Information Processing Coursework

Charmaine Louie, Alexandra Neagu, James Ong, Michal Palič, Václav Pavlíček, Matthew Setiawan

Purpose of the system

Medusa I/O aims to modernise the iconic 1980s game snake, expanding its functionality whilst preserving the signature game mechanics. New adapted features include a multiplayer mode, and power-ups such as increased speed or teleportation. Additionally, the Medusa I/O system features DE-10 Lite FPGA boards as a human interfaces, using the accelerometer as an input, and LEDs and 7-segment displays as some of the outputs.

Overall system architecture

Figure 1 shows the overall system architecture which consists of FPGA nodes, PC clients, a local game server, and an EC2 cloud server with access to a DynamoDB database.

At the core of the system are the individual PC clients. These render the game board and communicate with the local game server so that all the clients' game states are synchronized. They also communicate with their respective FPGA node to read the user inputs from the accelerometer and control the motion of the snake on the rendered game board.

The FPGA node communicates with its respective client PC over USB. The state of the development board buttons and switches can be read by the PC client by issuing a request to the FPGA. Similarly, outputs such as the LEDs or 7-segment display array can be written to via a request and are used by the game to show the user messages and effects related to the condition of their snake.

The local game server is hosted on a local area network. Here, a process is dedicated to each of the connected clients. The game server aggregates the changes in the game state from each of the client PCs and then communicates the updated game state back to them via UDP. The processes communicate through file operations, which allow them to keep a shared game state.

The client PC communicates with the cloud server hosting the database whenever the game ends, submitting their final score, which is then added to the leaderboards maintained in the database. These leaderboards are then processed, and the top entries are communicated back to the client PCs, where they are used to create an end-game screen that shows how well the player performed in this game compared to the historical highscores. During both data transmissions, the Json structured information was encoded to provide a minimum level of security. This information also propagates to the FPGA node where it is prominently displayed to the user.

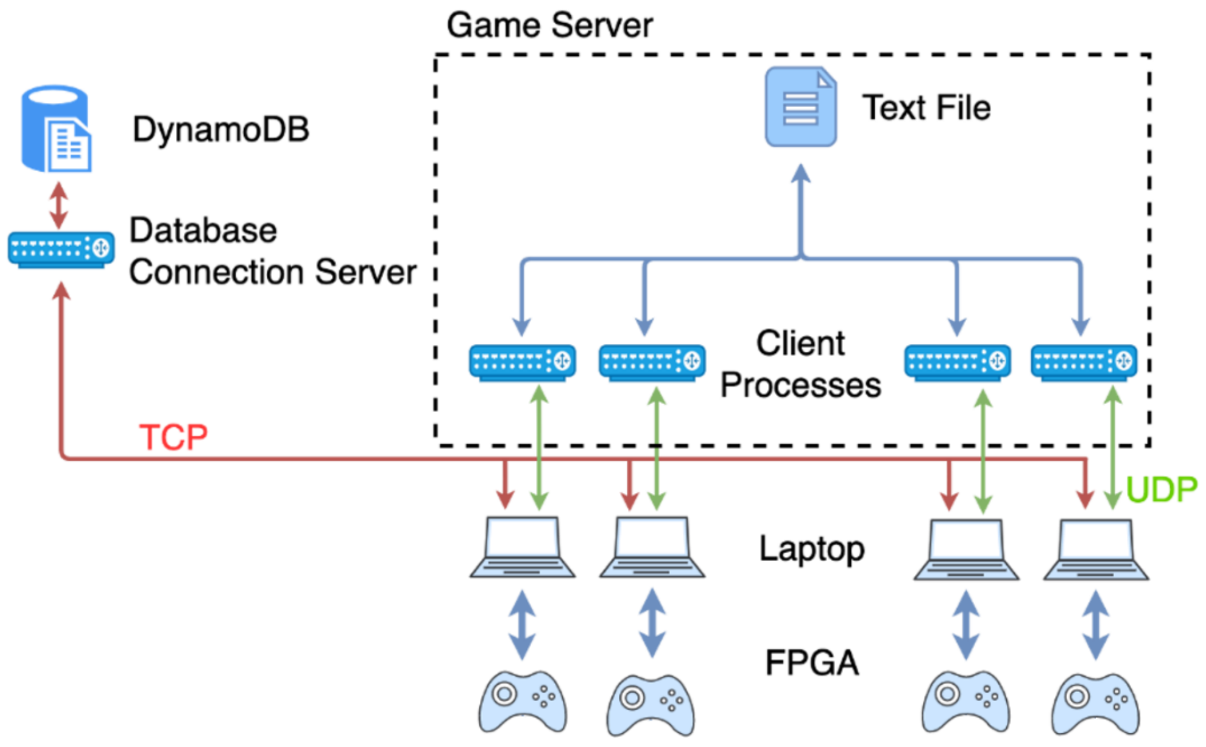


Figure 1: Overall system architecture

Software architecture

Medusa I/O software was primarily developed in Python, with the GUI being built using the Tkinter library. Tkinter was chosen due to its simplicity and good cross platform support, which was important as a large part of development took place on MacOS and Windows. The appendix contains a few screenshots of the game screens including the Login, Map, End and Leaderboards screens. Sound effects were also a big part of the user experience, hence the game included sound effects for player-to-powerups and player-to-player interactions. These were implemented using multiprocessing and PyDub libraries for cross platform compatibility and good performance.

