ECE532 Digital Systems Design

Multiplayer Slitherin Game

(Group Report)

Group 4 - Theodorus Budiwarman, Michael Chan, Patrick Wang, Mingzhou Zhang

April 14th, 2022

Table of Content

1.0 Overview

1.1 Motivation

Ever since the start of the COVID-19 pandemic, opportunities for social interaction have become very scarce. Most people are staying home, which means that the only way to connect with distant friends and family is through an online setting. This is the motivation for us wanting to make an online multiplayer game, which we have called *Slitherin*.

The Slitherin game takes inspiration from two well-known snake-based games, the first of which is the classic “Snake” game. Snake is a pixelated game, where the goal is to collect food items to increase the snake's length. Throughout the game, the snake must also avoid bumping into its own body, which becomes more difficult as its length increases. We have used this game as the base for Slitherin, and combined it with aspects from another popular online game called Slither.io. This has hopefully helped to create a more interesting, and interactive experience.

1.2 Goals

The goal of this project is to combine multiple software and hardware components into a new functional system that will run on the Xilinx Nexys 4 DDR FPGA. These components were designed and integrated to create a multiplayer, four-directional snake battle royale. We allow for multiple players to connect and play using their own FPGA clients, running a local version of our server-synchronized game. Though this game could also be implemented by a pure software approach like Snake and Slither.io, we took a predominantly hardware approach. Custom hardware IPs were implemented to help reduce latency and make the user's experience as smooth as possible.

1.3 Block Diagram

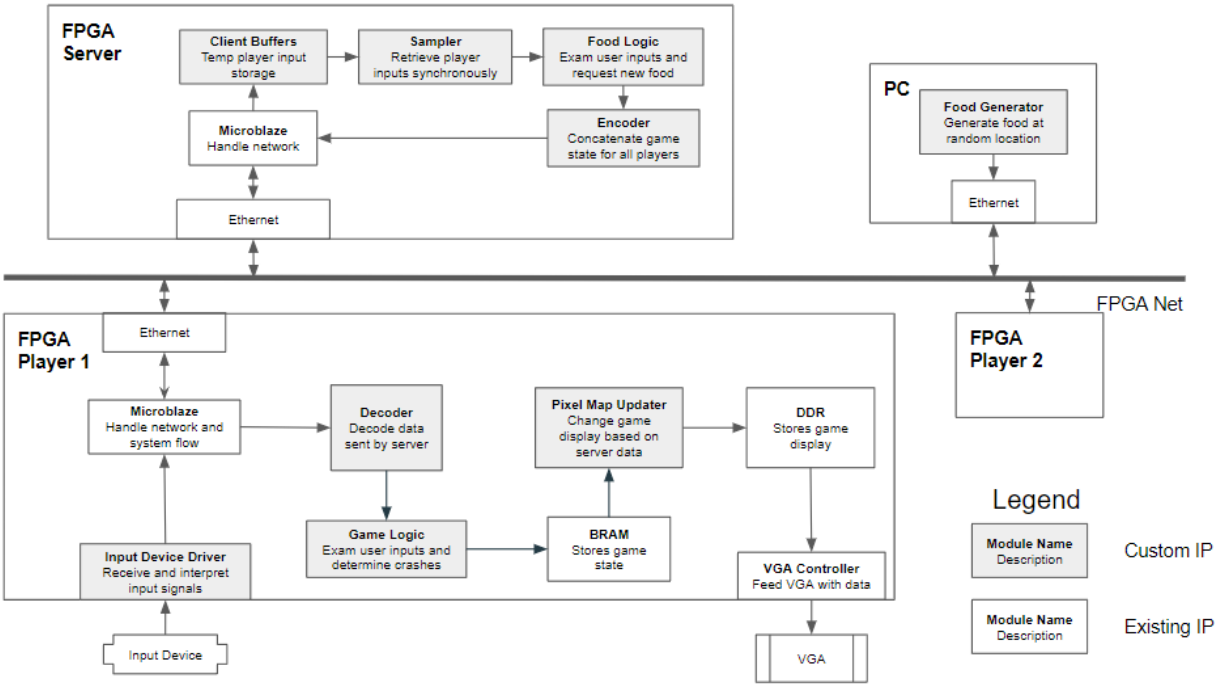

Figure 1. Block diagram of the system

1.4 Brief description of IP used, modified, created

IP Created:

- **Snake Game Logic**: Takes user input direction to control the snake movement and detect any input violations or snake collisions.
- **Snake Game AXI:** Communicates between the Snake Game Logic in hardware and software running on Microblaze.
- **Snake Game Client:** Integrates the Snake Game IPs with Microblaze and peripherals for a user to interact with the game.
- **Decoder:** Decodes packets sent through TCP from the server and store into slave registers for hardware to access.
- **Encoder:** Packs user input directions and new food locations into a single packet.
- **Pixel Map Updater:** Takes the game state stored in BRAM and draws on VGA.

