# Information Processing Laboratory Report

Group 5 - E.T. Go Home

March 2023

## 1 Introduction

This report highlights the architectural and testing approaches taken in making design decisions for each aspect of the game production. Through strong project management set up in a group Notion and updating code in GitHub repository (`https://github.com/mk1021/E.T.GoHome`) as a team it was possible to work effectively towards proposed goals within a specified time frame. The game is a multiplayer maze game based around the movie E.T. It involves two players competing to traverse rooms with obstacles while collecting powerup's that sabotage the other player. The aim of the game is to get E.T home by moving through all rooms to the end screen, the player with the shortest time wins, and players that score within the top five scores are presented on a leader board. The FPGA was used to move the cursor and display the score, while the server was used to communicate with both clients and store scores in a shared database.

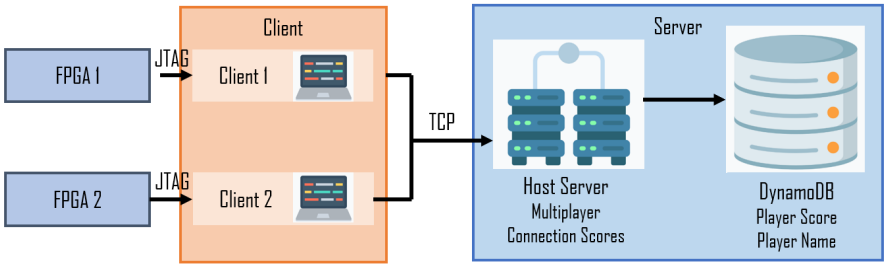


Figure 1: Architecture Diagram

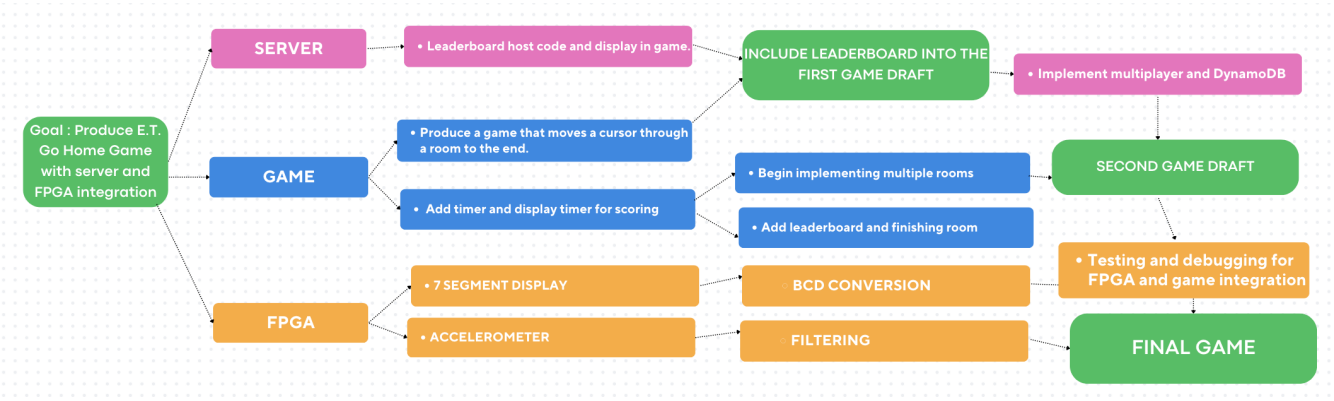


Figure 2: Our Flowchart

## 2 Game Graphics

### 2.1 Setting up the game:

The initial game idea was to trace a path without touching the borders of the maze created, as touching the walls would increase the player's time or alternatively end the game. This idea was quickly modified, due to the precision needed from the FPGA. The idea was then changed to a maze, where the player would have to get through the maze as fast as possible, touching the walls would be of no consequence, but the player would have to avoid the power ups as they could potentially impact the player negatively.