Different network transport protocols were selected for multiple roles within the project. The client to server communication for multiplayer games is traditionally implemented using UDP. We also favoured the low latency of UDP over the high transfer reliability of TCP as retransmitted packets were often irrelevant to the game by the time they arrived, due to the high synchronisation tick rate. And UDP retransmission was time efficient enough to tolerate the dropped packets. On the other hand, TCP was used to communicate the player’s score to the cloud database server and retrieve the player’s position in the global leader board. Here TCP was chosen since reliability is the most important characteristic of the database communication.

The mode of connection between the game server and client was iterated upon, until a suitably reliable and performant design was arrived at. Figure 2 shows the initial design which relied on code execution in a single thread. It received messages from clients in series, one after another, which caused problems. If one user crashed, the server would be blocked and the game frozen, until the crashed user timed out. In response to this, the design seen in Figure 3 was developed, where client communication was moved to separate processes. In this design, each client communicates with a separate process, which allowed the client communication to take place independently thus making it resistant to unreliable or dropped connections. Compared to a multithreaded approach, this also had the advantage of bypassing the python global interpreter lock and allowing multiple CPU cores to be utilised. This improved performance that can be translated into a higher possible tick rate for the server.

Splitting up the game server software into multiple processes also posed some new challenges. A method for sharing the game state between the processes had to be devised. Due to its simplicity and reliability, file handling was chosen for this purpose. The state of each client’s snake was maintained in a file on the game server and was updated as the client’s snake moved. When the shared state of the game board was to be sent to a client by its respective communication process, all the client files on the game server were read and the positions of the other players’ snakes were determined. One pitfall of this approach was that a text file could only be opened by one process at a time. While those collisions were rare, they did occur and were simply resolved by retrying the file operation until it is completed successfully. Due to the speed of these file operations, the introduced delay was still minimal.

To store the leaderboards for the game, a cloud database system was utilized. Data was defined in a format consisting of the player username as a partition key, a sort key and other attributes including the player's total and highest scores. The EC2 cloud server implements several functions primarily focused on extracting data from the database and returning the processed entries to clients, where a history of the game records is displayed.

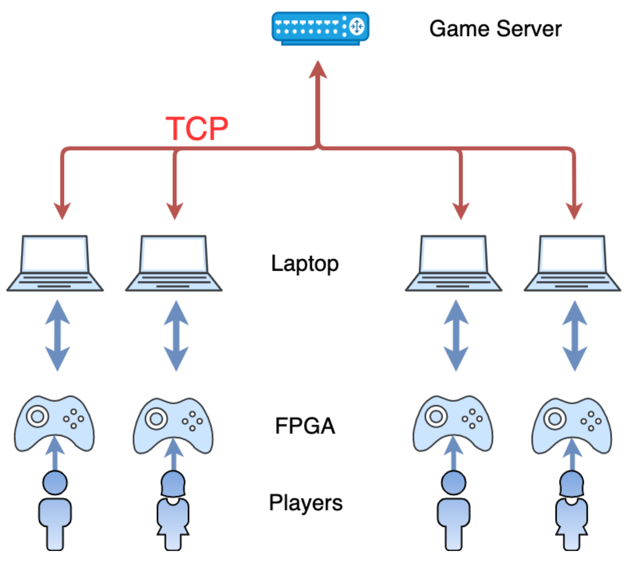


Figure 2: Initial game server design

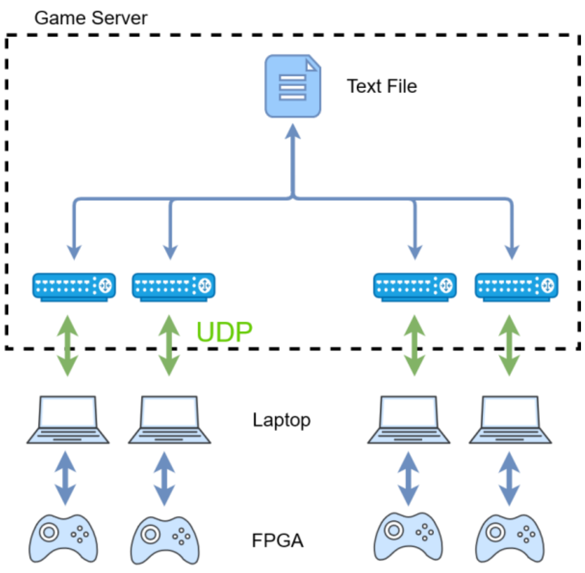


Figure 3: Final game server design

Software testing

Figure 4 shows the development and testing process of the whole system. The diagram presents the parallel development of the system's sections, as well as key milestones such as the integration of different components. Testing of each software part is described in the following sections.

