1. Game Overview - https://youtu.be/d-PNc1pg5Ww

"Slash Man" is a real-time multiplayer game where two players can engage in a duel using TCP/UDP connection architecture. The system uses two FPGA-based controller nodes, translating physical movements into in-game sword manoeuvres and player movement. Players must balance their stamina while depleting their opponent's health. The system also incorporates a persistent leaderboard mechanism that tracks player performance across multiple game sessions and records the highest individual kills/scores.

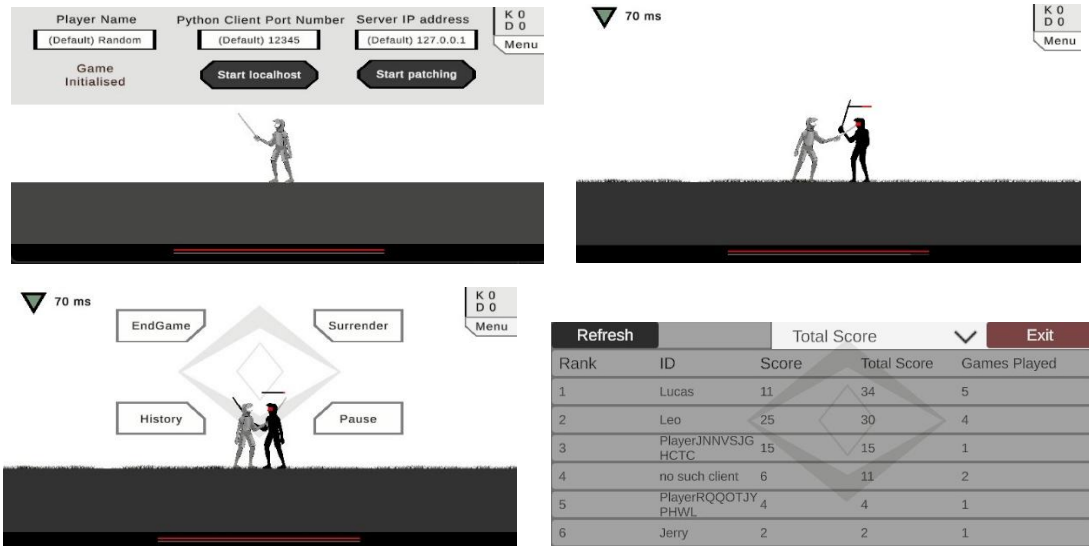

Figure 1. Slash Man UI

2. Overall System Architecture

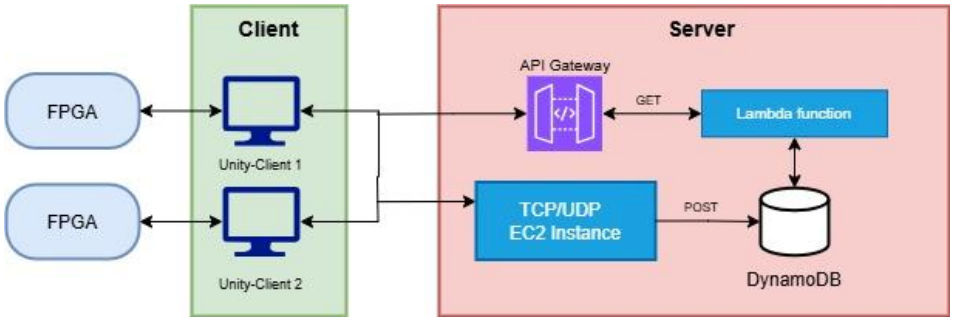

Figure 2. System Architecture Diagram

Functional Requirements:

- ✓ Local processing of the accelerometer data
- ✓ Establishing a cloud server to process events/information
  - Completed: The game server is hosted on AWS, allowing multiplayer online access.
- ✓ Communicating information from the node to the server.
  - Completed: The gaming movement is being sent from the node to the server: first starting as a pure acceleration value or button data, then being processed as character movements in the client device then being relayed to the server
- ✓ Communicating information from the server back to the nodes in way that the local processing can be impacted.
  - Completed: The player's health and position are information sent from the server back to the node. The player position is used by the client and the health feeds back to the FPGA to be displayed as a health bar
- ✓ Use of two nodes

3. FPGA Nodes

The FPGA framework architecture can be divided into 2 sections: Accelerometer Data Processing and CPU-to-Client communication. This structure utilizes parallel processing which aims to **minimize the Nios II softcore processor workload** and **increase data throughput** by focusing the CPU task on the feedback communication. With the accelerometer data fetching and filtering, the process can utilize a lot of the CPU resources so as a result, a separate hardware module has been constructed to perform all the real-time SPI communication with the ADXL345 accelerometer and filter the signal with a low-pass FIR.

