

6.2050 Final Project Report: SystemVeriVipers

1st Riley Davis

*Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
rileywd@mit.edu*

2st Matthew De Jesus

*Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
dejesus@mit.edu*

Abstract—This project implements a distributed multi-player snake style arcade game rendered across several FPGAs. Each client board is responsible for aggregating inputs for their respective players and for rendering all assets onto their screen. Up to 4 Joysticks can be connected to one FPGA board. One host board is responsible for collecting inputs, game advancement, curve-smoothing and transmitting game state to peripheral boards.

Index Terms—FPGA, digital systems, SystemVerilog, multi-player game, hardware acceleration, custom protocol

I. INTRODUCTION & PROBLEM SUMMARY

A. Introduction

Slither is an online multiplayer game that demonstrates unique challenges in scaling as well as synchronized game states. It can be used as a vehicle for demonstrating the exact kinds of problems FPGAs and digital systems are well equipped for: deterministic real-time control, parallel evaluation of game rules, low latency rendering of many overlapping sprites, and coordinating shared game state across distributed hardware over a custom inter-board bus protocol.

We decided on a host-client game setup to imitate Slither.IO servers and distributed players. The host board is responsible for game state initialization, updates and protocol control. The client board is responsible for reconstructing the game state, rendering assets (apples and snake cells) and replying to the host with player headings.

B. Communication

We sought to implement the host-client protocol over three wires, clock, Host-in and Host-out. Unlike SPI this protocol does not have a "Chip Select" this was chosen to allow daisy chaining of FPGAs and ensure that the number of clients could scale. Since no "Chip Select" is present, selecting a player occurs through decoding the host-out signal. Where all players can read the request and only the selected player responds on the Host-in line. Communication is fully synchronous with the host driving the global clock, ensuring consistent sampling at each client. The interface supports clock frequencies up to 25–40 MHz depending on cable length. Regardless, there are airlock latches on all inputs to prevent meta stability.

C. Host-Client

Challenges arise when using host-client setup. Since no game updates are done on the client, it doesn't store the game

state. This means the host board has to communicate the entire game state to all of the clients every frame. No two clients can respond on the host-in line at the same time, so the host has to meticulously cycle through each client telling them to respond. Additionally, since there is one host-out wire, the game board has to be serialized to broadcast, coupled with a 10MHz clock for data integrity, communication throughput is slowed significantly.

D. Project Checklist

We meet our commitments:

- Demonstrate Snake can be rendered on client
- Demonstrate 30bit packages module
- Clients will be able to receive updates from the host

Including, our goals of:

- Host will communicate game state, kills, food and new snake positions
- Client draws snakes on the screen (be it simple blocks)
- Input will be two buttons (left and right)

We did not meet our goals of:

- Demonstrate that clients will respond with player input when called upon
- Snake will be rendered in discrete arcs
- Snakes will have their own patterns

or our stretch goals of:

- Snake has drag and stiffness
- Use joystick to support velocity and continuous angles using a ADC
- Powerups
- Obstacles
- Bots

II. SYSTEM OVERVIEW

SystemVeriVipers implements a multiplayer Slither.IO game using a distributed host-client bus. One FPGA acts as the host and manages game state, while the client(s) FPGA respond with player inputs, reconstruct the game state, and render the game. All boards communicate using a custom three-wire serial protocol.

A. High-Level Architecture

The main components are:

- **Host FPGA:** Runs the game physics engine, polls each client for directional input, updates snake states, manages apple spawning, and broadcasts the complete world state every tick.
- **Client FPGA(s):** Captures joystick or button inputs and return `CMD_INPUT_DATA` to the host. Clients also reconstructs the entire game state from the broadcast packets.
- **Serial Bus:** Made of a transmitter and receiver module, the bus is able to send data bi-directionally at 25 MHz.
- **Game Physics Pipeline:** Updates head positions, manages growth, collisions, and performs follow-the-leader (FTL) smoothing.
- **Rendering Pipeline:** A sprite-based renderer that converts each snake's body coordinates into a sprite to be sent over HDMI.
- **Snake Storage:** Each snake's body is stored in a dedicated register indexed by segment number.