Game

The game was first tested in the single-player mode. When sufficient performance was achieved, multiplayer mode testing was conducted on a single machine for a no-latency simulation. The next step involved multiplayer testing across two different machines. These steps demonstrated that testing the game using a large-scale WLAN, as the College Wi-Fi, was not reliable enough and therefore we conducted further testing on a wired LAN with a dedicated router.

GUI components

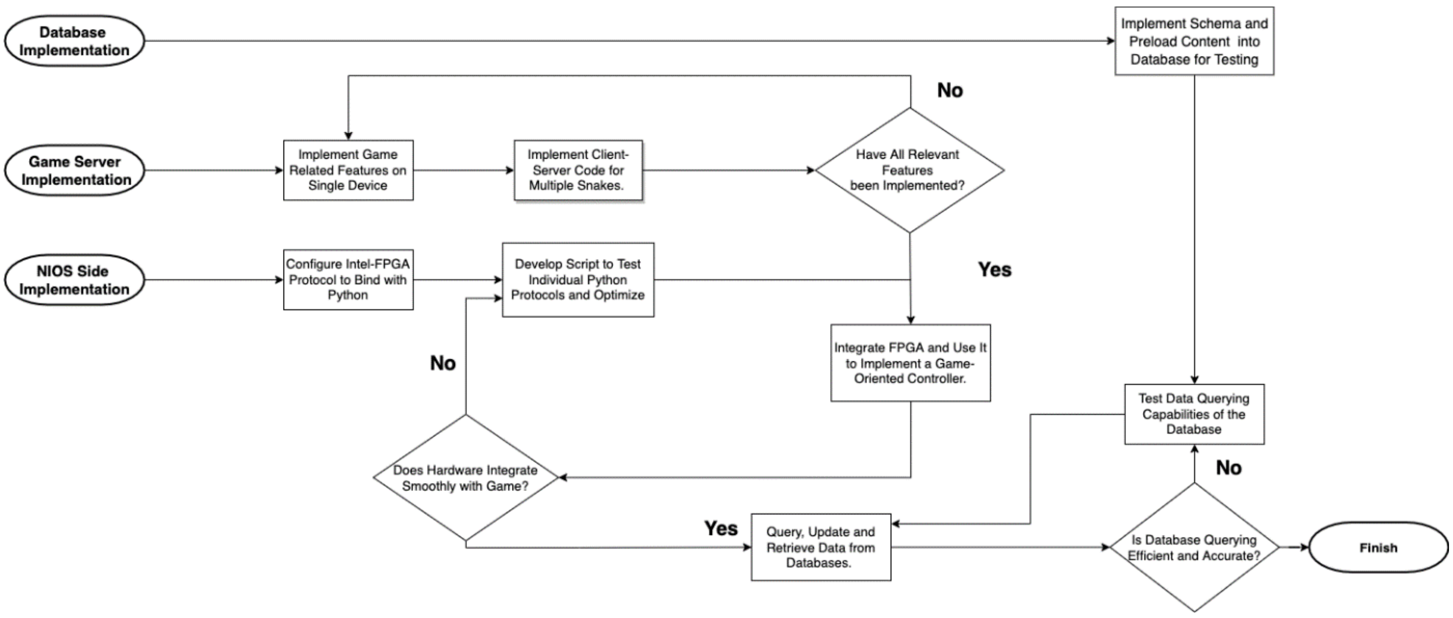
The game was built in a hierarchical manner, being split into different UI components to allow parallel development of each part of the game. These components were developed as separate Python classes that built their UI onto the Tkinter Frame received in the constructor parameter. Each component was initially tested using a separate script that provided an empty Tkinter Frame, data, and function reference to be used by the component. All GUI components were integrated after being tested separately.

Database

To effectively implement the database, a schema was defined, and individual scripts were written to update, insert data entries, and return the sorted data. For a complete testing scheme, dummy data was sent over the TCP connection in a predefined format to test whether data from DynamoDB could be retrieved and updated. A total of 100 entries were included in the database to determine whether entries could be sent efficiently and accurately to the client. After observing that the database could be updated through query methods with dummy inputs, it was successfully tested with inputs received from the game.

The whole system with hardware

Once all components of the system were tested separately, the integration testing was conducted. This testing was performed to make sure that components can interact with each other desirably. The key metric during this testing was the playability of the game meaning that the game does not crash, and the user can control the player as expected. This final testing showed only a small mismatch between components that were resolved, and it proved the functionality of the whole system.



