CG4002 Computer Engineering Capstone Project
2024/2025 Sem2

"Laser Tag++"

Final Report

Group B17	Name	Student #	Primary Component	Secondary Component
Member #1	Khairul Aizat Bin Md Halim	A0233315R	Hw Sensors	Comms External
Member #2	Steven Antya Orvala Waskito	A0244129H	Comms Internal	Hardware Sensors
Member #3	Brandon Owen Sjarif	A0245892R	Comms External	Comms Internal
Member #4	Ong Junzheng, John	A0196727M	Sw/Hw AI	Sw Visualiser
Member #5	Ren Tianle	A0211176N	Sw Visualizer	Sw/Hw AI

Table of Contents

Section 1 System Functionalities

1.1 Feature List

- Real-Time AR Overlay
    - Displays opponents, virtual objects (e.g., bombs, shields), and environmental anchors (e.g., spawn points) over the real-world camera feed.
    - Visualizes laser trajectories, explosions, and shield effects in 3D space.
- Opponent Tracking and Status
    - Shows opponents' real-time positions via AR markers (e.g., glowing icons).
    - Highlights shielded opponents with a glowing outline.
    - Triggers proximity alerts (e.g., arrows) when opponents are nearby.
- Game Action Visualization
    - Shooting: Displays laser beams from the crosshair to the target; flashes red on hit, gray on miss.
    - Bombing: Renders a 3D explosion radius anchored to the physical environment; persists until detonation.
    - Shielding: Activates a translucent barrier effect around the player.
- Player Stats and UI
    - Health Bar: Visible at the top-left corner (e.g., "HP: 100%").
- Multiplayer Synchronization
    - Syncs game state (player positions, actions, health) across all devices in real time.
    - Updates AR effects globally (e.g., all players see the same bomb explosion).
- Environmental Interaction
    - Anchors virtual objects (e.g., bomb zones) to real-world surfaces.
    - Detects collisions between players and virtual objects (e.g., entering a bomb radius triggers snowfall effect).

1.2 User Stories

As a...	**I want to...**	**So that...**
Player	Perform a Badminton swing	I can demonstrate my racket skills and technique
Player	Execute Boxing moves	I can showcase my punching combinations
Player	Throw a Snowbomb	I can hit opponents with a projectile attack

Player	Reload my weapon	I can continue fighting after depleting my ammunition
Player	Make a Golf swing	I can demonstrate my golf stroke technique
Player	Perform Fencing moves	I can demonstrate my sword fighting skills
Player	Activate my Shield	I can protect myself from incoming attacks
Player	View my health status	I can monitor my remaining health points
Player	Check my ammunition	I can know when I need to reload
Player	See shield cooldown	I can know when I can activate my shield again
Competitive player	View opponent's status	I can strategize my next moves effectively
Action player	Execute moves accurately	I can maximize the effectiveness of my attacks
New player	Learn the basic moves	I can participate effectively in the game
Experienced player	Chain different actions	I can create complex combat combinations
Defensive player	Time my shield usage	I can protect myself at critical moments

Section 2 Overall System Architecture

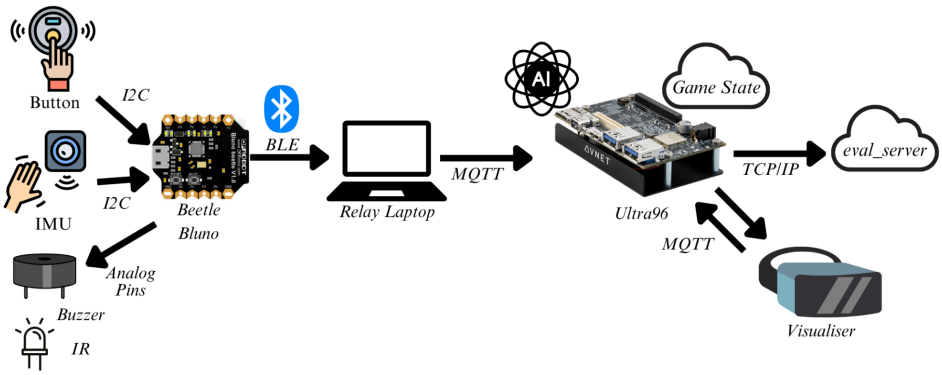


Fig. 1 Overall System Architecture

2.2 Description & Drawing

Our system's intended final form initially included three main subsystems — the Wristband, Vest and Gun. Each of these subsystems contained one Beetle. Firstly, the wristband will hold the MPU-6050 IMU for action recognition (Golf swing, activate shield, etc.). Secondly, the IR receiver will be placed on the center on the front of the vest. Lastly, the IR emitter will be placed on the gun and will act as the laser tag gun shooting IR "bullets" which will be detected by the IR receiver on the enemy's vest. However, we ultimately opted to use just two subsystems — the Glove and Vest. This will be explained in Section 3.8.

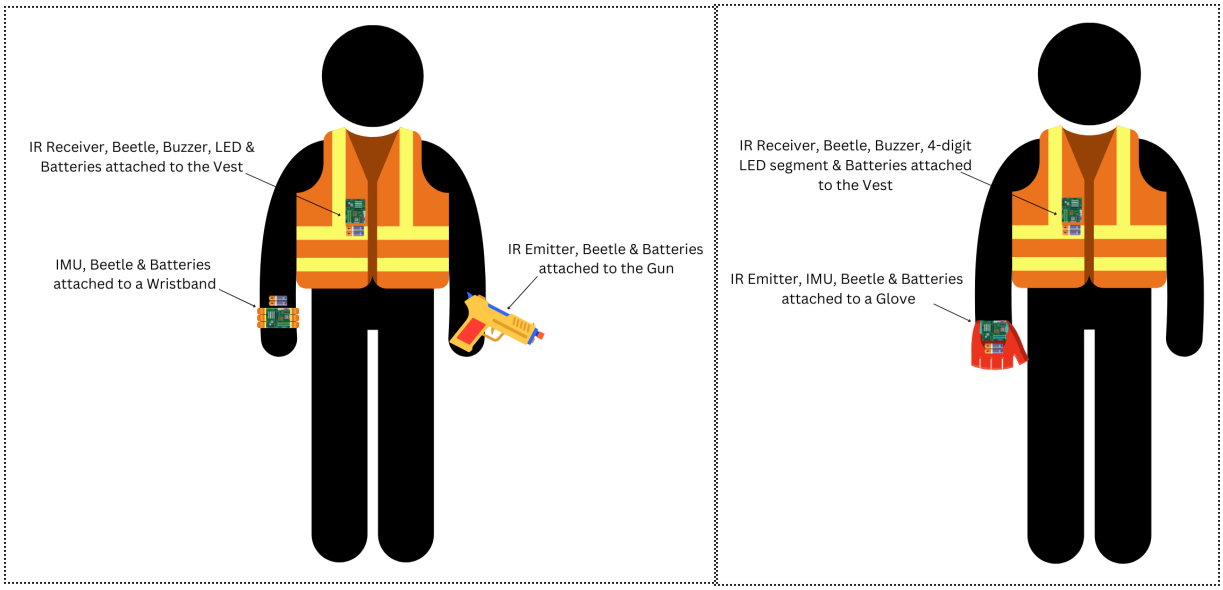


Fig 2. Mockups of the **initial** intended final form (left) and the **actual** final form (right)

2.3 Main Algorithm Breakdown

1	Initiate internal communication handshake between Bluno and relay laptop. Initiate external communication handshake between Ultra96, relay laptop, visualizer, and eval_server
2	Record the player movement's IMU data and other sensor data on the Bluno
3	Based on collected movement data from Bluno, data will be sent to the relay laptop via BLE
4	Relay laptop forwards movement data to local game state server hosted on the Ultra96 FPGA via MQTT.
5	The FPGA will forward this data to the AI algorithm for the data to be processed and waits for an action to be classified.
6	Throughout this process, the current game state is continuously being sent to and fro between the FPGA and the visualisers with an MQTT broker. The game state will continuously be reflected in the players' HUD.
7	Once an action has been classified, this will change the game state running on the FPGA and will then be reflected on the players' screen.
8	This will continue until one player wins the game based on the game state. At this point, the finite state of the program will be set to game-over, until restarted.
EX	In cases where eval_server is being used, the current game state will also be continuously sent to the server through TCP/IP through AES encryption and tunneling.

Machine Learning High level Algorithm

Our machine learning implementation for the AR laser tag system follows a structured four-step approach for action recognition and game state management. In the first step, we focus on movement and action detection through continuous data collection from IMU sensors, operating at a 100Hz sampling rate. The system monitors both accelerometer and gyroscope data to detect significant movements. When sensor values exceed predetermined thresholds, indicating potential game actions such as shooting, reloading, or activating shields, the system flags these data segments and forwards them to the Ultra96 for processing. Simultaneously, the system monitors IR sensor data for hit detection between players.

The second step involves comprehensive data pre-processing and feature extraction. This begins with software-based noise reduction and signal filtering to ensure clean data for analysis. The system segments the continuous data stream into 500ms time windows, enabling real-time action recognition. From each window, we extract key features in both time and frequency domains. Time domain features include mean, standard deviation, root mean square (RMS), and minimum/maximum values, while frequency domain features focus on peak magnitude and maximum phase. All features are normalized to ensure consistent model input.

