Information Processing EIE A – Group 5 Report

Jeremy Tan, Charlotte Gibson, Athanase de Germay, Junto Mabuchi, John Kelly

Submission Date: March 21, 2025

1 System Overview

This project implements an advanced motion-controlled gaming system that combines FPGA hardware processing, real-time cloud-based computation, and interactive 3D Web visualisation. Multiple FPGA nodes equipped with 3-axis accelerometers enable precise motion-based gameplay across three immersive mini-games. Each FPGA node performs local filtering and thresholding to ensure only significant motion events trigger game interactions. Events are communicated to a modular, AWS-hosted backend, which processes inputs, maintains authoritative game states, and synchronises multiplayer gameplay. A frontend built with React and Three.js renders synchronised game states, delivering a responsive, multiplayer gaming experience.

2 System Architecture

2.1 Hardware Layer (FPGA + Accelerometer)

The DE10-Lite FPGA interfaces with the onboard accelerometer sensors via SPI. To mitigate sensor noise and improve signal stability, a custom FIR filter was implemented in Verilog. The Nios II soft-core processor, instantiated on the FPGA, processes the filtered sensor data using threshold-based logic to differentiate deliberate player movements from natural jitter. When significant motion events are detected, they are transmitted to the client system via a UART communication channel, ensuring efficient and reliable data transfer.

2.2 Client Layer (Local Node)

The client receives data packets from the FPGA via a UART interface. Upon receipt, the event data is forwarded to the AWS server using a TCP connection, without additional processing. This streamlined approach ensures that the client layer remains lightweight, preserving the computational efficiency and performance advantages provided by the FPGA at the hardware layer.

2.3 Server Layer (AWS Modular Backend)

The server layer consists of multiple modules, each responsible for a specific aspect of backend functionality:

- **WebSocket Server**: Manages real-time bidirectional client communications.
- **FPGA Event Handler**: Decodes and validates incoming FPGA event streams.
- **Game Manager**: Acts as an intermediary between the WebSocket server and individual mini-games, managing shared gameplay data, delegating gameplay updates, and handling score updates.
- **Mini-game Modules**: Implement gameplay logic and mechanics, including event handling, game state management, scoring algorithms and dynamic game-content generation. Server-side authoritative game logic ensures synchronisation across nodes.
- **Flask Score Service**: Manages persistent storage and retrieval of global scores and analytics.

2.4 Frontend Layer (React + Three.js)

The frontend serves as a rendering client, animating game objects and players in real time based on server commands received via WebSocket. It ensures synchronised multiplayer visuals across nodes, while the backend handles game logic and state management. The frontend communicates settings to the server to support configuration and player positions after animation for accurate gameplay.

2.5 System Architecture Diagram

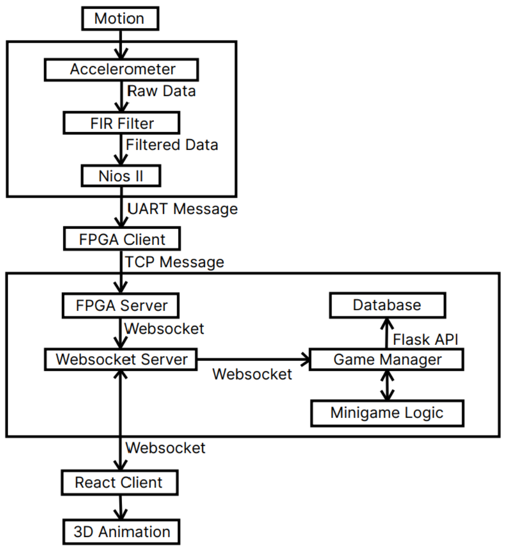

Figure 1: System architecture showing data pipeline from FPGA to frontend.

3 Performance Metrics

3.1 Primary Metric: Latency

Given the reaction-based gameplay, end-to-end latency was the critical performance metric. Strategies employed to minimise latency include:

- Early-stage hardware filtering and event-driven thresholding on the FPGA reduces data volume at the source, offloads processing from software and minimises transmission overhead.
- Efficient protocol usage, combining TCP for reliable transport and WebSocket for low-latency updates, ensures responsive communication across layers.
- Modular, non-blocking server backend enables scalable, concurrent processing of real-time gameplay events across multiple clients.