Database Implementation
Implement Schema and Preload Content into Database for Testing
No
Game Server Implementation
Implement Game Related Features on Single Device
Implement Client-Server Code for Multiple Snakes.
Have All Relevant Features been Implemented?
NIOS Side Implementation
Configure Intel-FPGA Protocol to Bind with Python
Develop Script to Test Individual Python Protocols and Optimize
Yes
Integrate FPGA and Use It to Implement a Game-Oriented Controller.
No
Test Data Querying Capabilities of the Database
No
Does Hardware Integrate Smoothly with Game?
Yes
Query, Update and Retrieve Data from Databases.
Is Database Querying Efficient and Accurate?
Finish

Figure 4: System development and testing flowchart

The Hardware

Protocol Design

To better parallelize the development of the system, as a first step, a protocol for communication between the host and the FPGA node was drafted. The goal of the protocol was to provide access to all the human interfaces of the DE-10 Lite development board using a convenient set of commands. The intent was to develop a more general interface that could be useful for most game designs utilizing the FPGA board as a controller. The commands supported by the final hardware are summarized in Table below.

Command	**Description**
R ALL	Returns the values for all the inputs.
R ACCRAW	Returns a formatted output with the raw accelerometer values for all the axes.
R ACCPROC	Returns a formatted output with the filtered accelerometer values for all the axes.
R ACCQUAL <number>	Sets the number of filtering coefficients at runtime. Range 1-64 allowed for hardware filtering.
R BUTTON	Returns the state of the two push buttons.
R SWITCH	Returns the state of the 10 switches.
R LEDFLASH <PATTERN>	Flashes LEDs without affecting other functions. ("11100" means ON for 300ms, OFF for 200ms)
R LEDWRITE <VALUE>	Displays the VALUE as a binary number of the LEDs, restored after R LEDFLASH
R HEXTEXT <TEXT>	Displays text on the 7-segment displays, scrolls if longer than 6 characters.
R DEBUG <1 or 0>	Collects timing information on interrupt routines and command latency.

Table 1: Hardware Commands

Software and Hardware Design

A solution was required to listen for commands from the host without blocking the other periodic features of the device, most importantly the sampling of the accelerometer. This was resolved by processing the input within the main loop of the program and delegating all periodic tasks to several interrupt service routines triggered by hardware timers.

The filtering hardware consisted of two shift registers, one for the sampled data from the accelerometer and the second for the filtering coefficients. The NIOS processor interfaced to the hardware using Parallel Input Output registers at the data input, control input and output of the hardware module. The shift registers can be advanced independently to allow the loading of filtering coefficients or sampled data. Figure 5 illustrates the structure of the hardware design.

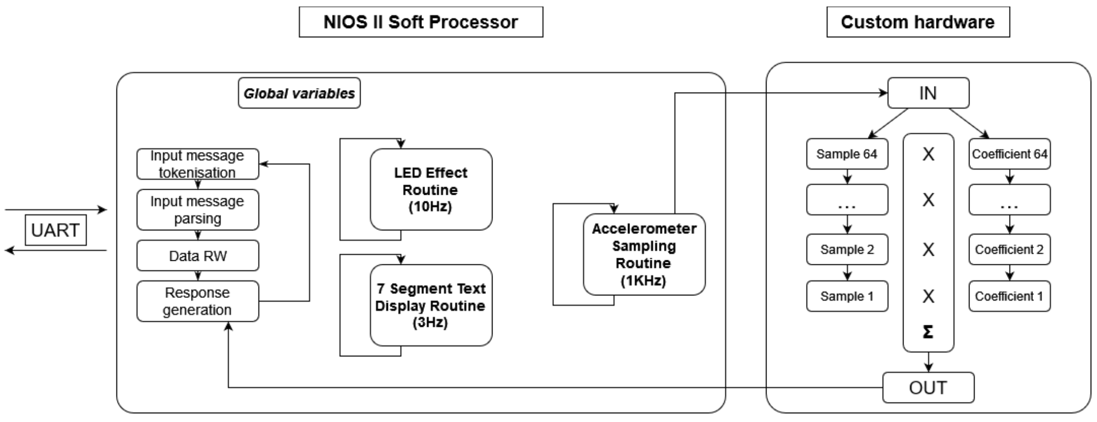


Figure 5: Code & Hardware structure

Performance

The key performance metric that was tracked and guided the development of the hardware was the latency of the R ACCPROC command. The latency for this command was critical because it was called by the host computer during every game tick to determine user input. High latency would result in a delay when reading input into the game. In this arcade game, the timing of the user inputs is critical to accurate controls of the snake, especially since some power-ups can dramatically increase the tick rate of the game. As such, the latency of the input must be minimized. The goal was for the input delay to be no more than one frame time at the computer. For a refresh rate of 60 Hz, this amounts to 15 ms, which was chosen as a target.

The processing for the accelerometer input was developed in three iterations. In the first, double precision arithmetic emulated by the NIOS CPU was used. In the second, fixed-point arithmetic using the alt_32 integer type was used. In the third iteration, this fixed-point algorithm was translated into hardware using System Verilog.

