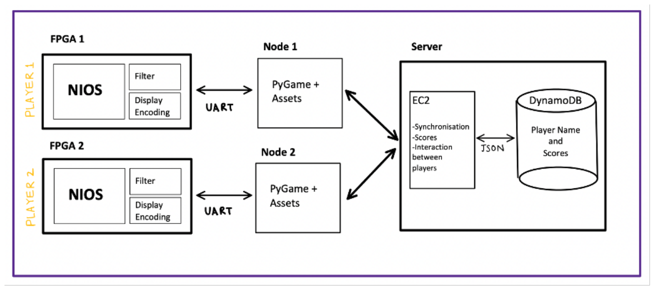
IP Group 10: Anish, Pengyuan, Nik, Andres, Hanbo, Kridhay

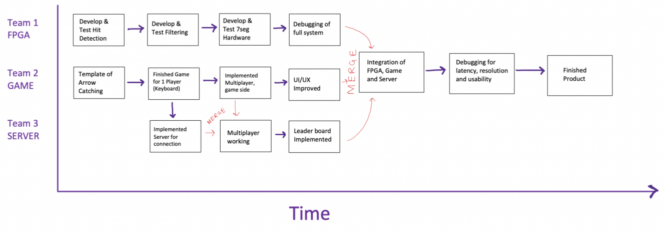
0.1 System Overview

The system is a game called Netflicks which uses a DE10 lite board FPGA, a computer and server. The game is a two-player rhythm game where arrows come down the screen and need to be caught by flicking the FPGA at the correct time. There are 3 levels, with the speed of arrows dictating the difficulty. There are 4 types of arrows corresponding to up, down, right and left FPGA flicks. There are 3 possible outcomes of an FPGA flick: perfect catch, good catch, and miss.

FPGA functionality requirements include hardware filtering, movement detection, and formatting the 7seg display. The game is multiplayer using an AWS server for communication, synchronizing game start, displaying the other player's score, and interactive power-ups. Score data is stored in a DynamoDB on the server to produce a top-10 leaderboard. Testing is done firstly in a modular fashion through unit tests, then the entire system is tested as a whole.

0.2 Overall System Architecture and Testing Flow Approach





Team 1 FPGA
Team 2 GAME
Team 3 SERVER
Develop & Test Hit Detection
Develop & Test Filtering
Develop & Test 7seg Hardware
Debugging of full system
Template of Arrow Catching
Finished Game for 1 Player (Keyboard)
Implemented Multiplayer, game side
UI/UX Improved
Implemented Server for connection
MERGE
Multiplayer working
Leader board Implemented
MERGE
Integration of FPGA, Game and Server
Debugging for latency, resolution and usability
Finished Product
Time

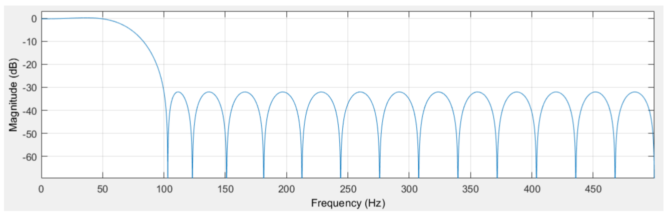
Figure 1: Flowchart of Overall System Architecture (left) Testing Flow Approach (right)

1 FPGA

The main requirements for the FPGA's functionality were: hardware filtering (DSP), movement detection (processing input), formatting 7seg display (output), and 2-way communication with the game. The design approach was to implement the most processing possible in embedded hardware and dedicate the NIOS II processor to 2-way communication with the game.

Hardware filtering involved writing an FIR low-pass filter in Verilog to remove high-frequency noise from the accelerometer readings. Design began by entering the specifications of the filter into MATLAB to generate a sequence of filter coefficients. The magnitude response had a satisfactory –30dB stopband gain.

