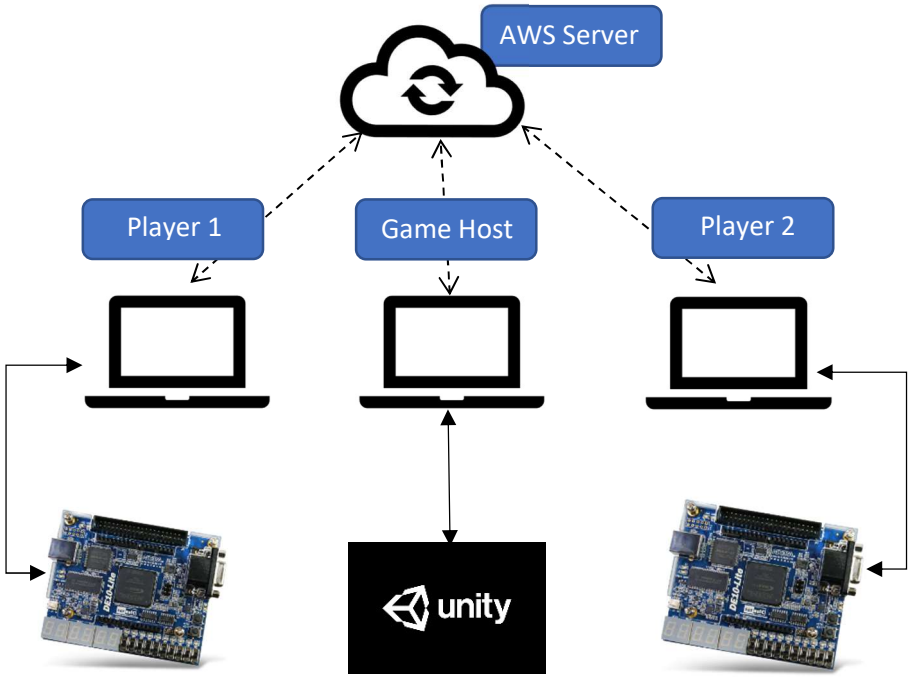
Information Processing Coursework

The purpose of the system

The given objective was to develop an IoT system with multiple nodes that process data captured by an accelerometer and interact with a cloud server to facilitate the exchange of information. Our team has decided to implement a remotely hosted game using the FPGAs as controllers:



Overall architecture of our system

Throughout a world pandemic, friends need to find ways to interact with each other and have fun from afar. This project puts together a “retro” 1 vs 1 Arena tank stand-off, that allows two FPGA owners to connect to a server from their laptops and use them to defeat their opponent in battle:

-RULES -

(1) Use your bullets to push your opponent outside of the map
(2) Do not touch the fire or you shall perish
(3) First to 10 points WINS

-CONTROLS-

Tilt the FPGA to the direction you want your tank to go.
Button 1: shoot bullet
Button 2: drop a bomb

The implementation of our system

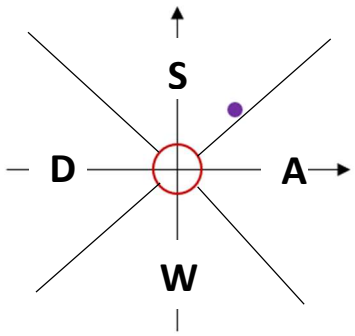
1) Communication between FPGA and PC

Communication between the FPGA and the PC is permitted through Quartus and its incorporated Eclipse tool. On a hardware level, the platform designer was used to configure the FPGAs so that the accelerometer, buttons, and LED lights were accessible.

Both players use the same C and Python programs designed to translate their FPGA movements into simulations of key presses. In Eclipse, the C program retrieves the position sent by the accelerometer from the tilt exerted on the FPGA. It then transfers that information to the Python file which will translate the position to a singular character "w", "a", "s" or "d" for Player 1 and Player 2. Other features such as “shooting” or “bombing” were added using the available buttons of the boards, transmitting the characters ‘e’ and ‘r’ respectively. These singular characters will then be transmitted one at a time to the server.

Our FPGA also undergoes real-time processing of the accelerometer values, using an FIR to smooth user input and minimize the effect of outside disturbances. After some playtesting, we ended up using a 2 tap filter, as using any more taps led to the player’s response feeling too sluggish as their tank was unable to perform more precise maneuvers. The filter is implemented using fixed floating-point representation to maximize computational efficiency, using the alt_32 integer data type and arithmetic shift operators to average out values instead of having to use the more expensive simulated floating-point operations.