This architecture separates physics, communication, and rendering, allowing for modularity and scalability.

B. Data Types and Definitions

All shared types are defined in `slither_pkg.sv`. Important types include:

- **cmd_t:** Lists all packet-level commands, including initialization, polling, acknowledgement, and body/food updates.
- **packet_t:** Packed struct used in serial communication including control bits, command opcode, target ID, and data payload.
- **pos_t and body_cell_t:** Represent (x, y) positions on the grid for snake heads, body segments and apples.
- **Player State Struct:** Combines head position, direction, body-cell array, length, and alive/active flags in a compact format.

These consistent typed structs allow the host and client to interpret game state identically and increase readability.

C. System Behaviors

The host FSM controls the game through four sub-FSMs

- **Initialization:** The host sends `CMD_INIT` and verifies clients through a PING/ACK handshake.
- **Input Polling:** Each tick, the host issues `CMD_POLL_INPUT`, and clients respond with directional data.
- **Physics Update:** The host updates all snake positions, manages growth, updates apples, and checks collisions.
- **State Broadcast:** The host streams new head positions (`CMD_GAME_STATE`) followed by body data using `CMD_BODY_START`, `CMD_BODY_DATA`, and `CMD_BODY_END`.
- **Client Reconstruction:** Clients rebuild all snakes in local memory for rendering.

This behavior ensures that all FPGAs have a consistent game board.

III. DETAILED DESIGN EXPLANATION

A. Data Types & Definitions

Our custom data types were key in completing this game, our host-client uses a variable width protocol that is tuned by the games constants such as number of players, custom commands and game resolution. For the example *Figure 2* we will demonstrate the protocol with 8 players, 16 possible custom commands *Table 1* and a resolution of 720p.

```
//Type Definitions
// Length of snake body
typedef logic [BODY_LEN_WIDTH-1:0] body_len_t;
// Heading direction 0-360 degrees
typedef logic [HEADING_WIDTH-1:0] heading_t;
// Player identifier
typedef logic [TARGET_WIDTH-1:0] player_id_t;
// Body cell array storage
typedef logic [MAX_BODY_DEPTH-1:0][DATA_WIDTH-1:0]
    body_t;

// Position/Cell Structure
typedef struct packed {
    logic [X_POS_WIDTH-1:0] x; //0-1280
    logic [Y_POS_WIDTH-1:0] y; //0-720
} cell_t;
```

```
// Packet Structure
typedef struct packed {
    logic [1:0] start_bits;
    cmd_t cmd;
    player_id_t target_id;
    logic [DATA_WIDTH-1:0] data;
    logic [1:0] stop_bits;
} packet_t;
```

```
// Snake State Structure
typedef struct packed {
    cell_t head_pos;
    heading_t heading;
    body_len_t length;
    logic active;
    logic alive;
    body_t body_cells;
} snake_t;
```



Fig. 1. Custom 3-wire packet format

1) *Serial Modules:* Receiving, Transmitting, Encoding, Decoding modules all work together to serialize, parallelize data and perform data health checks. These modules allow for a variable package length and ensures commands are understood.

B. Host Architecture

1) *Host Controller Overview:* The host controls the game loop. It coordinates four major FSMs: the initialization FSM, the polling FSM, the physics FSM and the broadcast FSM.

TABLE I
HOST-CLIENT COMMAND SET

Command	Code	Direction	Data Format	Purpose
INIT	0x0	Host → All	0x0000	Initialize all players; start discovery
PING	0x1	Host → Player	0x0000	Check if player is present
ACK	0x2	Player → Host	[2:0] = ID bits	Acknowledge host
SET POS	0x3	Host → Player	[15:8] = X [7:0] = Y	Set initial position
START GAME	0x4	Host → All	0x0000	Begin game loop
POLL INPUT	0x5	Host → Player	0x0000	Request joystick input
PLAYER INPUT	0x6	Player → Host	[16] = Button [15:8] = X [7:0] = Y	Send input to host
GAME STATE	0x7	Host → Player	[15:8] = X [7:0] = Y	Update XY after tick
BODY START	0x8	Host → Player	[7:0] = Length	Begin streaming body cells for player
BODY DATA	0x9	Host → Player	[15:8] = X [7:0] = Y	Body cell data (one segment)
BODY END	0xA	Host → Player	0x0000	End body stream for player
APPLE	0xB	Host → Player	[15:8] = X [7:0] = Y	Apple Cell Location