Fsampling	Fpass	Fstop	Order (taps)
1000	50	100	31



It was challenging to regulate the sampling rate of the filter, but it was achieved by creating a clock divider module which converted the 50MHz system clock to 1kHz. A 32x32bit synchronous write RAM was employed to store the acceleration samples and a 32x32bit ROM was used to store the coefficients. The multiply and accumulate stages were implemented in combinational logic. Taking the filtering load off the processor greatly increased the rate at which communication with the computer could be performed, crucially reducing input lag for a game based on reaction time.

Movement detection was implemented by limiting input to 2 axes, Y and Z. Positive acceleration and negative acceleration in the axes are mapped to one of the following outputs: up, down, left, and right. If acceleration exceeds a threshold, an action is detected and sent to the game. One issue encountered through testing at this stage of development was the calibration of input sensitivity. Another issue was the effect of gravity, where its components caused non-zero readings on each axis, even at standstill. This offset was at a maximum of 255 (when all force is acting on one axis), conveying that 1g of force was equivalent to an accelerometer reading of 255. This was observed in the vertical Y-axis.

The solution to these issues was implemented with a mix of hardware and software. Firstly, the built-in Y_OFFSET register of the accelerometer module was adjusted by -1g, meaning that this axis now read 0 at standstill. Secondly, a base threshold for action detection was set at 255, and the threshold was empirically tuned to account for the ergonomics of the wrist.

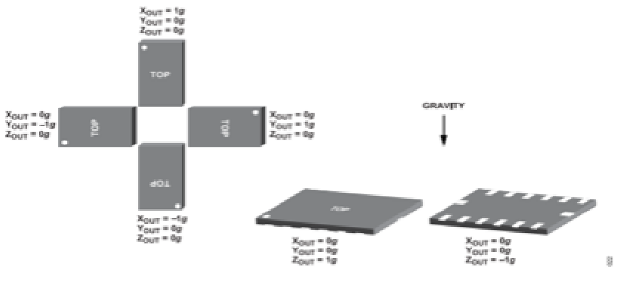


Figure 58. Output Response vs. Orientation to Gravity

Direction	Action Threshold multiplier
U (y-axis +)	1.5
D (y-axis -)	1.4(-1g offset for grav.)
L (z-axis +)	1.4
R (z-axis -)	1.5

Processing and encoding data from binary to 7seg was achieved mostly through hardware. 3 sub-modules were employed in doing so: char_to_7seg.v, hex_to_7seg.v and bin2bcd.v. char_to_7seg formatted the first 2 digits, allowing letter combinations such as Sc (score) and P1 (player 1) to be displayed according to inputs taken from the game. Hex_to7seg recycled the module from lab2 to purely display numbers. However, this was chained to the binary to BCD converter, allowing a 14-bit number to encoded in BCD with the double-dabble algorithm. Combining these produced the following display:

The final verilog top-level module nicely summarises how the embedded hardware was combined with the processor:

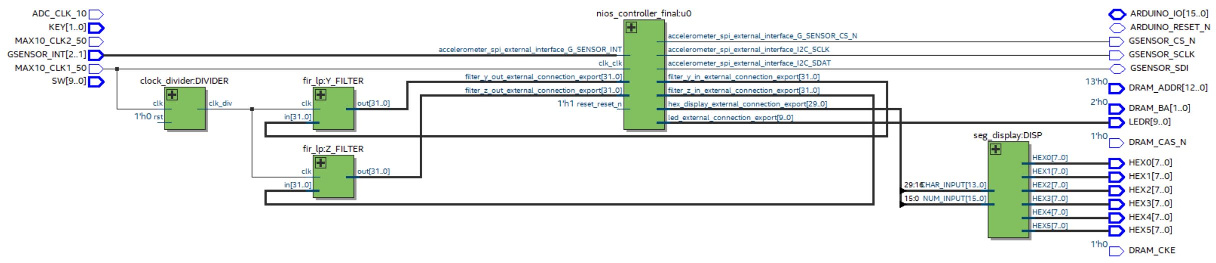


Figure 2: Top Level Module

During integration, there was an issue with the transmission rate from the board to the computer. The processor was outputting characters at an uncontrolled rate, which was much faster than the rate of the reception of the more complicated python code. This meant that a buffer of characters built up and there was an input delay. This issue was resolved by using a NIOS interval timer to toggle an output enable flag every 10ms. Whenever the flag was high, an if statement in the for loop executed output, effectively regulating the data output rate.