IP Modified:

- **Food Generator**: Generates a pair of random food locations upon request and sends them to the server.
- **VGA Drawing IP:** Supersizes a pixel 16x when drawing onto VGA

- **TCP Client:** Sends user input direction to server and feeds packets received to the decoder
- **TCP Server:** Accept and keep track of connections with the food generator and the clients. It sends requests to the food generator and sends packets to all clients packaged by the encoder.

2.0 Outcome

2.1 Results

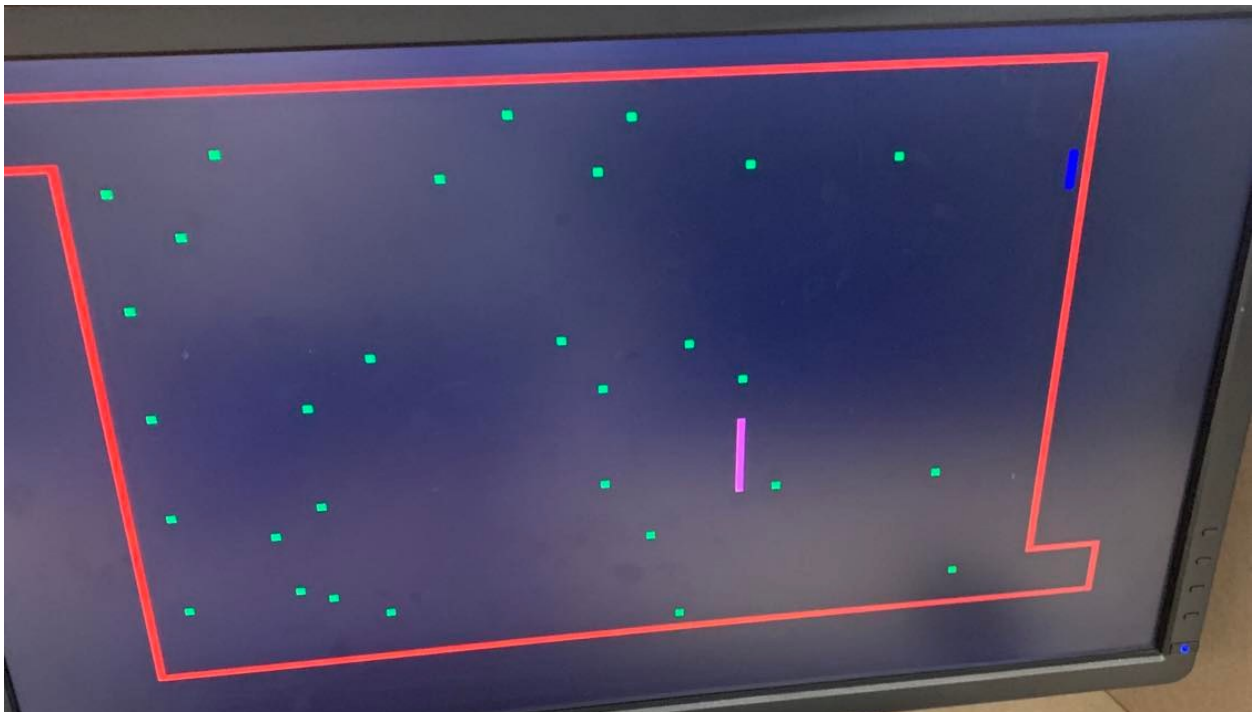

Figure 2. Image capturing the middle of a game.

Even though some modifications were made to the original proposed design, the team was still able to most of the acceptance criteria set out at the beginning of the design process. The requirements, acceptance criteria, and objective criteria have been summarized in table x below, followed by a more detailed description of individual results and decisions made by the team.

