Group Report - Snake

ENSC 452 - Advanced Digital System Design

2023/04/12

Group 03:
Jacob Forrest - 301360304
Chris Rosenauer - 301400868

Table of Contents

Table of Figures

1.0 Introduction

The overall goal of our project was to recreate the retro arcade game Snake on the hardware provided by the ZedBoard. The objective of Snake is to guide the serpent through a gridded play area, consuming as much food as possible while avoiding collision with the wall or the serpent's own body.

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.

To implement this project, we utilized two ARM processor cores in the Zynq-7000 SoC, the programmable logic on the Zynq-7000 SoC, and the I/O interfaces provided by the ZedBoard. To aid in the development of hardware and software required for the project, we made use of software such as Vivado, Vitis, and Matlab.

Development of the project was broken down into weekly milestone tasks. The following list provides a high-level description of the project sub-goals that were completed:

- Programming the software logic for the menu pages and main game loop on one of the ARM processor cores.
- Creating an API for rendering sprites.
- Building a pseudo random number generator hardware IP block for the fruit position randomizer mechanic.
- Building out the software logic for snake movement, fruit consumption, and collisions.
- Implementing a graphical user interface (GUI) for the menu pages.
- Implementing background music streaming on the second ARM processor core.
- Add menu and gameplay event-triggered sound effects to the audio playback on the second ARM processor core.
- Implement the background music volume adjustment controls in the volume menu.
- Implement the software logic to track gameplay scores and display the top five scores in the high score menu.
- 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.
- 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.
- In the options menu, Add the ability customizing the color of the snake and the type of food that spawns on the board in the options menu.
- Implement a death animation for the snake that can be triggered during gameplay by colliding with a wall or a tail segment.

2.0 Background

The origins of Snake began in 1976 with the release of ‘Blockade’ [1]. Throughout the decades, Snake-like games eventually evolved into the game we know today. Due to the games simple nature and fun game play loop, Snake has seen various ports to a plethora of hardware including the Nokia 6110 [1].

In Snake you control the serpent around a gridded play area, attempting to consume as much food as possible without colliding with the snake’s body or the play area’s bounding walls. The serpent’s movements are limited to only the cardinal directions, and the serpent can only change its direction on the play area’s grid tiles. After the serpent consumes a piece of food the snake grows longer, and a new piece of food is randomly positioned on the play area. The longer you play the harder the game gets, as the length of the serpent reduces the available area to maneuver the serpent.

Our implementation is created in C++, using a combination of provided and custom hardware blocks. The game receives keyboard user input via the ZedBoard’s UART terminal, outputs video display via the ZedBoard’s VGA port, and outputs audio via the ZedBoard’s AUX port. The game is implemented to use two cores: a primary core which performs gameplay logic and rendering, and a secondary core which outputs audio. The game necessitates randomness from the food’s random positioning. Pseudo-random number generation is facilitated through a custom LFSR pseudo-random number generator hardware block.

3.0 System Overview

3.1 Block Diagram

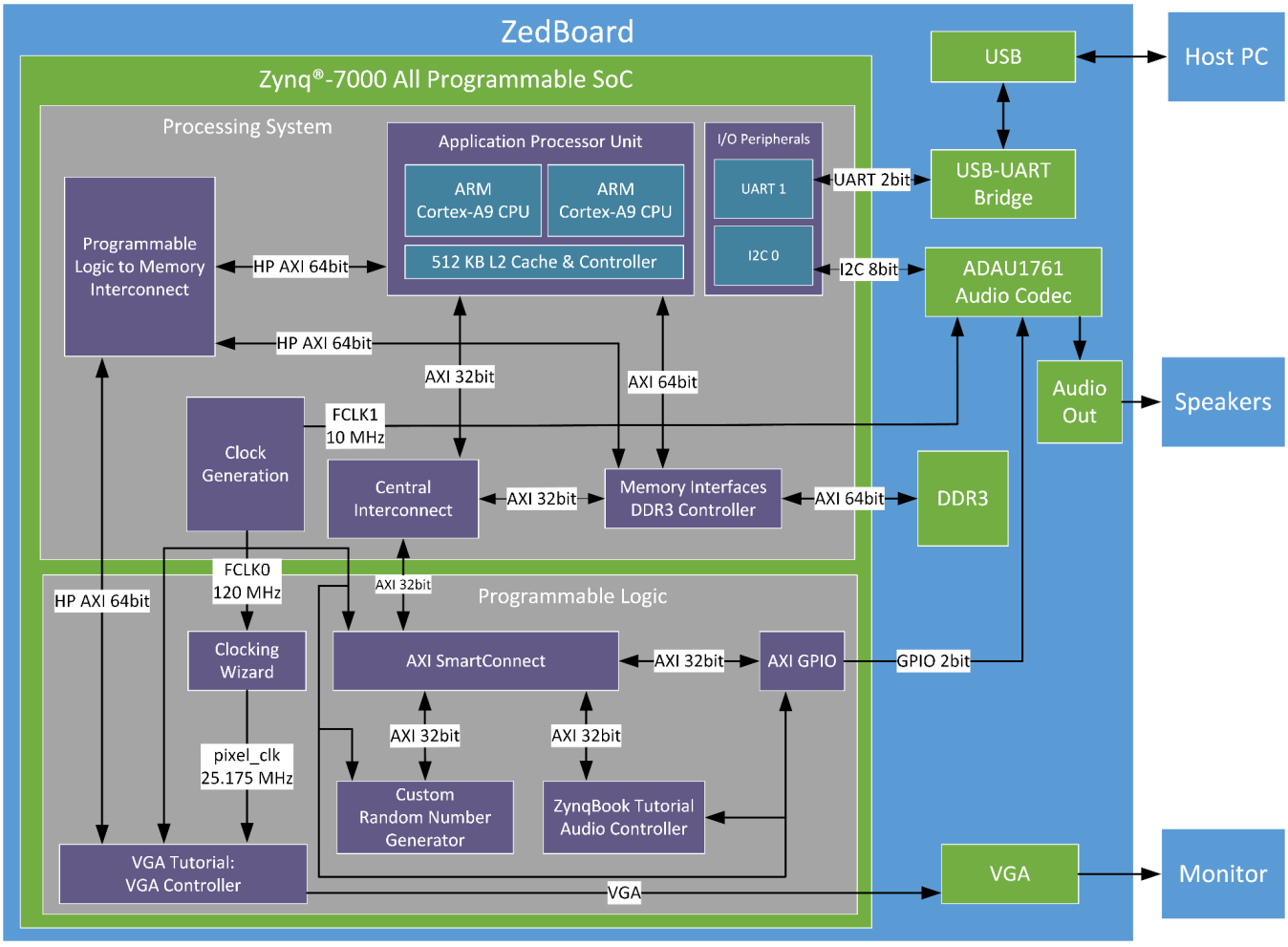


Figure 1: Zedboard System Block Diagram

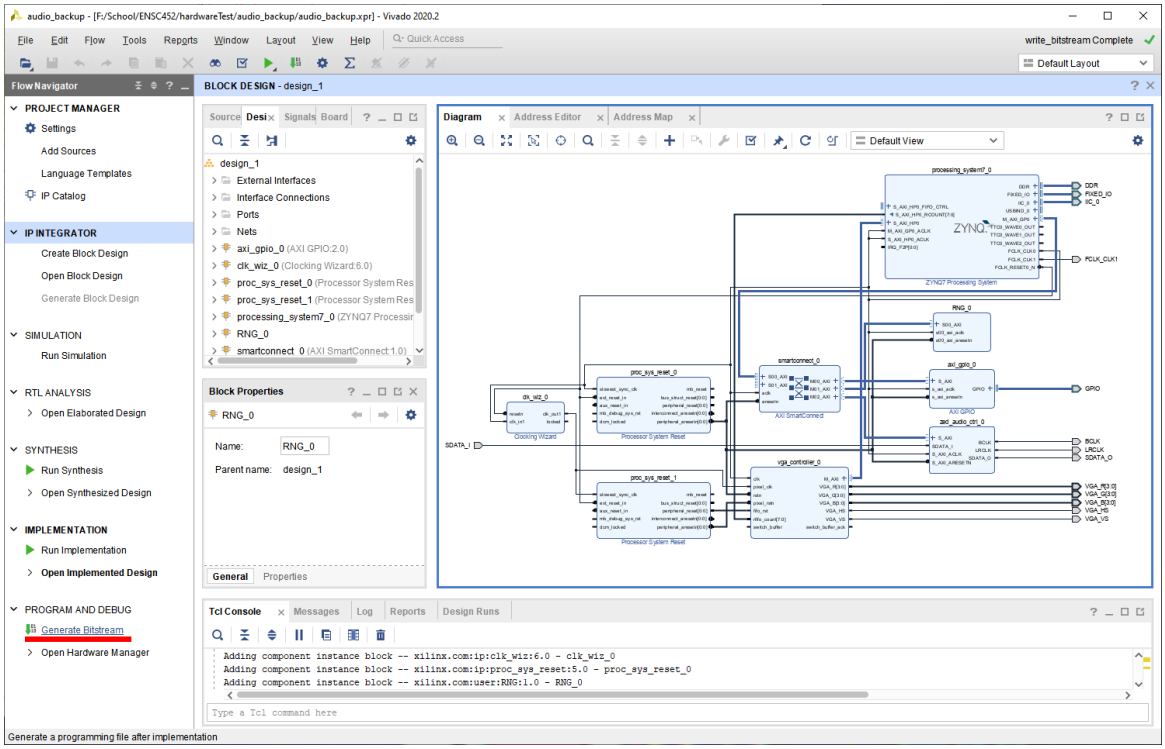
3.2 User Manual

3.2.1 Setting Up the Project

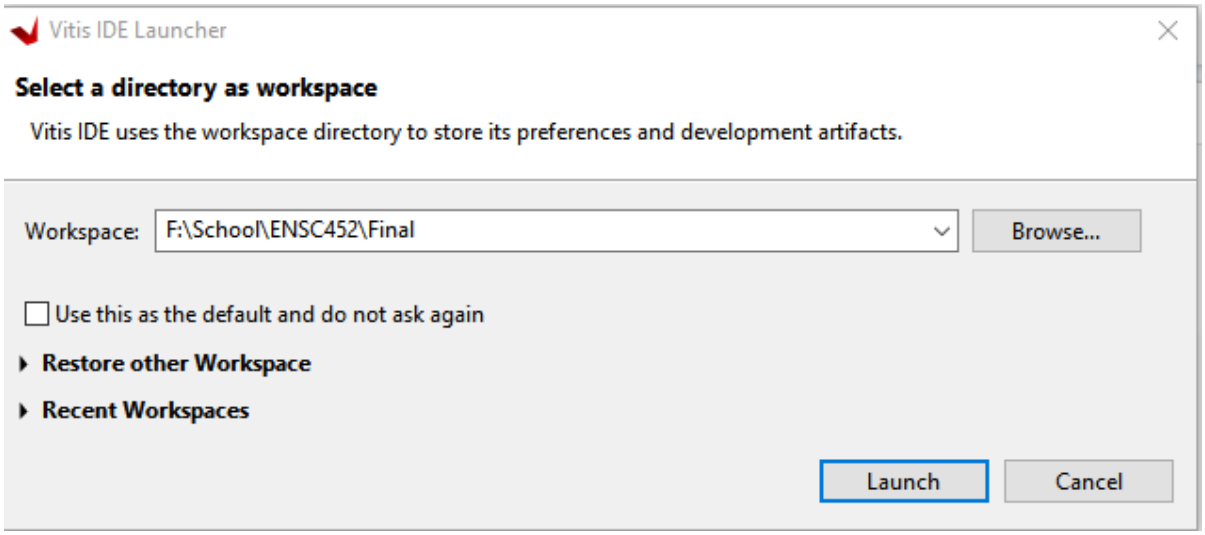
Note: Steps 1 and 2 are optional. Instead of generating the hardware's bitstream one may use the provided .xsa file, *audio_backup.xsa*.

1.) Open the hardware project *audio_backup.xpr* in the *VivadoProject/audio_backup* subdirectory.

2.) Select *generate bitstream* and select *OK.*



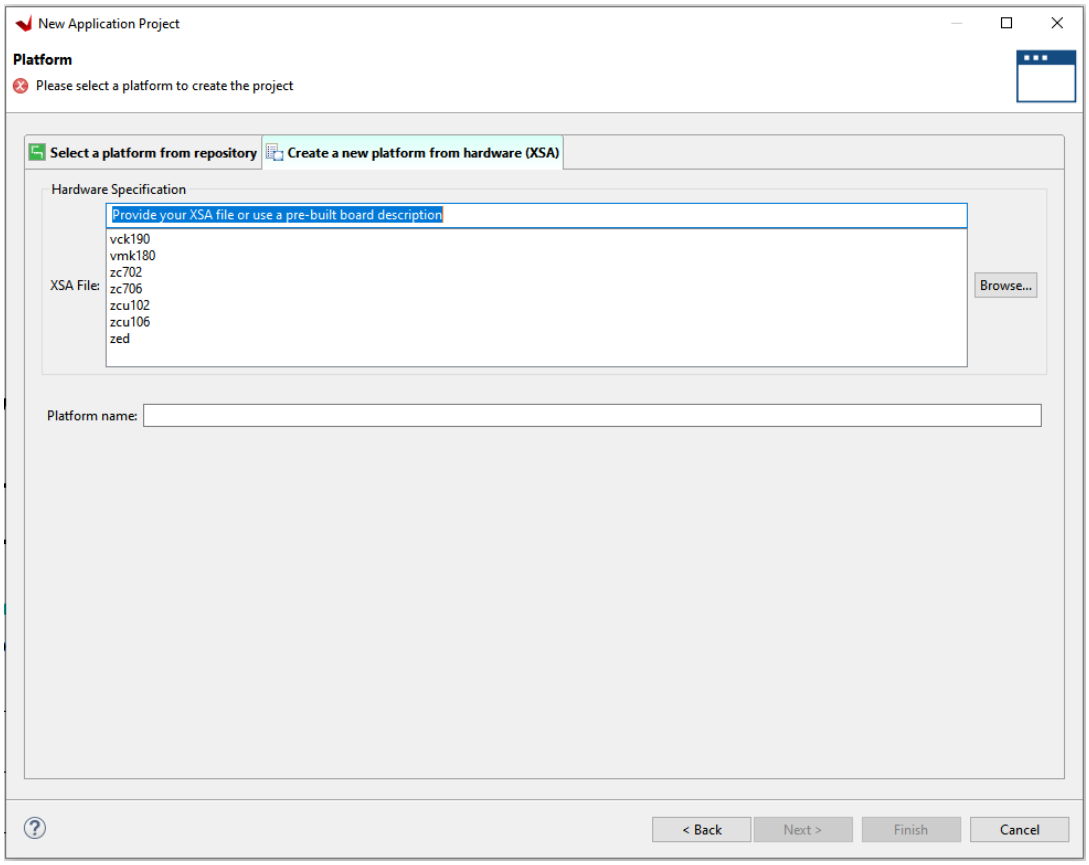
3.) Open Xilinx Vitis 2020.2 and select an appropriate workspace path, then select *launch.*



4.) Select *create application project* under project.