The host owns all game physics computation and updates its snake states each tick. A global tick counter determines when to poll inputs, when to run movement logic, and when to begin broadcasting updated state.

2) *Host Initialization FSM*: The initialization FSM sends out an INIT command to clear all of the clients memory, then it sequentially PINGS each player and waits for a response or timeout. After responses are collected it makes notes of active player and sets the starting positions for each snake. It then sends a game START command.

3) *Host Poll FSM*: The poll FSM rotates through each player sending INPUT_DATA and waits for a response. If there is no response it maintains the snake will maintain the previous heading. This player inputs are then passed to the physics module.

4) *Host Physics FSM*: The Physics FSM maintains the global game state, owns the snake representations and updates snakes and check for collisions and eaten apples.

C. Movement Logic

The physics module uses a heading direction calculated from button input to determine the direction of a step. This is done using a sine and cosine value lookup table to avoid excess computation and critical path delay. New x locations are determined by adding the signed location to the signed value of the step size multiplied by the cosine of the angle. New y locations are determined by adding the signed location to the signed value of the step size multiplied by the sine

of the angle. This ensures that within a margin of error each total step size remains the value of the STEP parameter while allowing movement beyond a rigid grid.

D. Collisions

We implement a two stage pipeline for path delay reduction and collision calculation. In the first stage, new positions for the heads of snakes are calculated based upon the aforementioned movement logic. These candidate head positions are stored in DRAM in head_x_latched[p] and head_y_latched[p]. This is then fed to the second stage which actually calculates collisions.

For each update, the module calculates all $\text{NUM_PLAYERS} \times \text{NUM_APPLES}$ and $\text{NUM_PLAYERS} \times (\text{NUM_PLAYERS} \times \text{PLAYER_LENGTH})$ collisions for snake-apple and snake-snake collisions respectively. The module checks for snake head locations within 8 pixels, the size of the apples, of the apple locations and the same for snake body cells.

E. Apple Collisions

If a collision is detected with an apple, the apple collision logic executes. In this case, the player length is incremented by 1 and a new pseudo-random apple location is chosen. Pseudo-random apple locations are determined using a 16-bit LFSR.

F. Snake Collisions

Snake collisions result in the alive state of the snake who owns the head involved in the collision changing to zero and the snake disappears from the screen.

1) *Host Broadcast FSM*: Once physics updates complete, the broadcast FSM streams the updated world state. It tells the clients its about to send the game state with a GAME_STATE command and rotates through each player sending one cell at a time with BODY_START, CMD_BODY_DATA and concludes with CMD_BODY_END.

G. Rendering Pipeline

1) Multiple Sprite Engine:

Rendering is accomplished with parallel sprite renderings that are placed into a frame buffer that is then output to the display. Each instance of a player has its own multiple_sprite module to handle rendering of all of that player's sprites. This means that each player can have unique patterns. Another instance of multiple_sprite is used to render the apples.

IV. EVALUATION

This section evaluates the performance of SystemVeriVipers with respect to latency, throughput, resource utilization, and design scalability. All results use highly conservative timing assumptions to ensure that the true system performance always exceeds the estimates provided here.