Table 1. Features/Requirement specification and their criteria

Features/Requirement Specification	**Description**	**Acceptance and Objective Criteria**
a) Output game display to VGA monitor	The snake game must utilize the VGA to display the output visuals and game state	**Snake game resolution:** Acceptance criteria: 160x120 resolution Objective criteria: 640x480 resolution **Color channel:** 8-bit RGB
b) Minimum of 2 Players Supported	The snake game must be able to support at least two players playing the same game in the server	Acceptance criteria: 2 Players Objective criteria: 4 Players
c) Minimum 5 Frames per Second (FPS)	The snake game must be able to be displayed in the VGA at a minimum of 5 FPS.	Acceptance criteria: 5 FPS Objective criteria: 20 FPS
d) 4-Directional Snake Movement	Snake's movement will be constrained to Up, Down, Left, and Right	n/a

a) The team was able to output and display the game on a monitor using the VGA connector, meeting the acceptance criteria of 160x120 pixels in resolution. The objective criteria of having a resolution of 640x480 was not met. After implementing the game at a resolution of 160x120, the team thought that the higher resolution would cause difficulties in being able to see each feature on the screen due to the fine resolution, and as a result the small object sizes.

b) The team was able to implement the game to support two players to play on a single instance of the game, meeting our acceptance criteria. The objective criteria of four players was not achieved due to time constraints caused by issues in the main game logic, and client-server communications, which have since been resolved. In addition,

the team has implemented the game in a way that allows for easy scalability, through modularised code, and parameterization wherever possible. We should be able to easily add additional players to the game now, if required.

c) The team was able to run the game, and display the updates through the VGA connector at 3 FPS, which does not meet the acceptance criteria of 5 FPS, nor the objective criteria of 20 FPS. The minimum criterion was not achieved due to two main factors:

1. Running the game too fast would cause synchronization issues between clients and their VGA outputs. We believe this was due to the inherent latency in the TCP communication process, which is out of our control.
2. After running the game at 3 FPS, the team felt that the speed of the snake was acceptable. Since the snake will move in each frame, running the game at 20 FPS may cause more difficulties in controlling the movement of the snake as it will move too fast and harm the overall experience of the players.

2.1 Further Improvements

Even though the main functionalities of the Slitherin game are complete, there are some additional features that could be added to further improve the game:

1. Peripheral inputs
   One further improvement to make would be to add peripheral controls for snake movement. In the beginning, the team attempted to make this work using the XADC and GPIO connections on the FPGA, however due to some unexpected bugs, this feature was set aside, as other critical game features needed to be worked on.

2. Win conditions
   As of now, the Slitherin game does not have any set win conditions. Players will be continuously respawned, and food will be continuously generated when eaten. Such win conditions may include:
   - A limited amount of lives, and whoever is the last snake standing wins
   - When a certain amount of food is eaten by a player
   - A time limit, after which the longest snake is the winner

3. Game aesthetics
   Another area of improvement for the future would be for the overall aesthetics of the game. Things such as adding a start screen/menu before all players have connected through the server, or highlighting certain features with borders, etc. A good looking game would definitely help to enhance the overall user experience.

4. Adding error correction for when the game becomes asynchronous
   As mentioned in section 2.1, currently when the game runs too quickly it may run into some synchronization issues. To potentially help with this, we may be able to implement some kind of error correction within our game logic for when players are receiving different packets (either previously missed or due to delays). Though the effects of doing so would still need to be tested and analyzed, it is expected that this may cause additional latency and affect the rate in which the game runs.

3.0 Project Schedule

3.1 Milestone 1

Original: Server and client implementation where client can continuously receive input that is non-blocking and/or research networking protocols that is high speed that can be used

Weekly Accomplishment:

- Implemented tcp client side communication that is non-blocking and to send a packet containing a string when switches change value. Depending on switch value, different strings are sent to imitate different directions. Used VIO to virtualize switch values, and GPIO to implement switch.
- Implemented tcp server side communication that allows for multiple client connections (two at the moment). Continuous input is taken in from the clients and a concatenated string made up of the data sent from all clients is broadcast back to all.

Discussion: During this week, we were on track with our original milestone and created the backbone of our server-client communication that our snake game will run on. However, we also decided to tweak our overall design to create the snake game logic in hardware instead of the proposed software implementation.

3.2 Milestone 2

Original: Basic Snake Game Logic in C & VGA Output using MicroBlaze. We are not outputting the snake game onto VGA yet, just establishing a VGA output.

Weekly Accomplishment:

- Implemented in verilog the snake body module and collision module
- Snake body module takes as input the directions, and outputs the updated position of the snake body in x-y coordinates. Module also checks for legality of moves (ignores input if immediately opposite from previous). Collision module checks if a snake collides with a food or a wall. If it is a food, increase snake length and update food map, if wall, send ‘died’ signal.

- Implemented a random food generator on pc, which takes the map size and generation frequency as input and sends the coordinate of the newly generated food to the FPGA server
- Researched Xilinx VGA modules, implemented TFT controller IP module to allow for output through VGA to monitor onto client side block diagram

Discussion: This week was roughly on track, except the change in decision to implement the game logic in hardware. This allows us to design our game in hardware as well as plan how we will communicate with the microblaze processor to instantiate and maintain our gamestates.