5.) Select *next*.

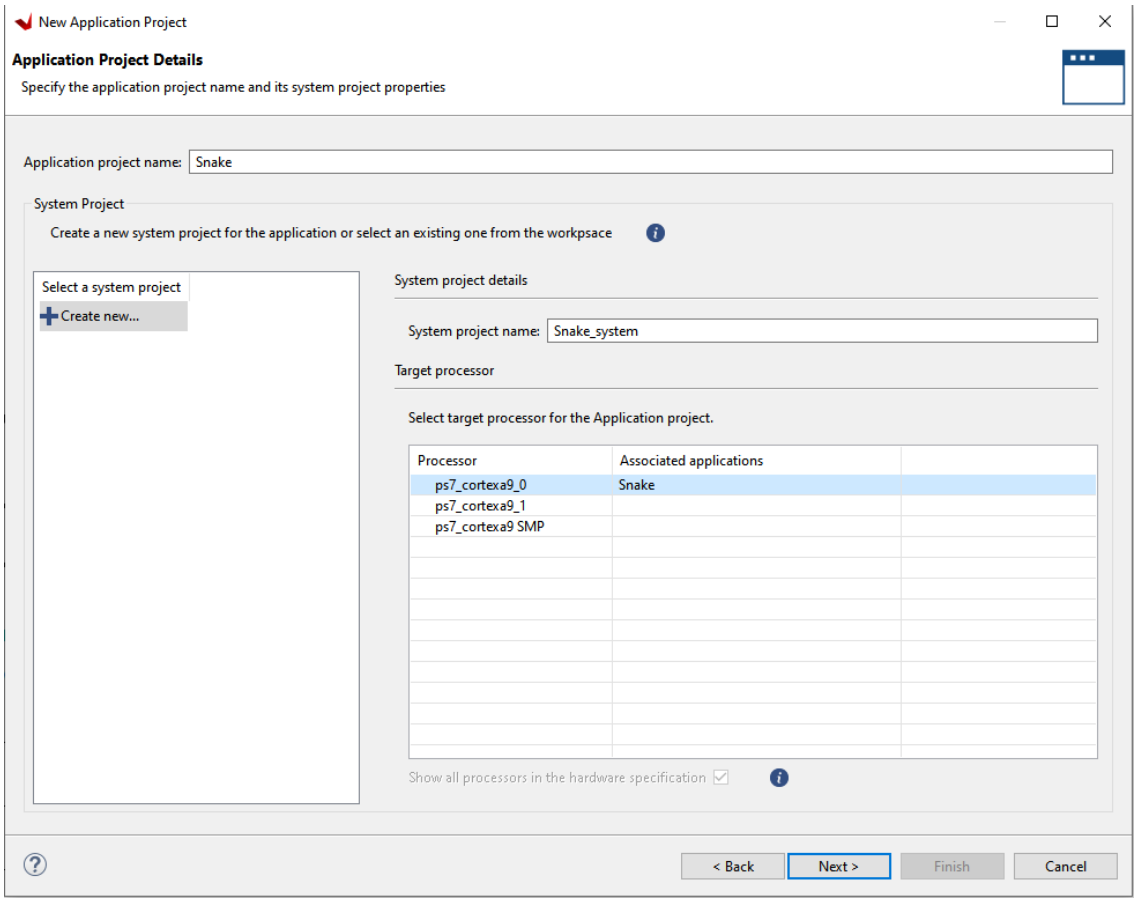
6.) Select the *create a new platform from hardware (XSA)* sub-window.



7.) Select *Browse* and path to the generated or provided .xsa hardware file, *audio_backup.xsa* from the *VivadoProject/audio_backup* subdirectory.

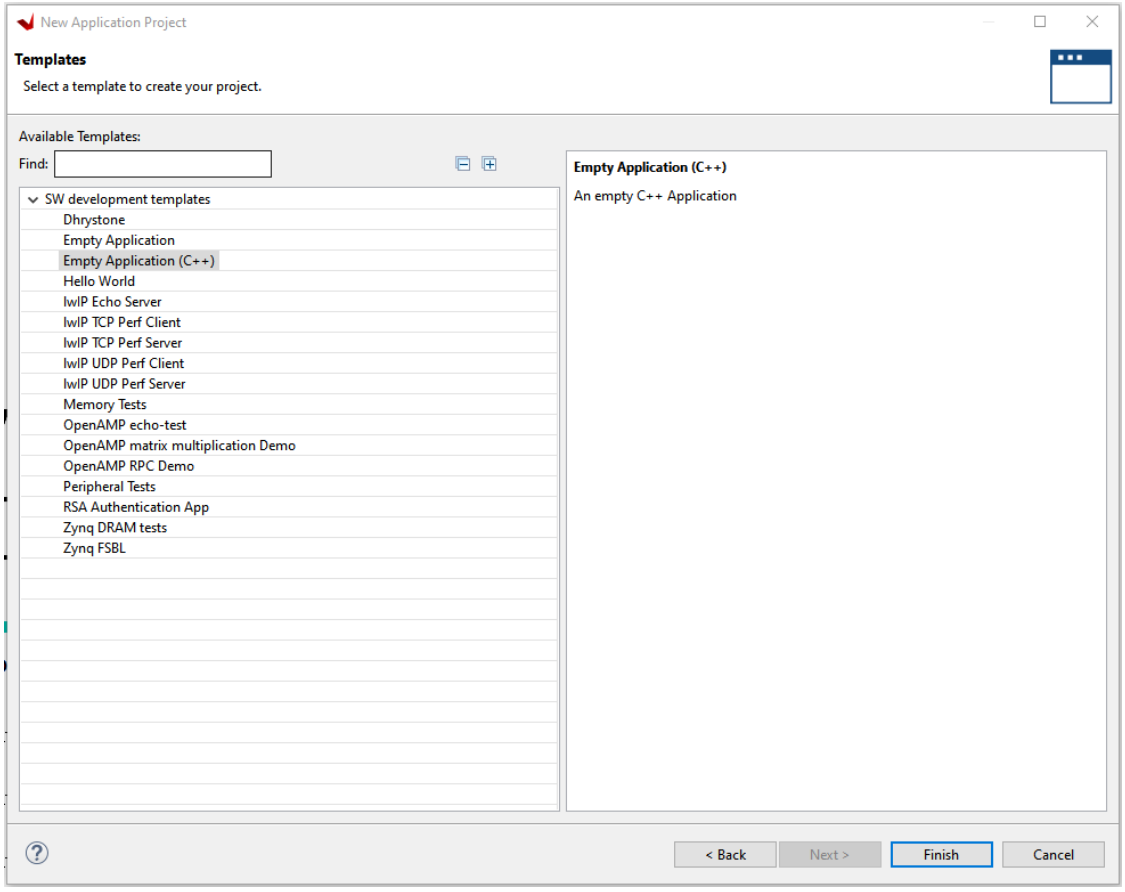
8.) Select *Next.*

9.) Provide an appropriate project name in the *application project name* box. Ensure that the selected process is *ps7_cortexa9_0*. Then select *next.*

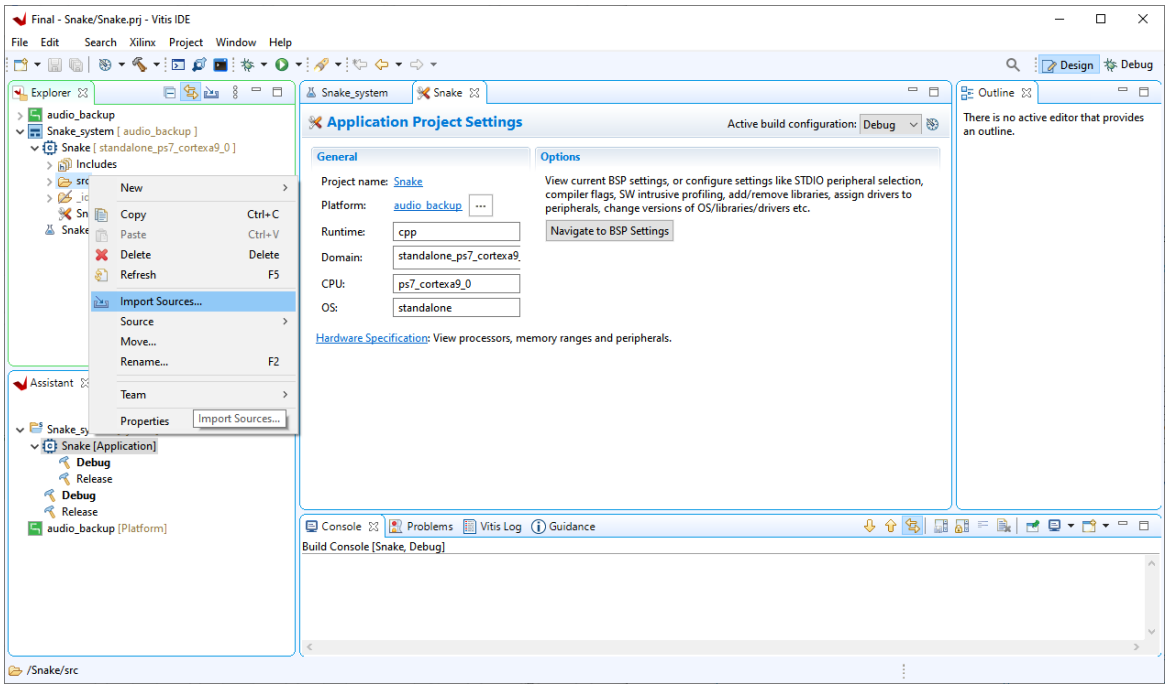


10.) Select *next.*

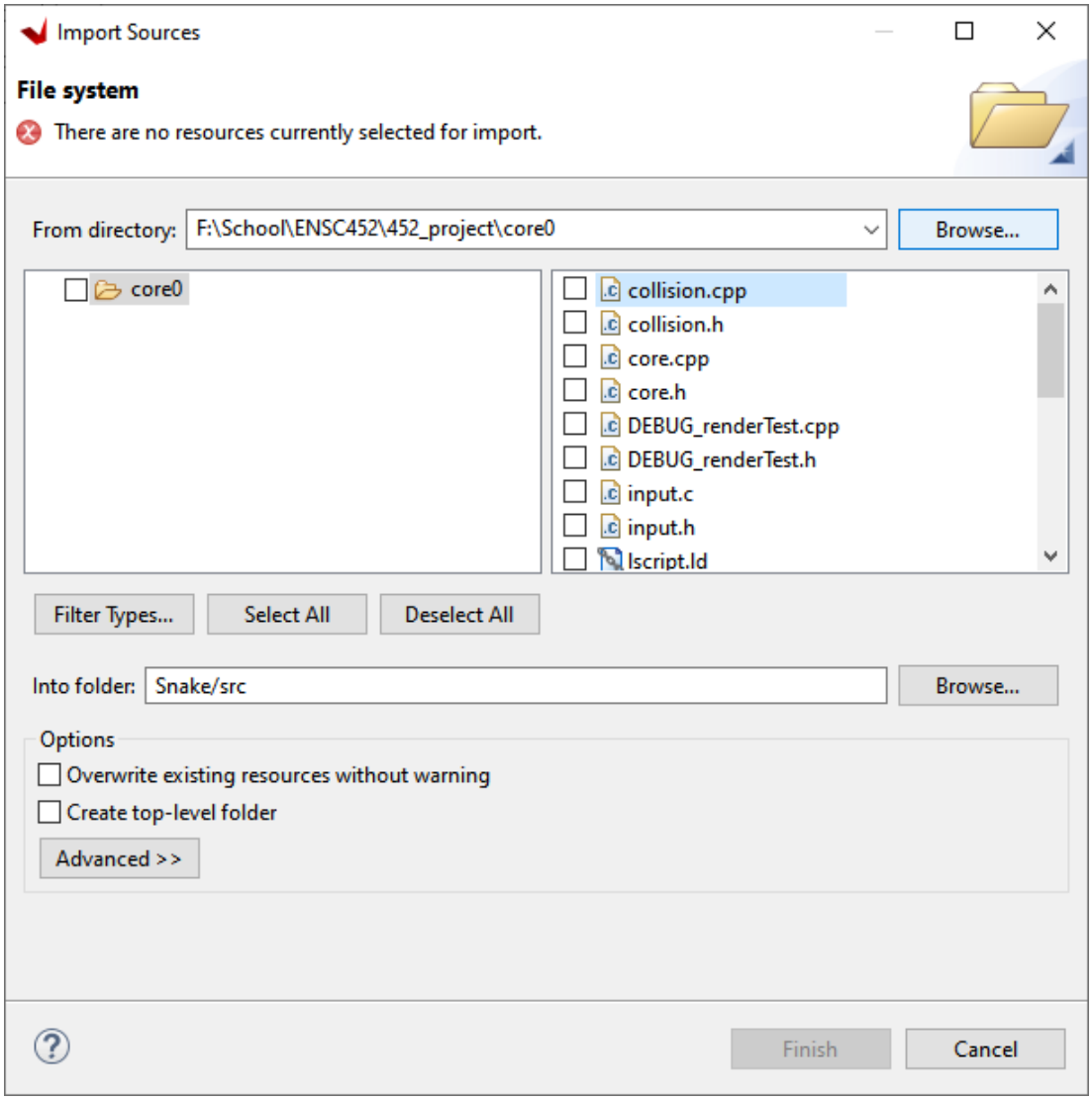
11.) Select *empty application (C++)*, then select *finish* and wait for the project to initialize.



12.) Right click on the *src* folder under the application project and select *import sources*.