A. Latency and Throughput

1) *Serial Transmission Time*: Each tick of the game requires the host to poll every client, receive input, and broadcast the complete world state. For a player with an average snake length L , the system transmits approximately $L+5$ packets per tick: two for input polling (host request and client response), and $L+3$ for head and body broadcast data. With N players, the total packet count per tick is

$$P_{\text{total}} = N(L + 5).$$

In our design, each packet is conservatively assumed to be **32 bits**, despite the actual packet format often producing shorter messages. Given a **25 MHz serial transmission rate**, the time to send a single packet is

$$t_{\text{pkt}} = \frac{32}{25 \times 10^6} \approx 1.28 \mu\text{s}.$$

The total serial time per tick is therefore

$$T_{\text{serial}} = P_{\text{total}} \cdot t_{\text{pkt}} = N(L + 5) \cdot 1.28 \mu\text{s}.$$

Using the most conservative configuration supported by the system, **7 players with a maximum body depth of 30 segments**, the serial subsystem transmits

$$P_{\text{total}} = 7(30 + 5) = 245 \text{ packets},$$

requiring

$$T_{\text{serial}} \approx 245 \cdot 1.28 \mu\text{s} = 313.6 \mu\text{s}.$$

Even in this worst case, serial transmission consumes only

$$\frac{0.314 \text{ ms}}{16.67 \text{ ms}} \approx 1.9\%$$

of the tick interval at a 60 Hz update rate. This demonstrates that the serial bus is operating well within timing margins and is not a bottleneck for the design.

2) *Physics Update Time*: The host computes all physics updates, including movement, growth, collision checks, and body shifting. The cost is approximately linear in the total number of body segments, NL . Let c_0 be the fixed per-tick overhead and c_{seg} the per-segment computational cost. To remain conservative, we assume

$$c_0 = 200 \text{ cycles}, \quad c_{\text{seg}} = 50 \text{ cycles/segment},$$

even though real measurements are significantly lower. For a 100 MHz host clock and worst-case configuration $N = 7$, $L = 30$, the physics engine performs

$$\text{cycles} = c_0 + c_{\text{seg}}NL = 200 + 50(7 \cdot 30) = 10,700 \text{ cycles}.$$

Thus,

$$T_{\text{physics}} = \frac{10,700}{100 \times 10^6} = 107 \mu\text{s}.$$

This is less than **0.65%** of the 60 Hz tick interval.

3) *Combined Latency*: Adding the conservative serial and physics estimates,

$$T_{\text{total}} = T_{\text{serial}} + T_{\text{physics}} \approx 313.6 \mu\text{s} + 107 \mu\text{s} = 420.6 \mu\text{s}.$$

This represents only

$$\frac{0.421 \text{ ms}}{16.67 \text{ ms}} \approx 2.5\%$$

of the available tick time. Therefore, even under extremely pessimistic assumptions, the system has more than an order of magnitude timing headroom.

B. Scalability Limits

Using the same conservative model, we can estimate the theoretical upper bounds on player count and snake length before the system would saturate its tick interval. Setting $T_{\text{total}} = T_{\text{tick}}$, solving for L gives

$$L_{\text{max}} \approx \frac{T_{\text{tick}} - \left(N \cdot 5 t_{\text{pkt}} + \frac{c_0}{f_{\text{clk}}}\right)}{N \left(t_{\text{pkt}} + \frac{c_{\text{seg}}}{f_{\text{clk}}}\right)}.$$

Even with conservative cycle costs and packet sizes, the theoretical time limit for a 7-player system at 60 Hz is approximately

$$L_{\text{max}} \approx 1300 \text{ segments per player},$$

far above the design's configured maximum of 30. Alternatively, with $L = 30$, the time-limited maximum number of players exceeds

$$N_{\text{max}} > 250,$$

again much greater than the architectural constraints of the design ($\text{MAX_PLAYERS} = 7$). This confirms that the system is not limited by serial or computational throughput.