The game was implemented in Pygame, since it was easier to customise the rooms to the requirements and add several mazes and power ups, to make the game more complex. Choosing Pygame also allowed more time to be spent on the hardware and cloud aspects of the game. The hardware only worked on two laptops in the group, therefore, if errors occurred, there would be no backup node to play the game on. The first version of the game used a simplistic design, to ensure functionality. A single obstacle was used within the room and only one room was implemented at first. This meant it was possible to test the basic functionality of the FPGA controls before implementing more complex features. After the first version, server connection was implemented and then FPGA integration. Once integrated, then design and UI considerations were implemented to improve the appearance and game play.

## 2.2 Rooms:

Five rooms of different levels of difficulty, were implemented. Each room was made in individual classes, where white outer walls were added, so that the player would not be able to move out of the rooms unless entering another room. Coloured walls were then implemented to create different custom designed mazes, and power ups added. These were placed so that it would be harder to reach the positive power ups (freeze other player and decrease time) and hard to avoid the negative one (add time):

```
class Room2(Room):
    """This creates all the walls in room 2"""
    def __init__(self):
        super().__init__()

        walls = [[0, -100, 20, 250, WHITE],
                 [0, 400, 20, 250, WHITE],
                 [780, -100, 20, 250, WHITE],
                 [780, 400, 20, 250, WHITE],
                 [20, 0, 760, 20, WHITE],
                 [20, 580, 760, 20, WHITE],
                 [190, 80, 20, 500, GREEN],
                 [590, 20, 20, 500, GREEN]
                ]

        addTime = [[380, 55, 40, 40, "icons8-add-time-32.png"],
                   [380, 280, 40, 40, "icons8-add-time-32.png" ],
                   [380, 505, 40, 40, "icons8-add-time-32.png" ],
                  ]

        subTime = [[80, 505, 40, 40, "clock.png"],
                   [680, 55, 40, 40,"clock.png" ],
                  ]

        for item in addTime:
            addTime = PowerUp(item[0], item[1], item[2], item[3], item[4])
            self.addTime_list.add(addTime)

        for item in subTime:
            subTime = PowerUp(item[0], item[1], item[2], item[3], item[4])
            self.subTime_list.add(subTime)

        for item in walls:
            wall = Wall(item[0], item[1], item[2], item[3], item[4])
            self.wall_list.add(wall)
```

Figure 3: Code snippet of how we created each room

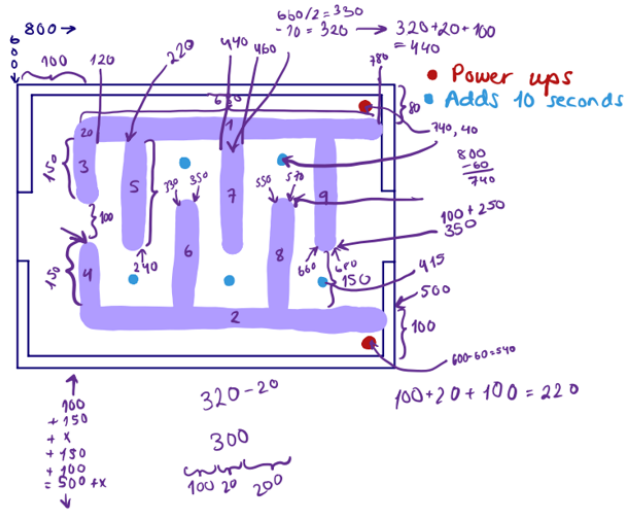


Figure 4: Customised Level annotated with calculations of maze walls and positions of power-ups

## 2.3 Personalisation:

To make the game more personal, an E.T. theme was chosen. The name of the game (Help E.T.!) was displayed at the top at the screen window's header and an icon of an animated E.T. was chosen as the pop-up icon at the bottom in the dock. The backgrounds, player icon and game music, were also customized with pictures of the moon, a black icon of E.T. in a basket, that changes direction with the player's direction, and the theme song from the original E.T. movie.

```
768 mixer.init()
769 mixer.music.load('bensound-summer_mp3_music.mp3')
770 mixer.music.play()
```