2 Communication Between Controller and Game

2.1 Brief explanation of Architecture

FPGA-Game communication was achieved by the UART using the JTAG interface provided on the board. There were two soft components – the C code compiled onto the FPGA, and the Python code that bridges the FPGA to the main game. One challenge was closing the nios-2 terminal using the 0x04 character: It yielded an unacceptable latency, communication was far too slow to be able to query the controller multiple times a second. This was an important factor in the development of the game.

DESIGN DECISION: Nios-2 terminal was kept open, the CtrlD signal was not sent. Instead, we used subprocess.Popen() and read/write from the stdin/stdout pipes respectively, taking care to flush them when required.

CORRESPONDING TRADEOFF: Handling the nios2-terminal was more complicated. If the program terminated without killing the nios2terminal properly, a "Broken Pipe" error would be produced the next time it was run. On linux this was fairly easily fixed by sending a kill system directly through the operating system:

```
import os
os.killpg(os.getpgid(self._nios.pid), signal.SIGTERM)
```

But on windows it took a bit more work. Steven found a workaround for this where Windows handled it appropriately. The tradeoff here was a large amount of time spent working around the bug. The Python code used multithreading to accomplish reading user inputs, sending chars to fpga, reading fpga chars and updating score simultaneously. It was written in an object-oriented approach, which has two advantages.

- **Abstraction**. The person in charge of the game code had absolutely minimal work, there were only 4 lines that needed to be added to the game and they did not require understanding of the underlying code:

  ```
  import connection
  Fpga= FPGA()
  Fpga.start_communication()  # starts all threads
  Fpga.read()  # gets the current action reading: U, D, L, R or 0
  Fpga.update_score()  # this changes the score currently being sent to the FPGA.
  ```

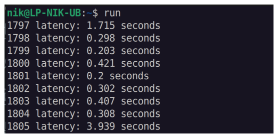
- textbfScalability. One of the ideas brainstormed was having multiple controllers per person, one for the left hand and one for the right hand. Going ahead with this, it would have been be as easy as fpga1 = FPGA() and fpga2 = FPGA().

On the FPGA side, the line

```
prompt = getc(fp)
```

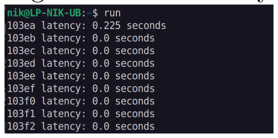
Presented a challenge, as the program waited if there was no character queued in the nios2-terminal. 3 Solutions to this were considered: 1. Multithreading the C code on sthe FPGA, 2. Implementing a timeout for the file pointer, 3. Constantly sending something in the nios2-terminal, even if it is not useful.

DECISION: Option 3 was picked because option 1 was deemed overkill, and option 2 would be wasting valuable processor time that could be allocated to something else, and would be a bad tradeoff. The python script is therefore constantly sending 5 characters in an infinite loop : 'S' indicating the start of the score, and 4 characters following it (the score).



```
nik@LP-NIK-UB:~$ run
1797 latency: 1.715 seconds
1798 latency: 0.298 seconds
1799 latency: 0.203 seconds
1800 latency: 0.421 seconds
1801 latency: 0.2 seconds
1802 latency: 0.302 seconds
1803 latency: 0.407 seconds
1804 latency: 0.308 seconds
1805 latency: 3.939 seconds
```