3.3 Milestone 3

Original: VGA Output and Peripheral Input for Snake Game for single player (basic snake game)

Weekly Accomplishment:

- The team has gone over the game flow and specified the functionality of each hardware block/IP, and the expected input and output. This way we have a modularized system to implement and verify piece by piece. Since we have milestones planned to build a single player game and then upgrade to multiplayer, this modularized approach is beneficial.
- We have also designed a protocol for the packet sent over the network containing game states such as players movements and new food locations.
- Experience was gained for AXI-Lite connection interfacing through assignment 1, which will hopefully help us to successfully output through VGA to a monitor

Discussion: During this week, our overall progress was delayed due to other coursework and materials. However, the team created a catch up plan to complete milestone 3 as well as work on milestone 4. Peripheral Input milestone has been pushed to milestone 4.

3.4 Milestone 4

Original: Connection between clients (players) and server allowing two players to each move one basic square pixel. Also upgrade simple snake game to slither io

Weekly Accomplishment:

- Established working **AXI4 interface** connection between our snake game module and BRAM. Now, the module is able to write data to BRAM.
- Added VIO to simulate inputs to the game, as well as an artificial synchronization signal which will eventually be sent by the server to maintain synchronization across players.
- Used an ILA to assist with extensive debugging due to issues with AXI4 protocol.

- Implemented software support in drawing random food onto VGA screen, as well as further game server development that will be responsible for the synchronization of the clients.

Discussion: Difficulties occurred when working with the snake game logic in hardware and implementing AXI4 that allows our logic to communicate to other IPs in our design. This delayed our plan of further developing our game, as well as the communication component between client and server. At this point, integration is one of the main areas of work as we developed separate IPs individually.

3.5 Milestone 5

Original: Implement slither io with multiplayer (client and server communication)
Weekly Accomplishment:

- Tweaked the existing game logic to be able to handle a new food generation by connecting a buffer-like register to input new food upon eating a food. This buffer-like register will contain a new food requested in the previous cycle, and upon eating a food, a new request will be sent to refill the buffer.
- Attempted peripheral input using an arduino to connect to an analog joystick to send interpreted input to the Pmod ports of the FPGA.
- Cleaning up existing VGA drawing logic, creating new functions for drawing/deleting snake, and working to make existing software scalable for future integration

Discussion: During this week, VGA output for a single player snake was achieved. However, implementation of multiplayer needed to be done. This freed some bandwidth for members to work on peripherals. Unfortunately, peripherals did not work leading to use using push buttons to control the movement of the snake. Original milestone was not met, but progress was achieved in completing the hardware game logic to work with software.

3.6 Milestone 6

Original: Game upgrades such as FPS improvement, Map Size increase, AI Snake to have “more” players
Weekly Accomplishment:

- Implemented and tested 2 player snake drawing using the previous drawing function. Drawing 2 snakes is functional, however, erasing the 2nd snake is not yet supported. At this point, the drawing function was still needed to be modified to work with TCP communication
- Set up PC (for random food generation) and game clients (sending input direction and food valid bit) to connect to single server

- Server is able to distinguish the sender of the data (pc or client), and it decodes the data accordingly. Data that needs to be sent (such as new food locations generated from PC) are encoded into a packet and sent out to the relevant clients

Discussion: Original milestone was to add improvements into our final game. However, this was not met and instead we focused on completing the core components of our game. During the milestone meeting, we were only missing some additional software changes to draw our complete game onto VGA.

4.0 Description of the Blocks

4.1 Snake Game Logic

The main logic of the snake game is processed in a custom hardware IP in a module called ‘snake_top’. As seen from the block diagram below, the snake_top module mainly consists of two modules; ‘snake_body’ and ‘collision’.

snake_body:
This module is responsible for calculating the current position of the snakes. Upon receiving the go_signal from snake_game_axi module, the module calculates the current position of the snake from:

- **input_dir:** The direction the snake is moving towards, which is an input from the player.
- **respawn:** A signal from the collision module that indicates whether a snake died, thereby respawning with a default size at a specific location
- **size:** A signal from the collision module that determines the current size of the snake.

The position is locally saved in two 64-wide register arrays of x and y coordinates, snake_x and snake_y.

collision:
This module is responsible for calculating collision-related events relating to each snake.