In Figure 6, the latency of the developed processing techniques can be seen as a function of the number of filtering coefficients. The latency figures include the delay due to the non-causal nature of the used filter, with the theoretical minimum plotted as the

Finite response series. While the latency of software-based methods increased linearly relative to the finite response limit, the hardware filter only incurred a small constant delay up to 64 coefficients. Its performance was close to the theoretical limits of the filter.

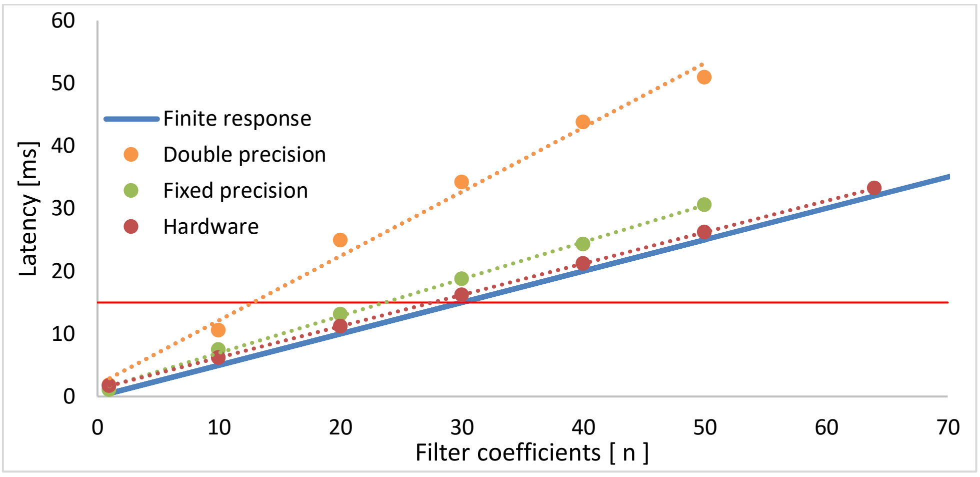


Figure 6: Performance comparison of filtering techniques

Testing

Hardware testing was a priority from the start. The hardware was designed and tested within the Icarus Verilog simulator. Random testing was used over 10000 sets of inputs and coefficients to ensure that bugs are not present. The reference value was calculated independently of the hardware within the testbench using double-precision arithmetic. Due to rounding, a tolerance of +-2 (on the output of -256 to 255) was allowed, though this was rarely necessary.

The communication between the NIOS and Python was tested by developing a Python testbench that sent a random sample of all possible commands to the FPGA. Outputs from commands along with the error codes returned by the NIOS system were then verified to ensure that no unexpected state was reached.

Table 2 shows the resource utilization for the final hardware design. Due to resource constraints, Hardware filtering was only applied on the X and Y axes of the accelerometer as these were the only ones relevant for the game input. This high resource utilization is the downside of the simple design that was developed, where the convolution of the filter coefficients and data was executed in one step. In total 130 32-bit multipliers were required. The resource utilization from the NIOS II/e alone was around 6% of the logic elements. Hardware filtering in one axis was sufficient to fully utilize all the embedded 9-bit multipliers and raise the Logic element utilization to 17%. The sharp increase to 95% logic element utilization when filtering for the second axis is a consequence of multipliers having to be constructed out of logic elements, a resource-intensive prospect. Using a smaller number of multipliers and computing the convolution over several clock cycles would allow for much fewer resources to be used while negligibly impacting performance. This would make a good future improvement to the system.

Resource	Number used	Utilization percentage
Logic Elements	47,196	95
Memory bits	11,264	<1
Embedded 9-bit multipliers	288	100
Phase Locked Loops	1	25