For the third step, model training and validation employs a systematic approach using train-test split with a 80-20 ratio to ensure robust performance. Our chosen architecture is a Multilayer Perceptron (MLP)

with 51 input features, derived from statistical analysis of accelerometer and gyroscope data. The model includes two hidden layers (128 and 64 neurons) using ReLU activation functions, and outputs classifications for eight distinct game actions (badminton, bomb, boxing, fencing, golf, logout, reload, and shield). We evaluate the model's performance through multiple metrics including classification accuracy, precision and recall for each action type, confusion matrix analysis, and latency measurements to ensure real-time performance requirements are met.

The final step encompasses real-time implementation and communication on the FPGA platform. We deploy the trained model on the Ultra96 FPGA using High-Level Synthesis (HLS) for efficient hardware acceleration. The system processes incoming sensor data in real-time, generating action classifications that update player states and game conditions. These results are then distributed to multiple system components: the game state manager for tracking player status, the visualization system for AR display updates, and the evaluation server for maintaining game scores and statistics. This integrated approach ensures smooth gameplay while maintaining the real-time requirements of the laser tag system.

Section 3 Hardware Sensors

3.1 Components & Devices

Component	Specifications	Datasheet
Bluno Beetle DFR0339	ATmega328P, 8 bit megaAVR MCU	DFR0339
IMU MPU-6050	3-Axis Gyroscope/Accelerometer, ±16g, 2.375 V to 3.46 V, QFN-24	MPU-6050
Feedback Systems		
IR Emitter OFL-5102	Infrared Emitter, 940 nm, 10 °, T-1 3/4 (5mm), 15 mW/Sr	OFL-5102
IR Emitter Resistor SFR16S (39Ω)	Supporting component for the IR Emitter - 39 ohm, SFR16S Series, 500 mW, ± 5%, Axial Leaded, 200 V	SFR16S
IR Receiver TSOP34836	Remote Control, 36 kHz, Side View Through Hole	TSOP34836
Buzzer ABI-009-RC	Electromechanical Buzzer, 4 V, 8 VDC, 30 mA, 85 dB, Black	ABI-009-RC
4-Digit LED Segment Display Module TM1637	Four 7-Segment LED Displays with 0.36″ high brightness red digits, Built-in TM1637 LED Driver, 2-wire serial interface, 3.3V and 5V compatible	TM1637
Power		
AAA Battery 4003211304.	Battery, 1.5 V, AAA, Alkaline, 1.25 Ah, Raised Positive and Flat Negative, 4	4003211304.
4xAAA Battery Holder MP006847	Battery Holder, 4 x AAA, Wire Lead	MP006847
5V DC/DC converter DFR0569	DC-DC Automatic Step Up-down Power Module (3~15V to 5V 600mA)	DFR0569
Miscellaneous		
Pushbutton R13-24A-05-BB	Pushbutton Switch, R13-24, 7.2 mm, SPST-NO, Off-(On), Round, Black	R13-24A-05-BB
Prototype Board MC01009	Prototype Board, Phenolic, 1.6 mm, 72 mm, 47 mm	MC01009
Prototype Board GC002-LF	PROTOTYPING BOARD, FR2, 40MM X 40.5MM	GC002-LF

3.2 Pin Table

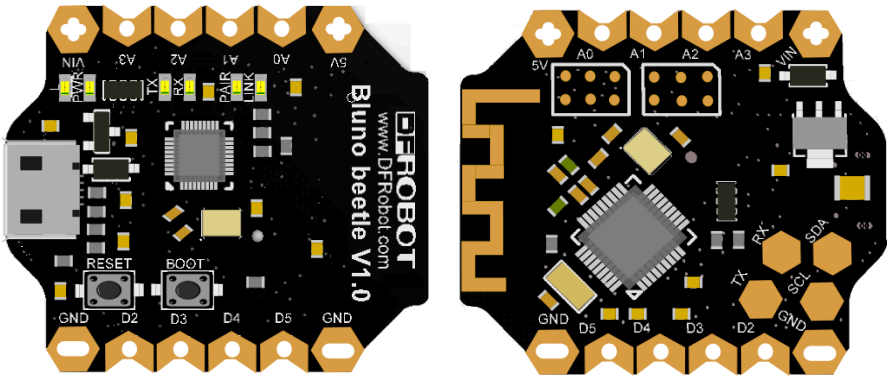


Fig 3. Bluno Beetle Pinout

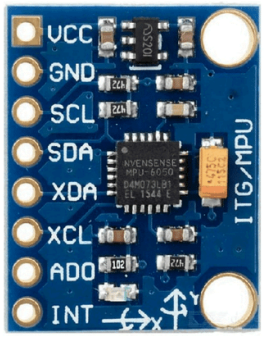


Fig 4. MPU-6050 IMU Pinout

The following table shows the connections between the pins of the Bluno Beetle and MPU-6050 IMU.

Bluno Beetle	**MPU-6050 IMU**
5V	VCC
GND	GND
SCL	SCL
SDA	SDA

3.3 Schematics

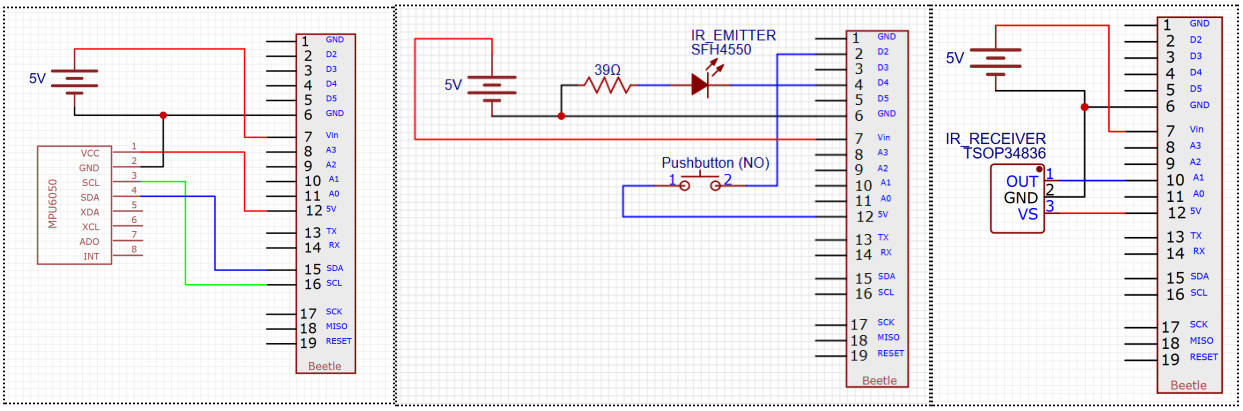


Fig 5. Schematics of **initial** subsystems: Wristband (left), Gun (centre) and Vest(right)

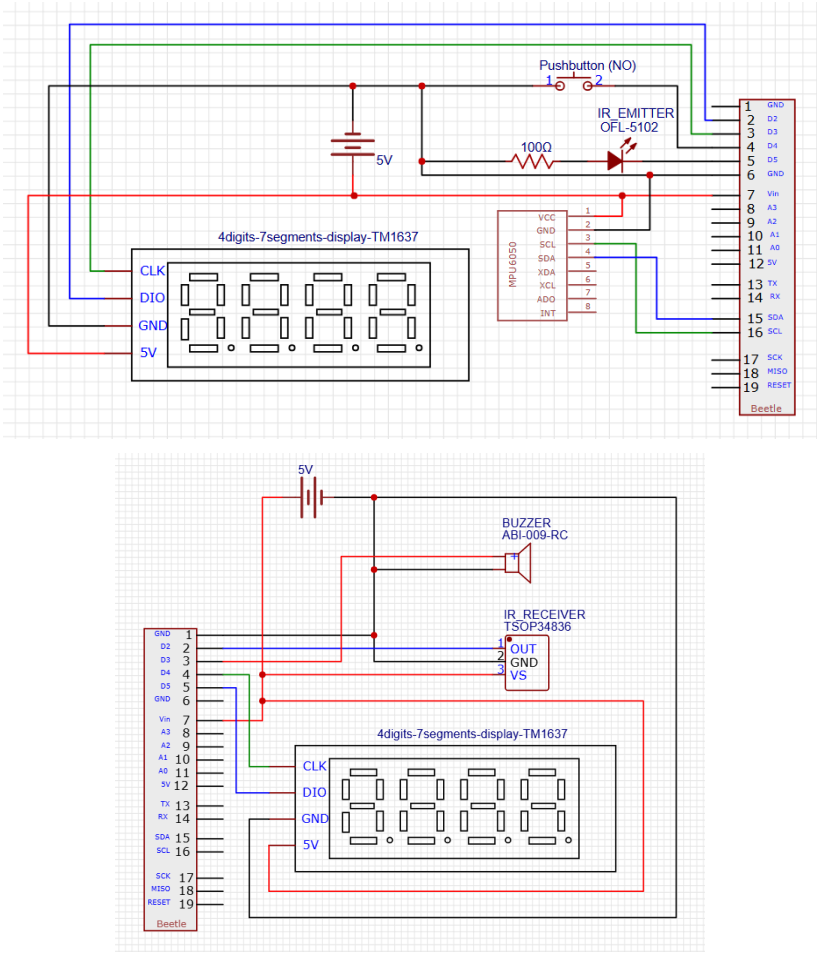


Fig 6. Schematics of **final** subsystems: Glove (top) and Vest (bottom)

3.4 Operating Voltage Level and Current Drawn

The following table shows the rated operating voltage levels, current drawn and calculated power consumption of each component.