- **Wall collision:** Upon colliding with a wall, or a body of a snake (itself, or the opposing player), the module will send out a ‘respawn’ signal, which indicates that the snake died, and should respawn at a different location.
- **Food collision:** Upon colliding with a food, the collision module will increment the ‘size’ register, which is connected to the snake_body. The snake_body will in turn draw one extra pixel, indicating that the snake has grown. Then, the module

will receive food_x and food_y from input, and save it in the local food buffer (A buffer that keeps track of all food locations), thereby spawning a new food.

Detecting collision related events is done by comparing the snake head's x-y coordinate against the coordinates of the wall, all food locations, and all snake body locations. If the snake head location is equal to any of the coordinates mentioned above, then a collision is detected

snake_top:
A top level module that connects signals across snake_body and collision. It receives as inputs the input direction signal, the go_signal, and new food locations food_x and food_y, and outputs a payload that includes information about size, snake position, and whether a food has been eaten.

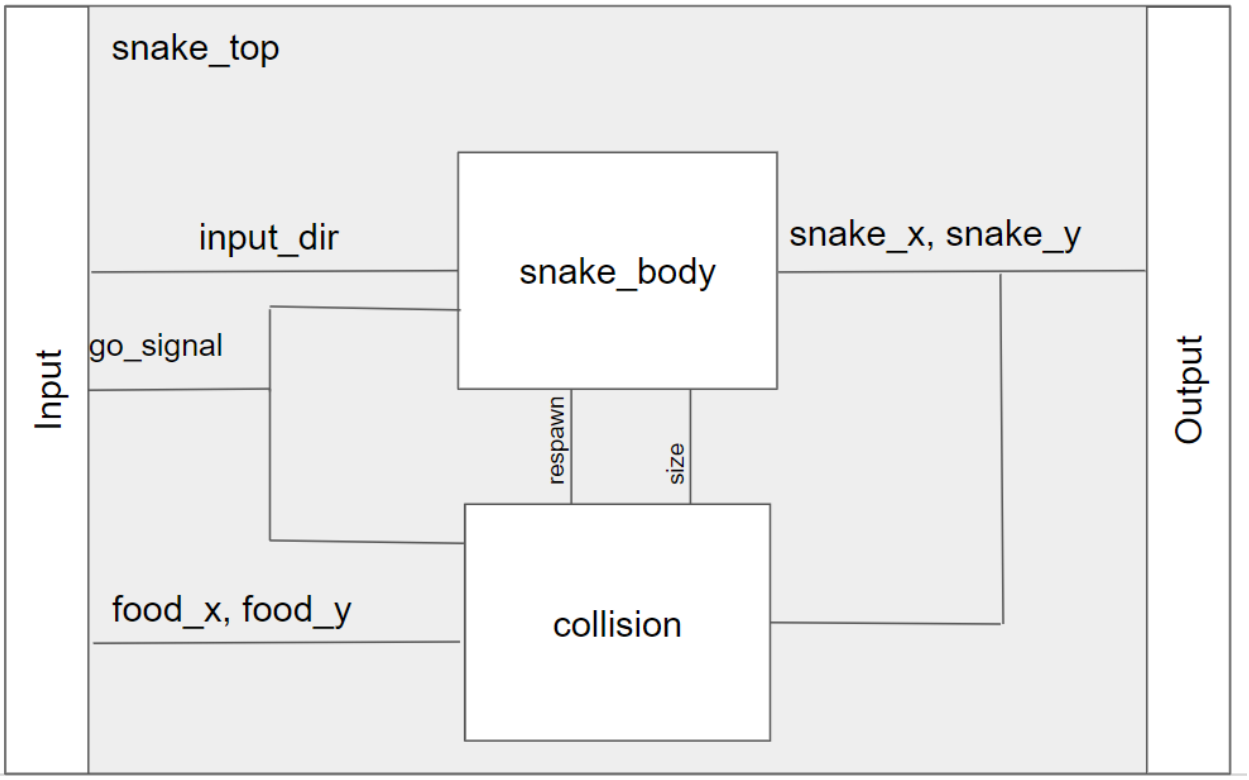

Figure 3. snake_top block diagram

4.2 Snake Game AXI Protocol IP

This module has 3 main parts, which are snake_top, slave_sub_ip, and M_AXI Interface.