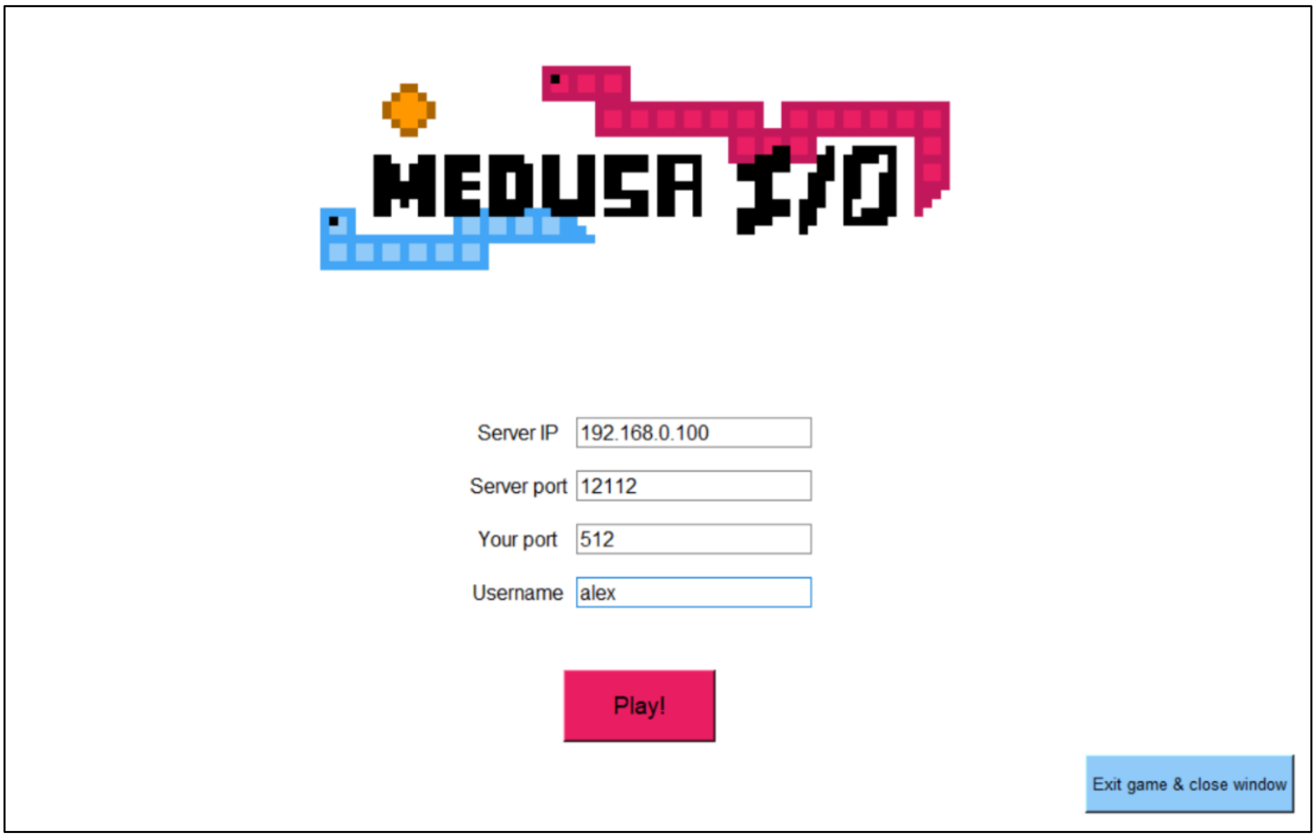
Table 2: Final DE-10 Lite resource utilization

Conclusion

The Medusa I/O system had a fairly large number of components. Robust network programming using UDP, allowed for low latency communication between clients and kept the game synchronized. A DynamoDB cloud database and its processing allowed the users to reliably store game information in the longer term. Timer driven NIOS code allowed periodic tasks to run even on a loaded processor. System Verilog was used to develop a fast hardware filter to process accelerometer data, ensuring responsive user input and adding as little as 1.3ms to the latency. Ultimately all these elements came together in this functional, performant and reliable system.