Component	Operating Voltage level (V)	Current drawn (mA)	Power Rating (mW)
Bluno Beetle DFR0339	5	10	50
IMU MPU-6050	5	1	5
IR Emitter OFL-5102	1.25	20	25
IR Receiver TSOP34836	5	0.7	10
Buzzer ABI-009-RC	4	30	120
4-Digit LED Segment Display Module TM1637	5	80	400
Pushbutton R13-509A-05-BB	5	-	-

We chose a 100Ω resistor to limit the current flowing through the IR emitter. Based off the IR emitter's characteristics, we did the calculations as follows:

Forward Voltage, V_f	Desired Forward Current, I_f	Power Supply, V_S
1.25V	20mA	5V

Using Ohm's law, $R = V_S - V_f / I_f = 5V - 1.25V / 20mA = 187.5Ω$.

For Power dissipation, $P = I^2R = 75mW$.

We then chose the closest standard resistor value, $R = 200Ω$ that has a power rating of >75mW.

However, we found that the emitter was underpowered and could not function at our desired 3m range with this resistor setup. Hence, we opted to use a 100Ω resistor instead.

3.5 Battery Design

We used 4xAAA Energizer Alkaline batteries to provide ~6V to a DC/DC converter that outputs stable 5V, 600mA peak current.

3.6 Algorithms & Libraries used

Arduino **MPU6050** and **IRremote** libraries.

3.7 Feedback Systems

When the player shoots a gun, the ammo count shown on the glove's LED display will decrease and the buzzer will sound. Everytime a player gets hit, the HP count shown on the vest's LED display will decrease (-5hp per shot & -10hp per other attacks).

3.8 Issues Faced

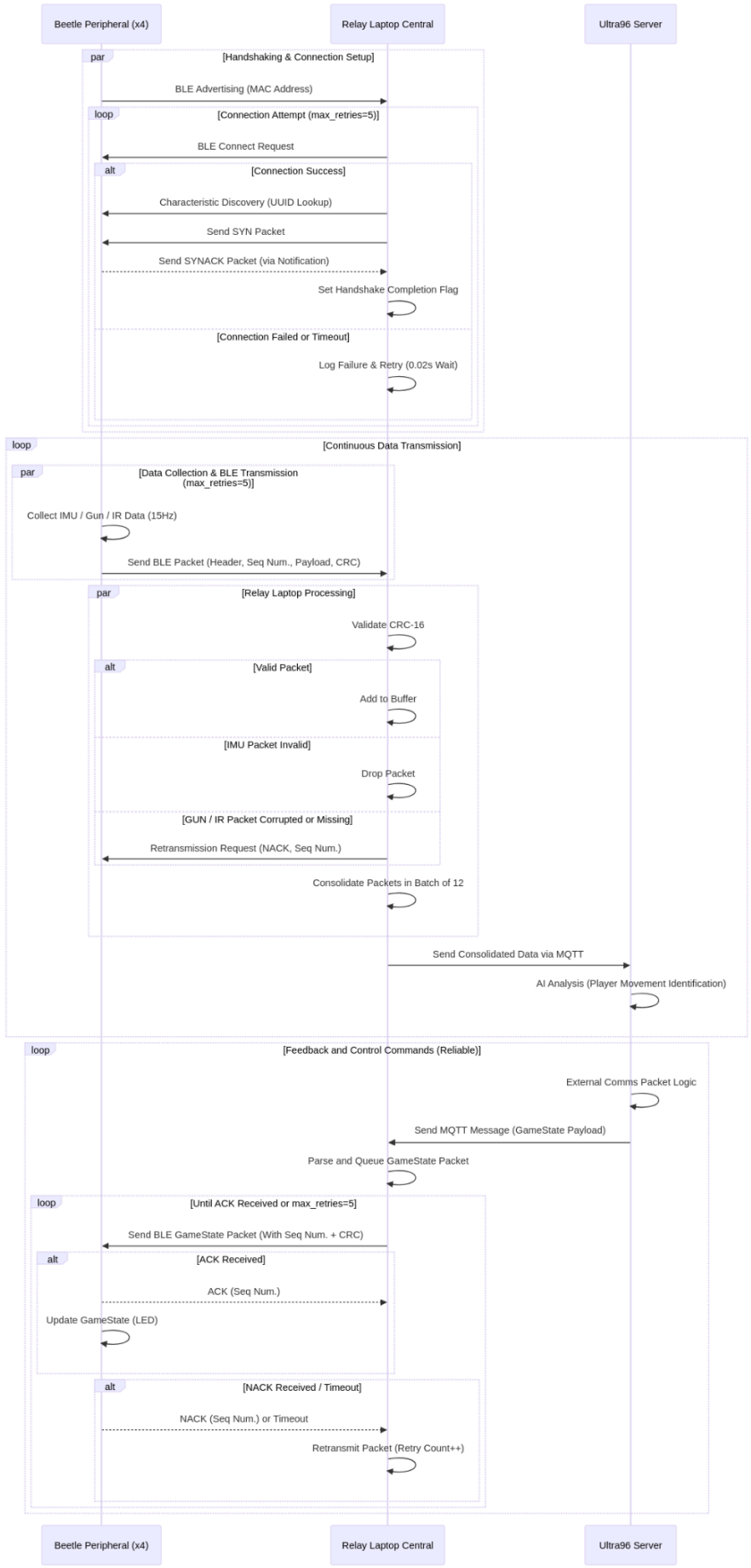
Initially, we planned on using individual 7-segment displays. We needed to use transistors to supply a 7-segment display with power from the 5V battery since relying on just the arduino pins would not be enough current for the display to show all digits.

A trick to get around using transistors and one resistor for each display segment is to switch between each segment at a very fast rate (each segment is lit up for 3ms) such that the human eye sees every segment being lit up at the same time. This allows the 7-segment display to rely on just the arduino pins as the arduino just has to drive a single pin at any point in time. (Source: https://github.com/gbhug5a/7-Segment-Displays-Multiplex-by-Segment/blob/master/MultiplexBySegmentDemo.ino)

However, this will require me to use all 8 pins of the Bluno Beetle (and an additional 2 pins to be connected to each of the CC so that makes 9 pins total). So, we could either use two Beetles or find a driver IC for the 7-segment display and use the traditional method (one resistor per segment and a NPN transistor to drive the display).

In the end, we opted to avoid this challenge entirely by using a TM1637 4-Digit 7-Segment display chip.

Section 4 Internal Communications



4.1 System Functionalities

The internal communication handles all communication to and from the microcontroller, in this case the DFRobot Bluno Beetle. The data that is transmitted from the Beetle includes IMU sensor data, IR sensor data, and gun button data. The data that is transmitted from external to the Beetle is game state data which includes ammo and health points. All of the communication between Beetle and external comms are done within the relay laptop, in which the internal communication program lives. The exchange of data relies on the use of packet, where each packet is like a small ship that carries information header and data payload, while a relay laptop is like a harbourfront that receives all of the data, rearrange it, and then dispatches it to the external communications to be used in other subsystems.

In a concurrent manner, the relay laptop will communicate with 4 beetles simultaneously. On top of that, the laptop will preprocess the packets from the beetle and consolidate them and send them to the relay laptop. The concurrent processes must be handled by a multithreaded application within the system.

In a reliable manner, the connection between all of the nodes are ensured to handle disconnection and reconnection quickly and safely, utilizing handshakes to reinitialize and synchronize between nodes.

The communication between the beetle and the relay laptop will be using Bluetooth Low Energy (BLE). The communication between the relay laptop and the Ultra96 (Server) will be using MQTT as described by the external communications team.

4.2 Duplex Connection

The internal communication happening between any node is a two-way connection. The connection between the relay laptop and Bluno happens both ways and so does the connection between the relay laptop and external comms.

The relay laptop needs to send game state information to the Bluno including HP data and Ammo data. On the other side, Beetle needs to send sensor informations and button informations including IMU data, IR hit data, and Gun Trigger data

The relay laptop needs to send Beetle information data including IMU data, IR hit data, and Gun Trigger data to the external comms. On the other side, the external comms needs to send game state information to the laptop relay including HP data and Ammo data.

4.2 Task on the Beetle

Each beetle is configured to collect and send IMU data at 15 Hz (67ms) rate to the laptop relay in an unreliable manner and button data to the laptop relay via BLE in a reliable manner. Initially, a 20 Hz number was chosen because of the hypothesis that human motion can be deducted at this rate. On top of that, 50ms is fast enough to become un-noticeable since the average human response time is in the 250ms range. Lastly, 20 Hz will hopefully not congest the internal and external communication and does not overwhelm the AI System. However, a decision of using 15 Hz is chosen because of three reasons, a

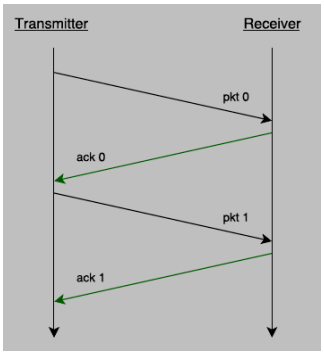
slower rate means a slower transmission that by experiment is less prone to packet corruption and packet loss. Secondly, since the Beetle is handling both reliable and unreliable packets at the same time, the slower rate will allow the reliable packet to not block the IMU unreliable data transmission for more than required. Thirdly, as we have tested, the AI detection with 15 Hz rate is performing with an accuracy of 94% which is satisfactory in our use-case.