S_AXI_Interface + Slave Sub-IP:

The slave_sub_ip module was initially an S_AXI Interface IP that was taken from Vivado's IP Catalog. The module was configured to include 2 slave registers, which will be used to receive the data from C-code (Microblaze) through the AXI4Lite protocol, and pass as input to snake_top module (go_signal, food_x, food_y, input_dir). To do this, small modifications were also made to this IP:

1. The slave_register that contains the go_signal had its logic slightly changed, such that upon setting go_signal to 1, it would be de-asserted immediately the next clock cycle. This is required so that the go_signal is pulsed, and thus will only update one frame of the game for each go_signal sent from software.
2. The slave registers that came from Vivado's library were set as 'reg'. Thus, the declaration of the registers needs to be changed to 'output reg', such that the signal can be passed into the snake_top module as inputs.

M_AXI Interface:

The M_AXI Interface is used to write the output payload from the snake_top module into the BRAM, where it will be read using C-code and used to display the game state into the VGA screen. The M_AXI interface logic is adopted from Vivado's M_AXI Interface from the IP Catalog. However, several modifications were required to fit our needs:

1. Since our M_AXI only needs to support writes, a slight modification was made to remove all read logic/signals (e.g. RDATA, ARVALID, ARREADY, RADDR).

2. Each snake requires ~1000 bits of data to be written into BRAM per frame((8 bits for x coordinate + 7 for y coordinate) * 64 bits for max_snake length + overhead bits for other game states). However, since AXI4Lite only supports a data width of 32 bits, a custom FSM logic is required to split the 960 bits into multiple transactions. Each transaction will thus have the following contents:

   The very first transaction will only include information about the size of the snake

unused[31:12]	size_2[11:6]	size_1[5:0]

   The second transaction only includes 2 food_valid bits, which indicates whether each snake ate a food in the current game frame

unused[31:2]	food_valid[1]	food_valid[0]

The next 64 transactions will include snake 1 and snake 2's locations.

unused[31:30]	snake_2_y[30:23]	snake_2_x[22:16]	snake_1_x [15:7]	snake_1_y[6:0]

Each of the above transactions are written into its own 4 bytes in the BRAM. Thus, the WADDR signal is also incremented by an offset of 4 after every transaction.

snake_top:

The snake_top IP described previously is instantiated here inside the snake_game_axi module. The slave registers inside the slave_sub_IP module is connected to snake_top as inputs, and the output payload is connected to the WDATA wire of the M_AXI Interface

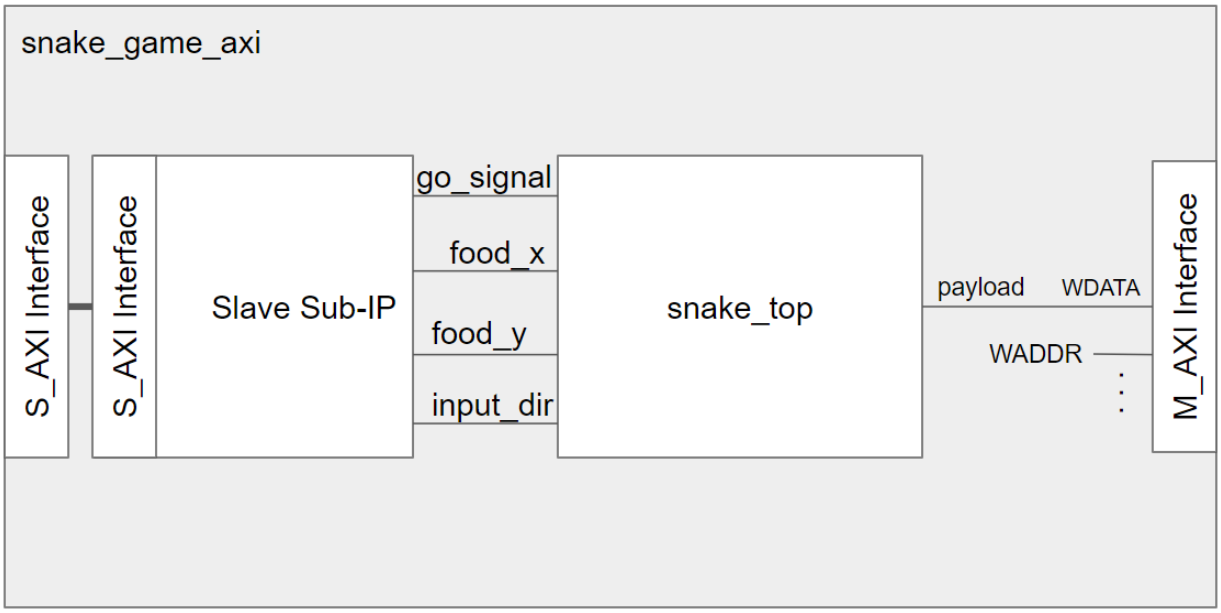

Figure 4. Block diagram of snake_game_axi module

4.3 Snake Game Client (hardware run on FPGA)

Snake Game Client is the hardware logic running on the FPGA that allows users to interact with the game. It is a top level module that contains buttons to collect user inputs, VGA to display the game image, Ethernet and Microblaze to handle network communications as well as the Snake Game IPs that process the game itself.

