Individual Report - Snake

ENSC 452 - Advanced Digital System Design

2023/04/11

Jacob Forrest - 301360304

Group 03

Table of Contents

1 Introduction

Our project aimed to recreate the classic arcade game 'Snake' on the ZedBoard hardware. In this game, the player guides a moving snake on a two-dimensional plane to consume fruit while avoiding walls and the snake's tail. Eating fruit grows the snake's tail, making the game more challenging. In our implementation the user inputs are directed to a terminal window on a host computer that connects to the UART port on the zedboard. Output to the user is facilitated through video via the onboard VGA port, and audio via the onboard 3.5mm aux port. Our implementation of the game also includes menu pages, and additional optional gameplay features. The menu pages provide an interface for the user to start gameplay, adjust game volume, view high scores, enable optional features, and customize the appearance of the game.

We utilized two ARM processor cores in the Zynq-7000 SoC, along with the programmable logic and I/O interfaces provided by the ZedBoard to create the game. To aid in development, we used software such as Vivado for configuring the Zynq SoC, Vitis for software development, and Matlab to create scripts to save game assets into C++ header files. Project development was broken down into weekly milestone tasks.

For a comprehensive documentation of the project, including instructions for interacting with the menu pages and gameplay controls, the group's collective work and contributions, readers are encouraged to review the group report.

2 Review of Technical Literature Used in the Project

2.1 The Zynq Book Tutorials: Adventures with IP Integrator

The ZynqBook “Adventure with IP Integrator“ Tutorial is a technical document produced by Digilent, the developer of the ZedBoard [1]. The tutorial provides code examples to guide users through the process of configuring and programming the ADAU1761 audio codec, which is integrated on the ZedBoard [1]. The tutorial's technical content is precise and focused on the use of the ADAU1761 codec [1]. The Zynq Book Tutorial provides guidance on designing and implementing an audio system and even provides code examples to demonstrate how to configure the codec registers [1]. I used this knowledge to implement a customized audio system on the ZedBoard.

2.2 Tutorial: Linear Feedback shift registers (LFSRs)

The EE Times tutorial on LFSRs is a reliable and trustworthy source for LFSR design. The article provides a clear explanation of LFSRs and insights into design and implementation [2]. I used this resource while designing a custom random number generator hardware block for the project project. For example, the article explains how the seed value and feedback taps should be carefully chosen to avoid repeating patterns and ensuring maximal sequence length [2].

2.3 AXI Reference Guide

The Vivado Design Suite AXI Reference Guide is a technical document produced by Xilinx, the developer of Vivado. As such, it is a highly authoritative source of information for designing AXI interfaces in Vivado.

The AXI interface is a standard interface for connecting IP blocks in a system-on-chip (SoC) design [3]. It provides a set of signals for data transfer and control between the master and slave components. For the RNG hardware block, I utilized an AXI slave interface to facilitate communication with the Zynq Processor. The reference guide provided information on implementing custom IP blocks using the Vivado IP integrator tool, which can generate HDL code for the IP block based on a graphical interface.

3 Project Implementation Details

3.1 Project Partitioning

The project was divided into several tasks and milestone deliverables, which were assigned to each team member based on our skills and interests. The following is a breakdown of the project task partitioning in order of the completion date:

Task / Feature Implementation	**Team Member**
Programming the software logic for the menu pages and gameplay loop.	Jacob
Creating an API for rendering sprites.	Chris
Building a pseudo random number generator hardware IP block for the fruit position randomizer mechanic.	Jacob
Building out the software logic for snake movement, fruit consumption, and collisions.	Chris
Implementing a graphical user interface (GUI) for the menu pages.	Jacob
Implementing background music streaming on the second ARM processor core.	Jacob
Implement the background music volume adjustment controls in the volume menu.	Jacob
Add menu and gameplay event-triggered sound effects to the audio playback on the second ARM processor core.	Jacob

Implement the software logic to track gameplay scores and display the top five scores in the high score menu.	Chris
Implement a feature that enables the ability for the fruit sprite to roam around the map during gameplay. Allow the user to enable or disable this feature in the options menu.	Jacob
Implement a hard mode feature that causes the snake to increase in speed as the gameplay progresses. Allow the user to enable or disable this feature in the options menu.	Chris
In the options menu, Add the ability customizing the colour of the snake and the type of food that spawns on the board in the options menu.	Chris
Implement a death animation for the snake that can be triggered during gameplay by colliding with a wall or a tail segment.	Chris

Table 1 - Project Partitioning

3.2 Individual Contributions

In this section, we will explore the various components of the Snake game that I was responsible for implementing. We will start with the top-level logic, which includes the menu pages and the main gameplay loop. Next, we will discuss the implementation of the Pseudo Random Number Generator (RNG) hardware block that generates random numbers for the game. Next, we will cover the Graphical User Interface (GUI) that I designed for the menu pages. Next, we will examine the audio components of the game such as the background music and the event-triggered sound effects in the game. Finally, we will discuss the roaming apple option, which enables the food sprite to randomly move around the grid during gameplay.

3.2.1 Top level logic - Menu pages & Main gameplay loop

My first milestone for the project was to design and implement the top level functionality of the project and implement stub functions for future features. Because the graphics had not been created at the time, I utilized the bi-directional communication of the UART to draw simple ASCII diagrams of the menu pages and gameplay loop. I based my implementation for this section off of the flow charts that I created in the original project proposal, though with some alterations.

To create flexible code that would allow for future expansion with additional menus I created a simple top level state machine program flow. The final list of states for the game includes: *main_menu*, *volume_menu*, *highscore menu*, *options_menu*, *gameplay*, *pre_gameover*, and *game_over*. For the menu interfaces, I implemented a cursor style navigation tool. By entering specific keys into the terminal connected to the Zedboard's UART port, the user could move the cursor up and down and select the highlighted option. Additionally, this navigation code could be easily implemented in future menu expansions. For example, the ‘Options’ menu was not originally planned for the project, however it was a very simple addition to the project because I could simply reuse the navigation code.

The main menu consisted of four options:

1. Adjust Volume: Enter the ‘Adjust Volume’ menu
2. View High Scores: Enter the ‘High Scores’ menu
3. Options: Enter the ‘Options’ menu
4. Play Game: Start the actual gameplay of Snake

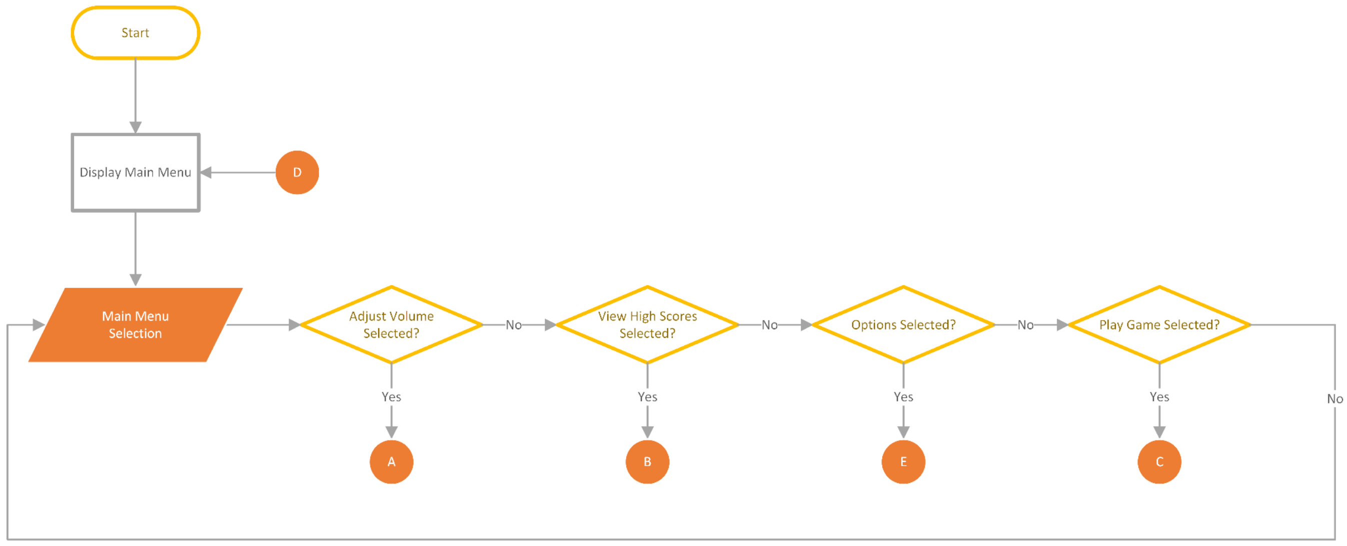


Figure 1: Algorithm Flowchart - Main Menu

Testing the main menu functionality involved manually verifying that the cursor control code was functional and that each selected option executed the proper action. For example, selecting the ‘Adjust Volume’ option would change the current state of the program to the *volume_menu*.