For the IR and Gun data, they are sent reliably. On the vest, we only send Gun data. On the gloves, due to the design of the hardware, we are to send IMU data and Gun data from the same Beetle. Since, IMU data is unreliable while Gun data is reliable, we employ a blocking wait_for_ack(). This is less optimal, but our only choice since the Beete's BLE handler does not work in interrupts. Another option is to multithread and use queue, however this further adds complexity, and may not be suitable for Beetle due to its low power and small memory.
All Beetle is configured to receive GameState from laptop using wait_for_ack() in reliable manner.

4.2.1 Unreliable Transmission

Initially, all data, including IMU is sent reliably, however, the stop-and-wait ARQ takes too much time on our experiment (~150ms RTT). So we decided to do unreliable transmission for IMU data. The IMU sensor data from Beetle to the laptop relay is sent unreliably, meaning that if a packet is corrupted or lost, the packet will be discarded. There will be no ACK sent by the laptop relay upon the recipient of the IMU data. A packet is considered corrupted and will be discarded if the packet CRC is wrong.

4.2.2 Reliable Transmission



The Gun data, IR data, and Gamestate data transmitted between the Beetle and the laptop relay is sent reliably using Stop-and-Wait ARQ (Nyangaresi et al., 2018).

The Beetle or the laptop relay will handle reliability by doing stop-and-wait protocol for retransmission, with a timeout of 300ms. When the Beetle receives NACK from the Relay laptop or vice versa, the sender will retransmit the packet. When the sender does not receive an ACK by the timeout duration, it will retransmit the packet. If by the fifth retransmission, the packet is not acknowledged, then the Sender will send an exception that the connection is offline and will try to reconnect with the node.

The stop-and-wait protocol is used because the projected latency (Tf + 2Tp) between Beetle and Laptop through BLE is minimum 3ms (Bluetooth, 2014)[6]. Initially, we are expecting each transmission to take a maximum of 10ms with a packet size of 20 bytes, from the max of 27 octets or bytes. On top of that, our latency is small enough; therefore, stop-and-wait arq will suffice in this case, and sliding window arq is not needed. However, during testing we observed that the round trip time between two BLE nodes is around 150ms (RTT), due to packet overhead and probably the BLE firmware and or library used. Therefore we decided to stick with this observation. This was also the reason that we chose to do unreliable transmission for IMU sensor data.

4.3 Multithreaded Application on Relay Laptop

The laptop has multiple threads that need to run concurrently. This includes:

1. 4 BLE Client (one for each beetle)
    a. Handshaking
    b. Reliability handler for BLE (NACK)
2. Preprocess of incoming packets
    a. Checksum validation
    b. Error detection
3. Consolidation of packets into a unified stream before sending to external comms
4. Transmission to Ultra96 via MQTT
    a. Reliability handler for Ultra96

Relay laptop must run these threads concurrently and ensure thread synchronization to prevent data corruption and resource contention such as deadlocks. Each concurrent thread will have a queue to process incoming and outgoing data. For example, the incoming data from Beetle will enqueue to ble_incoming_queue. The process of dequeuing and transmission will also be in a separate thread that constantly dequeues if there is a packet to process.

4.4 BLE Interfaces

We are using Bluetooth Low Energy (BLE) with Generic Attribute (GATT) profiles

Roles: Beetle as BLE peripherals and Laptop as central device

BLE Parameters
Due to Beetle's property, all beetles only have one service UUID and one characteristic UUID. The UUIDs across different beetles are the same, therefore we have to make use of MAC addresses to differentiate between beetles. On top of that, to differentiate between packet types, we use packet headers (e.g. IMU, Button Data).

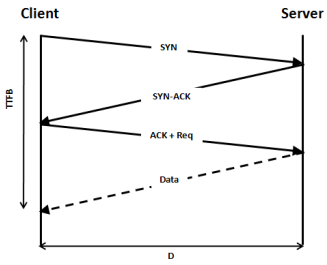
Data Rate: Calculating required throughput. Since each transmission requires 160 bits every 67ms, an ideal throughput of 2400 bits/second or 2.4 Kbit/s is needed for each beetle. Since the baud rate chosen is 115200 baud/second, and our data rate is well below the baud rate, the data rate is theoretically achievable. We ignore the reliable transmission data rate since 160 bits with a round trip-time of 300ms is significantly slower and the Beetle does a blocking call when doing a reliable transmission rate, therefore the worst case scenario for data rate is when we continuously send unreliable IMU data without any reliable data.

Range: We expect 10 meters of range for BLE 4.0 as described in the technical datasheet of bluetooth. However, in practice we suffer from a lot of bluetooth interference as BLE only has 32 channels and in case of a collision, it will disconnect and do channel switching. With more than 15 teams in a room at a time, the interference was a limiting factor. Our observation is that Beetle range is around 2 meters.

4.5 Internal Communication Protocols

4.5.1 Reliable Reconnection and Handshaking

The connection between the Beetle and the laptop relay is handled by the Beetle's firmware and the bluepy library respectively. These 2 components are not modified by us.



In the event of a disconnection, the bluepy library will throw an exception, in which the internal comms program will trigger a reconnection to the Beetle.

In the event of a connection or a reconnection, we sync the two devices with a handshake. Handshaking between Beetle and Laptop Relay will follow a 3-way handshaking protocol that is described in the diagram (Siracusano, 2016)[5]. The handshaking does 3 major things, flushing old data, clearing up the buffer for both nodes, and resetting the sequence number to 0 for both nodes.

4.5.2 Packet Format

Each packet is 20 bytes in length. We are going to use a big endian for every packet in the internal communication.

Each section is represented in bytes.

Packet Type [0]	Sequence Number [1]	Payload + Padding [2:18]	CRC-8 [19]

Padding is needed to ensure all packets are 20 bytes wide. Padding is repeating 0's
We are using 8-bit checksum using the CRC-8 0x07

4.5.3 Packet Types

There will be 3 distinct packets that are communicated between Beetle and Laptop Relay. Initially the control packet size was only 3 bytes (Packet Type, Seq, and CRC), however, we decided to change everything to constant 20 byte packets to make sure we can reliably handle packet fragmentation, packet drop, packet corruption through crc-check, and we can directly check the packet type every 20 bytes. (e.g. what if inside the data there was a data that is the same as the packet type we defined, since in the previous logic, we loop through incoming data and find the matched packet type header).

1. Data Packet: Sensor, IMU readings, and Physical buttons

IMU

Packet Type 0x41 [0]	Sequence Number [1]	Acceleration (x,y,z) in .2fp and gyro in .2fp. Each with 16-bit fp representation. 12 bytes needed in total. [2:18]	CRC-8 [19]

IR Receiver

Packet Type 0x42 [0]	Sequence Number [1]	Senses an IR represented as 0x1000…00 [2:18]	CRC-8 [19]

Shoot Gun Button

Packet Type 0x45 [0]	Sequence Number [1]	Ammo as payload 0x_AMMO_00..000 [2:18]	CRC-8 [19]

2. Game State Packet (from relay laptop to beetle)

Ammo

Packet Type 0x10 [0]	Sequence Number [1]	Ammo is zero is represented as 0x0100_AMMO_00.. [2:18]	CRC-8 [19]

HP

Packet Type 0x10 [0]	Sequence Number [1]	HP is zero is represented as 0x0010_HP_00…00	CRC-8 [19]

3. Control Packet:

ACK Packet Type= **0xC0**. NACK = **0x90**. SYN = **0xA0**, SYNACK = **0xE0**

Packet Type **0xC0** [0]	Sequence Number [1]	Padding 0x00000…00	CRC-8 [19]

4.6 Reliability Issues

4.6.1 Packet Fragmentation

Packet fragmentation, where packets are not sent as a whole, might happen because of physical limitations of the connection and interferences that might be encountered during the gameplay. Therefore we implement a simple algorithm on the laptop relay side. We implement a while loop where we wait for the whole packet before we are able to use the packet. Each beetle will have their own thread so as not to introduce a blocking loop in the relay laptop.

```
data = “”
packet_len = 0
while (packet_len < 20) {
        data += ReceiveData(20-packet_len)
        packet_len = len(data)
}
```

4.6.2 Packet Drop

The internal communication will handle packet drop with a timeout from the laptop relay that sends a NACK or a timeout from the Beetle when it does not receive ACK after a given amount of time, 10ms. The NACK or timeout will cause the beetle to retransmit the given packet.