The setup requires the use of a **Phased-Lock-Loop** (PLL) clock generator, particularly to maintain the pulses for SPI. The SPI communication involves an independent setup handshake with the accelerometer and a loop that continuously communicates with the accelerometer for data at a predefined update frequency. After each set of data read (all 3 axes), a data update pulse is signalled which shifts the signal into hardware filter modules that compute their corresponding axes value. To connect the filtered data with the DE-10, a custom peripheral is initiated as an Avalon slave with accessible buffer registers to connect to the Avalon bus. With this, the CPU simply needs to read the final x, y, and z data from the designated register location. During the construction, modular testing on the hardware modules was conducted by evaluating the waveform and utilizing the signal tap logic analyser to test the CPU system.

To achieve satisfactory results in our key metrics: **Latency, Signal Smoothing and Accuracy,** we must critically tune our key parameters. Latency is critical for coherent synchronization, signal smoothing maintains stability, ensuring that only our genuine motion is interpreted and accurate to help with precise decision-making.

With the maximum possible accelerometer sampling rate at 3200, we tried testing at that speed; however, the signal started experiencing too much noise, causing us to set the rate to 1600 which achieved great accuracy and responsiveness. The speed also matches the communication rate between the FPGA and the client (laptop) which averages around 2000 (tested using Python iteration counter). Though a faster communicate rate than the accelerometer frequency means a marginal oversampling, this is not a problem since the data has been filtered. In terms of latency, most of the delay will be accumulated during the FIR filter.

$$Delay = \frac{N_{Tap} - 1}{2} \cdot \frac{1}{f_s}$$

Using the equation and a max response delay of 20ms (Fast Pace Game), we set up the number of taps (N_{Tap}) to 64. With Matlab, we generate an FIR filter with the following coefficients: Taps – 64, Passband – 25Hz, Stopband – 50Hz.

Table 1. FPGA Key Parameters & Values

Key Parameters	Value
CLK FREQUENCY	25_000_000Hz
SPI FREQUENCY	2_000_000Hz
ACCELEROMETER FREQ	1600Hz
COMMUNICATION RATE	≈ 2000 Hz
F_{MAX} Clock Speed	107.27MHz
RESPONSE DELAY	20ms
TRANSFER DELAY	1.8ms

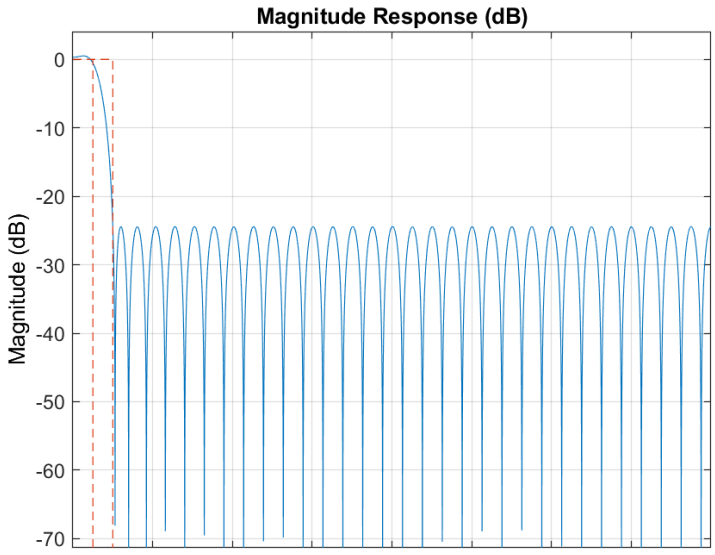

Figure 3. FIR Filter

4. Client

4.1 Transferring And Processing Data From Server To FPGA

The program utilises multithreading using the threading library to receive data from the server without interrupting the main loop. This is so we can ensure that all functionalities of the Python code do not interrupt each other, allowing responsive interactions. The subprocess involves receiving the player health data from the server, so that we can send a character back to the FPGA to inform it of how many LED's to light up, giving the player a real-time indication of their player health on their FPGA as they play. If full health, all 9 LED's will be lit up, decreasing as the health diminishes.

In the Python script, we first receive the player health from the server, then strip it of any newlines and spaces it may have. As we are only sending a number between 0-9 to indicate how many LED's to light up on the FPGA, we can send a singular ASCII character back to the FPGA, instead of an integer. We chose to send an ASCII character instead of an integer as it would be less complicated to process on the FPGA side, which reads one character at a time from the UART peripheral. We use the Python JTAG UART library to send the character to the FPGA.