The ‘Adjust Volume’ menu consisted of three options:

1. Increase Volume:
   - If the volume is less than the maximum, increase the background music volume
   - Stub function
2. Decrease Volume:
   - If the volume is greater than zero, decrease the background music volume
   - Stub function
3. Return:
   - Enter the main menu

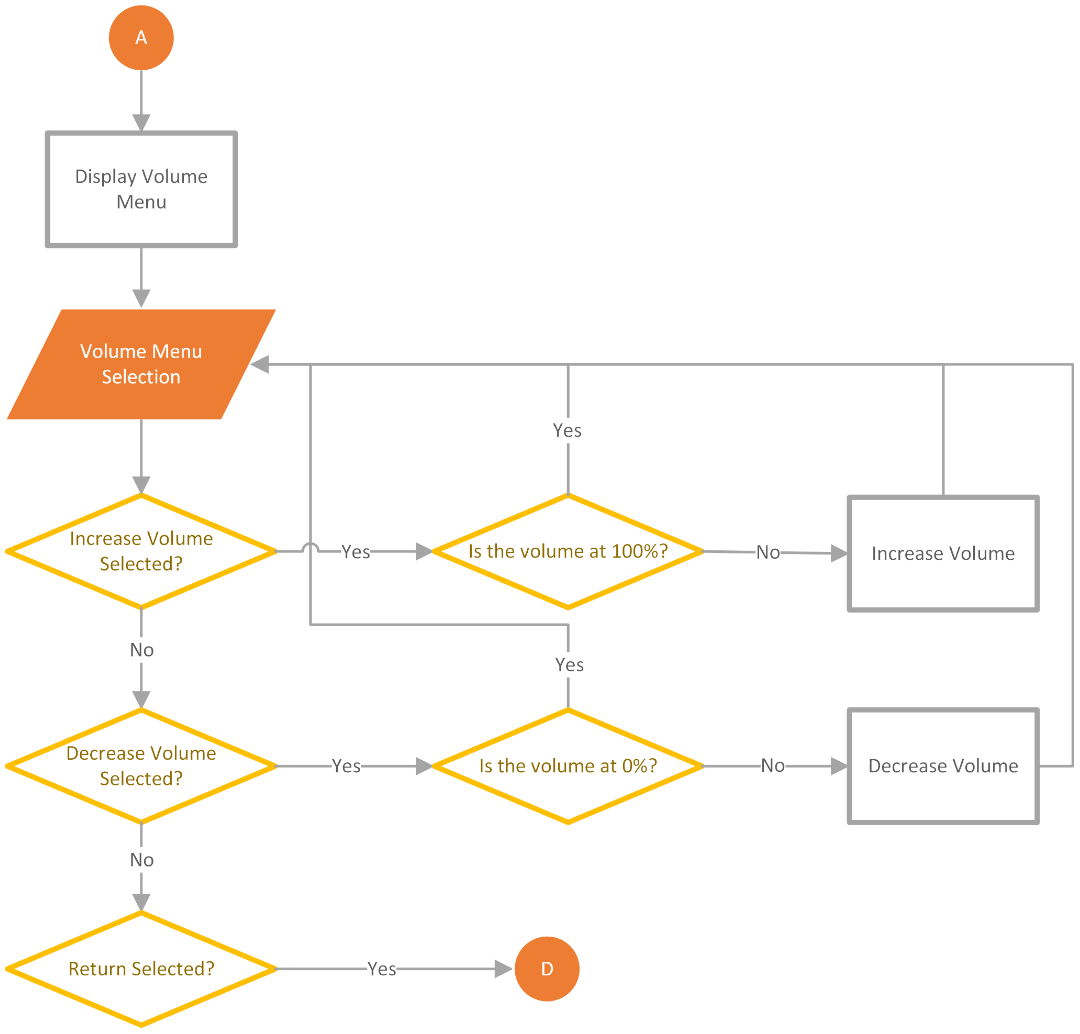


Figure 2: Algorithm Flowchart - Volume Menu

Testing the volume menu functionality again involved manually verifying that the cursor control code was functional and that each selected option executed the proper action. For example, selecting the 'Increase' option would increment the volume variable of the program. The conditional volume adjustment controls were also manually verified.

The 'High Score' menu consisted of a single option:

1. Return
   - Enter the main menu

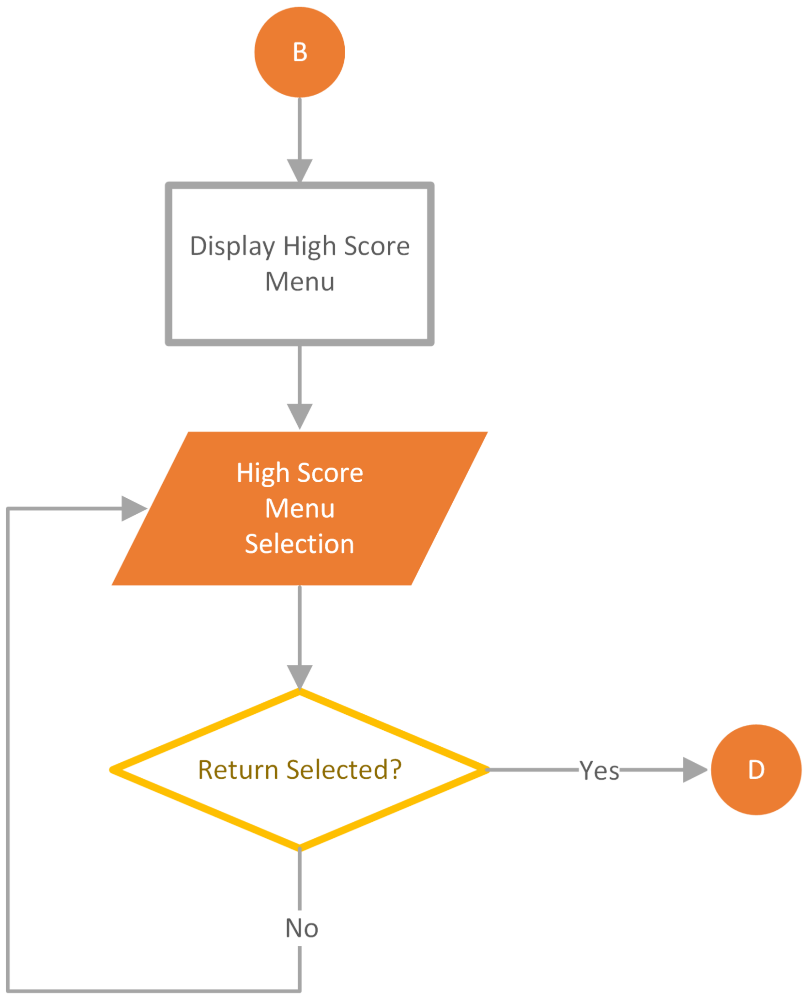


Figure 3: Algorithm Flowchart - High Score

Testing the ‘High Score’ menu functionality at this point in development was very simple because the high score saving feature was not implemented so the only thing to test was returning to the main menu.