Figure 3: TCP latency



```
nik@LP-NIK-UB:~$ run
103ea latency: 0.225 seconds
103eb latency: 0.0 seconds
103ec latency: 0.0 seconds
103ed latency: 0.0 seconds
103ee latency: 0.0 seconds
103ef latency: 0.0 seconds
103f0 latency: 0.0 seconds
103f1 latency: 0.0 seconds
103f2 latency: 0.0 seconds
```

Figure 4: UDP latency

2.2 Consideration of ESP32

ESP32 was considered to communicate wirelessly between the FPGA and the host computer. This would have allowed less restrictive movement, and allowed multiple controllers per player (I.e. one per hand). Testing of both TCP and UDP sockets was employed, and the latency recorded: As evidenced by the figures, if wireless communication were to be used then the UDP protocol would be far more suitable – occasional packet loss is okay but waiting 3.9 seconds for receipt of packet confirmation (as occasionally happened with TCP) is definitely not.

DESIGN DECISION: in the end the decision made to not use ESP32, due to restrictions of blasting to non-volatile memory – something that is necessary to be able to remove the cable. It was decided to strive for perfect integration first, a decision that was proved to be prudent given the complications that followed.

2.3 Tradeoff between responsiveness of action detection and score

Checking the contents of the UART port in every iteration of the FPGA code was found to have a significant latency on the action detection – a delay of around 5 seconds before the actions showed up. This is because of the way the UART buffer works – characters are queued there until read, at which point they are purged.

DESIGN DECISION: Looping the accelerometer code in a for loop before running the communication code was the way forward. This way, the ratio of how much processor time is given to accelerometer vs how much time is spent checking the UART buffer can be controlled.

Trying 10000 iterations in the for loop resulted in flawless action detection, but a large delay in score updates on the 7Seg display. Decreasing iterations to 1000 cycles gave good score updates, but still some delay in action detection. At 5000 cycles, score updates were still instant but with no noticeable delay in action detection.

In retrospect, it would have made sense to lean closer to action detection, since small delays in score updates on the board would not really affect the user experience.

3 Game Mechanics and Design

3.1 Scoring System and Gameplay

The game has a scoring system that rewards players for hitting arrows accurately. Hits are classified into three states: Perfect, Good, and Miss. A hit within 15 pixels of the centre of the net is Perfect, while a Miss is recorded if the arrow passes the net. Both Good and Perfect hits increase the player's combo by 1, which is multiplied by 1 or 2 depending on the hit's state. The player's score is then updated accordingly.

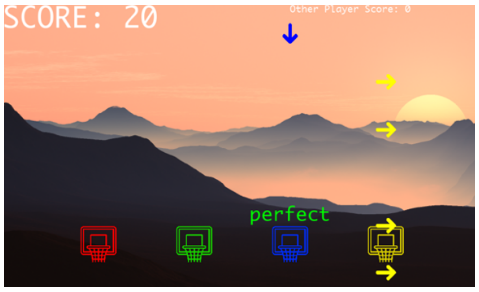


Figure 5: Perfect arrow hit

To add a twist to the scoring system, power-ups were added. Power-ups penalize opponents for 3 consecutive perfects. A message sent to the server sends a minus message to reduce the opponent's score by 10. A penalty message was attempted but couldn't be added due to the game's while loop. Ultimately, the message was not implemented.

3.2 User Inputs

Besides FPGA flicks, the game requires additional user inputs: player name, ready, selecting the level and navigating to the leader board. Testing the use of the FPGA for these inputs revealed that the feature did not enhance the user experience. The group decided that hovering over letters to type out the player's name would be too time taking.

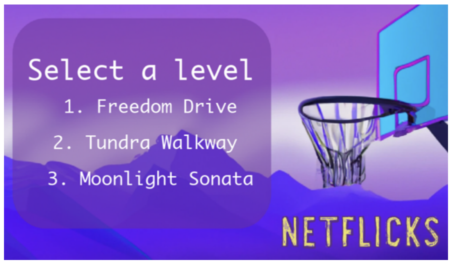


Figure 6: "Non game" input image

The other inputs could have been implemented with the switches on the FPGA. However, it was found that because a filter with many taps was implemented to ensure a high sampling frequency and low delay, the FIR filters took up too much of the logical block size available on the FPGA. Hence, there was not enough space to implement switches. Low-delay FPGA flicks were prioritised over using the switches for the "non-game" screens.