4.2 Receiving And Processing Data From FPGA To Host PC

In the main loop of the code, we simultaneously read and write from the UART communication peripheral, using the Python JTAG UART library. We first try to read from the UART port, and if there is data to receive, we decode this data and add it to a line buffer. We used a line buffer to store the incoming FPGA data, until a newline character was encountered, to which we would stop appending to the line buffer. Once we have the whole line in the line buffer, we need to extract the 4, 32-bit, words that make up the data line, 3 being the FPGA accelerometer data and the last word storing the FPGA push button state bits. We store the 4 words in an array so that we can access each one individually. The words are sent in a signed hex format, so we also need to convert the signed hex format words into integers. We use Python's casting expression to do this, while also accounting for the sign bit in the hex signed representation. Once we have integer representations of all the different data, we can start using this information to process different controls to send to the server.

For debugging and testing purposes, it was essential that we had a way to visualise the incoming data from the FPGA, so that we could tune, test, and improve the controls. We used a data buffer in the form of a deque data structure to store the last recent 4000 values to plot using matplotlib, which meant we were able to view a live rolling stream of the FPGA's accelerometer data.

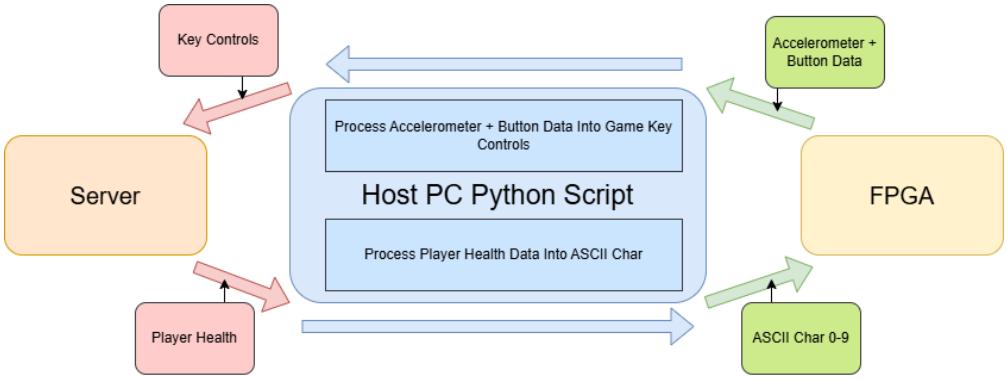

Figure 4 - System Diagram Between Server And FPGA

4.3 Processing Data To Send To The Server

Our game required 6 different command inputs; Left, Right, Jump, Down, Attack1 and Attack2. We settled on the x-axis of the accelerometer to control the left and right keys. By reaching a set threshold, the left and right keys would be separately triggered, with a negative threshold for the left key and a positive threshold for the right key, corresponding to tilting the FPGA left and right to a certain angle. In between these thresholds, both left and right keys are set to 0, corresponding to a stationary state.

For the jump and down movements, we decided that it was most intuitive and exciting to map the jump and down controls to the quick movement of the FPGA upwards and downwards in the z-axis respectively. These “impulse” movements of the z-axis for the vertical game controls, compared to the tilting movement of the x-axis for the horizontal game controls, required different handling and detection. We settled on a moving average approach to detect the impulses in the accelerometer data. We have a sliding window approach to determine the average data value of the accelerometer data and compare the average of the recent past data to the current accelerometer data. If the difference between the past average of values and the current value is greater than a customisable threshold, then we determine that an impulse has occurred. We are also able to customise the width of the sliding window, to be able to select from how far back we calculate the past average of values. A longer sliding window can identify “slower” and more gradual impulses, whereas a shorter sliding window requires faster and greater impulses to activate the control key.

4.4 Sending Data To The Server

A similar approach was taken for the Attack1 game control key, except the impulse would be made on the y-axis. We went through multiple iterations to ensure the game controls were intuitive and easy to use with minimal control errors. We soon noticed from extensive testing, that multiple control keys would be simultaneously triggered at the same time, causing control discrepancies and unnatural game logic. To stop this, we added “key blockers” where if a certain control key was triggered, it would prevent certain other keys from also being triggered, preventing game control conflicts that would inhibit intuitive gameplay. The last control key, Attack2, is set by the FPGA push button.

All of the game control keys are sent at once to the server in one line. Key commands are only sent if there is a key change in at least one of the inputs. This is to ensure compatibility with the server’s capabilities and reduce traffic to the server instead of a continuous stream of key commands.

5. Unity Game Instance

5.1 The Principles of Designing Servers and Clients

Protocols: TCP is used universally for reliability, while UDP is used for frequent, low-latency updates like positions.