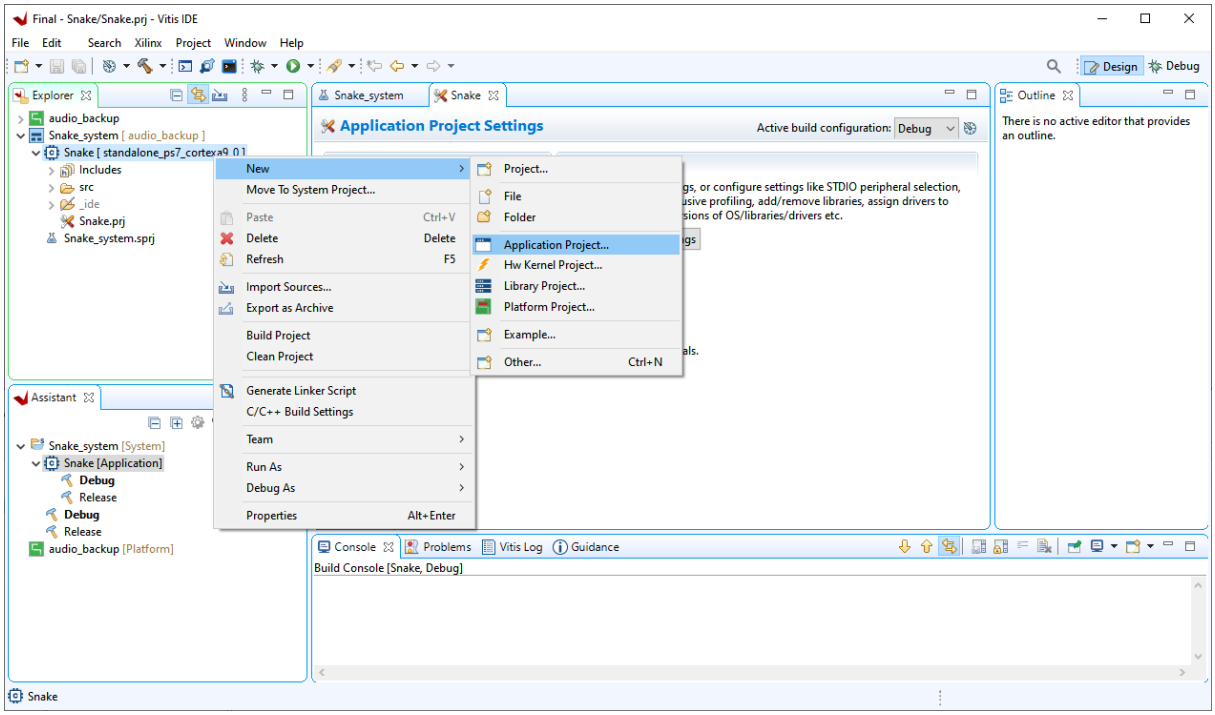


13.) Select *browse* next to the *from directory:* input box. Select the *core0* subdirectory.



14.) Select the *select all* button, then press finish. If you are prompted with file overwrite warnings select *yes to all*.

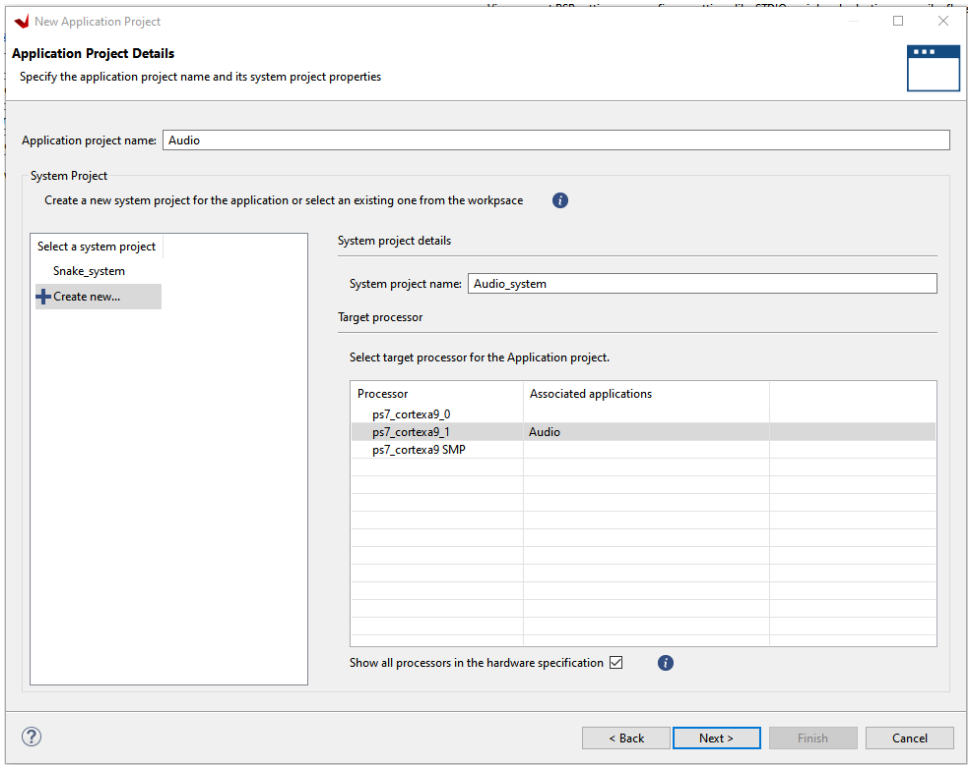
15.) Right click on the application project. Select *new, application project*.



16.) Select *next*.

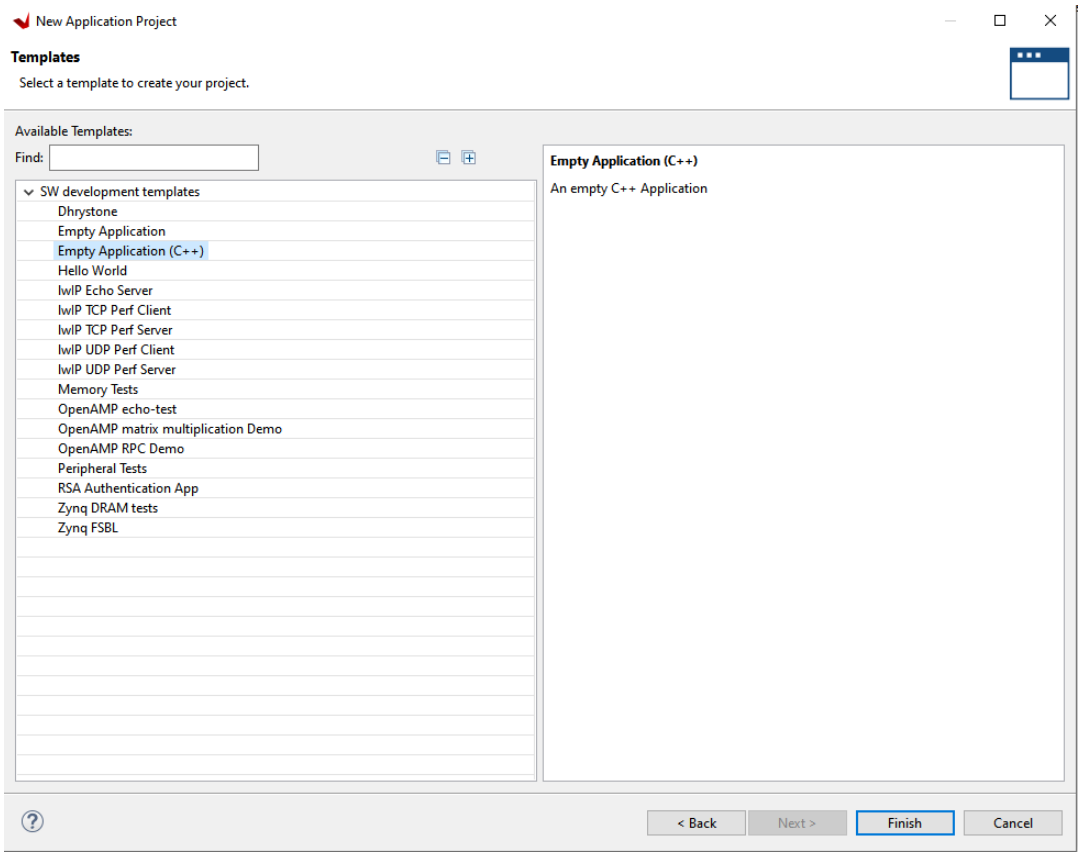
17.) Select *next.*

18.) Provide the application project with an appropriate name. Ensure that the tick box *show all processors in the hardware specification* is checked. Select *ps7_cortexa9_1* as the target processor. Then select *next.*

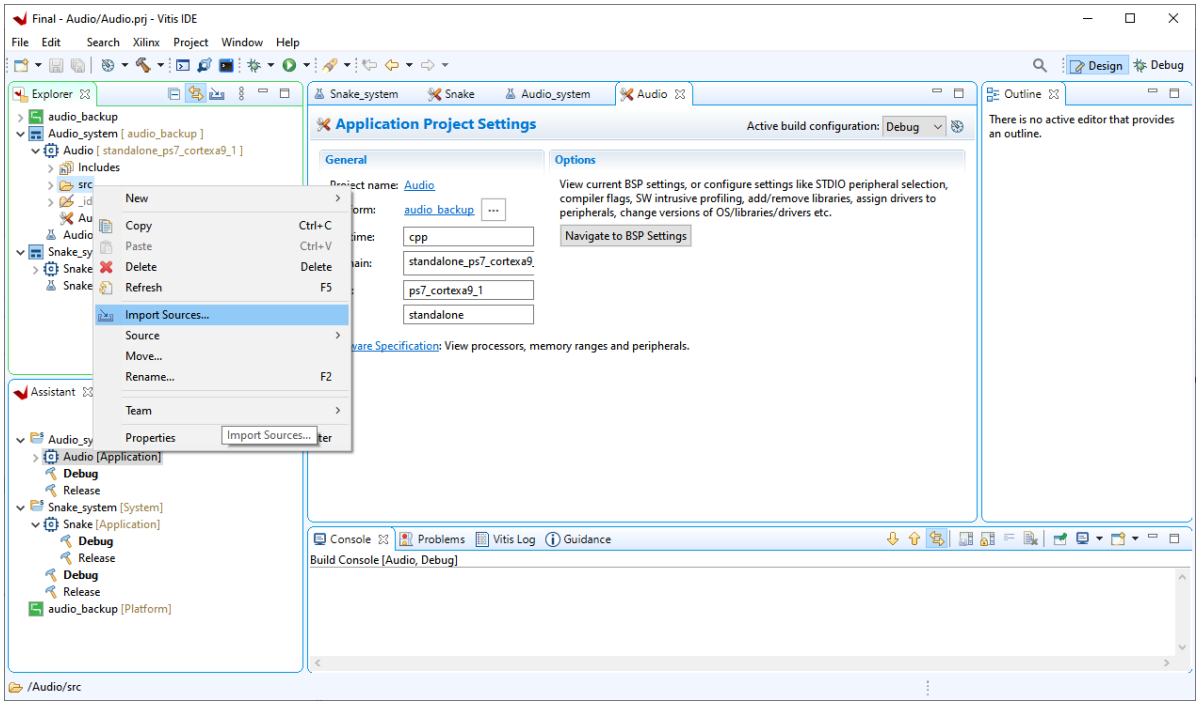


19.) Select *next*.

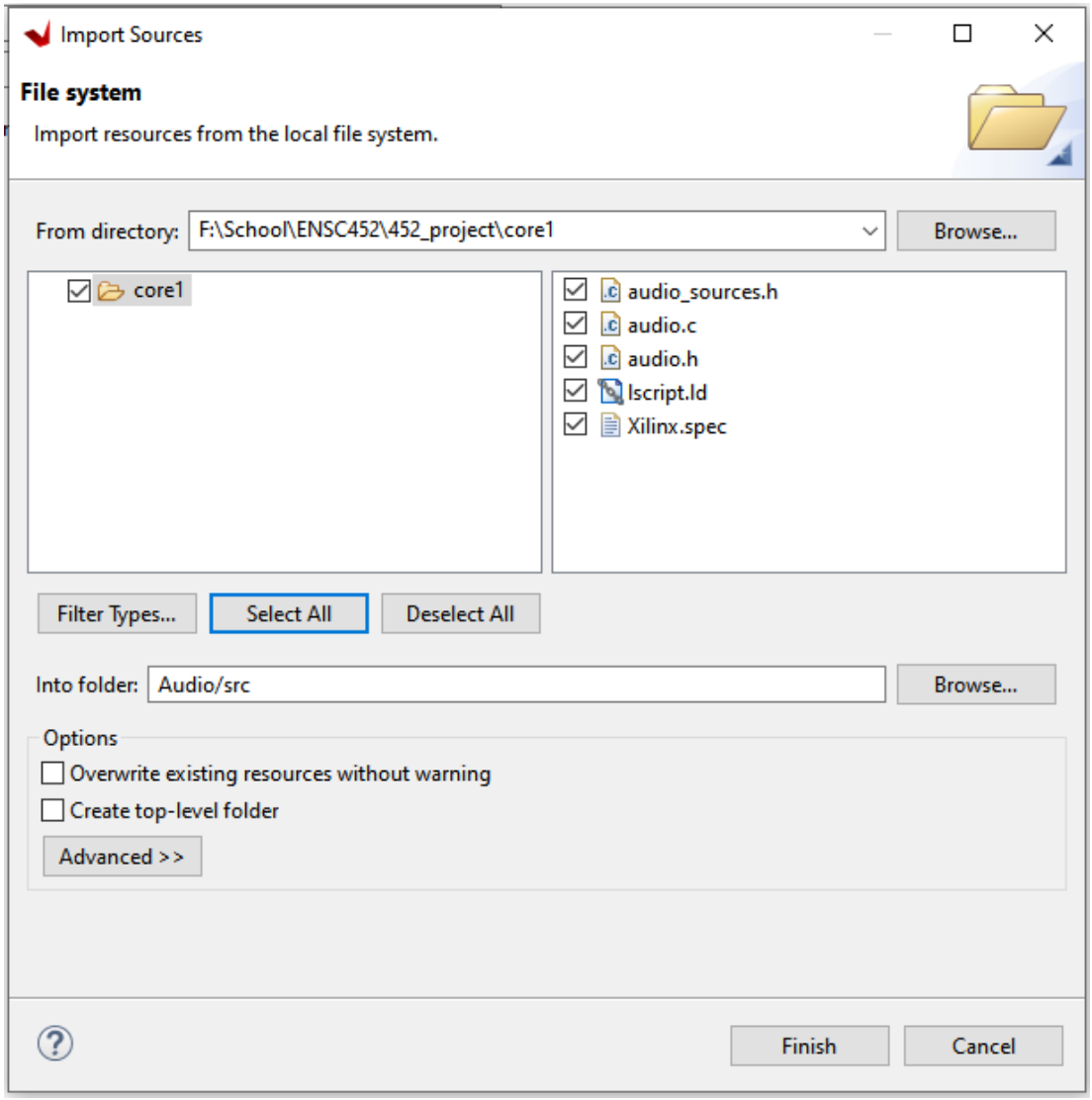
20.) Select *empty application (C++)*, then select *finish.*