Figure 5: Game Pop-up Icon, Player Icon and Game Music (Respectively)

## 2.4 Game Code Formatting:

The format of the code was created to ensure simplicity and readability. The player and room code were placed in specific classes which could then be inherited by other classes, this helped to reduce repetition and made it easy to implement extra features. A separate receive function was created, following testing, as there proved to be latency and inconsistency in sending packets to the server. Therefore, threading for both the FPGA and receive code was used to reduce latency by having the three processes (game, server and FPGA movement) run simultaneously. As time is

treated as the scoring mechanism of the game, multiple functions for updating, stopping and starting time were created to allow powerups to be implemented.

# 3 JTAG-UART Connection

The aim behind the JTAG-UART was to instantiate a connection between the accelerometer on the board and host. This was the key implementation for the player's movement as they control the FPGA. Starting with the code provided to in Lab 4, it was possible to initially send a character down to the board, however this had limitations:

1. Multiple characters (strings) needed to be sent for the accelerometer to communicate with the host. This could have been done by repeatedly opening and closing the subprocess whilst sending a character down each time however this would cause delays in communication. For a fast-paced game that required a quick response, this was not ideal and instead the subprocess command was changed. Instead, subprocess.Popen() was used to communicate with the subprocess whilst the game was being run (using the .communicate() method function) along with multithreading to read from the terminal too, which meant full lines could be read instead of characters.
2. The subprocesses would only return values once the it finished executing. This proved to be problematic as accelerometer values needed to be passed down "live" whilst the game was being played. Hence, this was solved using multi-threading.
3. Fast responses were required. Similarly as mentioned above, subprocess.Popen() and multithreading allowed for rapid reading from the terminal whilst the accelerometer simultaneously wrote to the shell.

# 4 AWS Server Implementation

## 4.1 Progression of ideas and testing:

### 4.1.1 Initial single player leader board:

It was decided the best way to achieve the basic requirement for connection between node and server, was to first implement adding scores to a leader board of top 5 player scores at the end of the game. An initial leader board was made in the host code but would later change it to be stored within a database. The implementation involved sending the players' scores to the server and appending them to a "CurrentScore" list in the host code. The performance metrics that were reviewed for this implementation, included correct output and ordering of player scores into the leader board and readable formatting that could be displayed on the screen. These metrics were tested using print statements in both the server and client code to debug issues such as ensuring the scores were in ascending order and only the top five were printed. Furthermore, to improve testing, when implementing new features a new file would be created to ensure multiple versions of the game and server code were saved before making adjustments.

### 4.1.2 Multiplayer:

After implementing basic functionality, the game was made multiplayer using a server. The first player to connect becomes player one, and the server sends messages to both players when the second player connects. When a player completes the final level, they send a message to the server with their score. The server sends the score to player two if the message starts with "Player1Score:". The metrics used for testing include waiting for two players to start the game, both players able to play the game once connected and both players able to receive the others score. Finally, to ensure robustness of the server to client communication, large packets of data were sent over a short period of time back to the server. Testing showed inconsistent results, threading was incorporated into the client code to constantly check for received data from the server. This resulted in more consistent results during testing, even with poor network connections.

### 4.1.3 DynamoDB leader board:

A DynamoDB database was implemented to store player scores and create a top five leader board. The partition key was determined as the server IP, with the scores designated as the sort key. This setup allowed for the scores to remain stored on the leader board as long as the IP did not change. The sort key was defined as the score to enable retrieval of the top five scores for each leader board. Additionally, this database meant that one could include attributes to the leader board such as the players name which could be inputted by the client at the end of the game. Additionally, the game was tested on multiple IP's and devices to ensure consistent results.