The gameplay loop was fairly challenging to implement due to the need to move the snake at a constant rate while also allowing the user to execute an action at any time. To ensure that the snake moves at a reasonable pace, a timer has been implemented which temporarily pauses the loop while allowing the user to enter commands to adjust the direction of the snake or pause the game. Once the timer runs out, the snake is moved one unit forward. The program then checks for a collision with an apple; if detected, the snake tail grows by one unit. Finally, the program checks for a collision with a wall or tail segment, and if detected, the program transitions to the gameover menu.

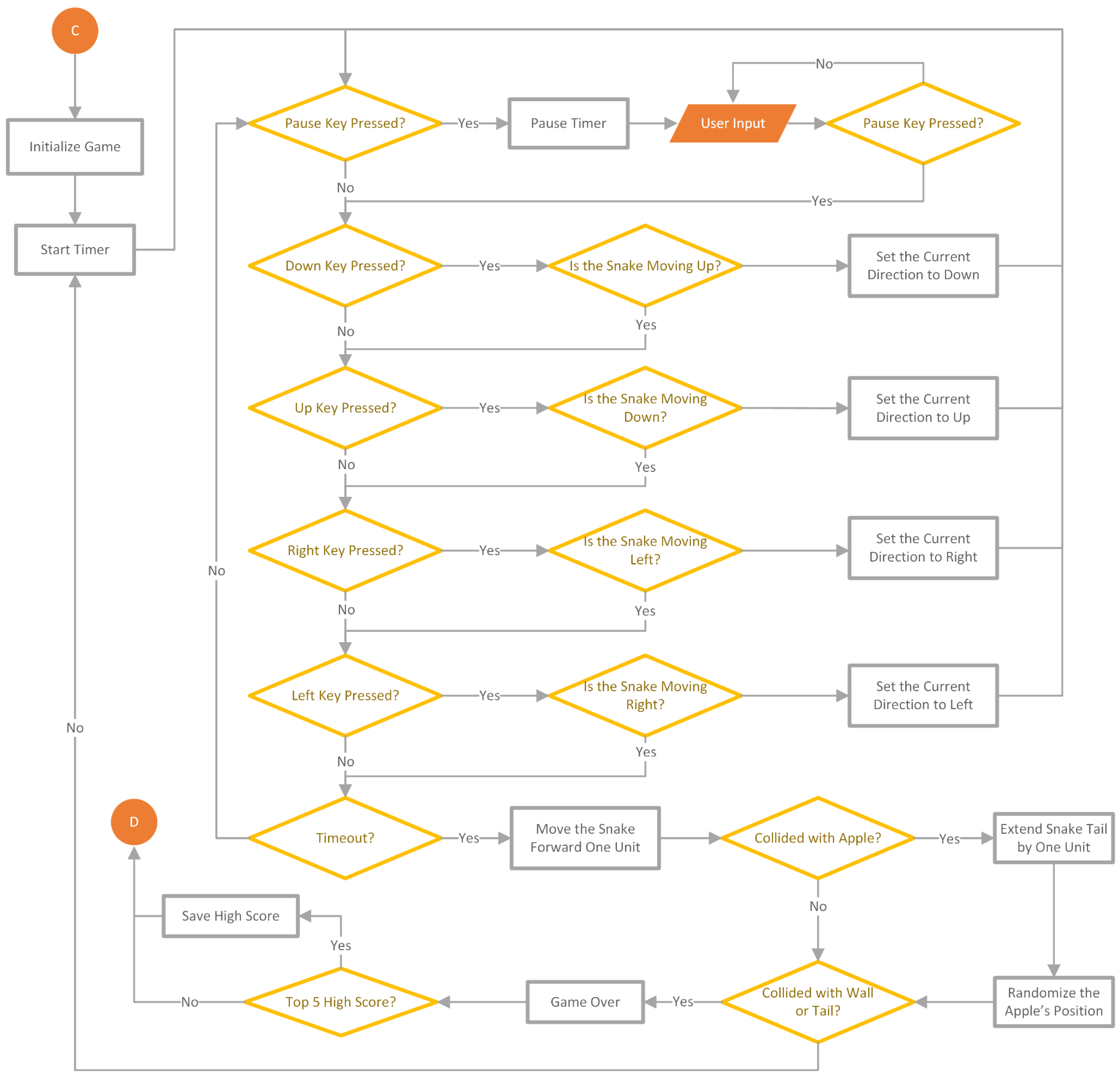
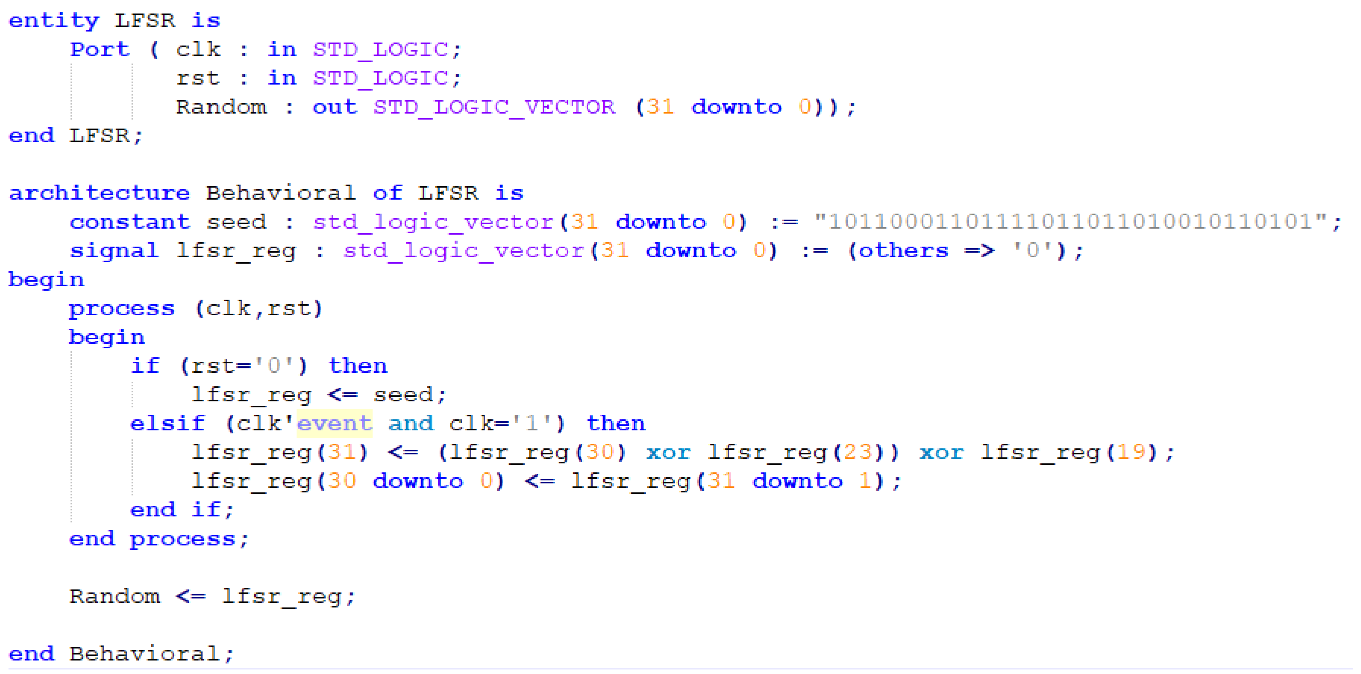