Appendix

Appendix 1: Initial connection screen



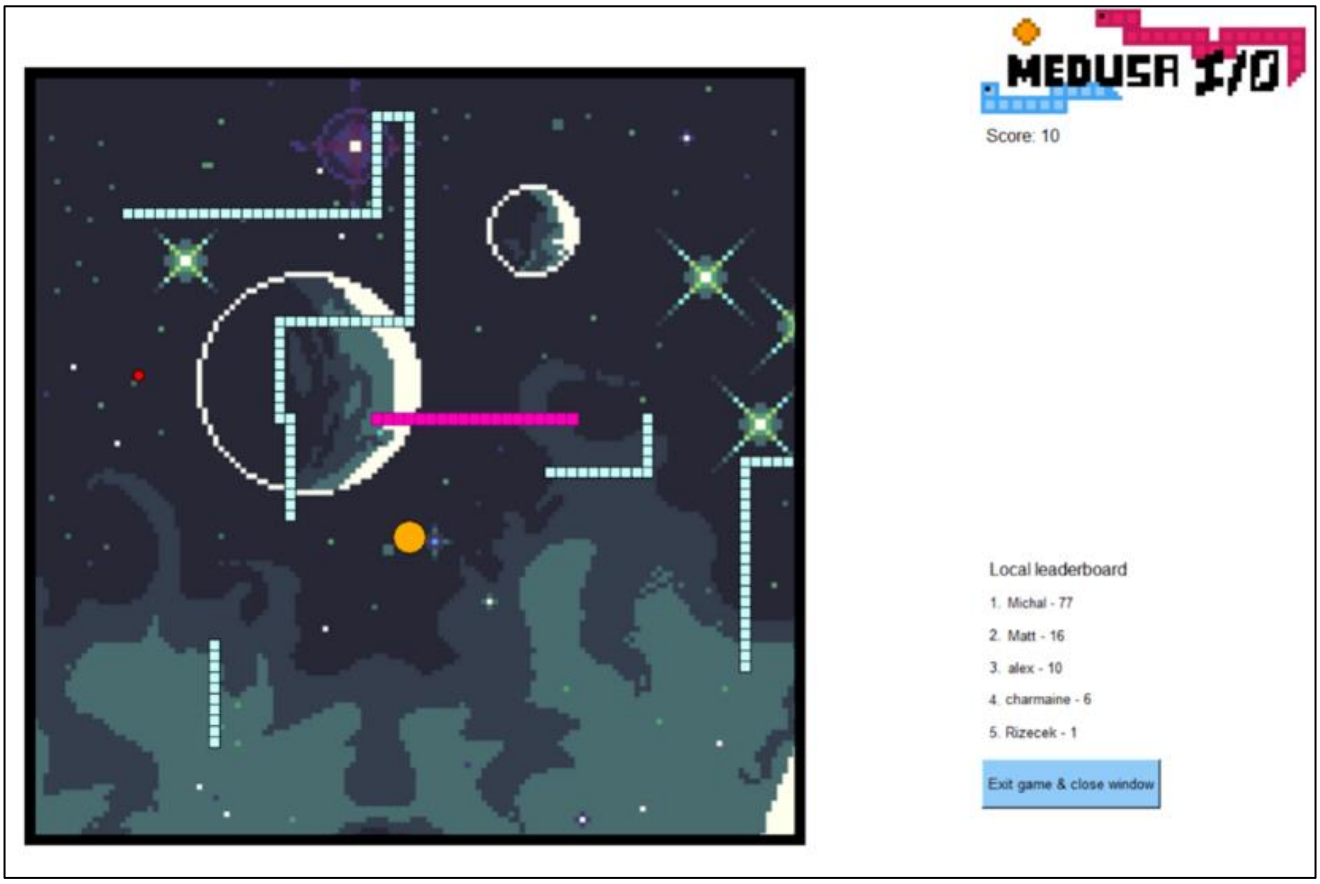
Server IP 192.168.0.100

Server port 12112

Your port 512

Username alex

Appendix 2: Main game screen

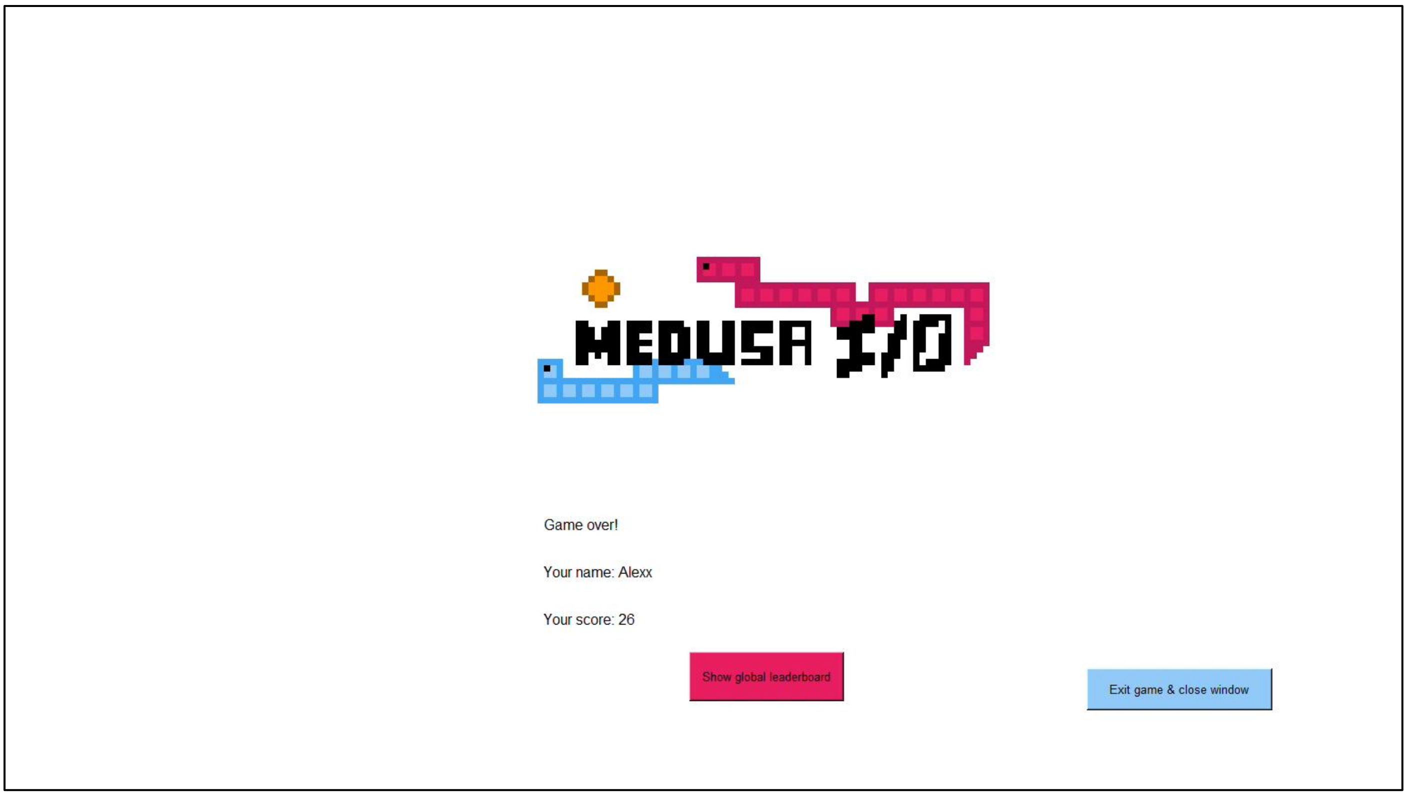


Score: 10

Local leaderboard

1. Michal - 77
2. Matt - 16
3. alex - 10
4. charmaine - 6
5. Rizecek - 1