21.) Right click on the *src* folder under the newly created application project and select *import sources.*



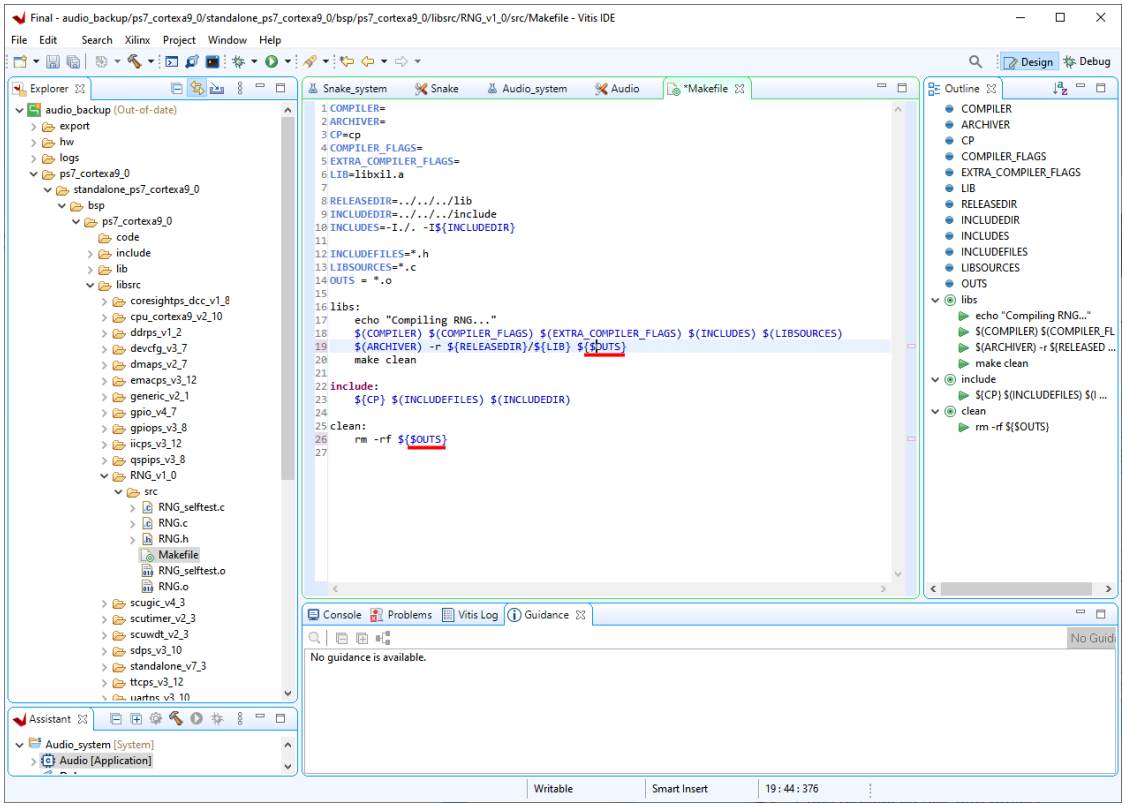
22.) Select *browse* next to the *from directory:* input box. Select the *core1* subdirectory.



23.) Select the *select all* button, then press finish. If you are prompted with file overwrite warnings select *yes to all*.

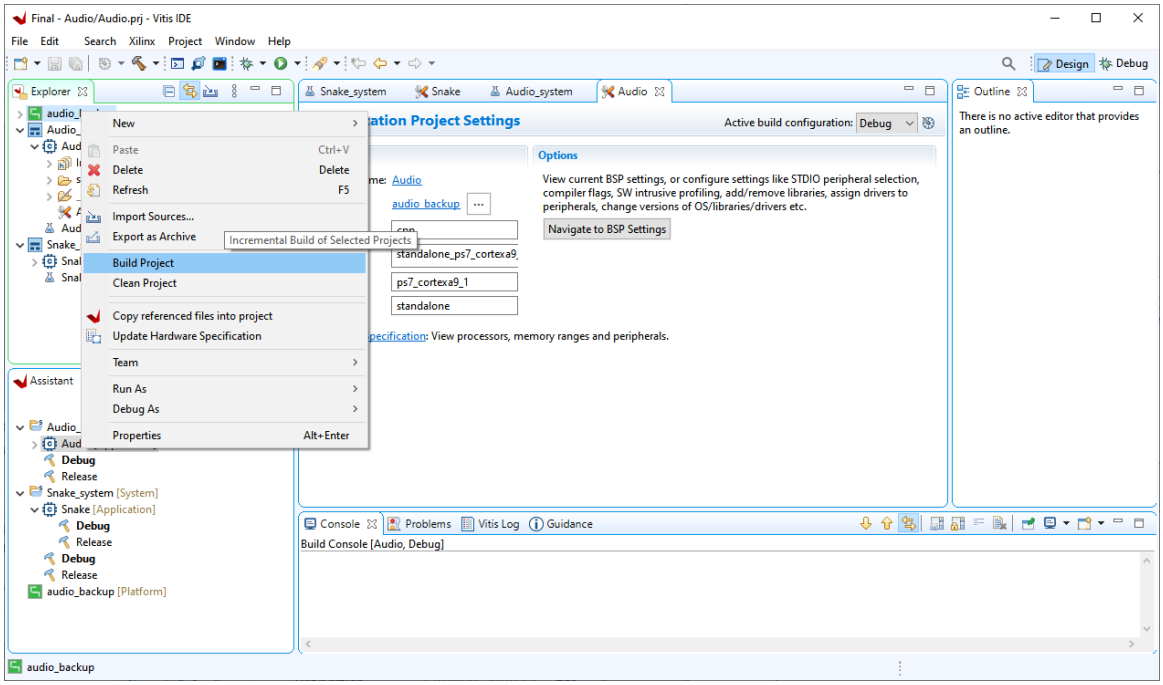
24.) Expand the hardware project and path to *ps7_cortexa9_0/standalone_ps7_cortexa9_0/bsp/ps7_cortaxa9_0/libsrc/RNG_V1_0/src/* . Open the file *Makefile.*

25.) Change all instances of *${OUTS}* with *${$OUTS}*. Save your changes.

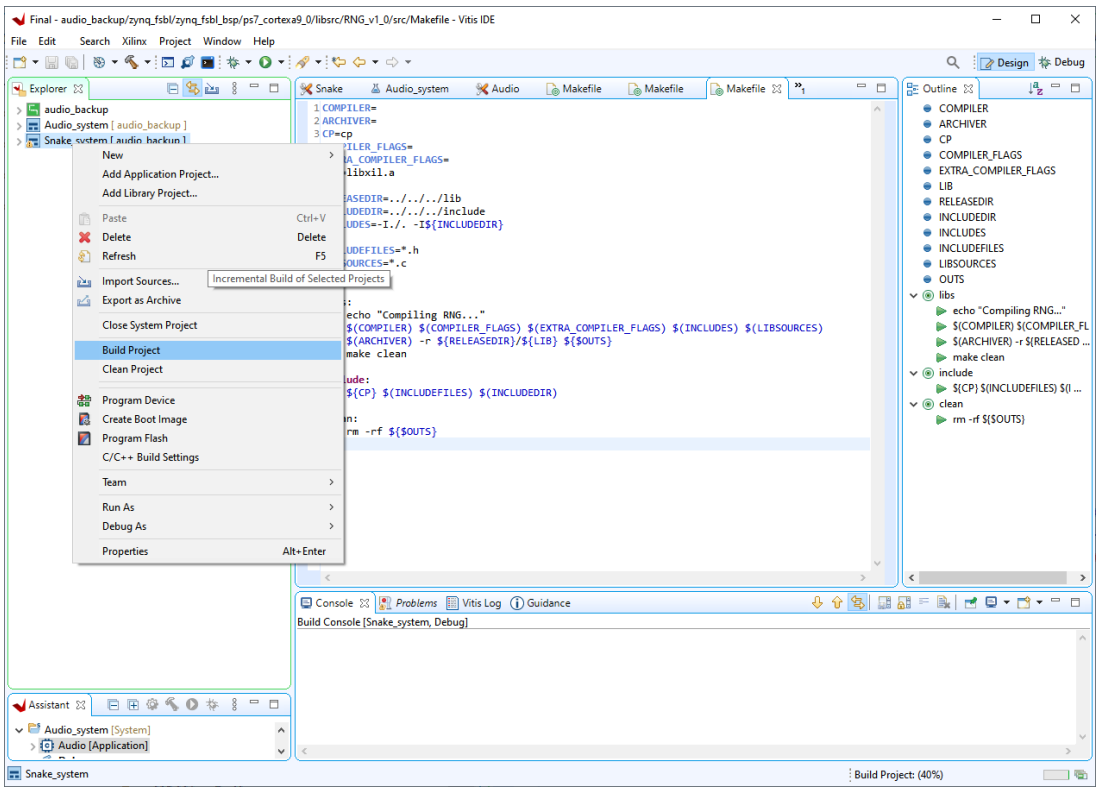


26.) Repeat steps 24 and 25 for the Makefile files in the following hardware project paths: *ps7_cortexa9_1/standalone_ps7_cortexa9_1/bsp/ps7_cortaxa9_1/libsrc/RNG_V1_0/src/* and *zynq_fsbl/zynq_fsbl_bsp/ps7_cortexa9_0/libsrc/RNG_V1_0/src/.*

27.) Right click on the hardware project and select *build project.*

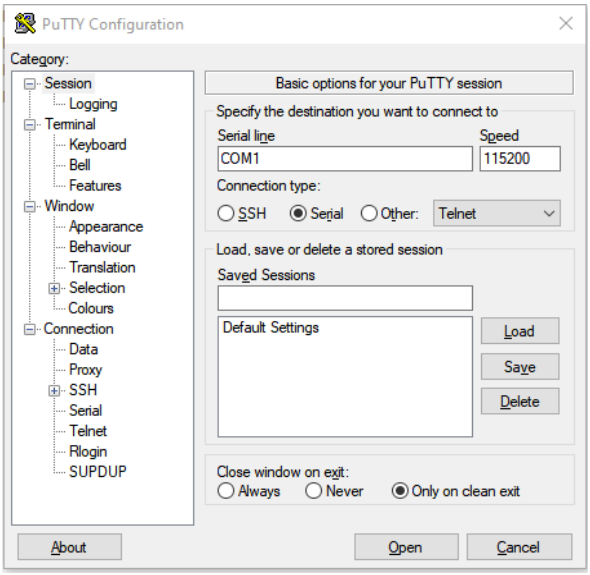


28.) For both the application projects: right click on the application project and select *build project*. If you encounter errors relating to the debug subdirectory of the application projects, clean and rebuild both application projects.



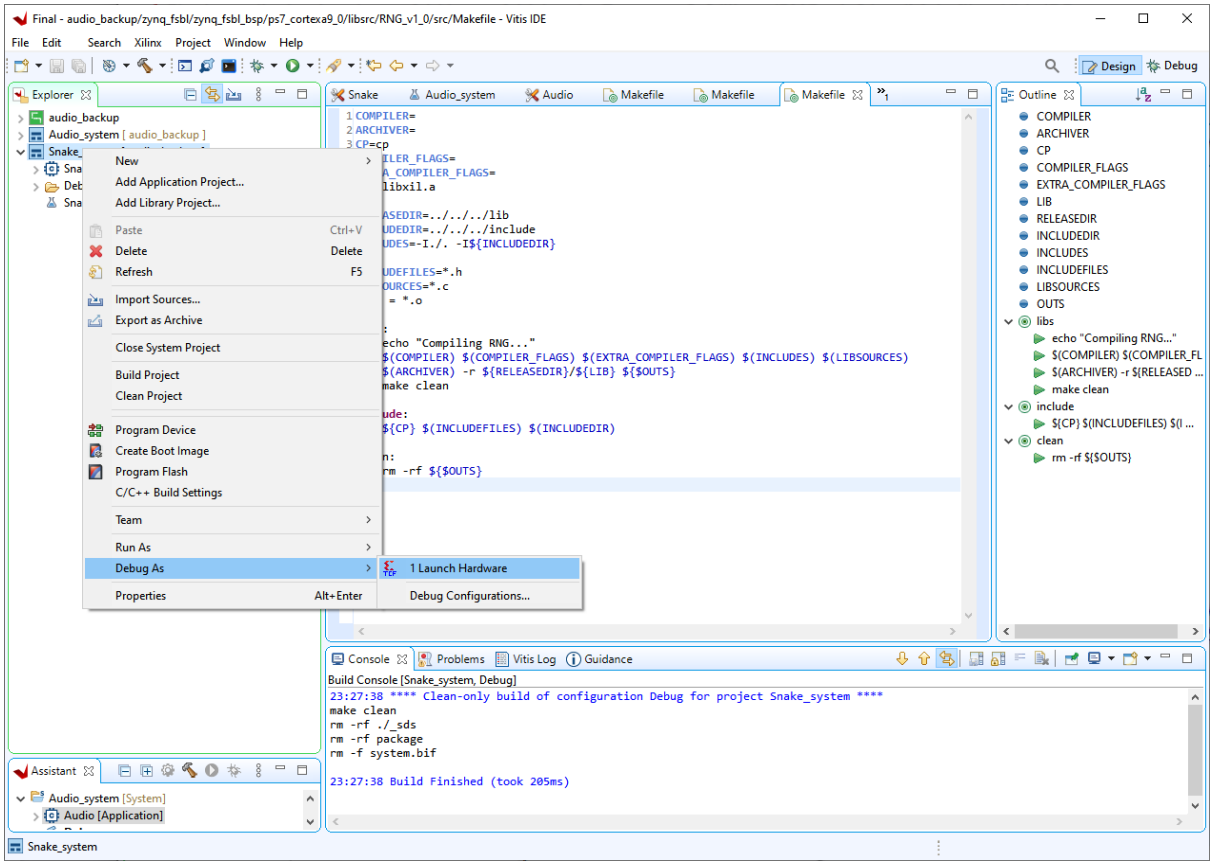
29.) Connect the ZedBoard to your system using the ZedBoard's JTAG/Debug and USB UART ports. Connect your VGA display to the ZedBoard's VGA port. Additionally, power the ZedBoard through the ZedBoard's power port and turn the ZedBoard on.