C
Initialize Game
Start Timer
Pause Key Pressed?
Yes
Pause Timer
User Input
Pause Key Pressed?
No
Yes
No
Down Key Pressed?
Is the Snake Moving Up?
Set the Current Direction to Down
Up Key Pressed?
Is the Snake Moving Down?
Set the Current Direction to Up
Right Key Pressed?
Is the Snake Moving Left?
Set the Current Direction to Right
Left Key Pressed?
Is the Snake Moving Right?
Set the Current Direction to Left
Timeout?
Move the Snake Forward One Unit
Collided with Apple?
Extend Snake Tail by One Unit
Randomize the Apple's Position
Collided with Wall or Tail?
Game Over
Top 5 High Score?
Save High Score
D

Figure 4: Algorithm Flowchart - Default Gameplay

3.2.2 Pseudo Random Number Generator Hardware

The goal of this milestone was to create a hardware implementation of a Random Number Generator. I implemented the Pseudo Random Number Generator hardware block using a 32-bit Linear Feedback Shift Register (LFSR). The LFSR is a commonly used technique for generating random numbers in digital systems [2]. The feedback tap configuration used in this implementation is 19, 23, 30. The design is initialized with a seed value of '10110001101111011011010010110101'. This implementation of an LFSR ensures that the generated sequence is highly unpredictable and can be used in our project to generate random numbers. One of the use cases for the RNG block is to generate random spawn positions for the fruit sprites.



```vhdl
entity LFSR is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           Random : out STD_LOGIC_VECTOR (31 downto 0));
end LFSR;

architecture Behavioral of LFSR is
    constant seed : std_logic_vector(31 downto 0) := "10110001101111011011010010110101";
    signal lfsr_reg : std_logic_vector(31 downto 0) := (others => '0');
begin
    process (clk,rst)
    begin
        if (rst='0') then
            lfsr_reg <= seed;
        elsif (clk'event and clk='1') then
            lfsr_reg(31) <= (lfsr_reg(30) xor lfsr_reg(23)) xor lfsr_reg(19);
            lfsr_reg(30 downto 0) <= lfsr_reg(31 downto 1);
        end if;
    end process;

    Random <= lfsr_reg;

end Behavioral;
```

Figure 5: VHDL Code for the LFSR Hardware Block

I used the Vivado “Create and Package IP” tool provided in Vivado to assist in implementing the LFSR hardware design into an AXI4 peripheral with a slave interface. To add the RNG block into our project’s hardware design, I used the AXI slave interface and an AXI SmartConnect IP block to facilitate communication with the Zynq Processor.

To test that the RNG block was functioning properly I wrote a simple C++ code snippet to collect the random numbers and then compared them to the expected output from the LFSR.

3.2.3 Graphical User Interface (GUI) - Menu Pages

The first step in creating the Graphic User Interface for the menu pages was to create text sprites for all of the available options. Using Paint.net, a freeware raster graphics editor program for Microsoft Windows, I created a transparent bitmap file of each menu option in Calibri font. Then, I used a Matlab script provided by Chris Rosenauer to translate the bitmap image file into a two dimensional array of raw hex data that then could be included in a header file. Using the menu text sprites, I used the sprite rendering API provided by Chris Rosenauer to write each sprite in the appropriate position to the frame buffer. The final step in creating the GUI for the menu pages was to simply incorporate the graphics rendering function calls into the top level program loop.

The following figures show the final graphical user interface for each of the menu pages:

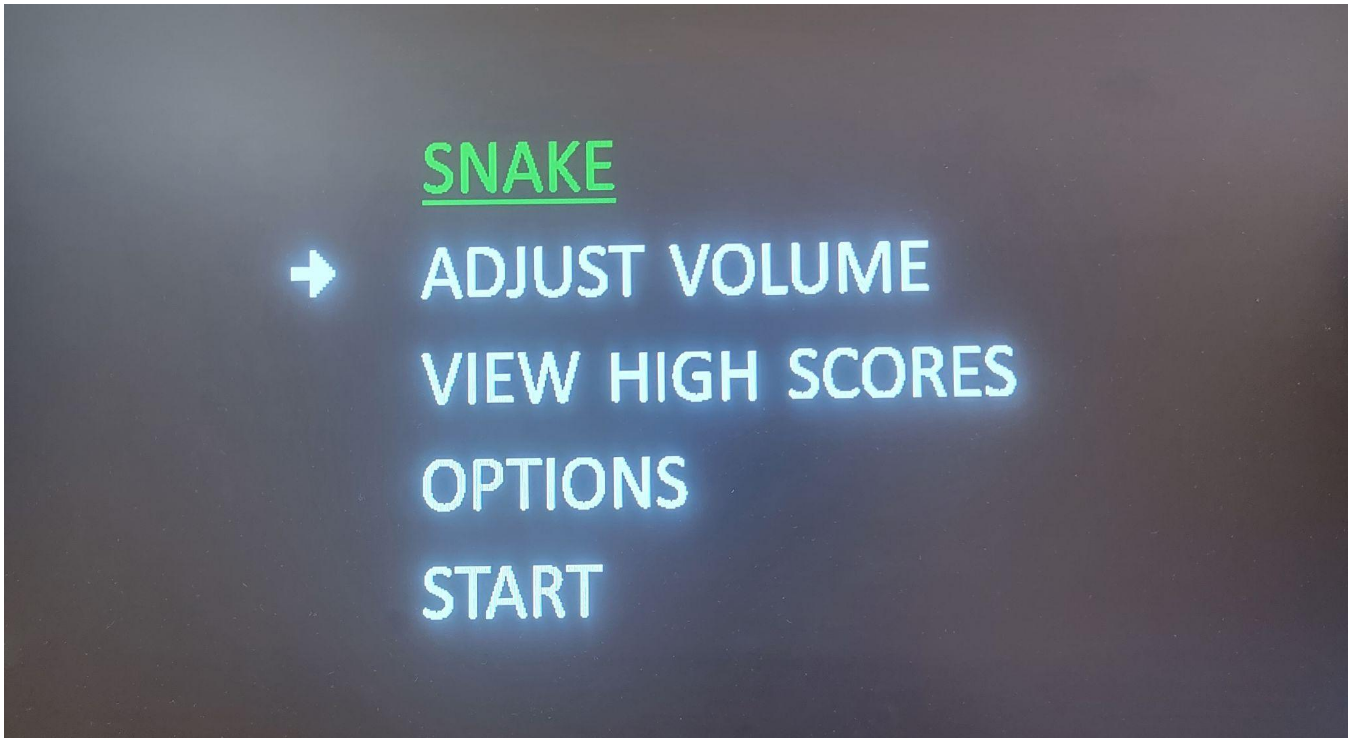


Figure 6: Main Menu GUI

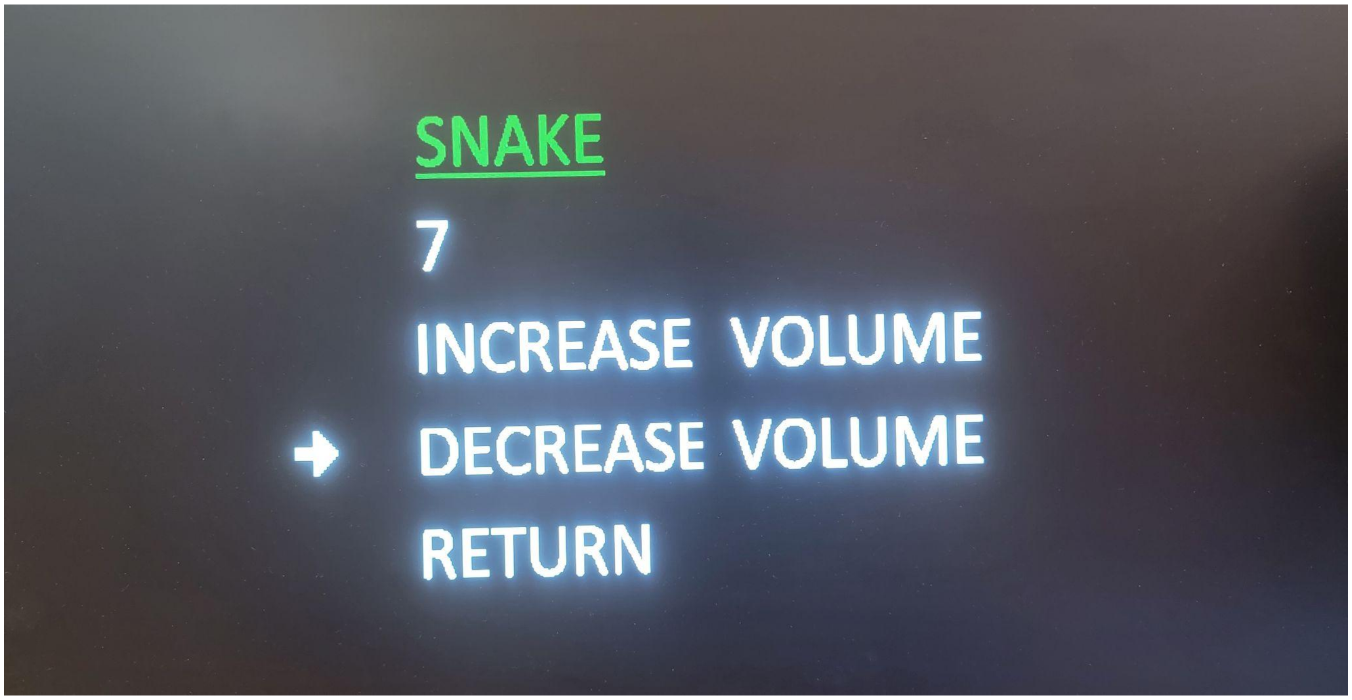
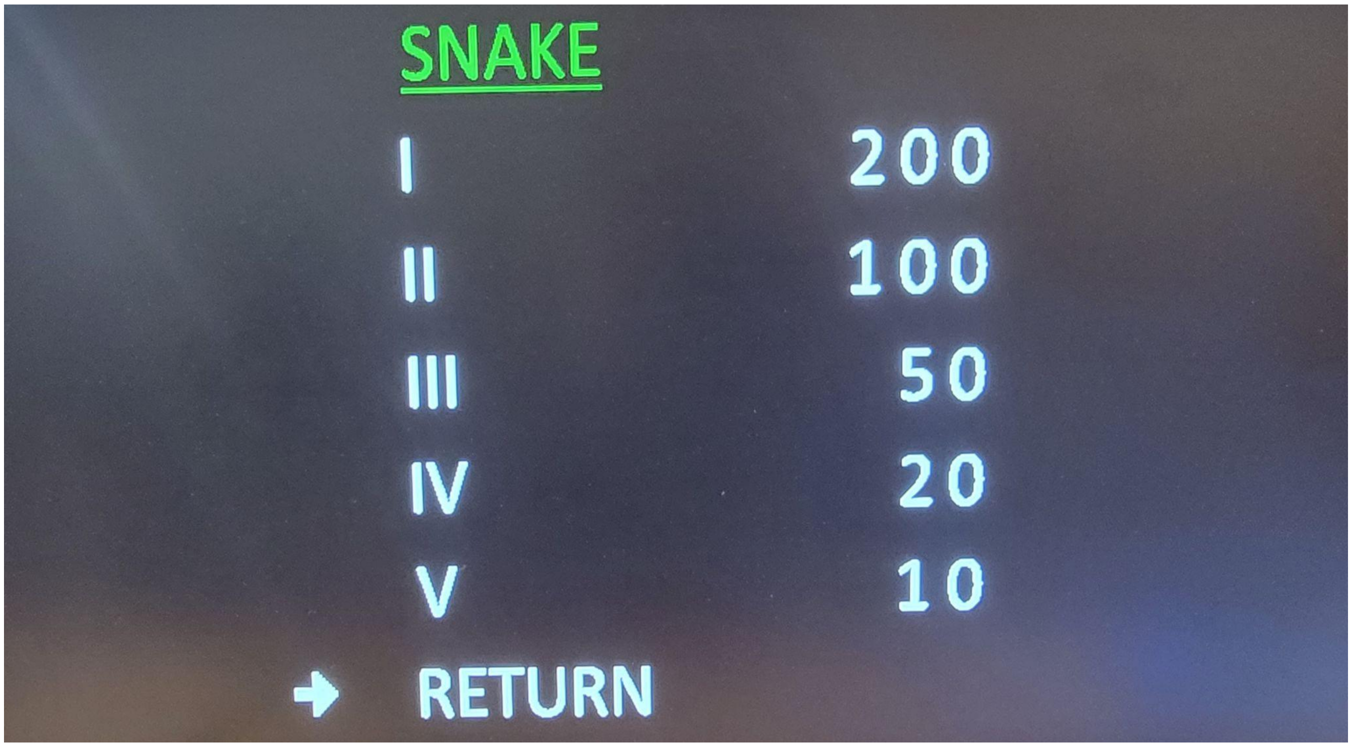
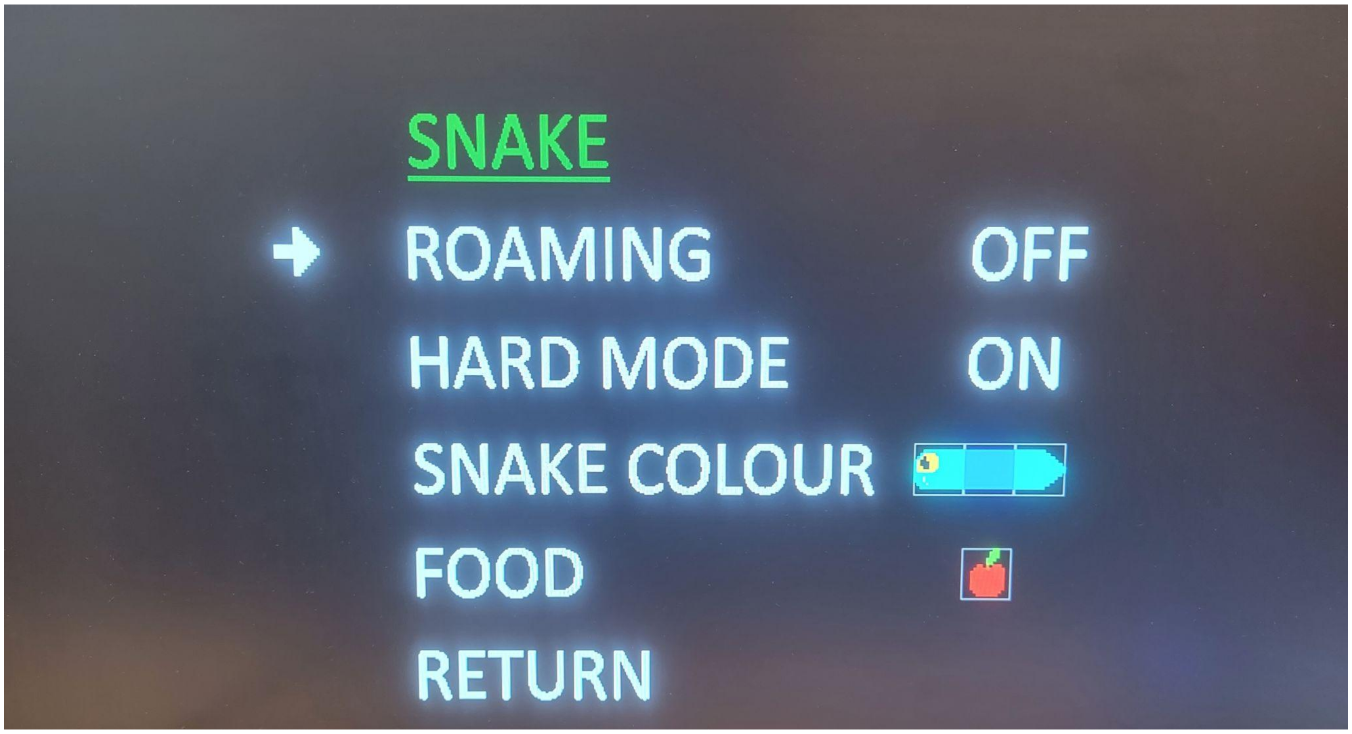