The Python file focuses on mapping the positions transmitted from the JTAG UART to a useable character for the PC. Each letter has a range of positions allocated to it. These ranges were defined through a series of try-outs and testing. For example, when tilting the board forward, the accelerometer was outputting negative y values and small x values. After multiple attempts, the diagram below illustrates the identified ranges necessary for the game:



The purple dot is an example of an x-y position that would be translated to the character “s” indicating the tank to move downwards.
The red circle illustrates the dead zone, in other words, the zone where no movement is executed and doesn’t send any output. After testing, the radius of this circle has been set to 50, to ensure that the program isn't too sensitive to inputs close to zero.

Our game only allows movement in the 4 cardinal directions, so whichever direction has the highest value of tilt will be translated to a character and sent to the server to be used as an input for the game.

The resources utilized for the use of the DE-10 Lite all refer to the labs taught in the first half of the term. All the handling of the buttons/ hex digits was inspired from Lab2, the handling of the accelerometer from Lab 3 and finally establishing the connection between eclipse and Python was taken from Lab4.

2) Communication between PC and server

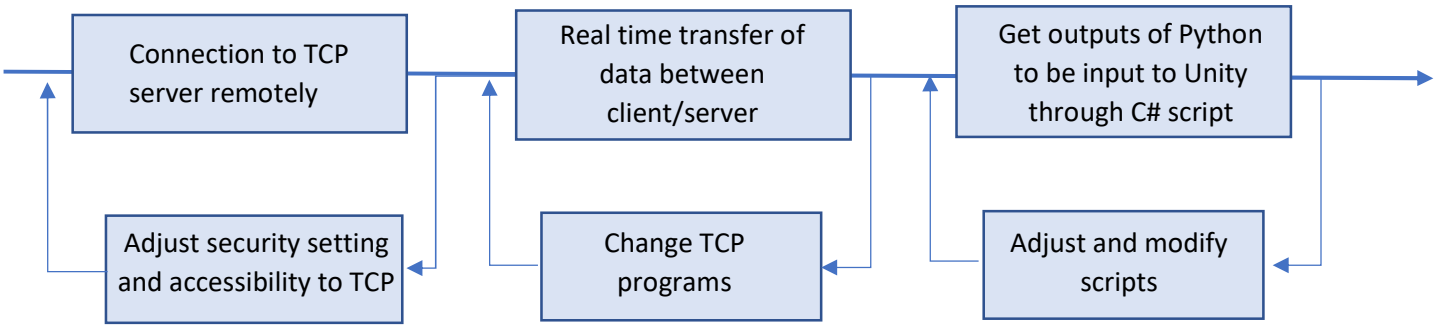
For this project, the main role of the server is to allocate the different roles to each of the three clients who connect to it and encode what it receives from two players, so it is understandable for unity. To transfer data across different computers there were numerous methods available. What we needed was an almost instant exchange of data amongst our computers. Protocols like TCP, UDP and HTTP used in socket programming offered this type of service. TCP was chosen for two main reasons. Firstly, as it is connection-oriented, it ensures the connection is established and maintained until the exchange between the client/server is completed and closed once completed. Since it can guarantee delivery of data to the destination, the likeliness of getting package errors is very small. Furthermore, transmission using TCP is fairly fast, so it is unlikely to act as a bottleneck in the scope of our system.

The server was decided to be ***structured*** in the following manner:

(1) The socket is set up and listens for incoming connections.
(2) Once three connections are made, it verifies the identity of each:
    a. One connection from Unity
    b. Two connections from two different players
(3) It notifies its clients that all connections are correctly established.
(4) Moves into a constant loop that transfers data from the players to Unity and vice-versa
    a. The server gives each player an identifiable prefix such as x for player 1 and z for player 2 which allows unity to distinguish which player is sending its information.

To achieve the desired TCP server/client connection, multiple stages of testing had to be undergone to ensure each step was working as desired. Steady improvement on the functionality of the scripts establishing communication with our devices was also made along the way.