V. CHALLENGES

A. Analog Joystick Inputs

The FPGA boards have no native Analog ports for Joystick Input, so we opted to use a 8 Channel ADC. Since each Joystick had an X and Y analog value, one joystick took two channels, this results in one ADC supporting four joysticks. Schematics were made *Figure 1*, the ADC required *inout*, ports with a shared addressing and data bus. We neglected the fact that the outputs of the ADC were 5V but the FPGA ports were 3.3V luckily we were able to get a bi-directional level shifter and convert them. Despite this, we were not able to

get a consistent reading on the output; it constantly flickered and did not properly latch to valid values. We tried to remedy this with an ADC and level-shifter bypass caps (We thought it was a noise issue), to no avail, *Appendix I*. After this we tried using the onboard Joystick port, but couldn't find the pins in the FPGA schematic to hook up to the XADC IP. We were able to connect the Joystick to the XADC but didn't work after two hours of debugging. We settled on using buttons to control the snakes left and right movements.

B. Custom Struct Test-benching

Custom structs significantly improved the modularity and readability of the design allowing a uniform package import to be created. While this allowed quicker programming, test benching our work became tirelessly complicated. Cocotb and Vicoco did interpreted packed structs indexing as one-bit values, switching to packed structs was unsupported. This made test-benching take roughly 75% of our time since we had to manually pack to write and unpack to read struct vectors in testbenches.

C. Lessons Learned

Our overall project workflow would have been greatly improved if we utilized Git better, since we each were able to work independently (client or host). The warrant to push to Git wasn't as strong as if we were working together on tightly dependent files. In the future, more time should be put into the ground work, custom types, sub-fsms. While we did end up with a very modular design we wasted a lot of time, trying to make more cumbersome designs work before rebuilding our framework. There were so many points in the project where we made progress further than what we are now, but didn't push to git. For example the logic analyzer was able to confirm that we had a working host-initialization sequence, but we did not screenshot it, made edits and were not able to return to that point.

and muxes to allow for multiple joystick channels. Ultimately, there were issues with the ADCs timing and the data retrieved was unstable and therefore unusable.

VI. RESOURCE UTILIZATION

TABLE II
RESOURCE UTILIZATION

Resource	Used	Available	Utilization (%)
Slice LUTs	16034	32600	49.18
LUT as Logic	15986	32600	49.04
LUT as Memory	48	9600	0.50
Slice Registers	2526	65200	3.87
Slices	5549	8150	68.09
Block RAM tiles	36	75	48.00
DSP blocks	0	120	0.00
Bonded IOBs	58	210	27.62
BUFGCTRL	3	32	9.38
MMCME2_ADV	1	5	20.00

$$\text{WNS} = 0.244$$

VII. CONTRIBUTIONS

Matthew worked on the the protocol framework, the custom structs host-side modules. He planned out four sub-FSMs controlled by the Host FSM. Matthew also created schematics for wiring a physical ADC to the FPGA using a level shifter