4.6.3 Packet Corruption

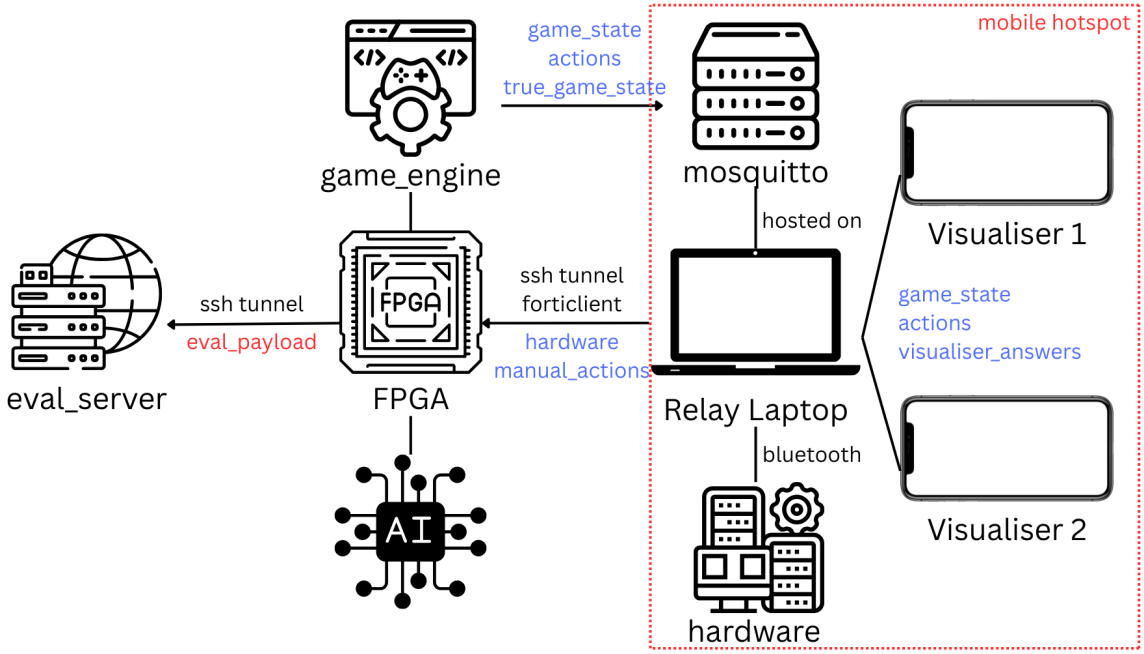
The packet corruption is handled with the CRC-16-IBM checksum with polynomial of $x^{16} + x^{15} + x^2 + 1$ and the laptop relay will send a NACK when the checksum is wrong.

4.6.4 Reliable Reconnection

As mentioned 4.5.1, we handle disconnection by performing reconnection and handshaking. However, it was observed on the demo day that a lot of interference caused frequent disconnection. We tried to increase the transmit power of the Beetle, but it increased the disconnection rate. We tried to decrease the transmit power of the Beetle, and it seems to have decreased the disconnection rate, however it performs very poorly in a crowded environment. We hypothesize that the Beetle hardware is not suitable for a noisy environment, maybe because of its BLE design and firmware. We strongly suggest not to use the Beetle for real-world use-case, rather we can look for alternatives such as Arduino Nano 33 BLE. We also suggest experimenting with other BLE libraries that are cross-platform and newer than bluepy.

Section 5 External Communications

Due to the nature of the SOC firewall preventing our visualisers from communicating with the FPGA, we will be using mobile hotspot connection on the relay laptop and the two visualisers. With this communication method, the relay laptop will be able to send packets to the FPGA through the SOC VPN and SSH tunnels.



System Architecture

As can be seen from the architecture diagram, the overall communications lines of the entire system include (1) Ultra96 and eval_server, (2) Ultra96 and Relay Laptop, (3) Ultra96 and Visualiser. MQTT topics are labelled in blue.

5.1 Communication between Ultra96 and eval_server

The Ultra96 acts as a client that will initiate the connection to the eval_server over TCP/IP. These messages will be encrypted following the encryption method of the server to ensure secure communication.

Due to the SOC firewall disallowing non-whitelisted devices from accessing SOC resources - i.e. our FPGA, we will establish an SSH tunnel from the laptop hosting the eval_server from port 8888 of the FPGA to the port established by the eval_server. On the FPGA itself, we will be posting our final game states to localhost on port 8888.

Eval_server expects a packet format of <packet_length>_<encrypted_packet> and an initial handshake message, “hello”. After the initial handshake has been done between eval_client and eval_server, the final

eval_payload will be sent to the eval_server following the same encryption method and packet format mentioned previously.

The format of these packages will follow the expectations of the eval_server as seen below - with game_state in the same format as eval_server:

Payload Format		
player_id (int)	action (string)	game_state (JSON payload)

eval_server Payload

Since the eval_server will send us a game state response with a data length header for every payload we send to it, we will need a receiver function to receive the true game state. To prevent data loss, we will continuously listen for data chunks until the chunks fulfil the correct data length. If there is packet loss, the receiver function will timeout and raise an exception.

5.2 Communication between Ultra96 and Relay Laptop

Communication within our internal devices will be done through MQTT, there are 3 main reason why we chose this model:

1	Publisher - Subscriber Model	Real-time and flexible communication through scalable topics ensures efficient communication between devices.
2	Lightweight and Battery Friendly	Reduced computational power primarily on the FPGA allowing for resources to be allocated for the AI model.
3	Reliable Communication	MQTT enables access to different levels of QoS allowing us to categorize messages in different levels of priority.

We will be using a local MQTT broker called mosquitto, which will be hosted on the relay laptop. There is an additional setting needed for the mosquitto.conf (configuration file), to allow anonymous users to connect to the broker. To allow MQTT packets to be sent to and from the relay laptop to the FPGA an SSH tunnel is established on port 1883 on both the board and the laptop.

The payload format sent from the relay laptop to the FPGA is as follows:

Payload Format		
PlayerID (int)	Acceleration (JSON payload)	Gyroscope (JSON payload)

5.3 Concurrency on the Ultra96 and Relay Laptop

On the Ultra96, there are a total of 5 processes: (1) MQTT subscriber, (2) MQTT publisher, (3) eval_client, (4) game engine, (5) AI driver. Due to the computation strain of all of these processes, we will be using solely multiprocessing on the FPGA.

There are a total of 8 multiprocessing queues used between all of the processes on the board, keeping track of hardware data, visualiser responses, manual and ai game engine inputs, final actions produced by game state, as well as the predicted and true game state. A dictionary was previously used, however this led to slower latency and response time from the FPGA. The additional memory cost of using multiple queues was far more manageable.

On the relay laptop, since not much computational power is needed, we will instead be using multithreading. In addition to the threads used for internal communications, two additional threads for MQTT publisher and subscriber were added.

5.4 Communication between Ultra96 and Visualisers

As an MQTT publisher and subscriber, the visualisers are one of the other final data endpoints aside from the eval_server. Naturally, the visualisers are subscribed to game state and action topics so as to update the user on their HP, ammo, etc. as well as display action animations from the AI predicted actions.

When there is a change in state for visibility and rain, the visualiser will send these packets in one topic for the game engine to compute when the player does their next action. The packet format is as follows:

Payload Format		
PlayerID (string)	packetType (string)	isVisible / numRain (boolean / string)

5.5 FPGA Data Pipeline

Depending on the type of packet sent by internal communications through MQTT, the data will either be sent to the AI driver to be predicted into an action or for the case of a gun or IR hit, they will be directly sent to the game engine.

As data is sent to the AI driver, IMU data for each of the players is compiled in their own lists until it reaches a certain threshold - number of data points for 2.5 seconds (conservative duration estimate of an action). This data will then be passed on to the AI driver for prediction and then forwarded to the game engine.

The game engine framework hosted on the FPGA is very similar to the calculations done by eval_server, however, it must be retrofitted to include visibility checks, the removal of player positions and the addition of manual actions such as gun and IR hit.

Originally, there was a check to ensure players could only do one action per round in “samurai style”, however, this logic would break in the case of action timeout from one of the players, causing game states in further rounds to be incorrect. Although it is possible to try and “debounce” this with a timeout, we have decided to remove it entirely as the hardware threshold was already high enough to prevent duplicate actions from players.

Based on the type of action sent from MQTT or the AI driver as well as the visibility and rain check from the visualiser, the game engine will update player game states accordingly and then send the predicted game states to the game state topic on MQTT. When there is a response for the true game state from eval_server, the game engine will update its own dictionary and send another game state object to the game state topic.

5.6 Challenges Faced

From a general overview standpoint, the true challenge of the external communications subcomponent lies in the overall network architecture of the system. Due to the constraints placed on our devices by the SoC firewall, it was challenging to identify the best architecture that would make effective use of our local MQTT broker while also maintaining secure and reliable communication between all devices.

Since our visualisers were unable to directly communicate to the local broker on the FPGA, we decided to move the broker to the relay laptop. But when the firewall blocked communications between devices in the secured network, we decided to move to using a mobile hotspot to connect the laptop and visualisers, allowing them to receive each other’s IPs and communicate through local MQTT. Naturally, this came with its own set of challenges, since the mobile hotspot we were using was spotty at times, the visualiser would sometimes disconnect and would need a clean restart to fix. The forticlient application on a Linux environment was also extremely buggy and could only work with one of our mobile hotspots. Whilst it was possible to boost the network through the use of a router and another SIM card, we felt that it would not be as cost-efficient for the demo. However if played on a larger scale an additional access point would definitely be a requirement.