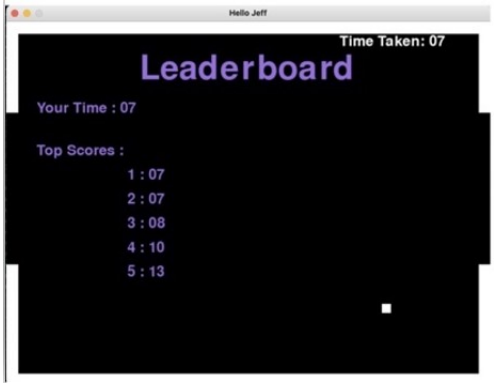


Figure 6: Intial Leaderboard

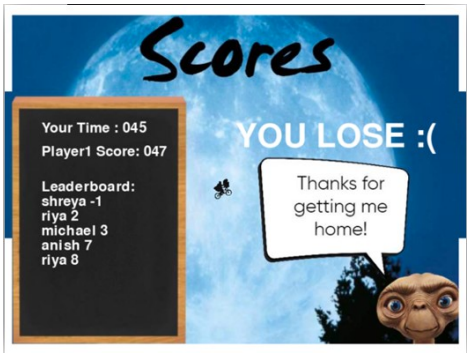


Figure 7: Final Leaderboard and Scores Screen

#### 4.1.4 Power-ups:

Finally, to incorporate continuous sending and retrieval of data from client to server and back, powerups were implemented. Two powerups were added, one freezing the other player for 5 seconds, and the other reversing their controls for 10 seconds. The powerups were implemented through different classes within the game code, and when a player collided with one, a message was sent to the server specifying the powerup type and the affected player's number. The server would then send a message back to the affected player. Testing involved verifying the collision, the server receiving the collision message, and the server sending the powerup message back to the player. The feature was tested first on arrow keys and then FPGA controlled games, with occasional latency and crashes, which were fixed by sending powerup status only once for accurate message reception by the server.


Figure 8: Power-up Display

# 5 Hardware

The purpose of the FPGA hardware functionalities in the game included configuring the accelerometer to provide on-screen movement and for the 7-segment display to illuminate with the final score, both of which communicate with the game via a JTAG-UART connection.
The architecture of the system involved setting up the hardware as shown in figure 9. Connections were memory mapped in platform designer with the appropriate IP cores instantiated to provide the features needed. In Quartus Prime, there was a top-level Verilog sheet that set up the input and output parameters as well as wires to connect components together. A $hex_to_7seg$ Verilog file was generated to ensure that each segment in a display had the correct bits determining whether it was on or off to show the correct hex number. All this hardware was blasted onto the board and connections made in hardware. The NIOS II software was where C code was written to provide filtering of raw noisy accelerometer values and a BCD conversion for the hex display.
The target performance metrics included: ensuring that filter code allowed the movement of the accelerometer to provide high precision movement on screen with a good balance of sensitivity, and that there was minimal latency between physical motion of the board and on-screen motion of the character.

## 5.1 Accelerometer

The target performance metrics included: ensuring that filter code allowed the movement of the accelerometer to provide high precision movement on screen with a good balance of sensitivity, and that there was minimal latency between physical motion of the board and on-screen motion of the character.