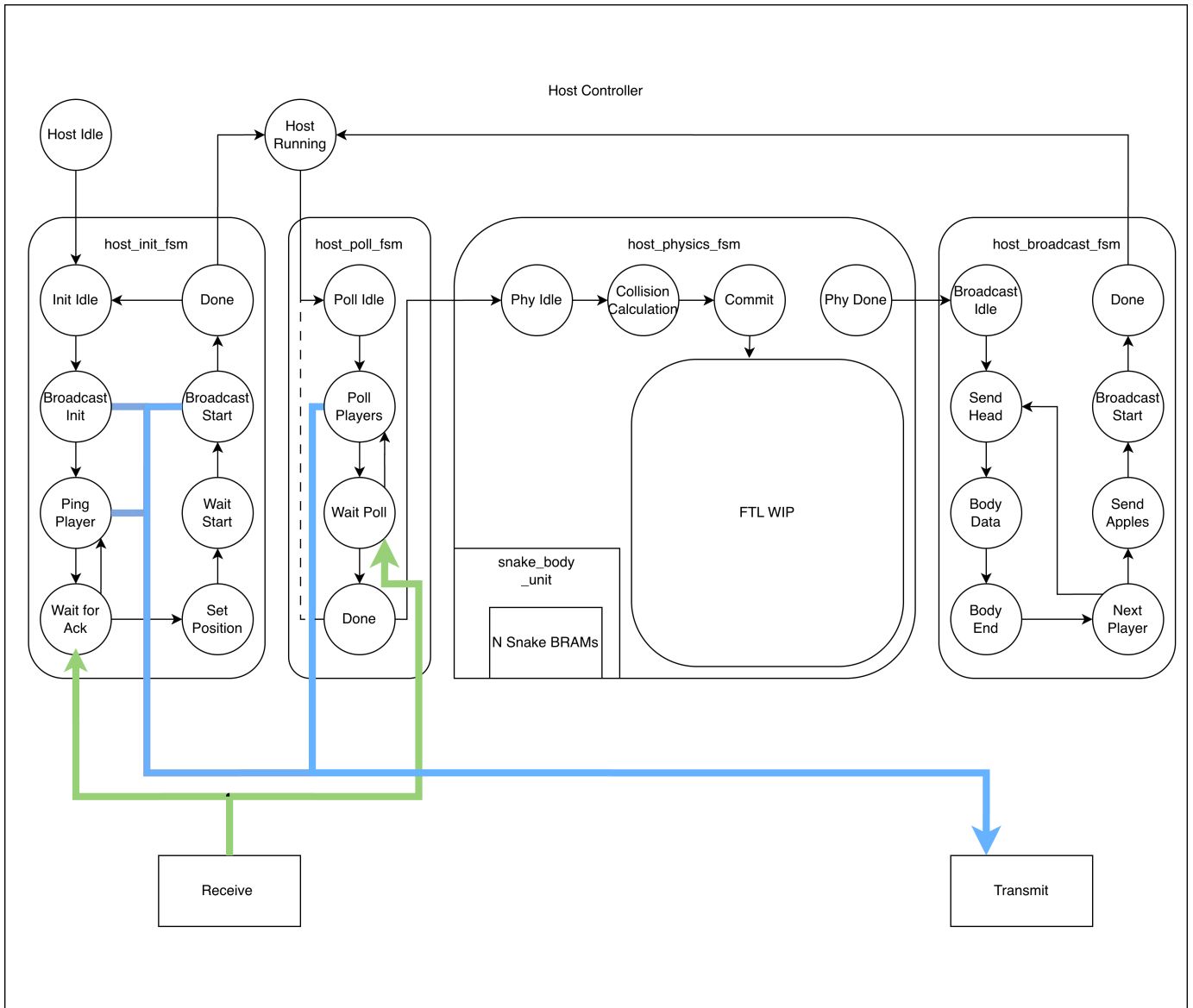


Fig. 2. One Main-FSM maintains Four Sub-FSMs

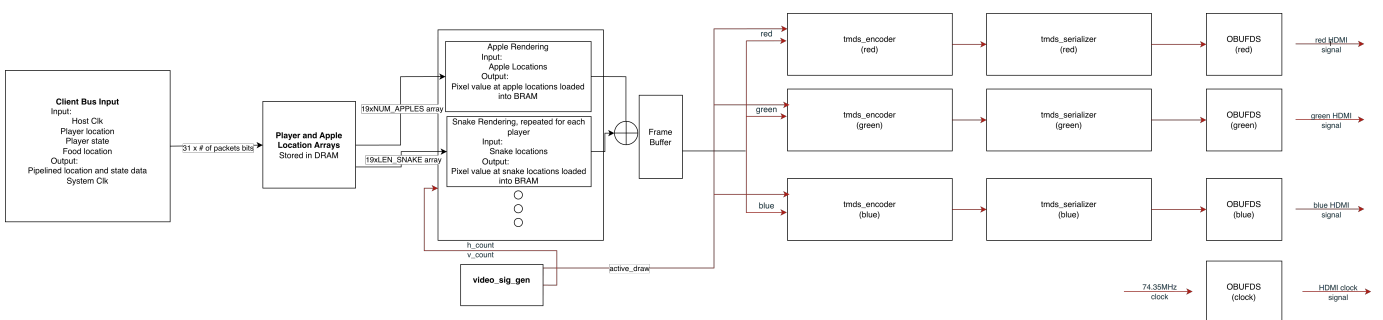


Fig. 3. Client FSM Diagram