter = 1, got expected oneDigit = 255
PASS: For ins =  10, counter = 0, got expected oneDigit =   0
PASS: For ins =  10, counter = 1, got expected oneDigit =   0
PASS: For ins =  11, counter = 0, got expected oneDigit =   1
PASS: For ins =  11, counter = 1, got expected oneDigit =   1
PASS: For ins =  12, counter = 0, got expected oneDigit =   2
PASS: For ins =  12, counter = 1, got expected oneDigit =   2
PASS: For ins =  13, counter = 0, got expected oneDigit =   3
PASS: For ins =  13, counter = 1, got expected oneDigit =   3
PASS: For ins =  14, counter = 0, got expected oneDigit =   4
```

Figure 10. Console Output for BCD

Regarding, there were a few problems encountered in the middle of development. The first issue that arose was with the version of Vivado used for initial development. Vivado 2016.2 was utilized to create a simple VGA controller, but as the game logic and overall modules became more complicated, this version would have many glitches, and the game would not operate as intended. It was found that Vivado 2022.2 was a better fit for our project.

The next step was one of the most challenging steps we got stuck on for a long time. It was the snake growth. We made the snake grow every time it ate the food, but there was a problem with shifting the body fragments after the snake's size increased after colliding with food. We used a for-loop to move the body segments, and we got an infinite loop because the procedural block (always) in the snake game module did not read the loop index properly due to blocking assignment (=). We still didn't fully understand why, but we got it to work by manually shifting all body segments by every snake's length. We know it was not the most efficient way, but our max snake's size is not high enough to make the code cumbersome to modify. With the final version, however, we were able to implement a for-loop without the need for manually shifting the snake's segments.

For the score, we were only able to implement one seven segment display, resulting in our maximum score being 9.

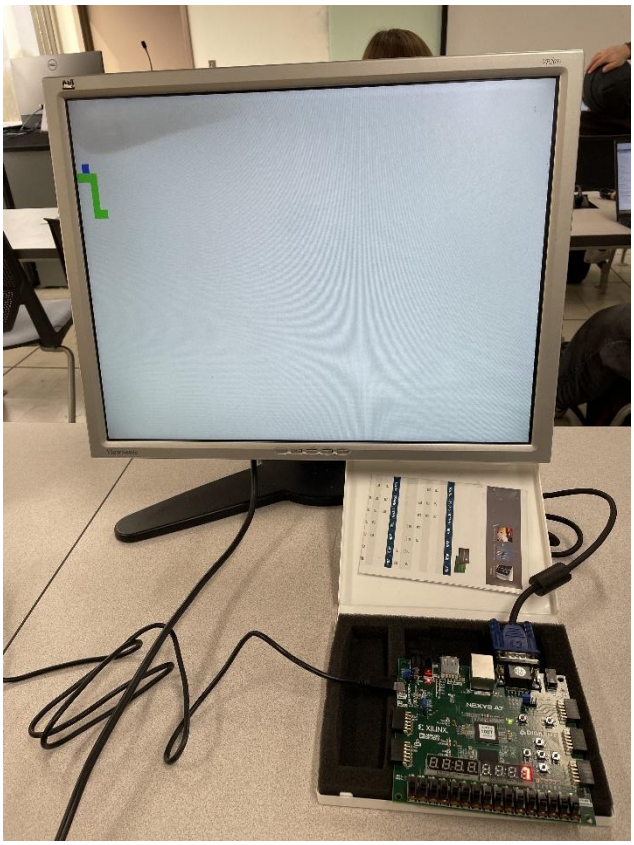

Figure 11. Working Snake Game to the VGA monitor
Note: the game has ended because we have collided with the wall!

V. Conclusion

In conclusion, our project operated relatively smoothly, with all our proposed modules eventually being implemented, other than the full implementation of our scoring module. The combination of the VGA controller, game logic, pseudo-random sequences, and pixel generation created a simple snake game for player enjoyment.

We were able to interface the Nexys A7 Board and its peripherals, as well as outputting the actual game to a VGA Monitor. The snake movements are quite smooth and easy for the player to control the snake. The food generation also creates a competitive element to the game, as the player needs to figure out how to navigate the snake to beat the game.

The creation of the game involved immense attention to detail, an open mind, and a strict adherence to the fundamentals of modular programming. Whether it was connecting the instantiated modules, or modifying the board constraint file, a good eye was essential. The debugging process took the most time in the overall implementation of the project, but that comes with any sort of engineering project.

With this final project, we were able to hone our coding skills, software and hardware debugging, as well as working with a team for an end goal.

Acknowledgments

This project utilized the assistance of Large Language Models (LLMs) as permitted by the professor during the semester of Fall 2024. They were used to assist

us in verifying the functionality of the code and minimizing glitches within the Verilog modules.

REFERENCES

[1] Chu, Pong P. “VGA Controller I: Graphic” FPGA Prototyping by Verilog Examples Xilinx Spartan-3 Version, J. Wiley & Sons, Hoboken, NJ, 2008, ch. 13, pp. 309-340.

[2] Brown, Arthur. “Nexys A7 Reference Manual.” Nexys A7. *Nexys A7 Reference Manual – Diligent Reference*, Diligent. nexys-a7_rm.pdf

[3] Instructables, “‘Snake’ on an FPGA,” *Instructables*, Nov. 30, 2015. https://www.instructables.com/Snake-on-an-FPGA-Verilog/

[4] Russell, “Linear Feedback Shift Register for FPGA,” *Nandland*, Jun. 09, 2022. https://nandland.com/lfsr-linear-feedback-shift-register/