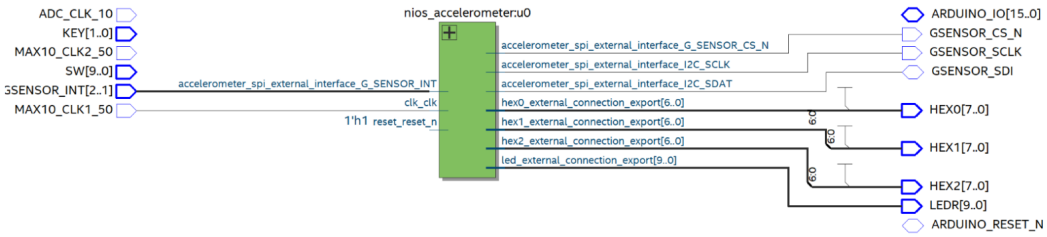


Figure 9: NIOS II Accelerometer Hardware Diagram

1. Unfiltered code: Extremely noisy and far too sensitive to small movements made to the board.
2. 49-Tap low pass FIR filter: 49 taps meant that filtering could be done to a higher resolution. However, filtering in floating point proved to be ineffective because NIOS does not support the multiplication of floats so would take many cycles emulating the operation through a series of fixed-point operations which introduced visible lag.
3. 49-Tap low pass FIR filter with fixed point: Multiplying the coefficients by 10,000 meant they were now in fixed point. This now solved the issue of latency introduced from the previous implementation. Initially it was decided to normalise the sum by 10,000 to only return 0, 1 or fffffff as shown in table 1. This meant a result was only noticed when the board was moved more than 45° This angle was chosen as it meant the movement was not too sensitive to small movements. This was necessary to test basic implementation, especially when integrating the board with the game. Whilst this worked, according to user testing feedback, it felt uncomfortable having to tilt the board at such a high angle. Therefore, it was decided to provide high-performance sensitivity movement by calculating at which angle it felt comfortable to move and also, not being too sensitive to small board motions.

| Direction | Output |
|---|---|
| Left ($\geq 45°$) | $x = 1, y = 0$ |
| Right ($\leq 45°$) | $x = fffffff, y = 0$ |
| Up ($\geq 45°$) | $x = 0, y = 1$ |
| Down ($\leq 45°$) | $x = 0, y = fffffff$ |

Figure 10: 2-axis motion of board normalised at 10,000

| Direction | Actual Angle measured | Output |
|---|---|---|
| Left | $20°$ to $90°$ | $x = 5$ to $e$, $y = 0$ |
| Right | $-20°$ to $-90°$ | $x = -a$ to $-2$, $y = 0$ |
| Up | $10°$ to $90°$ | $x = 0$, $y = 5$ to $e$ |
| Down | $-10°$ to $-90°$ | $x = 0$, $y = -a$ to $-2$ |

Figure 11: Preferred angle and its corresponding Output

4. Most Optimised FIR filter using coefficients from Matlab: To ensure that the accelerometer was able to have smooth readings being read to the terminal with low latency, there was compromise on certain elements of design for the FIR filter. To design a low-pass filter for an accelerometer, the cutoff frequency should be set lower than the frequency range of interest, whilst the passband frequency of the filter should be set as high as possible to allow the accelerometer to measure as much of the desired signal as possible. The stopband frequency should be set low enough to effectively remove high-frequency noise and interference. Taking all of these factors into consideration, the chosen passband frequency was: 425 Hz, with a stopband frequency of 450 Hz, passband ripple of 0.5 dB, and stopband attenuation of 65 dB. After using these parameters in a Matlab script the coefficients were implemented into the code for the accelerometer. Furthermore in the code, by increasing the size of the buffer in the memory allocation the accuracy of the averaging operation increases, however the performance is effected. The coefficients for this buffer were trialed and tested to find the optimal value. The optimal number of taps in the filter (49) was used to improve the frequency response and provide better attenuation in the stopband as this improves the sensitivity of the accelerometer by reducing the noise and interference it might pick up by the sensor. As increasing the number of taps also increases the computational complexity of the filter and may lead to longer processing times a careful balance was reached. Hence by continuously testing the filter with the FPGA ensured that the data that it sends can be perceived as smooth on the game with low latency and delay.

## 5.2 BCD in 7-seg Display

The 7-segment displays on the board were configured so that the final score received from the game could be displayed. Code was implemented in NIOS II software where the number received was passed through a BCD algorithm to ensure each digit could be displayed rather than a hexadecimal number. The process involved splitting the number into its ones, tens and hundreds. A modulus division by 10 was performed to get the digit and sent to a CASE implementation where the digit was mapped to its corresponding binary code for display (code in GitHub).
Testing was conducted through running the terminal with different numbers. Initially, each digit was tested to ensure the binary implementation was correct. Then the units, tens and hundreds were tested separately and then together to determine that the value will always be displayed accurately.