As mentioned previously, latency was also an issue. To solve this, the code was refactored and improved by changing the existing architecture into multiple messaging queues and removing potentially wasteful formatting checks. As MQTT was inherently reliable, we did not feel the need to go through the data packets one by one to check for inconsistencies. Ensuring payload formats between the different subcomponents and having checks during messaging also acted as sanity checks and improved code integrity.

It was also quite challenging to keep track of the entire network architecture and multiple rough diagrams were drawn to visualise and explain the entire system to my teammates. There were also a lot of scripts that needed to be run and hence a lot of terminal tabs, however this was quickly solved by bash scripting and tmux.

Section 6 Software/Hardware AI

6.1 Ultra96 Synthesis and Simulation Setup

The implementation approach utilizes two distinct workspaces to develop and deploy the neural network system. The first workspace focuses on model development using PyTorch, providing a robust environment for training and validating the neural network. This environment will be used to prototype the model, ensure its effectiveness, and extract the necessary model parameters(weights&biases) for FPGA implementation.
The second workspace centers on the FPGA development using Vitis HLS and Vivado, where the trained PyTorch model will be translated into hardware. The neural network's architecture(layers and number of neurons), weights and biases will be converted from PyTorch to a hardware description that can be synthesized for the Ultra96-V2[8] board. This dual-workspace approach allows for rapid prototyping in software before committing to hardware implementation.

6.2 Neural Network Model and Pipeline Details

PyTorch was chosen over Tensorflow because it seems more pythonic and straightforward.
From our research, there is no correct amount/number or formula used to select the number of layers and neurons, and it is mostly empirical(so the individual could experiment, test and change multiple times and adjust it as the individual wishes to).

Our PyTorch model architecture has a three-layer architecture. It consists of the Input layer, First hidden layer, Second hidden layer, and the Output layer. The first layer has 128 neurons employing ReLU activation functions, the second layer has 64 neurons also with ReLU. Modules installed includes numpy, scikit-learn, seaborn, and more. The number of extracted features determines the size of Input layer. Features we extracted include Mean, Standard Deviation, Minimum, Maximum, Median, Skewness, Kurtosis, for all three x,y,z axes, and for both accelerometer and gyroscope. The Output layer size is 8, corresponding to eight(8) activity classes(badminton, bomb, boxing, fencing, golf, logout, reload, and shield).

The PyTorch model training order is : load data → scale raw data → extract features → select features → encode labels → split data → train model.
Before training, all features are standardized using scikit-learn's StandardScaler to ensure consistent scale, while action labels are transformed using LabelEncoder.
To optimize model performance, highly correlated features with correlation coefficients exceeding 0.9 are identified and removed, reasons for doing so includes to reduce multi-collinearity, improve computational efficiency and more stable training, ultimately to maintain most of the information while eliminating redundancy.
The training process employs cross-entropy loss, apt for this multi-class classification problem. Adam(Adaptive Moment Estimation) optimization is used and configured with a learning rate of 0.001. Using a batch size of 32 and training over 20 epochs, the model demonstrated rapid improvement, with

loss values decreasing from 1.0469 initially to 0.0410 by completion. Data is partitioned with an 80-20 train-test split, using a fixed random seed for reproducibility.

We took around 100~150 samples per activity per person.

For our laser tag action recognition system, a Multilayer Perceptron (MLP) architecture has been selected, instead of e.g. Convolutional Neural Networks(CNN), as CNN seems to be more apt for other use cases e.g. analyzing visual data. For details of MLP, can refer to Prof Shiuan's lecture slides.
The choice of MLP is motivated by several factors such as 1. Computational efficiency for real-time processing, 2. Suitable complexity for the classification task, 3. Straightforward implementation on FPGA hardware, 4. Good balance between performance and resource utilization.
MLP requires precise number of elements. I used a python script "check_number_of_elements_in_array.py" to double check the number of elements in the array, alternatively using LLM can help to check too.

The network will process sensor data from accelerometers and gyroscopes to classify different laser tag actions. For initial testing and validation, several publicly available datasets could be used, e.g. UCI's Machine Learning Repository's Human Activity Recognition dataset, Kaggle's Run/Walk Accelerometer dataset, etc.

One critical point that I have learnt and took very long to debug/figure out during the first part of the semester for the individual AI subcomponent, is that the test sample data loaded inside the HLS testbench file for Csimulation, must be scaled in the same process as the PyTorch model. Similarly and more importantly, when the FPGA is doing real-time live inference, cannot just use raw data, and the raw data must exactly match the process(e.g. Feature extraction and scaling) as training the PyTorch model.

Another important point in the Pynq driver script, is that the line "self.mlp.register_map.CTRL.AP_START = 1" is necessary. This is because for "#pragma HLS INTERFACE s_axilite port=return", the "ap_ctrl_hs" protocol, keeps the accelerator idle until the AXI-Lite's "AP_START" control register bit is set to 1.
(One alternative is to use "#pragma HLS INTERFACE ap_ctrl_none port=return", but I choose to follow TA)
Another important point is that in the HLS header file, the HLS IP's AXI port is declared as ap_axis<32,0,0,0>, meaning each transfer is 32 bits wide. In the Pynq driver file, the data bit width must match too.

6.3 Training and Testing Methodology

The training process will follow a structured approach to ensure robust model performance. The dataset will be split into training (80%) and testing (20%) sets. The following methodology will be employed:

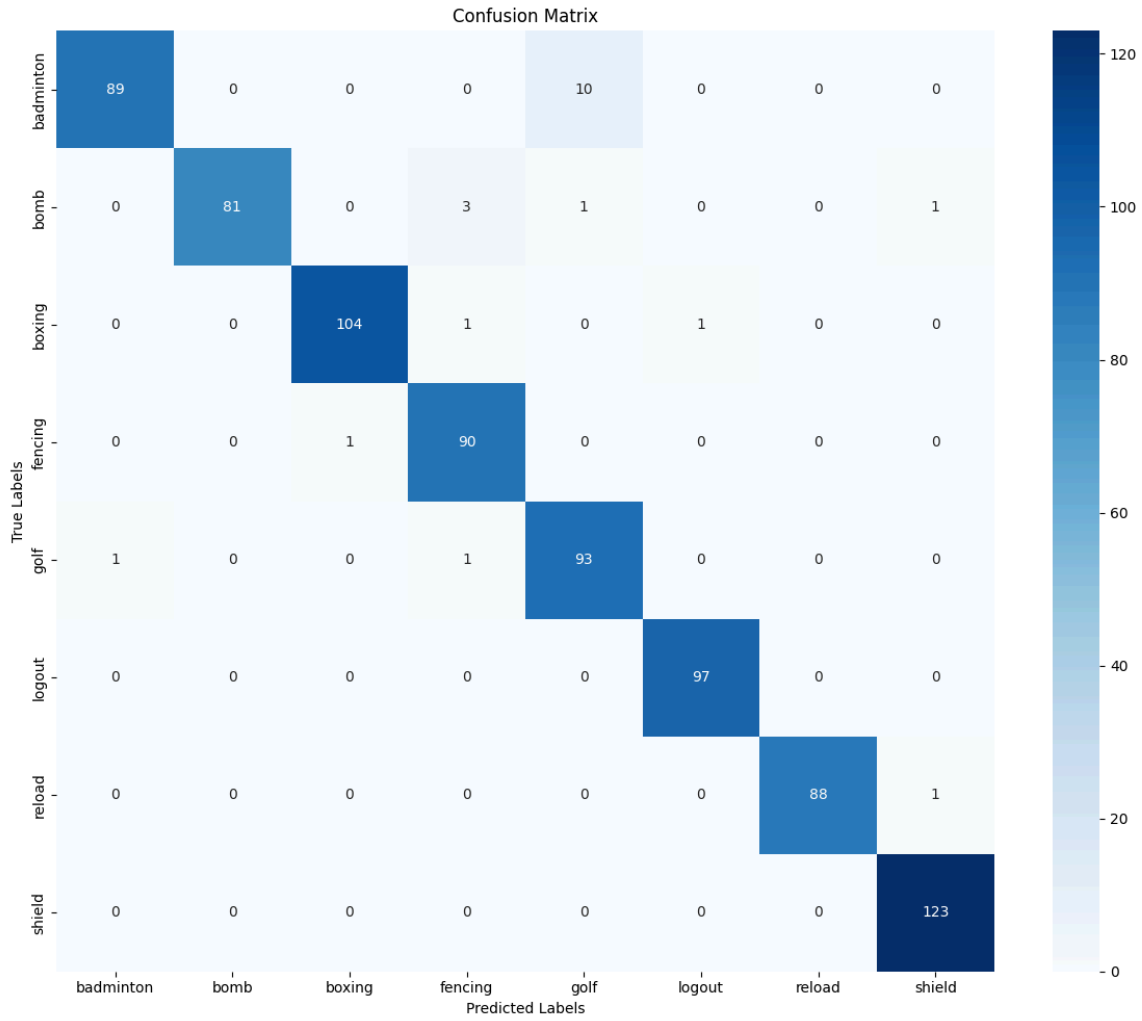
- Data preprocessing for noise reduction and normalization
- Feature extraction from raw sensor data
- Cross-validation to ensure model robustness

- Performance evaluation using accuracy, precision, and recall metrics

	precision	recall	f1-score	support
badminton	0.99	0.90	0.94	99
bomb	1.00	0.94	0.97	86
boxing	0.99	0.98	0.99	106
fencing	0.95	0.99	0.97	91
golf	0.89	0.98	0.93	95
logout	0.99	1.00	0.99	97
reload	1.00	0.99	0.99	89
shield	0.98	1.00	0.99	123
accuracy			0.97	786
macro avg	0.97	0.97	0.97	786
weighted avg	0.97	0.97	0.97	786

Overall Accuracy: 0.9733

Performance metrics showed good classification capabilities, with an overall accuracy of 97.33%. Per-class precision ranges from 89% for golf actions to nearly full accuracy for bomb and reload classifications. Recall values span from 90% for badminton to near full accuracy for logout and shield actions.