4.4 VGA Controller (software run on Microblaze)

This is the controller software that performs all functions for drawing to the monitor through VGA. This software module was built upon the VGA tutorial and sample code provided by tutorials. Its responsibilities are to:

- Draw and colour pixels on the monitor
    - Game hardware writes to BRAM the game information (described in section 4.2).
    - Snake Game Client reads from the BRAM and writes to DDR the features that need to be drawn
    - Values stored in DDR are read by the TFT controller (hardware module) and output through VGA
- Supersize pixels to align with the game's resolution
    - Pixel size is defined as a global variable that can be changed to create the resolution needed
    - The current pixel size is 16x the original, and monitor resolution is 640x480, so resolution of the game would be 160x120

4.5 TCP Server (software run on Microblaze)

TCP Server is adapted from the Echo Server C files provided in previous tutorials. Since the server needs to keep track of which connection corresponds to which receiver, upon accepting a connection, it stores the new Protocol Control Blocks (PCB) to a separate variable. By doing so, once the server receives a packet, it can identify whether the sender is a player or the food generator, and decodes the packet accordingly.
The server also has a counter which is used as a timer to broadcast the latest user input and food locations to all of the players. Once the set time is reached, the server will package these pieces of information into a single string and send it to all of the players.

The packet sent/received from the server is in the form of concatenated strings, delimited by a colon. Below are the protocols for different types of messages:

- Sent from player to server:
    - <player_dir:food_valid>
- Sent from server to player:
    - <player_1_dir:player_2_dir:food_loc_1_x:food_loc_1_y:food_loc_2_x:food_loc_2_y:reset>
- Sent from Food Generator to server:
    - <food_loc_1_x:food_loc_1_y:food_loc_2_x:food_loc_2_y>

4.6 TCP Client (software run on Microblaze)

TCP Client is adapted from the TCP client C files provided in previous tutorials. It first establishes connection with the server, and then enters an infinite loop where it sends to server whenever the user input changes or a food has been eaten, as well as receives packets from server and processes it.

Within the TCP Client, additional software IPs are added to complete the functionality we require.

- **Pixel Map Updater:** Using the microblaze, we will read the game state directly from the BRAM which contains size of snakes, a food valid signal, and the positions of both snakes. Using this information, we can draw the snake using our functions to write to VGA.
- **Client Encoder:** Upon user input, the client will encode a packet with the new user input and send the packet to the server to represent a new direction in movement by the snake. In addition, the food valid bit may be 0 or 1 depending on whether hardware has detected a recently eaten food. Refer to TCP Server to see the message that will be sent from client to server.
- **Client Decoder:** Upon receiving the server packet, the client will decode and write to slave registers the data that was contained within the server packet such as the go_signal, player_dir, new_food_locations

4.7 Food Generator (software run on PC)

The Food Generator is adapted from the TCP client python script provided in previous tutorials. We chose to put it on the PC because it is more capable of handling random processes. The food generator first tries to connect with the server running on an FPGA, and goes into an infinite loop where it listens to the server. Every time it receives a packet, it will generate two sets of random food locations bounded by the size of our game map. Then it will concatenate them into a single packet and send it to the server.

5.0 Description of Your Design Tree

Github: https://github.com/michaeltfchan/Slitherin_FPGA
The tree consists mainly of two parts, which are the ‘client’ and ‘server’ folders. Each folder has its own subfolders, described below:

Client:
The client folder is located under **snake_game_project/client/,** and contains the following sub-directories:

- **VGA_mar_7_work:** This ridiculously named folder includes the main snake_game_client hardware. The VGA.xpr file is the main Vivado project, which is where the block diagram exists, and where you would launch the SDK project from.