3.2 Latency Measurement

A pipelined latency measurement approach was taken to localise timing variability, improve accuracy across longer measurement periods, and prevent error accumulation. The pipeline captured latency at four stages: Hardware filtering latency, FPGA-to-client event latency, Client-to-frontend render latency, and Client-to-game interaction (e.g. hit registration).

3.2.1 Filter Performance Comparison

Table 1 presents the latency of various filtering approaches, highlighting significant optimisation when using hardware filtering compared to software filtering relative to the raw data benchmark.

Metric	No Filter	Software Filter	Hardware Filter
Mean Latency	19.15 μs	3.556 ms	31.02 μs

Table 1: Filter Performance Comparison

3.2.2 Pipeline Latency Breakdown

Measurement Stage	Minimum	Maximum	Mean
FPGA → Frontend Receive	96.51 ms	196.26 ms	∼133 ms
Frontend Receive → Render	4.5 ms	17.1 ms	∼10 ms
Frontend → Game Logic Update	20.08 ms	71.27 ms	∼43 ms

Table 2: Pipeline Latency Breakdown

3.2.3 Clock Calibration and Synchronisation

To mitigate clock drift between the FPGA and the frontend, a two-point calibration was introduced. The FPGA timestamps events with $alt_timestamp$, while the browser logs reception times using $performance.now()$. The offset between the FPGA and browser clocks is estimated by correcting for transmission delay and clock rate differences, enabling alignment of FPGA timestamps with browser time. This offset is then applied to incoming events for more precise network latency measurements. Recalibration is supported by resetting the calibration state as needed to prevent drift.

3.3 Performance Enhancements

Building on the low-latency strategies, additional optimisations were implemented to improve system performance. The FPGA-level thresholding significantly decreased the server load by discarding insignificant movement at the source. The FIR filter was tuned using LED visualisation, allowing real-time feedback to balance responsiveness and smoothness. All communications use interrupt-driven I/O to minimise transmission delays and maximise throughput efficiency.

4 Design Decisions

1. **Dedicated Hardware Filtering on FPGA**
   The hardware FIR filter ensured low-latency signal processing. FPGA-level data processing reduced the computational overhead on the client and avoided OS-induced variability.
   Alternative: Software filtering.
   Reason Rejected: Software filtering introduces platform-dependent timing inconsistencies and higher latency due to CPU scheduling.

2. **Three.js 3D Rendering**
   Three.js provided seamless integration with react and full compatibility with modern web browsers, allowing for efficient deployment. Three.js provided smooth and lightweight rendering well-suited for dynamic 3D animations and a responsive user interface.
   Alternative: Unity/Unreal.
   Reason Rejected: These engines, introduce a steep learning curve, less flexibility for frontend customisation, and heavier runtime overhead, making them less suitable for web-native applications.

3. **WebSocket Communication**
   WebSockets were chosen to support persistent, bi-direction communication between the server and clients. This enabled the transmission of gameplay data, including motion events and score updates, with minimal overhead and latency. The connection remained open throughout the session, ensuring efficient handling of frequent, low-payload messages.
   Alternative: REST APIs and polling
   Reason Rejected: Rest APIs or polling increase latency due to the overhead of request/response cycles and are not well-suited for an event-driven gameplay model.

4. **Server-Centric Game Logic**
   Gameplay logic and computation were centralised on the server, ensuring consistent state synchronisation across clients for accurate rendering. This architecture enforced authoritative control over game outcomes, minimised client-side discrepancies, and facilitated cleaner frontend development through clear separation of concerns.

5. **Modular Server Design**
   The modular server architecture enhanced system maintainability, supported scalable development and facilitated easier debugging and testing of individual components.

5 System Testing

5.1 Software Testing

Testing was carried out in phases to independently validate components before integration:

1. Keyboard-Controlled Prototypes: All mini-games were first prototyped with keyboard input for UX iteration.
2. FPGA Motion Integration:
   - Motion-triggered events verified using UART and threshold tuning.
   - Confirmed successful FPGA-Driven Control throughout the system, with FPGA commands producing the correct frontend animations.
3. Multiplayer Synchronisation Testing: Concurrent FPGA-client setups validated system-wide synchronisation and low latency under realistic gameplay conditions.
4. Cloud Deployment: AWS-based deployment confirmed system scalability and performance under distributed conditions.