The diagonal values of the Confusion Matrix shows correct predictions, whereas the off-diagonal values are the misclassifications.
The confusion matrix exhibits minimal misclassification between action types. However, there is a notable confusion occurring between badminton and golf movements. This is likely due to similarities in badminton and golf motion patterns.

In the Csimulation testing step, one feature we felt that is unique and interesting is that, we loaded the "feature_data.txt" file in the testbench file. When testing different sample data of different activity, we simply change the sample index number of our "load_test_sample" function in "mlp_layers_tb.cpp", this would correspond to and load a different row of data sample. This approach is far more efficient and elegant, compared to repeatedly copying and pasting hardcoded test data values. I also prefer to use absolute path, rather than relative path, so that it is certain that the correct path and correct data file is used.
I initially manually counted rows which very inefficient, I then used Notepad++ to see the row number. (Note that e.g. for sample index 0 in the feature data text file, I need to use row 1 of Notepad++.)

(Note : While “C simulation” can use file I/O operations, “C synthesis” cannot use I/O operations. This is the reason that, copying and hardcoding the weights&biases into HLS header file, is necessary.)

6.4 FPGA Implementation Details

The implementation of our neural network on the Ultra96 FPGA follows a comprehensive approach utilizing High-Level Synthesis (HLS) with C++. Our implementation strategy encompasses several crucial phases to ensure efficient hardware deployment and optimal performance of the laser tag action recognition system.

The foundation of our implementation begins with the translation of our trained PyTorch model into FPGA-compatible components. This process starts with extracting the trained weights and biases from our PyTorch model and converting them into text files.

One critical step in this phase requires us to choose the data type of our sensor data values. Initially, balancing accuracy with hardware efficiency is important, but after multiple tests, we prioritise the precision and accuracy of our model inference; correct prediction is most important. Although fixed point numbers representation takes less memory/resources and is more efficient compared to floating point number representation, correct prediction is of utmost importance. 16-bit fixed point representation has shown to be quite inaccurate, 32-bit fixed point is a good improvement, and we ultimately chosen floating point numbers as it is the most accurate. The speed/latency trade off is minimal, merely milliseconds.

We have analysed and have determined that the optimal bit width for both weights and activations is 32bits, ensuring we maintain classification accuracy while having optimal resource usage. These optimized parameters are then stored in the FPGA's block RAM for efficient access during operation.

The hardware architecture implementation forms the core of our design, where we develop several key components. We create specialized matrix multiplication units to handle the neural network's layer computations efficiently. The design includes ReLU activation function modules, implemented with careful consideration of hardware resources. To maintain continuous data flow, we implement a sophisticated input/output buffering system, complemented by memory controllers that manage weight access efficiently. This architectural foundation ensures smooth data processing through the neural network pipeline.

System integration represents a crucial phase where we connect various components into a cohesive system. We implement an AXI interface to facilitate communication between the processing system (PS) and programmable logic (PL) components. Direct Memory Access (DMA) channels are established for efficient data transfer, minimizing processing overhead. The system includes carefully designed control logic that manages operation sequencing and ensures proper timing of all components. A particular focus is placed on integrating the sensor data input stream, ensuring seamless data flow from the IMU sensors through the processing pipeline.

Once the AI Subcomponent is tested to be in working order, we thought deeply about some optimization strategies, to enhance system performance. Initially we have also tried using HLS

optimisation/performance pragmas, such as to partition memory and parallelize the matrix operations instead of using sequential processing, but ultimately decided not to use HLS optimisation pragmas. Reasons are to opt for simplicity instead, and reduce risks of buggy behaviour. HLS pragmas also increase FPGA resources usage, e.g. the BRAM and LUTRAM. Another secondary reason is using HLS pragmas increases the time for synthesizing bitstream in Vivado, from minutes to hours.

6.5 Performance Evaluation Strategy

- The evaluation of the implemented system will focus on three key aspects:

Timing Analysis

- Measurement of inference latency
- Real-time performance validation
- System response time assessment

Power Consumption

- Static power measurement
- Dynamic power analysis during operation
- Overall power efficiency evaluation

Resource Utilization

- FPGA resource usage monitoring
- Memory utilization assessment
- System bottleneck identification

This comprehensive evaluation approach will ensure that the final implementation meets both the functional requirements and hardware constraints of the laser tag system.
The success of the implementation will be evaluated based on:

- Classification accuracy above 90%
- Almost Real-time response
- Efficient resource utilization (< 70% of available FPGA resources)
- Power consumption suitable for portable operation

Section 7 Software Visualizer and Game engine

7.1 Visualizer Design

Our visualizer shows the following critical data in real time:

- Primary Game Stats:
    - Player Health (HP): Displayed as a health bar or numeric value at the top-left corner.
    - Shield/Status Effects: Visual overlays (e.g., glowing outline for shielded players).
    - Number of shields left.
    - Number of bombs left.
- Opponent Status:
    - Player Health (HP): Displayed as a health bar or numeric value at the top-left corner.
    - Shield/Status Effects: Visual overlays (e.g., glowing outline for shielded players).
    - Number of shields left.
    - Number of bombs left
- AR Effects:
    - Trajectories: Laser beams or projectile paths when shooting.
    - Fencing and boxing effect: Achieved with sword or glove prefab.
    - Shooting effect: Achieved by moving the gun prefab back/forth or tilt quickly.
    - Hit/Miss Feedback for gun shot.
    - Bomb Zones: Snowfall until the end of the game.
    - Shield effect: Show the shields of both self and opponent.

7.2 Visualizer software architecture

1. Framework discussion

   Initially, AR Foundation was chosen as the core AR framework due to its native support and cross-platform capabilities. However, it was later replaced with Vuforia after observing its superior performance in detecting QR codes at longer distances, which is crucial for our game environment where visual markers may be far from the camera. Vuforia's image target tracking proved more stable and reliable for maintaining accurate overlays on moving opponents and objects.

2. Visualizer game object architecture

   The visualizer is structured with modular prefab components representing each AI action—such as golf, badminton, and others. Each prefab is governed by its own controller script, responsible for handling state changes and animations based on incoming game data.

A canvas is persistently overlaid in the camera space to display key UI elements like the scoreboard, action prompts, and AR overlays.

For compatibility with Google Cardboard and other head-mounted displays, a CardboardManager component is integrated to enable stereoscopic split view rendering.

3. Communication with the game engine

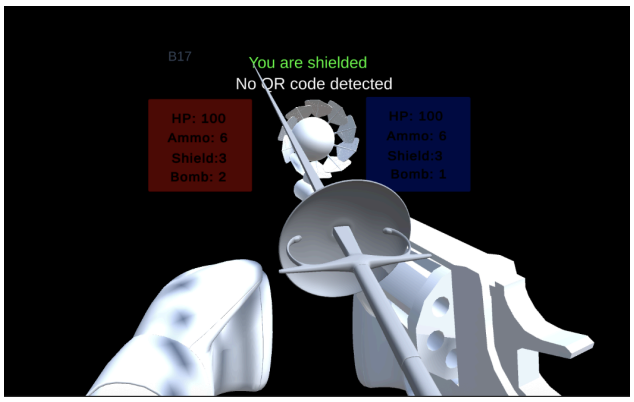
The communication between the visualizer and the game engine is handled using MQTT, a lightweight and efficient publish-subscribe messaging protocol. The visualizer subscribes to two key topics—`game_state` and `ai_actions`—to receive real-time updates on game progress and AI-driven opponent moves. In return, it publishes messages to the `visualiser_answers` topic, which contains the results of visibility checks (i.e., whether a player or object is currently in view) and rain detection information, which may impact gameplay visuals and strategies.

7.3 Visualizer overlay

We used a world-space canvas anchored to the camera to ensure that critical game information remains constantly visible to the user regardless of their view orientation. This overlay includes the scoreboard, and player status indicators. To maintain immersion while providing real-time feedback, the UI elements are semi-transparent and positioned strategically to avoid obstructing the main action area.

The scoreboard updates dynamically based on incoming data from the `game_state` topic via MQTT, reflecting changes in HP, shield, ammo, etc.

7.4 Design of the Visualizer



7.5 Issues faced

1. Controlling Golf Ball and Badminton Trajectory

Achieving a realistic and controllable projectile flight for the golf ball and badminton proved challenging. Initially, I attempted to rely on Unity's built-in physics to simulate force and gravity. However, this approach led to unpredictable and often unsatisfactory arcs, making gameplay less intuitive and harder to control.