Server/Clients Communication testing approach



To begin with, a clear connection to a TCP server remotely (i.e. on an AWS Server) had to be ensured. To establish it, a bash script was written to be ran from a local PC: the script securely connects to the server and uses SSH agent-forwarding to pass the relevant GitHub permissions to the AWS instance. A script which installs required tools on the instance and pulls needed files to run the TCP server from our repository is executed. Then, following compilation and execution of those files, the server is ready and awaits connections. Since the whole process is automated and transferable, any PC with the necessary GitHub permissions to access the repository can run the script and the server will automatically set up from their host. This way, that PC is now ready to connect to the server and play the game.
Some security setting adjustment had to be made so incoming TCP connections were permitted from any IP.
The next step consisted of verifying the real time transfer of data between the client and the server, which was tested by altering the TCP programs.
Once the overall shape and function of the client and server scripts were understood it was necessary to port them to the platforms that would eventually run them (C#/Unity and Python) and ensure that it remained working as expected.

As observable, most of the data processing is done within the scripts and the server is solely focused on ensuring that information is properly transmitted to its correct destination.

```
Ping statistics for 18.134.146.61:
    Packets: Sent = 50, Received = 50, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 12ms, Maximum = 144ms, Average = 21ms
```

The diagram shows diagnostics for pinging the AWS server from a client PC. We see that the average time for a round trip is 21ms, which is extremely fast and likely unnoticeable to us in the scope of the entire system. However, we can also see that there was a maximum time of 144ms which is almost seven times slower than the average. While infrequent spikes to this degree are unlikely to have a detrimental effect, it may introduce a noticeable delay for user inputs and if the round trip to were increase beyond the measured maximum there might be worse effects.

```
kai@Kai-VM:~/Documents/InfoProc/InfoProc_Project/TCP/CppC$ ./Player
Socket Created
Valid IP
Connection Established with Server
Hi this is the server. Who am I connected to?
ID Sent
Ready
```

```
Read from Unity zi
Sent to Player1
Read from Player1 w
Sent to Unity
Read from Player2 a
Sent to Unity
Read from Unity xa
Sent to Player2
```

The snippets of code show what we output to terminal during the execution of the programs. The right shows the Server while it creates the socket and connects to all the clients and above is the client's perspective. The top right shows the server's actions in the process of the game. It reads information as it receives it and sends it to the correct destination. For data received from Unity it decodes the destination from the first character in the message and sends the relevant information to either Player 1 or Player 2.

```
kai@Kai-VM:~/Documents/InfoProc/InfoProc_Project/TCP/CppC$ ./Server
Initialising
Socket Attached
Listening
Connection Established with 127.0.0.1
Hello Sent,
We are connected to Unity
Listening
Connection Established with 127.0.0.1
Hello Sent,
We are connected to Player
Listening
Connection Established with 127.0.0.1
Hello Sent,
We are connected to Player
Unity Assigned
Player1 Assigned
Player2 Assigned
All good!
Connections Configured. It's going alright!
Player1 5
Player2 6
Unity 4
```

3) Communication between unity and server

Once the accelerometer records the FPGA's position, which is then outputted by the JTAG UART to the Python script, an individual character is sent to the server. It will process it and add a prefix to it: "x" for player 1 and "z" for player 2. This is where Unity's C# program does its part. It receives from the server a pair of characters such as "zw" or "xw" and proceeds to obtain its needed information. The first character give information on which player wants to move thus indicating Unity which tank will have to move. The second character represent the letter indicating the direction the player wishes to move its tank towards.

```
if (!string.IsNullOrEmpty(x))
{
    if (x[0] == 'x')
    {
        x2 = x[1];
    }
}
if (x2 == 'w')
{
    vertical = 1;
    horizontal = 0;
}
else if (x2 == 's')
{
    vertical = -1;
    horizontal = 0;
}
else if (x2 == 'a')
{
    horizontal = -1;
    vertical = 0;
}
else if (x2 == 'd')
{
    horizontal = 1;
    vertical = 0;
}
else if (x2 == 'e')
{
    horizontal = 0;
    vertical = 0;
    fire = true;
}
else if (x2 == 'r')
{
    horizontal = 0;
    vertical = 0;
    setbomb = true;
}
```

The figure on the left is an extract of the C# file which processes received strings from the server.