3.3 Levels

A function was created to randomly input 50 arrow directions into a text file, creating random levels for each player to prevent pattern memorization. The difficulty of the levels were based on the speed of falling arrows instead of the order of arrow patterns, which was used in an earlier iteration. The speed settings had to be adjusted after testing with the FPGA revealed that its flicks were slower than keyboard taps. Upon further testing, a design flaw was discovered in which the randomly generated text file at Player 1's node was different from Player 2's. This was fixed by having only Player 1 generate the text file and sending it as a string to Player 2. An "infinite game" was considered, where the game continues until a miss occurs, but this was abandoned due to latency issues.

3.4 Icons, Background Images, Text, and Music

All icons, images, font package and music were obtained from free sources and credited in the python code. During UI development, there were instances where the positions of text, icons and images were "hard coded" with respect to the screen size. This was not a problem on the two devices used for the development, but when the code was tested on a third device with different resolution settings, there was a mismatch. To fix this, some of the code was redone to ensure all positions were set with respect to the resolution settings by implementing position based on the fraction of the screen height and width.

4 Backend and Network

4.1 While loops and Flags

While loops and flags played a crucial role in controlling the game flow and ensuring correct behaviour in this rhythm game. In the game, different screens are shown depending on the state, and while loops update and render the game screen in real-time. Flags indicate events like game start, in-game, game over, and leaderboard display. Initially, the game used a single while loop, but testing revealed issues with stage separation. Thus, flags were introduced, and variable initialisation was moved outside the loops to ensure proper functioning.

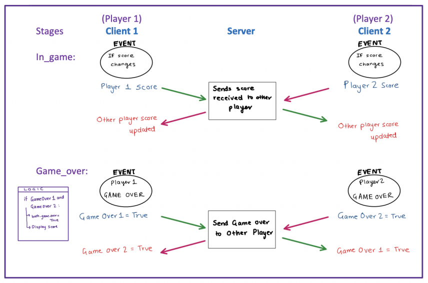


Figure 7: Network communication

4.2 Server Communications

In the communication aspect of the game, messages were sent by the client in the while loops to request actions such as is_ready and get_score. These messages were received by the server, which then sent a confirmation or follow-up message to the clients. Initially, message receiving was done in the game loop without a separate thread, causing the code to get stuck at the line until the next message was received. To solve this issue, a separate thread calling a receive function was implemented for message receiving. On the client side, the received message action was stored in global flags so that it can be received in the game thread.

To test the robustness of the server-to-client communication, stress testing was carried out. Lots of data packets were sent in a short period of time to the server. It was revealed that certain data packets were corrupted because previous packets were being merged with current ones. To fix this, a delay of 0.2 seconds was added after every data transmission event. After doing some more testing to find the lowest delay time possible which would produce no packet corruption, the ideal delay time was found. Unit testing was also carried out to test the individual functions of the entire game and host backend code, which includes communications and game logic.

4.3 Database

A database was implemented to store player scores and construct a leaderboard of the top ten scores. Initially, a Postgres Database on AWS RDS was used but as the data structure often had to be changed, DynamoDB was switched to. Although less rigid than SQL, DynamoDB allowed for fast iterations and experimentation with data structure. The partition key was set as the level of the game and the scores were assigned as the sort key to allow for the grouping of scores based on level and querying of the top ten scores for each leaderboard. The latency with using DynamoDB was low and querying was efficient. Functions related to DynamoDB, such as actions to fetch or update the leaderboard, were modularly tested as well as tested as a whole.