To solve this, I implemented Bezier curves to manually model the flight path of each shot. The curves were dynamically generated based on player input and the target opponent’s location. This method gave us full control over the shot's shape, allowing us to simulate realistic arcs while maintaining smooth and predictable gameplay. It also made it easier to synchronize animations and improve hit accuracy.

2. Updating External Comms for the Bomb Check Mechanism

Our game includes a bomb check feature, which depends on environmental anchoring to track. However, during testing, we noticed that these anchored objects would occasionally drift from their intended positions due to slight mismatches between virtual anchors and the real-world environment.

This positional drift introduced a major issue: it became extremely difficult to verify the correct number of “bomb hits” or to detect impacts accurately. To mitigate this, I added an additional loop check everytime the opponent moves to a new positon.

3. Wrong Initial Choice of Image Tracker Package

In the early stages, We built the AR-based tracking system using AR Foundation, assuming it would suffice for our multiplayer setup. However, as development progressed and we tested the two-player gameplay mode, we discovered that AR Foundation’s image tracking lacked precision—particularly for targets more than 1 meter apart. This severely impacted the game's ability to recognize and sync objects in a shared AR environment.

Faced with a tight deadline, we made the decision to migrate the entire image tracking system to Vuforia within a single day. Vuforia offered better long-range image detection and more consistent tracking across devices. While the migration was intense, it paid off: we regained stable and accurate tracking between players, enabling a much smoother multiplayer experience.

Section 8 Future Work : Societal and Ethical impact, Extension

While I would argue our design would be a bad example for laser tag, our achievements in AR, On-Cloud AI, and IoT integration is a huge technical feat. A more generalization of our system branches into the 3 categories.

IoT is edge devices that we use every single day. Having improved our lives for the past decade, the possibility of IoT is endless. The IoT used in the laser tag system can be generalized into wearables such as our smartwatches, where it can talk with our phone (relay laptop), and that the phone communicates with the cloud (Ultra96). Or a cool way is that in the event of a natural disaster, our Beetle's can act as nodes, to send and receive vital information such as medical supplies shortage and emergency cases. In a more individual manner, we can equip the traditional medical equipment and sensors to the Beetle and allow remote monitoring of patients on Cloud or local network. Another use-case could be in the agricultural sector, where smart low-power edge devices are equipped with soil and rain sensors, and another is equipped with fertilizer dispenser actuators, and that these smart device mesh can automatically maintain the farm, and alert the farmers. Or even, the maintenance of our global warming, where the Beetles are deployed on forests to check for early signs of forest fire, or ice caps, to check for ice cap melting. We can also deploy Beetles in industrial sites to make sure that the waste made by the plants are in accordance with regulations. The generalization that we seek to discuss in case of Beetle's, Laptop Relay, and FPGA integration is that using smart IoT connected to bluetooth and cloud communication, we can sense any physical phenomena and analyze it in cloud. Additionally, we have a choice of doing it reliable or unreliable.

Our AR laser tag system, though originally developed as a fun and competitive game, has the potential to scale far beyond that. With minimal changes, it could be adapted for training simulations, team-building exercises, or even emergency services coordination. In a disaster relief scenario, for instance, responders wearing AR glasses could use the system to identify danger zones, find resources, or get navigational support in real time.

Of course, as AR wearables become more immersive and ubiquitous, they raise important ethical concerns, particularly around privacy and data security. When everything from gaze direction to physical movement is being tracked, it's essential to ensure that this data is stored securely, anonymized where possible, and never used without user consent. Public spaces, especially, must be protected from turning into zones of constant surveillance under the guise of augmented utility.

One exciting addition to the AR laser tag system could be a smart, automated mini car, a physical robotic companion that acts like a copilot or support drone for the player. This car could:

- Follow the player around the arena using visual or Bluetooth-based tracking.
- Act as a mobile shield, deploying temporary cover in tactical situations.
- Carry ammo or power-ups that the player can virtually "grab" through the AR interface.
- Function as a scout, moving ahead to map terrain or highlight hidden opponents.

- Even emit light or sound effects to distract enemies or signal team actions.

This robotic teammate wouldn't just add tactical depth, but also a fun, futuristic twist, blurring the line between physical robotics and digital AR. In future versions, it could even communicate with the player via voice or gestures, making it a real-time interactive assistant in both game and non-game scenarios.

Our Human Activity Recognition(HAR) laser tag system demonstrates the power of wearable technology combined with real-time AI action recognition. There are other use cases beyond gaming applications, that could have meaningful societal impact.
Our AI activity recognition system could be employed in patient monitoring and preventive healthcare. By retraining our MLP model on movement patterns associated with Parkinson's disease, stroke recovery, or mobility impairments, medical professionals could gain continuous, objective assessment of patient conditions outside clinical settings.
In the context of SG's ageing population, reliable detection of fall risk behaviors in elderly in HDB apartment flats, lobbies, and stairwells, could be useful to alert caregivers, resulting in more efficient care and a better peace of mind for working adults and caregivers.
In addition to detecting falls, our system could be adapted to detect changes in gait that precede falls, enabling proactive interventions. The model's ability to distinguish between similar actions (as demonstrated in our confusion matrix analysis) provides the foundation for detecting subtle movement changes that might indicate deteriorating health conditions before they become critical.

One profound societal impact lies in the domain of accessibility applications. Our HAR technology could be repurposed to create gesture-controlled interfaces for individuals with limited mobility. By training the neural network on personalized gesture sets, people with physical impairments could interact with digital devices and smart home systems through movements they can comfortably perform.

One improvement we really think that would be very useful is Edge-Cloud Hybrid Processing; While our current implementation runs inference directly on the FPGA, we could implement a hybrid approach where complex environmental analysis happens in the cloud while time-sensitive action recognition remains on the edge device. This could leverage compute resources from the cloud, instead of worrying about the limited resources of the Ultra96-V2 FPGA.
Another improvement that could potentially make the free-play game more fun would be, having Cross-Reality Integration, in a way that the system could blend physical objects in the environment into the game mechanics. For instance, real-world obstacles could be automatically detected and incorporated as cover elements within the game.
Regarding privacy, our system currently tracks physical movements, but future iterations might incorporate more personal data. The continuous collection of biometric data raises questions about data ownership, consent, and potential surveillance. Clear opt-in policies and transparent data handling practices would be essential.

If we combine all the three things, we get the AR laser tag system. The laser tag system looks very niche, however it is actually quite generalizable. Imagine if we deploy this in the hospital, where the nurse can use the AR glasses, and each patient's vital information is anchored on the door! So when the nurse walks through the hallway, she can just see the vital information outside the door and prioritize her schedule?

This would enable real-time triage, reduce manual chart-checking, and minimize errors in high-stress environments. The same system could guide maintenance technicians through complex repairs with overlaid schematics or help warehouse workers locate items efficiently. The key innovation is contextual, hands-free information delivery, a paradigm shift from screen-based interfaces to spatially aware computing.

However, this potential comes with profound ethical responsibilities. As a rule of thumb for us, any system collecting biometric or behavioral data must adhere to three principles:

1. Minimal Viable Data: Collect only what's absolutely necessary (e.g., limb movements for activity recognition, not facial features).

2. Explicit Consent Loops: Require opt-in permissions that explain data usage in plain language, with granular controls (e.g. "Share my movement data with researchers but not advertisers").

3. Frictionless Anonymisation: Automatically strip identifiers from data streams at the edge device level before cloud processing.

While our system may not revolutionise laser tag, its true significance lies in demonstrating how IoT, AR, and edge-cloud AI can converge to solve real-world challenges, from healthcare and disaster response to accessibility and environmental monitoring. The technology framework we've developed is not just a game, but a scalable platform for context-aware computing, prioritizing ethical design (minimal data, explicit consent, and anonymisation) alongside technical innovation. By focusing on human-centric applications, like AR-assisted nursing or AI-powered fall prevention, we've shown how playful experimentation can seed solutions with profound societal impact. The future of this work isn't just better gadgets, but intelligent systems that enhance lives while safeguarding privacy and trust.

References

[1] https://wiki.dfrobot.com/Bluno_Beetle_SKU_DFR0339

[2] https://www.instructables.com/Accelerometer-MPU-6050-Communication-With-AVR-MCU/

[3] https://www.researchgate.net/publication/337664623/figure/fig3/AS:831395677093888@1575231584108/MPU6050-with-the-pin-layout-and-axes-orientation.ppm

[4] Nyangaresi, Vincent & Silvance, O & Abeka, Silvance & Arika, Rebecca. (2018). CSEIT1833460 | Low Latency Automatic Repeat Request Protocol for Time Sensitive GSM-Enabled Smart Phone Video Streaming Services.

https://www.researchgate.net/figure/Stop-and-Wait-Protocol_fig1_324653865

[5] Siracusano, Giuseppe & Bifulco, Roberto & Kuenzer, Simon & Salsano, Stefano & Melazzi, Nicola & Huici, Felipe. (2016). On the Fly TCP Acceleration with Miniproxy. 44-49. 10.1145/2940147.2940149.

[6] https://web.archive.org/web/20140214120404/http://www.bluetooth.com/Pages/low-energy-tech-info.aspx

[7] https://www.geeksforgeeks.org/advanced-encryption-standard-aes/

[8] Ultra96 Datasheet and Guides : https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/ultra96-v2/