30.) Open PuTTY. Select *serial* connection type. Input the serial line corresponding to the ZedBoard's serial line (for Windows users this can be found in the Device Manager). Set the speed to *115200*.



31.) In Vitis, right click on the first application project you created select select *debug as, launch hardware.* The program will upload onto your ZedBoard. Note this process may take some time. Additionally, you have need to click on both application projects and

select continue after the program is uploaded to the ZedBoard



Congratulations! If all the previous steps were completed properly the program is not running on your ZedBoard you can now play snake!

3.2.2 Playing the Game

After successfully loading the game onto the ZedBoard you will be greeted by the main menu. On every menu you can press *W* on the keyboard to move the cursor up, *X* on the keyboard to move the cursor down, and *S* to select the option selected by the cursor.

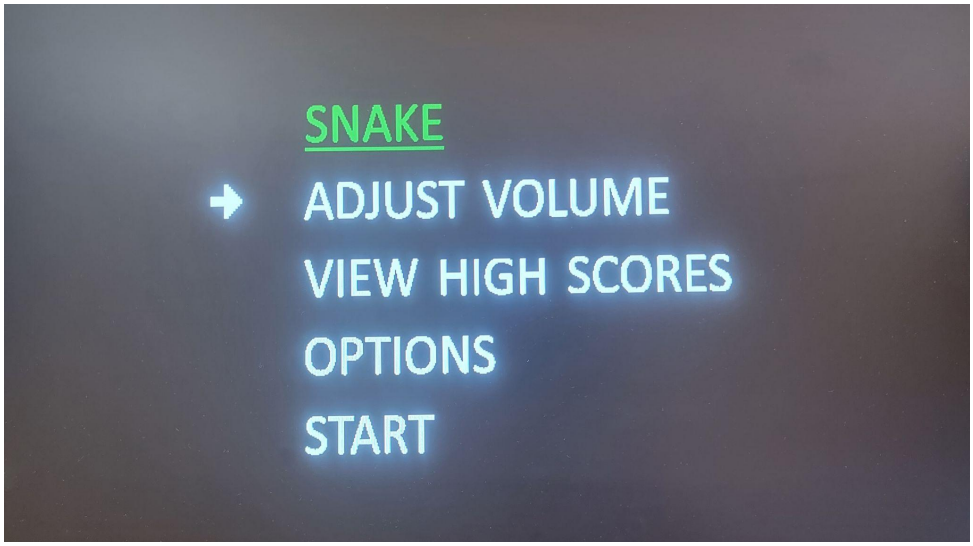


Figure 2: The game's main menu. The first screen the user will see upon starting the game

3.2.2.1 Main Menu

Selecting the *ADJUST VOLUME* menu item sends the user to the volume menu. Selecting the *VIEW HIGH SCORES* menu item sends the user to the high score screen. Selecting *OPTIONS* sends the user to an additional options menu. Selecting *START* starts the game.

3.2.2.2 Volume Menu

The volume menu gives the user the options to increase and decrease the game's audio volume. The audio volume is represented as a number that ranges from 0 to 10. A volume of 0 ensures that no game audio will be outputted through the ZedBoard's AUX port. A volume of 10 ensures that the game volume will be the loudest as supported by the game. Selecting *INCREASE VOLUME* increases the volume by 1. Selecting *DECREASE VOLUME* decreased the volume by 1. Selecting *RETURN* will return the user to the main menu.

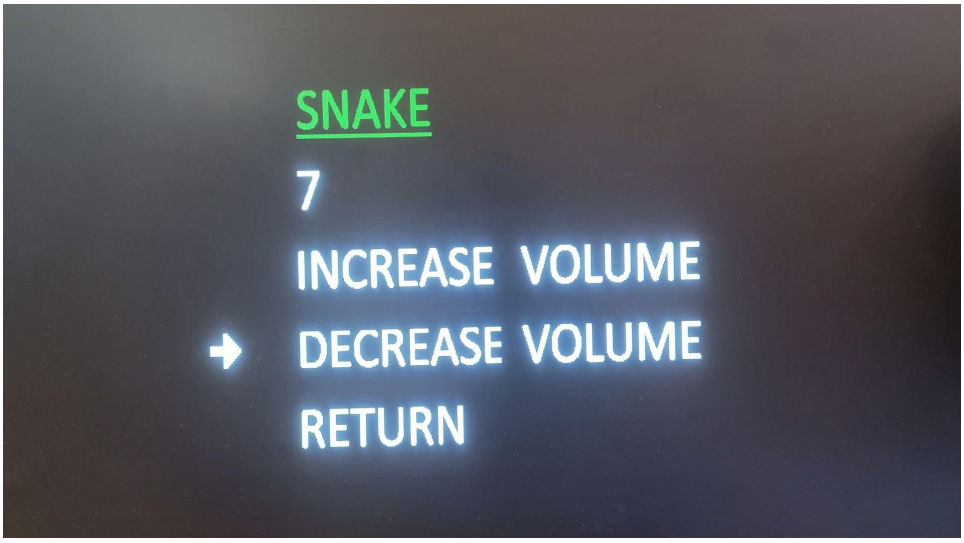


Figure 3: The game's volume menu

3.2.2.3 High Score Menu

Upon entering the high score menu the user will be shown the top five scores achieved during the current play session. The user has only one option on this menu, *RETURN*, which will return the user to the main menu.

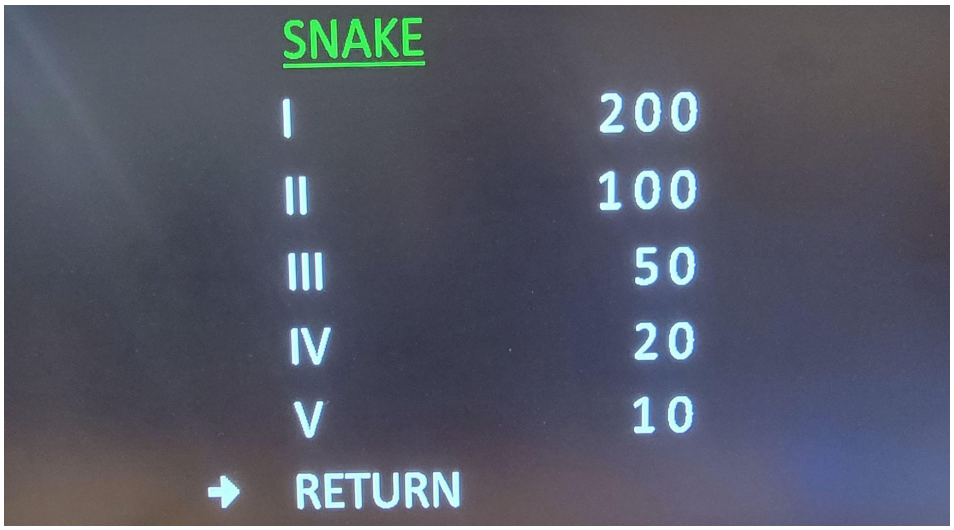


Figure 4: The game's high score menu

3.2.2.4 Options Menu

The options menu provides several different gameplay and visual options. Selecting *ROAMING* toggles the roaming game mode, an option where the game's food roams around the play area. Selecting *HARD MODE* toggles hard mode, an option where the snake progressively moves faster the more food you eat. Selecting *SNAKE COLOR* cycles the snake's color between three options: green, blue, and red. Selecting *FOOD* cycles the food's sprite between

five options: apple, orange, cherry, meat, and cheese. Selecting *RETURN* will return the user to the main menu.

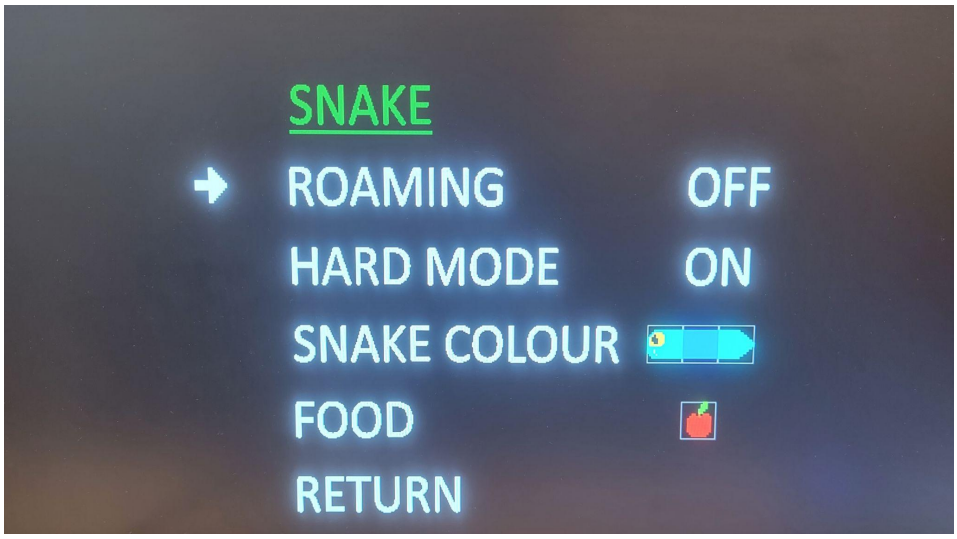


Figure 5: The game's options menu

3.2.2.4 Gameplay

During gameplay you control the snake. Your objective is to navigate the snake to eat pieces of food while avoiding colliding into yourself, or the play area's bounding walls. However, there's a catch. Every time you consume a piece of food your snake will grow longer, shrinking the available space through which the snake can maneuver.

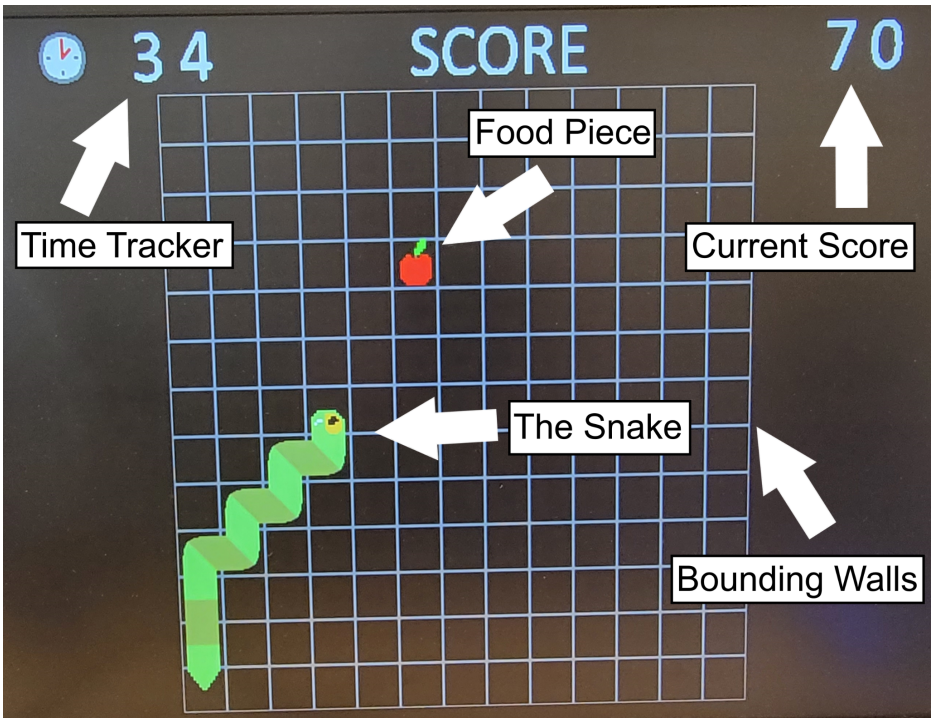


Figure 6: A Capture of Gameplay with Labeled Important Items

The snake is controlled through the *W, A, S,* and *D* keys of the keyboard and controls the snake to move up, left, down, and right, respectively. The snake cannot turn around instantly. If the snake is moving right you must first move up or down before you can move left. The player can also pause the game by pressing the *P* key, and unpause the game by pressing the *R* key. While the game is paused the snake will not move and the game time will not increment. The player can determine if the game is paused by looking for the pause icon on the right side of the screen.

Figure 7: A Capture of Paused Gameplay

If the user moves the snake's head to the food piece the food piece will reappear at a new location on screen, the snake will grow slightly, and the player's score will increase. Score does not increase linearly. As the player eats more food the score will increase faster. At first the score will increase in increments of 10. After the player reaches a score of 100 the score will increase in increments of 100. After the player reaches a score of 1,000 the score will increase in increments of 1,000. After the player reaches a score of 10,000 the score will increase in increments of 10,000. The time tracker displays the number of seconds elapsed in the current game.

If the play moves the snake's head to collide with another part of the snake, or one of the play area's bounding walls the game will trigger a game over. Upon triggering a game over a death animation will play, followed by the game proceeding to the game over screen. If the player's score is sufficiently high upon reaching a game over, the player's score will be stored with the high scores in the high score screen.