Appendix 3: Game over screen

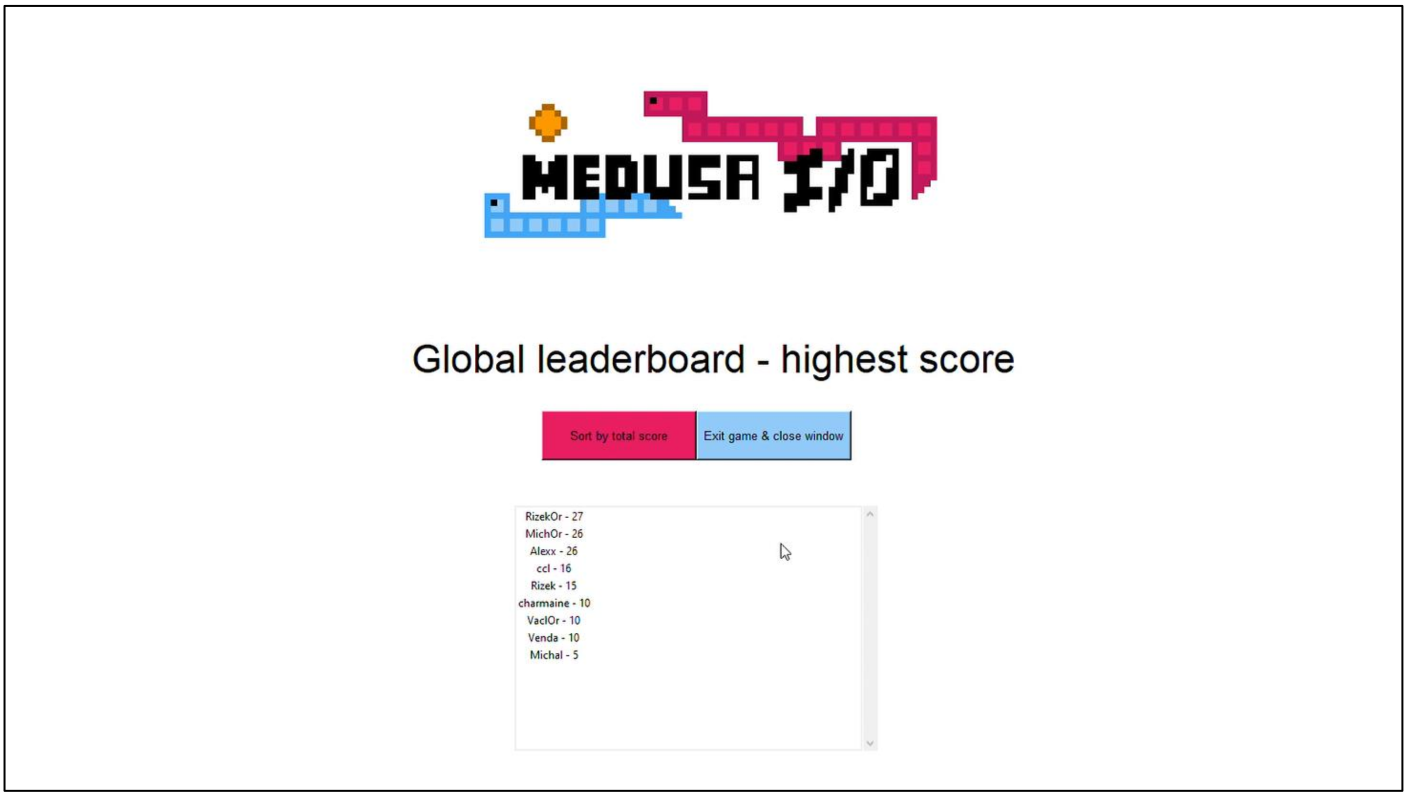


Game over!

Your name: Alexx

Your score: 26

Appendix 4: Global leaderboard screen



Global leaderboard - highest score

```
RizekOr - 27
MichOr - 26
Alexx - 26
ccl - 16
Rizek - 15
charmaine - 10
VaclOr - 10
Venda - 10
Michal - 5
```

Appendix 5: Sample use case