The software code is located under this same folder, in the following directory: **VGA/VGA.sdk/snake_game_client_2_player/src/.** This is where all software code resides, which includes code for VGA drawing, as well as server-client communication. Below are the main files of interest:

  - main.c: The main file that contains software logic including
    - Client-server communication
    - Parsing inputs from server
    - Passing inputs into hardware IP through slave registers
    - Reading game state from BRAM, and updating VGA display based on read game state
  - vga.c: Contains all VGA-related functions used in main.c
  - vga.h: Header file for vga.c, contains function prototypes

- **snake_game_axi4_full_folder:** This folder contains the files related to the snake_game_axi module.
  - snake_game_axi4_full: A subfolder that contains the block diagram for this module
  - snake_game_slave_axi_folder: A sub folder that contains the block diagram for the slave_sub_IP, which has slave registers used by the game client to receive inputs. This folder also has another sub-folder named “snake_game_slave_axi_ip” that contains the IP repository of slave_sub_IP.
  - Snake_game_2_player_final_demo_food_fix: Another ridiculously named folder that contains the IP of the snake_game_axi module. This IP repository will be referenced by the main snake_game_client module (in VGA_mar_7_work)

- **snake_game_files:** This folder contains all custom Verilog source code required for the game client IP.

- **snake_top:** This folder contains the Vivado project for the snake_top module (the main game logic IP).

- **snake_game_ip:** This is where the game logic IP (generated from snake_top above) is located.

Server:
Within the server, it contains our server vivado project as well as the food generator that will run on the PC. This parent directory of the server is in: **snake_game_project/server/**

- **lab_demo_2**: This folder contains the vivado project that has our server. It also contains the software files to run the server under **lab_demo_2.sdk/tcp_server/src/**
- **main.c:** This file contains the logic to setup the TCP server and handle the communication between clients and servers. The server is responsible for sending and receiving packets that may alter the game states. It decodes the input direction from each client and also encodes the packet that will be sent periodically to the clients. In addition, upon seeing the food_valid bit is high from a client, it will send a request to the food generator and receive a response back containing the new food locations that will be sent back to each client.
- **food_gen.py:** This python script is used to generate food randomly and has a TCP connection to the server.

How to run our Project
To run our project, you will require 3 DELS-A machines connected to the Xilinx Nexys 4 DDR4 FPGA boards as well as connection to the FPGANet in the labs. Also run ipconfig on command prompt to identify the IP address of the machines. Make note of these IP addresses as they will be critical in connecting the machines in a later step.

Initial Client Setup (2 machines):
To launch the clients, we will need to open the client project from the **snake_game_project/client/VGA_mar_7_work/VGA** directory and launch the vivado project called VGA.xpr. After vivado is launched, launch the SDK. We also need to open 'hardware manager' on Vivado, program the device with the generated bitstream, and launch the VIO. This VIO will be used to reset the hardware game logic. Go back to SDK, locate the main.c file, change the IP addresses to match the server IP address on **lines 70-78 of main.c** and also match the IP address of the source machine (current machine that client is on). For more information on IP addresses required, refer to documents in the course page:

ECE532_DESL_Network_Connectivity_updated.pdf.

Then, program the hardware, and go back to the VIO to reset the hardware by toggling the wire 1 -> 0 -> 1. Repeat for machine 2. Do not run the software on SDK for the clients yet. Now this is where we move onto the server.

Server setup (1 machine):
To launch the server, open the vivado project (ridiculously) called lab_demo_2.xpr in the directory **snake_game_project/server/lab_demo_2/**. Open hardware manager on vivado, program the device with the generated bitstream, and launch VIO as well. This VIO will control the restart of the game and send the synchronizing go_signals that run the game. Launch SDK, configure IP addresses similar to client setup in main.c in **lines 134, 163, 165, 219 and 221, configure the port # in echo.c line 208**, program the hardware, connect to com6 with baud rate 9600 on the SDK terminal, and run the software using run configurations. Ensure the software application is selected properly. Now, launch food_generator python script. If the connection is established correctly, the python script will print “here”.

Connection between Clients and Server and Game Start
With the server setup, we can go back to the clients and connect to the server. Connect to com6 with baud rate 9600 on the SDK terminal. Then, run the software on SDK for the clients using run configurations. Ensure the correct application is selected. From here, we can see in the SDK terminal the TCP communication setup being printed. We can move onto the VGA screen now. After the client and server are connected properly, we should see the map of the game drawn as well as the initial food. We can start the game by toggling the VIO on the server side from 1 -> 0 which will start sending the go_signals. Players can now use the push buttons. The game will continue indefinitely unless restarted by toggling the VIO from 0 -> 1 -> 0.

6.0 Tips and Tricks

- When using DESL machines, work on the C drive and save the zipped work folder on Google Drive. It’s faster than copying from and to the W drive.
- Have sufficient time if you have connecting external peripherals to the board, e.g. sensors or microcontrollers.
- When in doubt, close everything and reopen as Vivado can be quite buggy to work with.
- Run Elaboration if some changes did not appear to be implemented.
- Synthesizing/Implementing in Vivado will take a long time! Try your best to make each iteration of the hardware thorough and efficient.