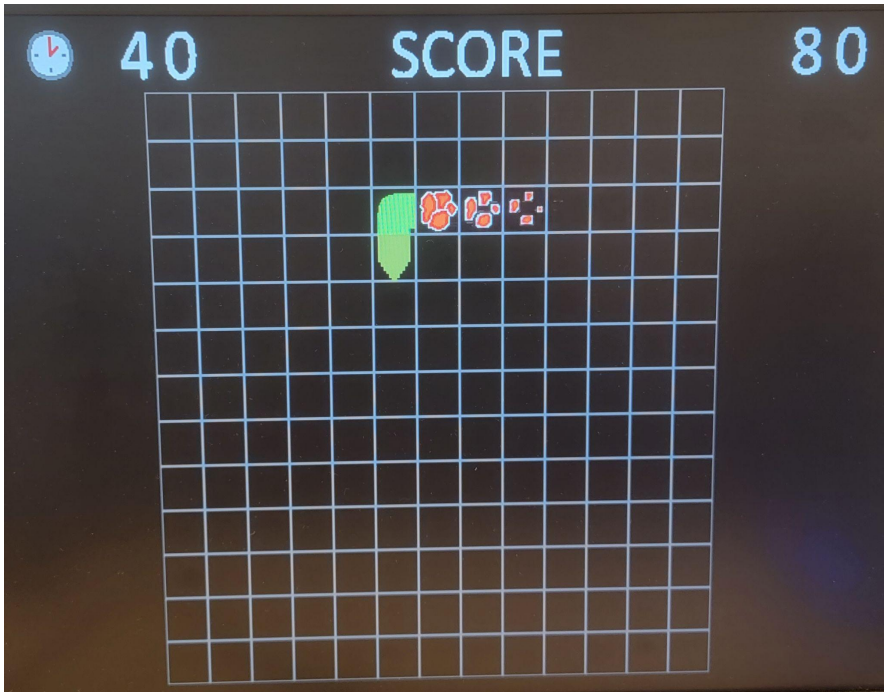


Figure 8: A Capture of the Game Over Animation

3.2.2.5 Game Over Screen

The game over screen is reached when the player triggers a game over by having the snake collide with itself, or with one of the play area's bounding walls. The game over screen displays the player's achieved score and a singular menu option: *RETURN*. Selecting *RETURN* will send the player to the main menu.

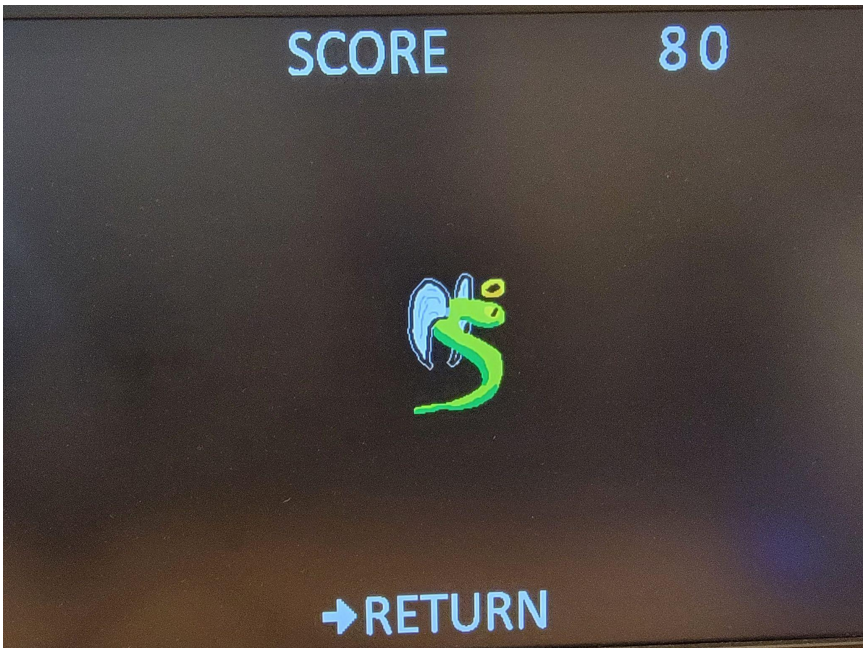


Figure 9: The Game Over Screen

3.2.2.6 Additional Gameplay Options

The game provides two additional gameplay modes: roaming and hard mode. These options are both toggleable through the game's option menu. When roaming is enabled the food will move around the map in a straight line, occasionally changing direction. However, only the snake's head can eat the moving food. If the food collides with the snake's body the food will bounce off of the snake. When hard mode is enabled the game will get faster as the snake eats more food. The speed at which the snake moves will double when the player reaches 100, 1,000, 10,000, and 100,000 points.

3.3 IP

Notable hardware blocks that were implemented in our Vivado project include:

- The Xilinx ZYNQ7 Processing System IP
- A custom RNG implementation
- The Xilinx AXI GPIO IP
- The zed_audio_ctrl IP provided with the "Adventures with IP Integrator" tutorial [2]
- The VGA Controller IP provided with the "VGA Controller" tutorial [3]
- The Xilinx Processor System Reset IP
- The Xilinx Clocking Wizard IP
- The Xilinx AXI SmartConnect IP

4.0 Outcome

Milestone	Team Member	Deliverable
1	Jacob	Implemented the top level functionality of the menu pages and gameplay loop, Implemented stub functions to interface with future feature integration.
	Chris	Created gameplay sprites, created helpers scripts to convert sprites into C++ header data, created an interface to render sprites to the VGA output, integrated VGA controller into FPGA hardware.
2	Jacob	Implemented a Pseudo Random Number Generator hardware block using a 32-bit Linear Feedback Shift Register (LFSR).
	Chris	Implemented snake movement, snake growth and food consumption, food repositioning, collision detection with food and the snake's body.
3	Jacob	Created menu sprites, Implemented the Graphical User Interface for the menu pages.
	Chris	Expanded collision detection to detect collisions with the play area's

		bounding walls.
4	Jacob	Modified the FPGA hardware design to include the zed_audio_ctrl IP block, Created a MATLAB script to extract audio data from MP3 files and convert it into C++ header files containing left and right channel integer arrays, Added a second application project for the second ARM core. Developed a C++ program to stream background audio, Created a simple protocol to enable communication between the processor cores. Added functionality to the ‘Adjust Volume’ menu.
	Chris	Implemented a game over animation, and the game over screen.
5	Jacob	Implemented event-triggered sound effects for the menu pages and gameplay.
	Chris	Implemented in-game score and time tracking, implemented high score storage, insertion, and retrieval, expanded upon the high score screen, created sprites for the updated snake visuals, implemented the snake’s updated visuals.
6	Jacob	Implemented the optional roaming apples gameplay feature. Created a new menu for additional gameplay customization features. Implemented the GUI for the new ‘Options’ menu.
	Chris	Implemented hard mode, created sprites for alternative snake colors, created sprites for alternative food sprites, implemented the usage of the alternative graphics, updated the game over animation, added animation to the snake’s movement.

Table 1: The Achieved Milestones

Overall this project was a success. We were able to meet all of the project’s main milestones and meet the majority of bonus milestones. The game has responsive and fun gameplay, has intuitive menus, and multiple extra options and gameplay modes, while working within the constraints of the given hardware.

Fulfilled technical requirements for this project include the following:

- Rendering sprite-based graphics through the ZedBoards VGA output
- LFSR implemented pseudo-random number generation
- Playing background audio and sound effects through the ZedBoards AUX port
- UART based user input

It should be noted that user input was originally intended to be implemented using the ZedBoard’s push buttons. However, part way through the project we decided to use keyboard inputs via the UART port. This choice was made as the keyboard provided an easier to use, and better feeling interface than the push buttons.

The gameplay aspect of the project fulfilled the following milestone requirements:

- Snake movement
- Collision detection between the snake, onscreen food, and the play area's bounding walls
- Growing the snake upon consuming a piece of food
- Snake movement animation
- Snake death animation
- Displaying the current game time
- Displaying the current score
- High score storage
- Hard mode
- Roaming food gameplay mode
- Alternative graphics
- Gameplay-based sound effects

The game's menus provide an interface to access and enable the following requirements:

- Changing the game's volume
- Setting hard mode
- Changing the snake's color
- Changing the food's sprite
- Viewing high scores

Our produced project can be improved in several ways:

- Adding input buffering
- Implementing additional bonus gameplay options
- Improving the fluidity of the snake's movement

Currently, the gameplay portion of code only acts upon the user's last legal input of that gameplay loop. This method of interpreting user inputs can lead to cases where the user provides inputs faster than the game is set to update. For example, say the snake is moving right and the user wants to change the snake's direction to move left. The user quickly inputs up then left. If the user inputs both of these inputs during the same gameplay loop the snake will only move up. One could add input buffering such that the user can supply inputs to be acted upon in later gameplay loops.

We were able to implement the majority of bonus milestones. Future iterations may implement these leftover gameplay options. These gameplay options include: an additional gameplay option to have multiple pieces of food on screen at the same time, and an additional gameplay option to have walls randomly spawn on the game field.

Lastly, the animation of the snake's movement could be improved. Currently the snake's movement has a two frame animation. This animation appears somewhat lackluster compared to the more fluid death animation. One could improve the game by increasing the frame rate and fluidity of the snake's movement animation.

5.0 Description of the Blocks

5.1 Hardware

The following subsections describe the hardware IP blocks and constraints files used in our Vivado project.

5.1.1 ZYNQ7 Processing System

In addition to the programmable logic, the Zynq-7000 SoC includes a processor hardware block that consists of a ARM Cortex-A9 MPCore processor and various peripherals such as UART, PS-PL AXI interfaces, PS-PL clocks, memory controllers, etc. For our hardware project, we utilized both ARM cores in the ZYNQ7 Processor System IP block. We used the AXI interfaces to implement a communication protocol between the ARM cores and the IP blocks implemented in the programmable logic. We also used the ZYNQ7 Processing System block for two PL Fabric Clocks; FCLK_CLK0 (120 MHz) was used as the AXI clock for the PL IP blocks and FCLK_CLK1 (10 MHz) was used for the Audio Codec. For I/O peripherals, we utilized both UART, and I2C in our hardware design.

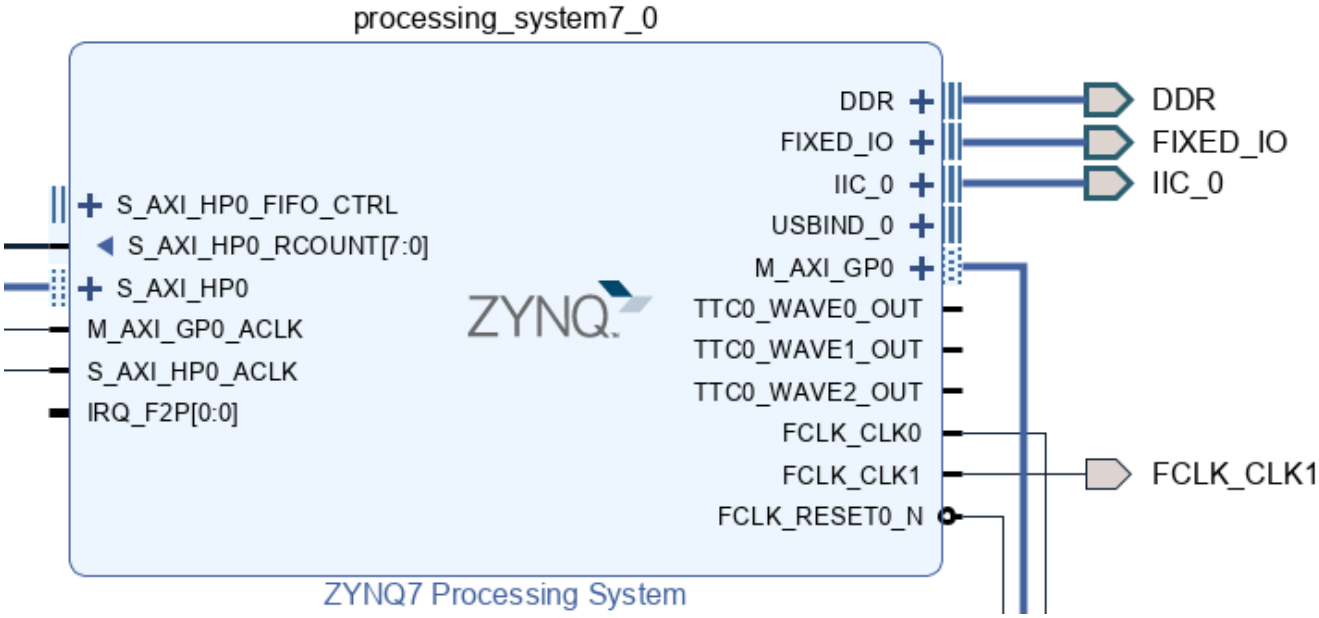


Figure 10: ZYNQ7 Processing System IP Block

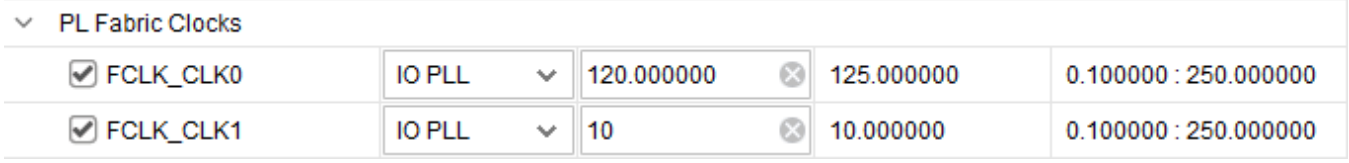


Figure 11: ZYNQ7 Processing System IP Block - PL Fabric Clocks

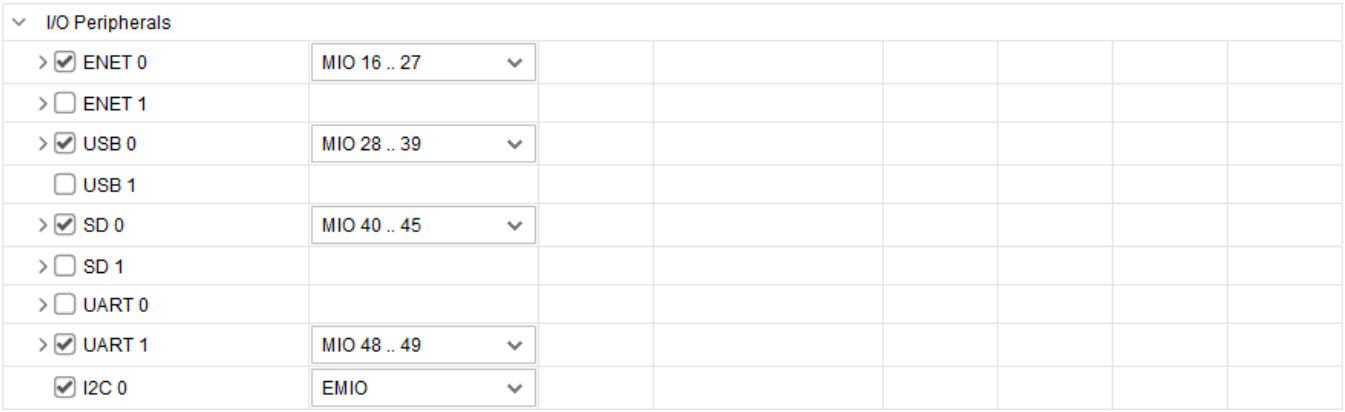


Figure 12: ZYNQ7 Processing System IP Block - I/O Peripherals

5.1.2 RNG

For the RNG IP block, we used a 32 bit LFSR implementation. LFSRs are commonly used in digital systems to create random numbers [4]. In this particular implementation, we use a feedback tap configuration of 19, 23, 30 and a seed value of '10110001101111011011010010110101'. The resulting sequence is highly unpredictable, which makes it perfect for generating random numbers in our project. The LFSR component is instantiated in an AXI4 slave interface that is auto generated by Vivado's "Create and Package IP" tool.