Figure 7: Volume Menu GUI


SNAKE

I	200
II	100
III	50
IV	20
V	10

Figure 8: High Score Menu GUI


SNAKE

➜ ROAMING	OFF
HARD MODE	ON
SNAKE COLOUR	
FOOD	

Figure 9: Options Menu GUI

3.2.4 Background Music & Volume Adjustment

Implementing the background music required making changes to the hardware Vivado project. First, I added the zed_audio_ctrl IP block to the hardware design and connected the AXI slave interface to the ZYNQ7 Process system via the AXI SmartConnect IP block. For the ZYNQ7 Processing System, I enabled an I2C pin and a 10 MHz PL Fabric Clock to interface with the Zedboard's Audio Codec. I also added a 2-bit single-channel GPIO to connect to the Audio Codec's I2C ADDR pins. Finally I copied the constraints file provided with the 'Adventures with IP Integrator' tutorial to assign the signals to the correct pins on the Zedboard.

To store audio files into the Zedboard's memory, I wrote a matlab script to translate an mp3 format file into two one dimensional integer arrays for left and right channel audio so that they could be easily incorporated into a C++ header file.

Implementing the continuous playback of the background audio required creating a second application to run in parallel with the gameplay application. Therefore, for the Xilinx Vitis project, I created a second software application and configured it to run on the Zynq-7000 SoC's second ARM processor core. In the new software application, I used the code provided with the 'Adventures with IP Integrator' tutorial to configure the Audio Codec's PLL, the I2C data structure, and the Zedboard's Line in and Line out ports, namely, the *IicConfig*, *AudioPllConfig*, and *AudioConfigureJacks* functions [2]. I then wrote a simple loop program that continuously playback the background audio at ~48 kHz by writing each sample to the audio controller and then waiting for 21 microseconds.

Finally, I instantiated an atomic integer variable pointer to the memory location *0xFFFF0000* so that the background audio application and the main gameplay application could communicate the value of the background volume. Additionally, I filled in the volume adjustment stub functions in the 'Adjust Volume' menu.

3.2.5 Event-triggered Sound Effects

The goal for this milestone was to implement sound effects for specific in-game events such as consuming an apple, crashing into a wall or tail segment, and selecting a menu option. The first step I took in implementing the event triggered sound effects was to utilize the Matlab script that I created in the previous milestone to add the sound effects to the C++ audio header file. I then instantiated three atomic integer variable pointers to specific memory addresses to serve as a simple flag communication protocol between cores. For example, if the user selected an option in the menu, then the program running on the main core would load the value '1' into the *MENU_VAL* memory address which could then be read and reset by the program running on the audio core.

During this milestone, I refined the design by iterating through multiple implementations of the playback code for event-triggered sound effects. In the first design iteration, if one of the sound effect flags was set, then the background audio would be replaced with the sound effect until it was finished playing. This implementation was unappealing due to the abrupt audio source changes and the fact that the entire sound effect would have to be completed before another event could trigger a new sound effect.

The second design iteration improved the experience of the audio by outputting a sum of the background audio sample and the sound effect audio sample. However, this implementation still did not resolve the issue that sound effect events could not be processed while one of the sound effects was still playing.

The final design iteration used a queue to allow for overlapping sound effect playback. In this design, the audio program checks the sound effect flags on loop iteration. If any of the flags were set, then the sound effect would be added to the queue. The output audio sample of each iteration of the main loop is a sum of the background audio sample, and any sound effect audio sample that is queued. When all of the audio samples from a queued sound effect has been output to the audio controller, the sound effect is removed from the queue.