Data Protection: Data is wrapped in a key-value structure. Regex extracts the key and content, ensuring data integrity and proper processing.

Authoritative Server: The server is the sole authority for the game state (e.g., player positions, health, stamina, scores). Clients send key action data to the server, which updates the game scene. This prevents cheating and ensures consistency across clients.

5.2 Implementations on Server and Clients

TCP Handshaking: Initially, clients send their IDs to the server which maps client IDs to internal player IDs, sends the mapped ID set to clients and waits for acknowledgements. Once all 2 clients respond, the game starts, and both TCP and UDP servers begin sending data.

UDP Handshaking: During TCP handshaking, a UDP server starts. The TCP server requests the TCP clients to start the UDP clients which resend player IDs. The UDP server then matches them to TCP records. Clients update positions of local game objects based on internal player IDs. These ensure no mix of player positions.

TCP Game End Handshaking: If a player requests to end, the server prompts the other client. If both agree, the server will disconnect all clients and restart both TCP and UDP server, meanwhile clients display results (e.g., "VICTORY," "DEFEAT," "DRAW"), and a "press mouse to continue" prompt.

Sharing player actions: Key changes ('1' or '0' for pressed/released) are sent from TCP client only when detected from python local host. The client has 2 arrays storing current and past key to identify key changes. This significantly reduces data traffic comparing to sending key directly. Timestamps are also included in this packet to measure delay. Broadcast player status data: Health, stamina, and scores are updated less frequently via TCP for reliability.

5.3 Game Scene Setup

Player Control: Includes control/health/attack scripts, skeletal animations, and links to TCP/UDP server/client scripts.

Scene Construction: Pre-configured with TCP/UDP server/client objects, cameras, UI, and two player objects. A pseudo-random map is generated using a seed sent from the server after handshaking.

6. Database Integration and hosting on AWS

The database implementation utilizes AWS Lambda functions to handle API requests and a DynamoDB providing noSQL database storage. The Unity game client connects to this backend through RESTful API calls, relying on open API gateways. In an environment without AWS Academy restrictions, an implementation utilizing AWS Cognito with IAM user pools to manage player identities would have been more robust.

Since we only use the database for storing the leaderboard data, we use a simple DynamoDB schema only using one table with "playerID" as the partition key. Each player record stores the highest individual game score, the cumulative score across multiple game sessions, number of games played and a timestamp of most recent activity. Time-to-live (TTL) functionality is also used to automatically remove inactive players with no scores.

6.2 Hosting game server on AWS

For the game's real-time multiplayer functionality, we initially implemented a local Unity-based server using TCP and UDP protocols to synchronize game frames between clients. After some feedback from the game demonstration, we decided to migrate the game server to an AWS EC2 instance, updating the client connection endpoints to target the EC2's public address instead.

7. Testing Framework

The firmware and networking components each had its own separate branch for testing and development. The game design and front-end was all done in Unity. Any experimental features were also done in a separate branch to avoid corrupting a working design.

Our testing for Slash Man follows the flowchart in Figure 5. We conducted multiple end-to-end test sessions between pairs of clients, verifying that the FPGA movements translated into smooth and intuitive in-game actions. Throughout testing we monitored latency times, enabling us to iteratively tune our communication protocols and hardware filter parameters. For integration testing, we validated the complete data flow from end-game score submission to the leaderboard display and verifying the aggregation of scores across sessions.

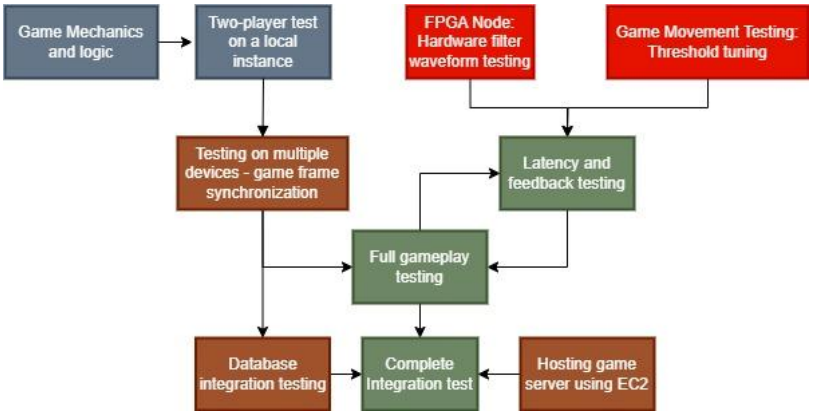

Figure 5. Testing Methodology for Slash Man