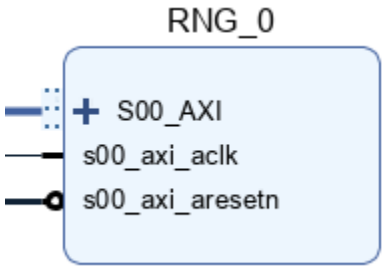


Figure 13: Custom RNG IP Block

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

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 14: 32 Bit Linear Feedback Shift Register VHDL Code

5.1.3 AXI GPIO

The AXI GPIO IP block, provided by Xilinx, is a programmable digital I/O peripheral that allows communication between an AXI-based system and external devices through GPIO signals. In our project it was used to connect to the Audio Codec's I2C address pins.

Figure 15: AXI GPIO IP Block

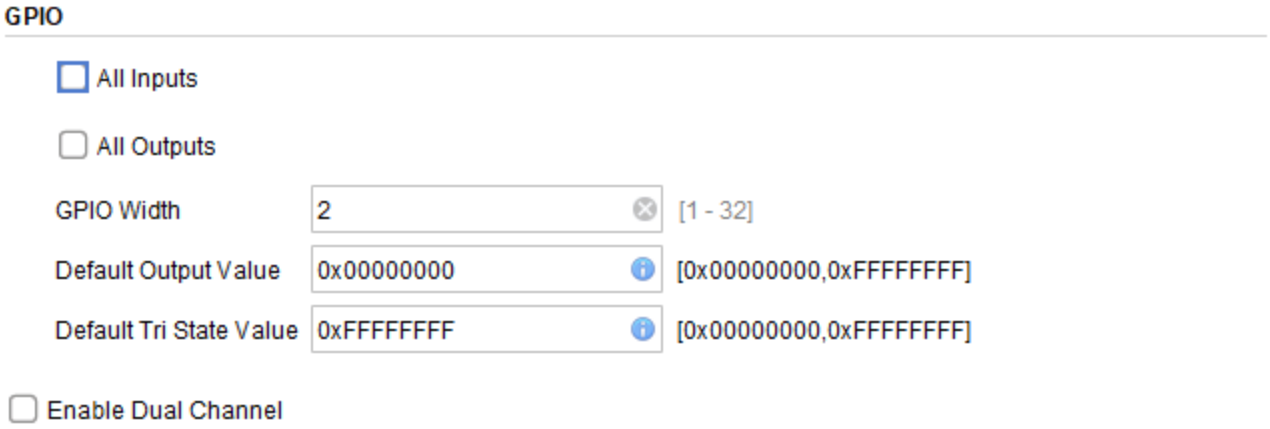


Figure 16: AXI GPIO IP Block Configuration

5.1.4 Zed Audio Ctrl

The zed_audio_ctrl IP block is an IP block provided with the Zynq Book tutorials for controlling the ZedBoard's Audio Codec [2]. We did not make any changes to the configuration of the Audio Controller IP in our project.

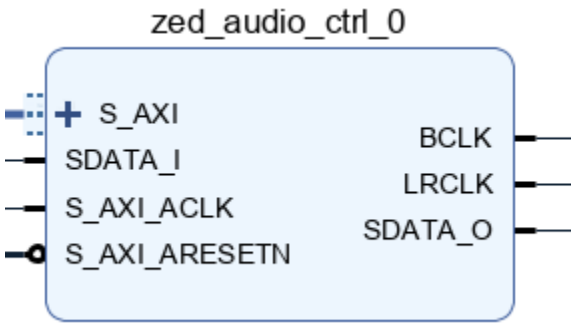


Figure 17: Audio Controller IP Block

5.1.5 VGA Controller

The VGA Controller IP block was provided with the VGA Controller tutorial [3]. The VGA IP block displays graphical output from the frame buffer in memory to the ZedBoard's VGA port and has been configured to display at 640 x 480 @ 60 Hz.

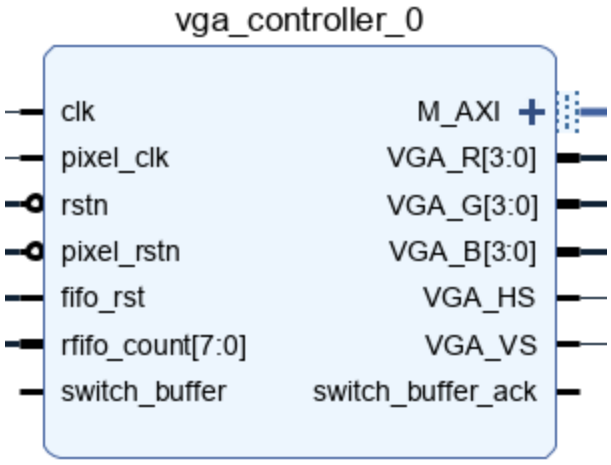


Figure 18: VGA Controller IP Block

Parameters	Values
horizontal_length	640
vertical_length	480
h_front_porch	16
h_sync_pulse	96
h_back_porch	48
v_front_porch	10
v_sync_pulse	2
v_back_porch	33
h_sync_polarity	"0"
v_sync_polarity	"0"
image_buffer2_baseaddr	0x010E9001
image_buffer1_baseaddr	0x00900000
RSTN.INSERT_VIP	0
PIXEL_RSTN.INSERT_VIP	0
CLK.INSERT_VIP	0
PIXEL_CLK.INSERT_VIP	0
M_AXI.INSERT_VIP	0
FIFO_RST.INSERT_VIP	0

Figure 19:
VGA Controller IP Block - Configuration

5.1.6 Clocking Wizard

The Clocking Wizard IP, provided by Xilinx, was used in our project to configure a Pixel Clock for our VGA Controller; in our project we requested the clocking wizard to output a 25.175 MHz clock rate for our desired resolution of 640 x 480 pixels.

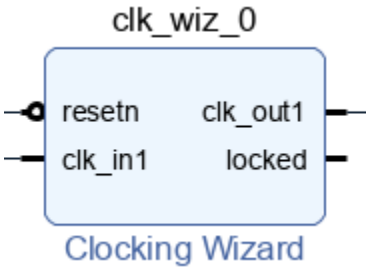


Figure 20: Clocking Wizard IP Block

Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)		Drives	Use Fine PS	Max Freq. of buffer
		Requested	Actual	Requested	Actual	Requested	Actual			
clk_out1	clk_out1	25.175	25.17397	0.000	0.000	50.000	50.0	BUFG		464.037

Figure 21: Clocking Wizard IP Block - Configuration

5.1.7 Processor System Reset

The Processor System Reset IP Block provides system level resets for an entire system, including the peripherals and main processing system to ensure a predictable start-up state. For our project, we required two instances of the IP; one for the peripherals synchronized to FCLK_CLK0 (120 MHz) and one for the VGA Controller IP's pixel_clk input.

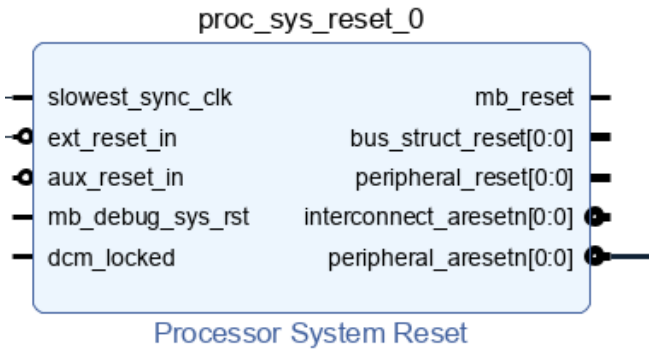


Figure 22: Processor System Reset Block

5.1.8 AXI SmartConnect

The AXI SmartConnect IP, provided by Xilinx, provides additional master AXI interfaces to facilitate communication between all AXI peripherals in the PL with the ZYNQ7 Processing System. In particular the AXI SmartConnect was connected to the slave AXI interfaces of the RNG, AXI GPIO, and Zed Audio Controller IP blocks. The VGA Controller was connected directly to the high performance slave AXI interface on the ZYNQ7 Processing System.

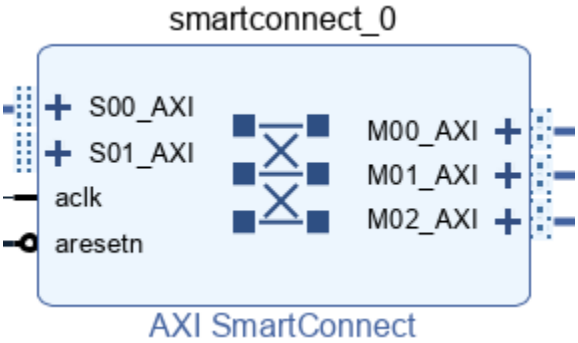


Figure 23: AXI SmartConnect IP Block

5.1.9 Constraints

The hardware for this project required two constraints files: *zedboard_master.xdc,* and *audio.xdc*. The constraint file *zedboard_master.xdc* was used to configure various elements of the hardware and was obtained from the VGA Controller tutorial [3]. The constraint file *audio.xdc* was used to configure the zed_audio_ctrl IP block and was obtained from the audio tutorial [2].

5.2 Software

The following subsections describe the most important blocks of our code that were created for our project.

5.2.1 Graphics Rendering API

5.2.1.1 Sprite Data Generation

Sprite generation is facilitated through the provided MATLAB helper scripts located in the *helper* subdirectory. Particularly *FolderDump.m* and its contained helper function, FolderDump. *FolderDump* takes a given path with images, and an output file path and provides a C++ header file. The generated C++ head file contains constant arrays of each sprite's graphical data, height, and width. Additionally, the output file contains enumerators for every sprite in the provided file. These enumerators can be used to index the sprite data in the generated arrays.

FolderDump

Argument	Description
path	The folder path which contains the images to convert to C++ header data.
fOutput	The path to the desired output folder.

Everytime new sprites are added to the project the sprite header file must be generated and reimported to the project. For compatibility with this project, the generated sprite header file must be named *sprites.h*.

5.2.1.2 Accessing Sprite Data

Sprite data is intended to be accessed through helper functions located in *render.h*. The sprite data must be initialized before it can be accessed. Initialization is performed through the function *Init.* It is intended (though not strictly enforced) that sprite data be accessed through *GetSprite*.

Init

Argument	Description

	This function accepts no arguments.

Init initialized all existing sprites from *sprites.h* such that these sprites are accessible from the function *GetSprite*.

GetSprite

Argument	Description
spriteIndex	Sprite index is the enumerator of the desired sprite. The enumerator is generated through the aforementioned MATLAB helper function.

GetSprite returns a sprite structure which encapsulates all data needed to render the desired sprite. Sprite structures can be directly passed to drawing functions.

5.2.1.3 Sprite Rendering

Sprite rendering is performed through functions in *render.h.* The software renders sprites to a 640 x 480 display via the ZedBoard's VGA port.

ClearCanvas

Argument	Description
canvas	A pointer to the base address of the canvas which will be cleared.

ClearCanvas resets the desired canvas, such that the canvas displays an entirely black screen. This function can also be used on the frame buffer, however this isn't recommended.

DrawBackground

Argument	Description
canvas	The canvas which the background will be drawn to.
background	The background which will be drawn to the canvas. Expected to be 640 x 480 pixels.

DrawBackground copies a desired background to the desired canvas. Static backgrounds can be pre-rendered and copied to the canvas to improve performance. This function can also be used on the frame buffer, however this isn't recommended.

PaintToCanvas

Argument	Description
canvas	The canvas which the sprite will be drawn to.

sprite	The desired sprite to be drawn.
xPos	The x coordinate where the sprite will be drawn from.
yPos	The y coordinate where the sprite will be drawn from.

PaintToCanvas draws the desired sprite to the desired canvas at the desired position. Transparency is not supported, and drawing a sprite will overwrite any data on the canvas in the area where the sprite will be drawn. This function can also be used on the frame buffer, however this isn't recommended.

GetCanvas

Argument	Description
	This function accepts no arguments.

GetCanvas returns a pointer of the base address to the canvas. It is intended that sprites are drawn to this canvas.

GetFrameBuffer

Argument	Description
	This function accepts no arguments.

GetFrameBuffer returns a pointer of the base address to the frame buffer which will be displayed via the VGA port. Users are not recommended to use this frame buffer directly for performing rendering.

Draw

Argument	Description
	This function accepts no arguments.

Draw copies the canvas to the frame buffer in a way that prevents visual artifacts from memory caching. It is intended that this function be called after all sprites have been drawn to the canvas.

getEnumNum

Argument	Description
num	A number from 0-9

getEnumNum returns the sprite enumerator corresponding to the sprite of the provided numeric.

It is intended (though not strictly necessary) that sprites are drawn to the canvas provided by *GetCanvas*, instead of being drawn directly to the frame buffer. After drawing all desired sprites to the canvas the user should call *Draw* to copy the canvas to the frame buffer.

5.2.2 Gameplay Logic API

5.2.2.1 Gameplay Functions

snakeHelper.h contains several functions to access, mutate, and act upon currently stored gameplay data.

InitSnakeComponents

Argument	Description
	This function accepts no arguments.

InitSnakeComponents initialized all snake components such that the only initialized snake components are the snake's head, and the starting tail. This function is expected to be called at the beginning of gameplay.