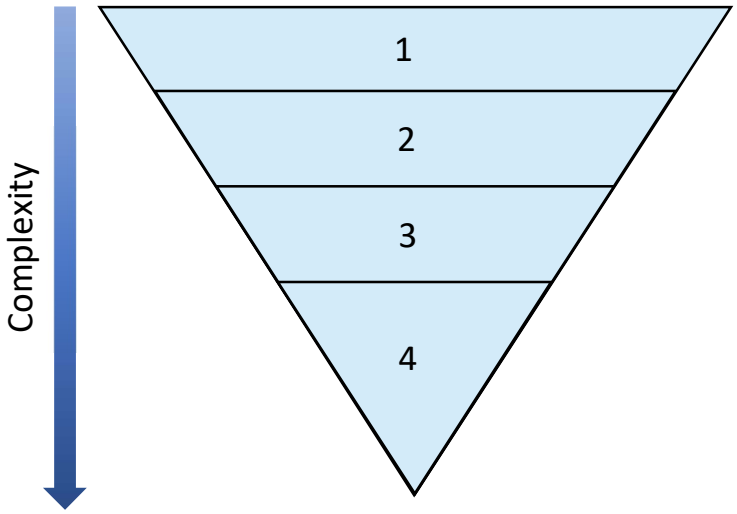
As the code displays, the first if statement verifies the first character of the string to identify the player. Once the player is identified, x2 is then updated: depending on the received letter the program will output its mapped directions, so the tank moves accordingly in the game.

The same process is executed for the second player only its identifiable character is an "x" and not a "z".

When designing the gaming platform, the decision of making a C# script that allows multithreading, and, allocating one set of keys for two players were taken. Usually when using TCP, once the connection is established the client only has the possibility to listen to what the server is sending. Writing a script allowing multithreading is essential when designing a real time game. It allows Unity to listen to the server while effectuating changes to its game at the same time thus making the game-time smooth and responsive.

Once the game was up running, perfecting the connectivity with the server and the clients was gradually improved through tests at different scales:

Full game's functionality testing approach:



1) Playing with from the host's PC to test all keys functionality and the tanks movements
2) Connection between one remote player's PC and the host's PC with the same IP, with different IPs, from a virtual machine connected to the United States. This allowed to test how stretched the connectivity was and keep an eye on the latency from the different locations.
3) Connection between two players PC with the host PC, same level of testing as previously to ensure everything is still working.
4) Connection of one/two players using the FPGAs as their consoles connected to their PCs to test the full game experience.

Performance metrics of the system

After finalizing the system, we edited the Python script to retrieve performance measurements of our system.

1. Timing analysis of communication between the FPGA and the PC

The C file retrieves positions from the accelerometer which are then processed by the JTAG-UART and receives characters from the Python file when players shoot or drop bombs. All actions related to the FPGA such as reading from the x/y axis, reading a button value, incrementing/resetting the scoreboard and printing loser/winner take **2ms**.
The JTAG_UART takes a total of ***997ms*** to send a character to retrieve the position, actually retrieve the position and process it to the Python file. This timing can be broken down into three main steps:

- Sending a character to the NIOS II terminal using JTAG-UART: ***8.90μs***
- Receiving character from NIOS II terminal using JTAG-UART: ***995ms***
- Processing the output from the NIOS II terminal: ***196μs***

2. Timing analysis of communication between the PC and the server

Once the Python file processes the positions, it outputs a character to the server. The time taken to send a character to the server is of ***6.57e-05 seconds***.
Conversely, when the server wants the FPGA to display the points on the LEDs or who is the winner/loser on the HEX display, Python receives a character from the server. The time taken to receive a character from the server: ***7.50 e-06 seconds***.
Therefore, the component contributing the largest time delay is receiving a character from NIOS II using JTAG-UART, causing a significant bottleneck to the performance of our overall system.

3. Further implementations to improve the performance of our system

We tested how affecting the number of taps in our FIR filter affected the overall communication time between our FPGA and PC, to see if there were any further optimizations to be made. When comparing a 4 tap FIR filter to an FPGA implementation without FIR filter, we noted that there was no detectable delay in receiving characters from the NIOS II terminal, which was expected as the primary bottleneck of our system is delay from the use of the JTAG-UART to receive characters from the NIOS II terminal.

To conclude, the communication between our nodes was successfully implemented such that the delay caused by communication doesn't hinder the experience of the player. The tanks also move almost instantaneously after the board is tilted.