5.2 Hardware Testing

LED Spirit Level (Lab 3) was used for visual FIR tuning.

5.3 Latency Testing

System latency was evaluated through repeated test runs, with both the FPGA and browser environment reset between iterations to ensure unbiased results. High-resolution timestamps were captured at each pipeline stage, enabling precise measurement of delays.
Clock synchronisation between the FPGA and frontend was maintained via offset calibration, enabling accurate alignment of cross-domain timestamps. Latency was measured at multiple stages: transmission latency was computed by comparing calibrated FPGA timestamps with frontend arrival times; render latency was approximated using requestAnimationFrame() at animation trigger; and hit detection latency was derived from the delta between client-sent motion timestamps and server-side processing times.
This process enabled the characterisation of the system's latency under test conditions.

5.4 Testing Flow Diagrams

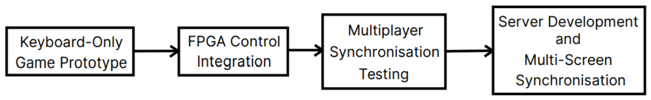
Keyboard-Only Game Prototype
FPGA Control Integration
Multiplayer Synchronisation Testing
Server Development and Multi-Screen Synchronisation

Figure 2: General Testing Flow

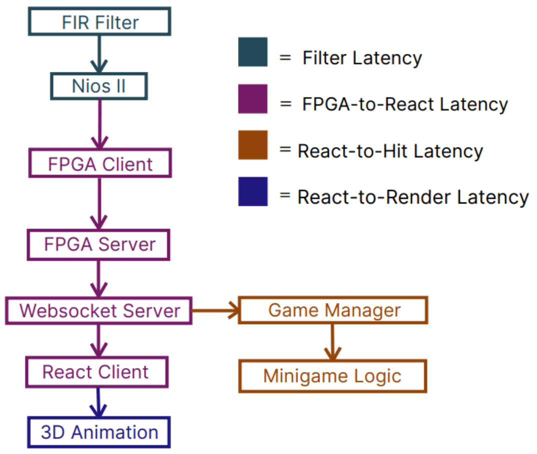

Figure 3: Latency Measurement Flow

6 DE10-Lite Quartus Resources

The FPGA design and development process was built using various tools on the DE10-Lite board, including the Nios II Soft Processor for threshold logic, UART communication, and motion detection. A custom FIR filter implemented in Verilog to smooth accelerometer data in real time, with SPI communication to interface with the accelerometer module. The hexadecimal LED display was used to indicate the player number, and dedicated buttons were also integrated to trigger special animations within the games. In addition, a LED debugging interface was used throughout the development phase to visualise the signals.

7 Functional Requirements and Extensions

Requirement	**Technical Implementation Details**
FPGA-based accelerometer processing	FPGA hardware FIR filtering, Nios II processor thresholding, and UART event communication
Cloud-based event handling	AWS EC2 server processes input events, delegates game logic via Game Manager, and synchronises multiplayer gameplay.
Bidirectional node-server communication	Reliable TCP data transfer to AWS server, real-time control commands sent via WebSockets to FPGA clients.
Multiplayer node support	Fully implemented and tested synchronisation across multiple FPGA nodes and clients, supporting concurrent multiplayer interactions.
Frontend visualisation and synchronisation	React and Three.js frontend dynamically render synchronised multiplayer states based on authoritative server data.

Table 3: Functional Requirements and Technical Implementation

7.1 Extended Functionalities

During this project, we expanded the functionalities beyond the initial requirements to fully showcase the controller's capabilities. Each of the three mini-games demonstrate unique features designed to evaluate the capacity of the controller, including it's responsiveness, smooth 2D character movements, and the capability to connect multiple nodes for multiplayer game play. To enhance player engagement, we integrated audio tracks with the mini-games and sound effect when an animation is played. Finally, the scoreboard is linked to a database, enabling automatic score saving and sharing between players to ensure a consistent multiplayer experience.

Project completeness supported by attached demonstration video and source files.

YouTube link: `https://www.youtube.com/watch?v=0eh9FeRNpHk`

TOTALLY NOT
MARIO PARTY

7.2 Peer Review