GetComponentCount

Argument	Description
	This function accepts no arguments.

GetComponentCount returns the number of snake components currently active. In essence, it returns the length of the snake.

GetHead

Argument	Description
	This function accepts no arguments.

GetHead returns a pointer to the head of the snake.

SetApplePosition

Argument	Description
xPos	The x coordinate of the desired food position

yPos	The y coordinate of the desired food position

SetApplePosition sets the currently active food piece's position to the desired coordinates.

MoveApplePosition

Argument	Description
xPos	The current x coordinate of the food piece.
yPos	The current y coordinate of the food piece.
dir	The direction that the food piece will move.

MoveApplePosition attempts to move the apple one unit in the desired direction. If the apple collides with a wall or tail segment, the direction is reversed. If the apple is stuck between two objects, it does not move.

GetApplePosition

Argument	Description
outX	The variable where the food piece's x coordinate will be stored.
outY	The variable where the food piece's y coordinate will be stored.

GetApplePosition returns the coordinates of the currently active food piece.

move_snake

Argument	Description
currentDirection	The direction that the snake will move in.

move_snake moves the snake in the desired direction.

UpdateScore

Argument	Description
	This function accepts no arguments.

UpdateScore increments the player's current score. Score is not incremented in a linear fashion. Instead score increments by 10 until the player reaches a score of 100, then by 100 until the player reaches a score of 1,000, then by 1,000, until the player reaches a score of 10,000, then by 10,000 afterwards.

ResetScore

Argument	Description
	This function accepts no arguments.

ResetScore resets the stored player's score to 0.

GetScore

Argument	Description
	This function accepts no arguments.

GetScore returns the player's current score.

IsHighScore

Argument	Description
score	The provided score which will be compared to the existing high scores.

IsHighScore returns if the provided score is sufficiently great to be a high score.

GetHighScores

Argument	Description
outHighScores	An array which will contain currently stored high scores.

GetHighScores returns an ordered array of the 5 highest recorded scores.

UpdateHighScores

Argument	Description
score	The score to be added to the stored high scores.

UpdateHighScores attempts to insert the provided score into the currently stored high scores. The provided score will only be entered into the stored high scores if the provided score is larger than the lowest stored high score. Upon inserting a new high score the previous lowest high score will be removed from the stored high scores.

UpdateTime

Argument	Description

us	The elapsed time in microseconds which will be added to the currently tracked game time.

UpdateTime increments the current play session's time by the provided elapsed duration in microseconds.

ResetTime

Argument	Description
	This function accepts no arguments.

ResetTime resets the tracked current play session's time to 0 seconds and 0 microseconds.

GetTime

Argument	Description
	This function accepts no arguments.

GetTime returns the number of seconds, and fractional microseconds that have elapsed in the current play session.

SetHardMode

Argument	Description
isHardMode	A boolean specifying the desired state of hard mode.

SetHardMode enables or disables hard mode, based on the provided argument *isHardMode*.

GetHardMode

Argument	Description
	This function accepts no arguments.

GetHardMode returns if hard mode is currently active.

GetTimeOut

Argument	Description
baseTimeOut	The base delay used for ensuring the gameplay's update loop does not proceed too quickly.

GetTimeOut alters the duration of the gameplay loop's time delay to accommodate hard mode's increased game speed. If hard mode is enabled *GetTimeOut* halves the provided time if the player's score exceeds 100 points, quarters the provided time if the player's score exceeds 1,000 points, and eighths the provided time if the player's score exceeds 10,000 points. If hard mode is disabled *GetTimeOut* returns the provided argument *baseTimeOut*.

SetSnakeColor

Argument	Description
newColor	The desired snake color.

SetSnakeColor sets the snake's color to the provided color.

GetSnakeColor

Argument	Description
	This function accepts no arguments.

GetSnakeColor returns the currently active snake color.

SetFoodSprite

Argument	Description
newFood	The desired food sprite.

SetFoodSprite sets the food's sprite to the provided food.

GetFoodSprite

Argument	Description
	This function accepts no arguments.

GetFoodSprite returns the currently active food type.

5.2.2.2 Gameplay Related Graphics Helpers

In addition to gameplay logic based functions, *snakeHelper.h* also contains graphical helper functions used to render gameplay and menus.

Init

Argument	Description

	This function accepts no arguments.

Init initialized graphical data used in gameplay. Specifically, it initializes all sprites stored in *sprites.h*, and creates a static background used for the grid pattern seen in gameplay. This function must be called before any graphics can be rendered.

DeInit

Argument	Description
	This function accepts no arguments.

DeInit deallocates any data created in *Init*.

GetGameplayBackground

Argument	Description
background	A pointer of size 640x480 pixels where the game's grid patterned background will be created.

GetGameplayBackground creates a static background which is used in gameplay.

Render

Argument	Description
gameState	The current menu/gameplay state of the game.
showHead	Optional. Specifies if the snake's head should be rendered in gameplay.
renderSnake	Optional. Specifies if the snake should be rendered in gameplay.
highScore	Optional. Specifies if the player's score should be rendered as a high score in gameplay.
cursorPosition	Optional. Specifies the cursor index of the cursor in the current menu page.
volume	Optional. Specifies the current volume of game audio. Used in the volume menu.
moving_apple	Optional. Specifies if the roaming gameplay option is enabled. Used in the options menu.
explosionIndex	Optional. Specifies the index at which the game over animation should be drawn. Used for the game over's cascading explosion animation in the pre game over state.

Render renders the current screen to the VGA port. *Render* acts upon every menu/gameplay state, and has the ability to render every menu, along with gameplay.

AltRender

Argument	Description
direction	The direction of the snake's head.
showHead	Optional. Specifies if the snake's head should be rendered in gameplay.
renderSnake	Optional. Specifies if the snake should be rendered in gameplay.
highScore	Optional. Specifies if the player's score should be rendered as a high score in gameplay.

AltRender only renders gameplay. It is used to render the off-frame of the snake's movement animation.

AltDrawDirectionfulSnakeComponent

Argument	Description
thisDirection	The direction of the current snake body part.
nextDirection	The direction of the snake body part next closest to the snake's head.
xPos	The x coordinate of the current snake body part.
yPos	The y coordinate of the current snake body part.
ns	The sprite of a snake's body part moving from south to north.
nw	The sprite of a snake's body part moving from west to north.
ne	The sprite of a snake's body part moving from east to north.
ew	The sprite of a snake's body part moving from west to east.
en	The sprite of a snake's body part moving from north to east.
es	The sprite of a snake's body part moving from south to east .
sn	The sprite of a snake's body part moving from north to south.
sw	The sprite of a snake's body part moving from west to south.
se	The sprite of a snake's body part moving from east to south.
we	The sprite of a snake's body part moving from east to west.
wn	The sprite of a snake's body part moving from north to west.
ws	The sprite of a snake's body part moving from south to west.

AltDrawDirectionfulSnakeComponent renders one sprite of the snake at the provided coordinates, based on the directions of the current, and next snake body parts.

GetFoodSprite

Argument	Description
outSprite	The sprite structure which will contain the current food sprite.

GetFoodSprite writes the currently active food sprite to the provided sprite structure, *outSprite*.

GetSnakeSprite

Argument	Description
isOffFrame	A boolean representing if this function should return the sprites corresponding to the animation off-frame of the snake.
headN	The sprite structure which will contain the sprite of the snake's head moving north.
headW	The sprite structure which will contain the sprite of the snake's head moving west.
headE	The sprite structure which will contain the sprite of the snake's head moving east.
headS	The sprite structure which will contain the sprite of the snake's head moving south.
lEW	The sprite structure which will contain the sprite of the light-colored variant of the snake's body moving from west to east.
dEW	The sprite structure which will contain the sprite of the dark-colored variant of the snake's body moving from west to east.
lSN	The sprite structure which will contain the sprite of the light-colored variant of the snake's body moving from north to south.
dSN	The sprite structure which will contain the sprite of the dark-colored variant of the snake's body moving from north to south.
lNE	The sprite structure which will contain the sprite of the light-colored variant of the snake's body moving from east to north.
lWN	The sprite structure which will contain the sprite of the light-colored variant of the snake's body moving from north to west.
lSW	The sprite structure which will contain the sprite of the light-colored variant of the snake's body moving from west to south.
lES	The sprite structure which will contain the sprite of the light-colored variant of the snake's body moving from south to east.

dNE	The sprite structure which will contain the sprite of the dark-colored variant of the snake's body moving from east to north.
dWN	The sprite structure which will contain the sprite of the dark-colored variant of the snake's body moving from north to west.
dSW	The sprite structure which will contain the sprite of the dark-colored variant of the snake's body moving from west to south.
dES	The sprite structure which will contain the sprite of the dark-colored variant of the snake's body moving from south to east.
taillN	The sprite structure which will contain the light-colored variant sprite of the snake's tail moving north.
taillW	The sprite structure which will contain the light-colored variant sprite of the snake's tail moving west.
taillS	The sprite structure which will contain the light-colored variant sprite of the snake's tail moving south.
taillE	The sprite structure which will contain the light-colored variant sprite of the snake's tail moving east.
taildN	The sprite structure which will contain the dark-colored variant sprite of the snake's tail moving north.
taildW	The sprite structure which will contain the dark-colored variant sprite of the snake's tail moving west.
taildS	The sprite structure which will contain the dark-colored variant sprite of the snake's tail moving south.
taildE	The sprite structure which will contain the dark-colored variant sprite of the snake's tail moving east.

GetSnakeSprite returns all snake sprites of the currently active snake color.

PrintTime

Argument	Description
seconds	The number of seconds which have elapsed in the current play session.
xPos	The x coordinate where the time will be drawn.
yPos	The y coordinate where the time will be drawn.

PrintTime prints the provided time elapsed and a clock icon. The number of printed digits is only 3 digits long. Times longer than 999 will be shown as 999.

PrintScore

Argument	Description
score	The current play session's score.
xPos	The x coordinate where the score will be drawn.
yPos	The y coordinate where the score will be drawn.
showScoreWord	Optional. Specifies if the word "SCORE" will be drawn alongside the provided score.

PrintScore prints the provided score, and prints the word "SCORE" if specified. *PrintScore* supports printing of scores up to 9,999,999.

5.2.2.3 Collision Detection

The header *collision.h* provides methods for obtaining information about collisions between game objects. Game objects that are acted upon by collision detection include: the snake, food, and the game's bounding walls. Note that all of these functions are designed to only work while the game is in the gameplay state.

DetectCollision

Argument	Description
	This function accepts no arguments.

DetectCollision performs collision detection on all existing game objects and returns a bit-flag based on the detected collisions. Specifically the function returns if the snake's head collided with an apple, if the snake's head collided with a bounding wall, if the snake's head collided with the snake's body, and/or if the apple collided with any part of the snake.

Overlap

Argument	Description
first	An OverlapDims structure which represents the box collider of the first object in the desired collision query.
second	An OverlapDims structure which represents the box collider of the second object in the desired collision query.

Overlap performs a collision query on the two provided box shapes. The function returns if a collision is occurring between these two shapes.

IsOutOfBounds

Argument	Description
xPos	The x coordinate of the desired object.
yPos	The y coordinate of the desired object.
width	The width of the desired object.
height	The height of the desired object.

IsOutOfBounds performs a collision query to determine if the object described by the passed dimension characteristics is outside of the game's play area.

5.2.3 Audio Configuration Software

To configure the Zed Audio Controller IP block and the Zedboard's audio codec, we utilized the low level C code provided with the "Adventures with IP" tutorial [2].

Functions utilized:

- **IicConfig**: sets up the I2C communication interface between the board and the audio codec.
- **AudioPllConfig:** configures the Audio PLL to generate the clock signal that drives the audio codec.
- **AudioWriteToReg:** writes data to specific registers on the audio codec using I2C commands.
- **AudioConfigureJacks:** configures the Line in and Line out ports on the board.

The code also provided the header file audio.h, which defined enums for the internal registers of the audio codec and the audio controller.

5.2.4 Audio Data Generation

5.2.4.1 Audio Data Generation

Audio data generation is facilitated through the provided MATLAB helper scripts located in the *helper* subdirectory. Particularly audiodump, which extracts the left and right channel audio data from an MP3 format file and converts it into constant arrays compatible with C++ syntax and stores it in a txt file. The create_header function simply combines all of the created txt files in the current directory into a single header file for convenience.

audiodump

Argument	Description
path	The file path to an MP3 file.

create_header

Argument	Description
	This function accepts no arguments.

6.0 Description of Your Design Tree

The following subsections describe the files contained in each subdirectory of the submitted archive folder.

/audio/

- The /audio/ directory includes the source audio in the form of MP3 files and unsigned 32 bit integer arrays.

/core0/

- The /core0/ directory includes the source code files for graphics rendering, gameplay, and menu navigation. It also includes the linker script for the Vitis application project.

/core1/

- The /core1/ directory includes the source code files for outputting background music and sound effects. It also includes the linker script for the Vitis application project.

/graphics/

- The /graphics/ directory includes the source graphics used in the project. The image file formats vary between .bmp and .png images.

/helper/

- The /helper/ directory includes MATLAB helper scripts used to convert graphical and audio data into constant arrays stored in C++ header files.

/IP/

- The /IP/ directory contains the source for the hardware blocks used in this project, which were not provided through Vivado.

RNG_1.0

- The /RNG_1.0/ subdirectory includes the source code for the LFSR pseudo-random number generator created for this project.

vga_controller_ip

- The /vga_controller_pi/ subdirectory includes the source code for the VGA controller used in this project, which was provided by the VGA tutorial.

zed_audio_ctrl

- The /zed_audio_ctrl/ subdirectory includes the source for the audio controller used in this project, which was provided by the Audio tutorial.

/VivadoProject/

- The directory /VivadoProject/ includes Vivado hardware project, developed for and used in this project, and project's exported bitstream, *audio_backup.xsa*.

7.0 References

[1]	A. Angelos, "The history of snake: How the Nokia game defined a new era for the Mobile Industry," It's Nice That, 23-Feb-2021. [Online]. Available: https://www.itsnicethat.com/features/taneli-armanto-the-history-of-snake-design-legacies-230221. [Accessed: 12-Apr-2023].
[2]	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.
[3]	Aguila, Z. (2020, January 16). Vivado and Xilinx's Vitis Tutorial: VGA Controller. ENSC 452 Tutorial, Simon Fraser University.
[4]	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].