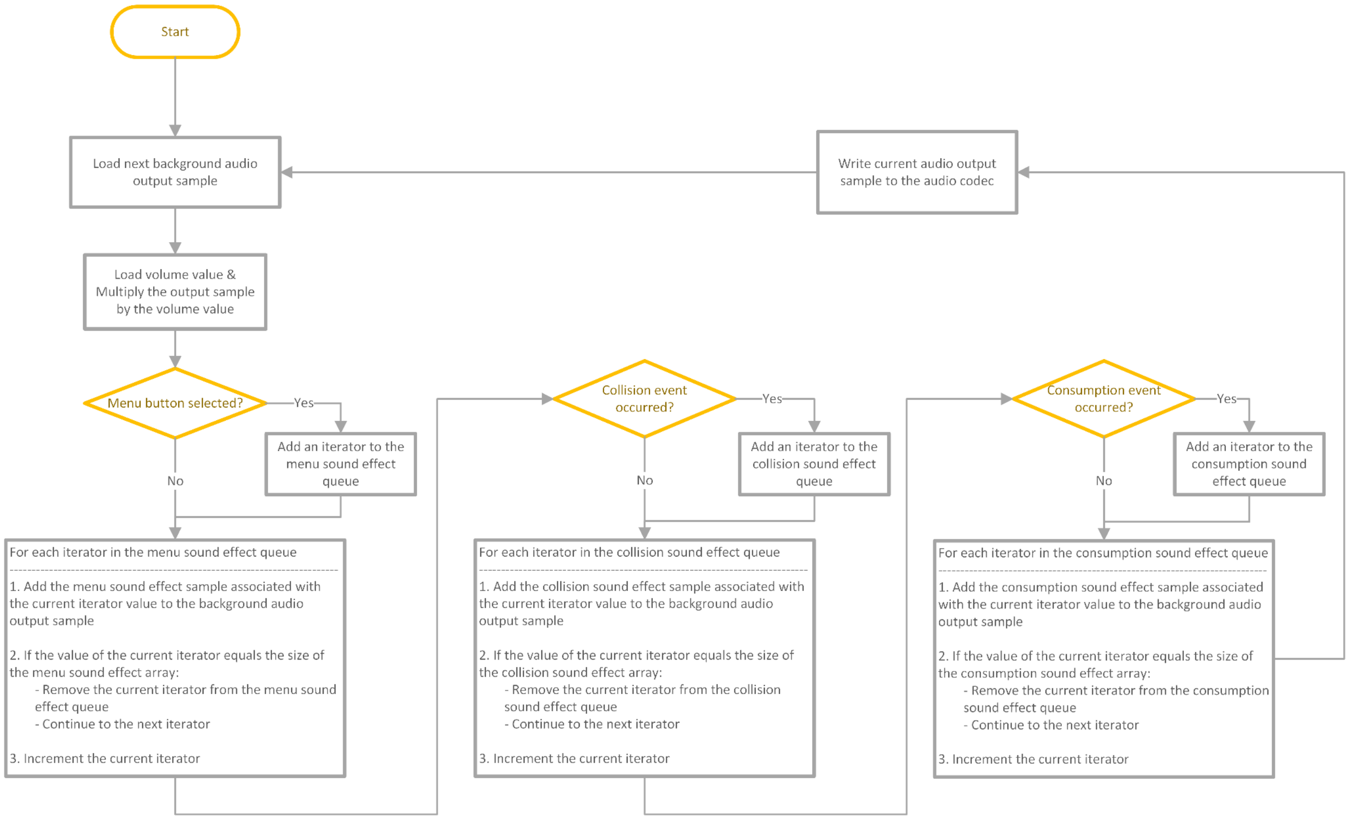

Start
Load next background audio output sample
Write current audio output sample to the audio codec
Load volume value & Multiply the output sample by the volume value
Menu button selected?
Yes
No
Add an iterator to the menu sound effect queue
For each iterator in the menu sound effect queue
1. Add the menu sound effect sample associated with the current iterator value to the background audio output sample
2. If the value of the current iterator equals the size of the menu sound effect array:
- Remove the current iterator from the menu sound effect queue
- Continue to the next iterator
3. Increment the current iterator
Collision event occurred?
Yes
No
Add an iterator to the collision sound effect queue
For each iterator in the collision sound effect queue
1. Add the collision sound effect sample associated with the current iterator value to the background audio output sample
2. If the value of the current iterator equals the size of the collision sound effect array:
- Remove the current iterator from the collision sound effect queue
- Continue to the next iterator
3. Increment the current iterator
Consumption event occurred?
Yes
No
Add an iterator to the consumption sound effect queue
For each iterator in the consumption sound effect queue
1. Add the consumption sound effect sample associated with the current iterator value to the background audio output sample
2. If the value of the current iterator equals the size of the consumption sound effect array:
- Remove the current iterator from the consumption sound effect queue
- Continue to the next iterator
3. Increment the current iterator

Figure 10: Algorithm Flowchart - Audio Program

3.2.6 Roaming Apple Option

The goal of this milestone was to add an optional feature that would increase the difficulty of the game by making the fruit sprite move around the grid during gameplay. Because this feature was originally a bonus milestone, I needed to create a new menu page where the user could enable and disable the roaming feature. I implemented the navigation and GUI for the new 'Options' menu using the steps described in sections *(3.2.1)* and (*3.2.3*). The following flow chart shows the final implementation of the 'Options' menu with additional bonus features:

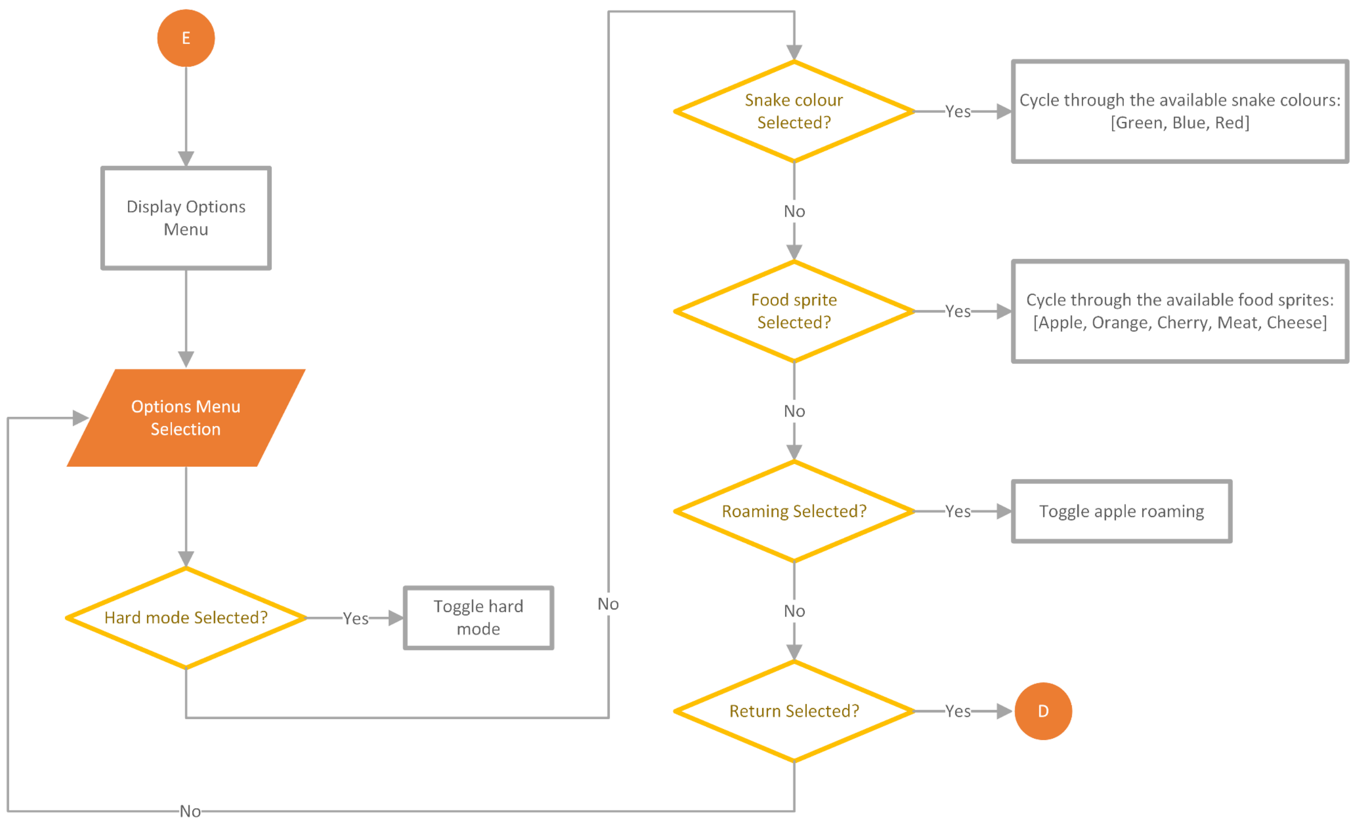


Figure 11: Algorithm Flowchart - Options Menu

For the implementation of the roaming apples feature, I utilized the RNG hardware block to randomize the direction of the roaming apple once every seven movements. Additionally, I configured the main gameplay loop to move the apple at half the speed of the snake. Through my test playthroughs, this combination of predictable movement and slower speed provides the most enjoyable experience.

To avoid the apple colliding through the wall or the snake's tail segments, there are conditional alternative movements available for the apple. The following flowchart showcases the implemented moveApple algorithm and the edge cases that were considered and tested:

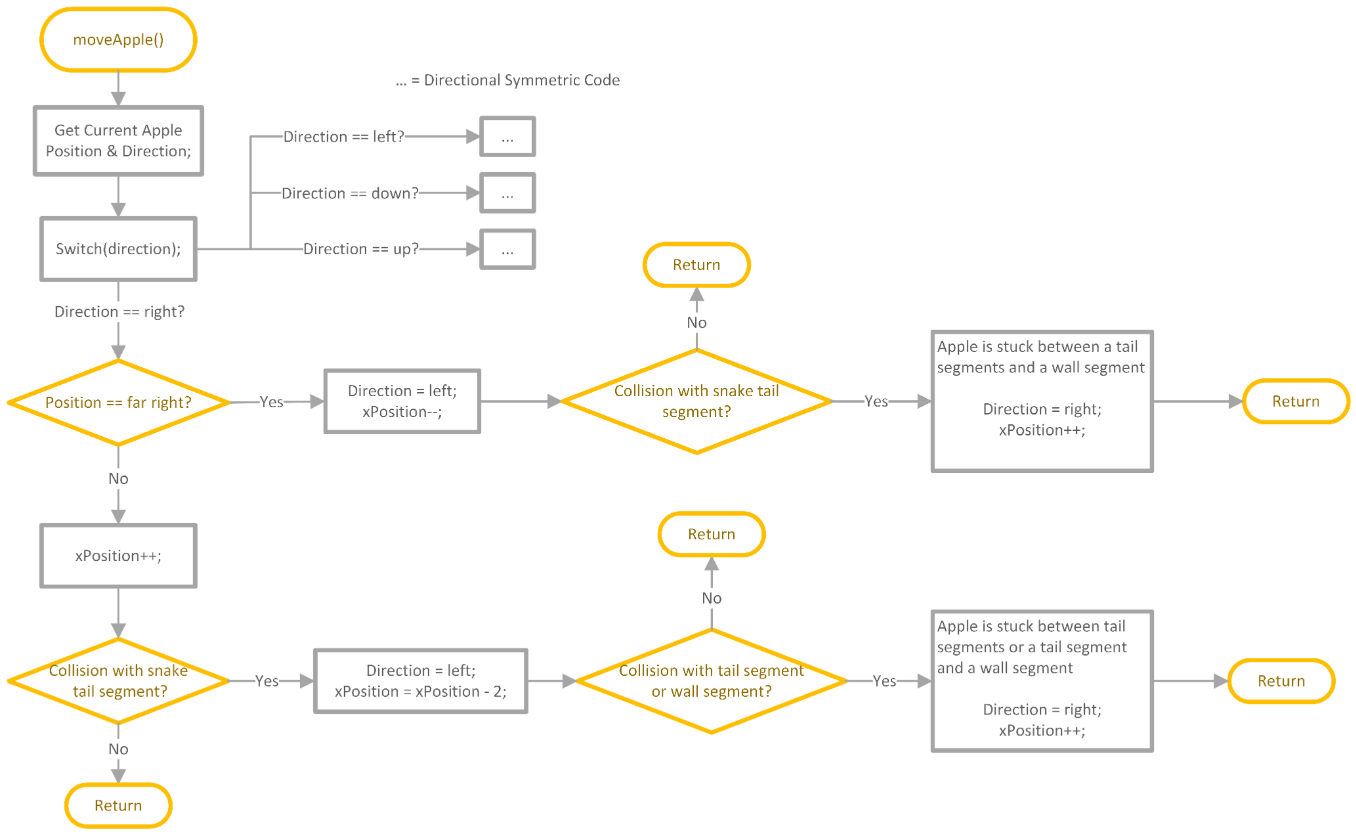

moveApple()
... = Directional Symmetric Code
Get Current Apple Position & Direction;
Switch(direction);
Direction == left?
Direction == down?
Direction == up?
...
Direction == right?
Position == far right?
Yes
No
Direction = left; xPosition--;
Collision with snake tail segment?
Return
Apple is stuck between a tail segments and a wall segment
Direction = right; xPosition++;
xPosition++;
Collision with snake tail segment?
Direction = left; xPosition = xPosition - 2;
Collision with tail segment or wall segment?
Apple is stuck between tail segments or a tail segment and a wall segment
Direction = right; xPosition++;
Return

Figure 12: Algorithm Flowchart - Roaming Apple (Direction == right)

3.3 Collaboration with Team Members

For source control, we initially used timestamped project folders for managing the project development, which worked well when the project was small. However, as the project complexity increased, we recognized the need for a more robust solution and switched to git for source control. The use of git allowed us to add, merge, and revert changes very easily which greatly improved our collaboration efficiently.

Creating the top level logic for the main program was very helpful for simplifying collaboration efforts because it allowed us to develop and integrate individual code modules throughout development. I suspect that attempting to merge all of our individual code modules near the end of the project would have been substantially more difficult and less successful. Additionally, I designed the top level program to follow a finite state machine program flow loop so that it would be very simple for Chris to add new states to the project.

During the portions of my milestones that involved displaying sprites on screen, I leveraged the graphics rendering API that Chris developed.

3.4 Tools

In our project, various tools were used including Vivado's "Create and Package IP" feature, the Vitis development environment, and Matlab.

The "Create and Package IP" tool in Vivado is a feature that allows users to create IP blocks with different interfaces such as AXI4, AXI4-Lite, AXI4-Stream, AHB, etc…. from user-created custom IP designs. It uses a graphical user interface (GUI) to provide an intuitive workflow for users to create, modify, and package IP blocks. In our project, I used the "Create and Package IP" tool to implement the LFSR (Linear Feedback Shift Register) hardware design into an AXI4 peripheral with a slave interface. The AXI4 protocol is a widely used standard interface for connecting processors and peripheral devices in FPGA design. The slave interface is a type of interface that responds to read and write actions initiated by a master device. By packaging the LFSR with an AXI interface I was able to easily integrate it into the overall hardware design.

I also learned about using the Vitis development environment. Specifically, I learned how to create platform projects for exported hardware designs in the Xilinx Shell Archive (.xsa) format and how to set up multiple application projects to run on different cores provided by a platform project. In our project, we created two software applications to run on both of the hardware platform’s ARM processor cores. To ensure that there were no issues with memory addressing, we had to modify each application’s linker script.

I also utilized Matlab to write a script to parse mp3 audio files into C compatible integer arrays so that I could store the audio data in header files.

3.5 Ensuring Success & Lessons Learned

One important design decision that helped us achieve success in the project was that we employed a mix of top-down and bottom-up methodologies. I started implementing a project with a top level implementation of the main program logic complete with stub functions which helped reduce time spent on code integration.

Another design decision that helped achieve success was the use of modular reusable code. For example, I implemented the top level design of the project as a finite state machine which made it very easy to add new states to the project such as new menu pages. Another example of reusable code that I created was the menu navigation system; I created a cursor style navigation system that was easy to replicate on all menu pages.

Another important task that helped ensure success was the creation of algorithm flow charts. Flow charts were designed prior to implementation to provide a visual representation of the system. The visualization of all of the potential program flow paths helped me identify corner cases that were necessary to address. Overall, flowcharts helped facilitate efficient implementation of the software modules, and ensure sufficient testing coverage.

4 Community Contribution

As a community contribution, I provided assistance on the discussion board Piazza (https://piazza.com/class/lcgz3yp4sti4ta/post/103) by sharing a fix for an issue that many students ran into when integrating audio into their project and documented the steps to resolve the issue. After adding the zed_audio_ctrl IP block to their hardware project, Vivado did up automatically update the HDL wrapper file to include the I2C ports which would cause the software programs to freeze during the call to *AudioPllConfig()*.

5 Feedback to Xilinx

One inconvenience that I found with the Xilinx tools were the limited export and project sharing options. I found it to be very difficult to export and share the projects between different PCs or with other team members. I would like to see more intuitive and seamless ways of exporting and importing projects. It would also be great to have a direct integration with Git repositories.

Another issue I ran into with the Xilinx tools was a bug that affected the auto generated makefiles for the Custom IP blocks. The syntax in the makefiles was incorrect and required manually fixing each time the hardware project was updated in order to properly compile the software application.

6 Course Feedback

I found the project timeline to be generally well-planned and executed. Having at least a week between each milestone demo provided enough time for development and testing. I also found that the open lab concept worked well for me, as I appreciated the flexibility to work at my own pace and it helped me to stay on track with the weekly milestone modules.

In terms of course credit hours, I felt that four credit hours was appropriate given that the amount of work required was manageable.

Overall, I thought the lectures were engaging and informative, although I found that the pace of the lectures to be quite fast at times, and would have appreciated more time for hands-on activities and problem-solving exercises.

Overall, I enjoyed the course and felt that it provided a good introduction to FPGA design. I appreciate the opportunity to work on a real-world project and gain hands-on experience with the tools and technologies used in industry.

7 References

[1] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, N. David, and R. W. Stewart, “Adventures with IP Integrator,” in *The ZYNQ book tutorials: For Zybo and zedboard*, Glasgow: Strathclyde Academic Media, 2015, pp. 119–150.

[2] M. Maxfield, “Tutorial: Linear Feedback shift registers (LFSRS) - part 1,” *EE Times*, 07-Sep-2022. [Online]. Available: https://www.eetimes.com/tutorial-linear-feedback-shift-registers-lfsrs-part-1/. [Accessed: 09-Apr-2023].

[3] Xilinx, “AXI Reference Guide,” *Vivado Design Suite*, 2017. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide. [Accessed: 09-Apr-2023].