DECODER-IO DOCUMENTATION

Overview:

Our IoT system implements a controlled gaming platform which allows multiplayer games to be selected and played within a hub. An "admin" FPGA device is assigned at the beginning and has selection control in the menu to which game is played or the real-time leader board. Additional FPGA devices can act as users and join the platform, waiting until a game has been chosen. We have incorporated wireless communication between each player's controller through the use of Arduinos to enhance the portability of our gaming platform.

Two different multiplayer games are employed on the platform as examples. The two games relate to guessing a code which is an unknown combination of movements (left, right, forwards, backwards) and switch flicks. The first game is inspired from the currently popular game "Wordle" by the Times. The second game tests the users speed and accuracy in a race to input the correct code. Both games aim to engage the users' problem-solving skills and agility, using tailored feedback corresponding to a player's input.

Game 1: MasterMind

- In this game, the database contains a code (e.g. left, switch1, backwards, switch3, right, forward) which is unknown to all the players.
- The players take turns, one by one, to guess the code. Local processing on the FPGA will allow a display message on the 7-segment display to let each player know when it is their turn.
- To guess the code, a player will perform a series of movements and switch combinations and will then push the send button to signal that this is the guess they would like to put forward.
- A reset button is incorporated to allow the user to reset their guess before they send.
- This player's guess is assessed by the database and all the players receive the feedback from the database on the validity of this guess through the user interface. This displays the player's guess, and whether the inputs were:
    - In the right place (Green)
    - In the code but guessed in the wrong place (Yellow)
    - Not in the code (Red)
- All the players can use this feedback to adjust their guesses accordingly.
- The players continue to play one-by-one until a player guesses the code correctly.
- Once a winner has been declared, the "admin" user is reassigned and can navigate through the main menu to another game or the leaderboard.

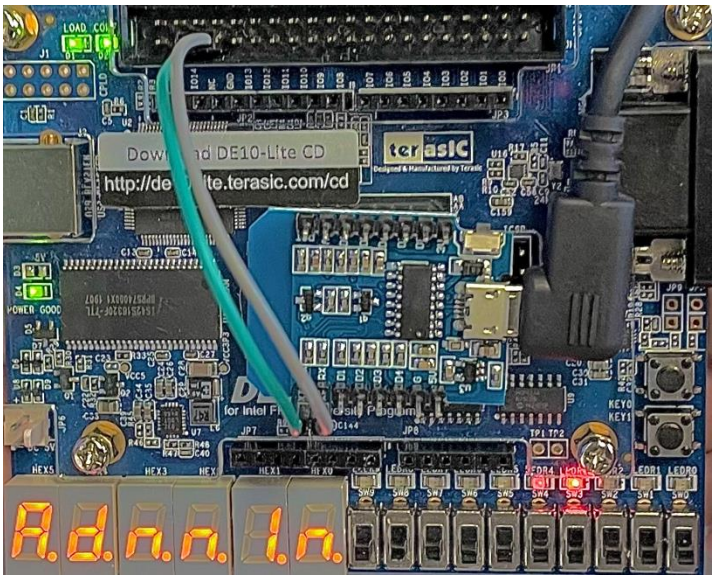
Game 2: Speed

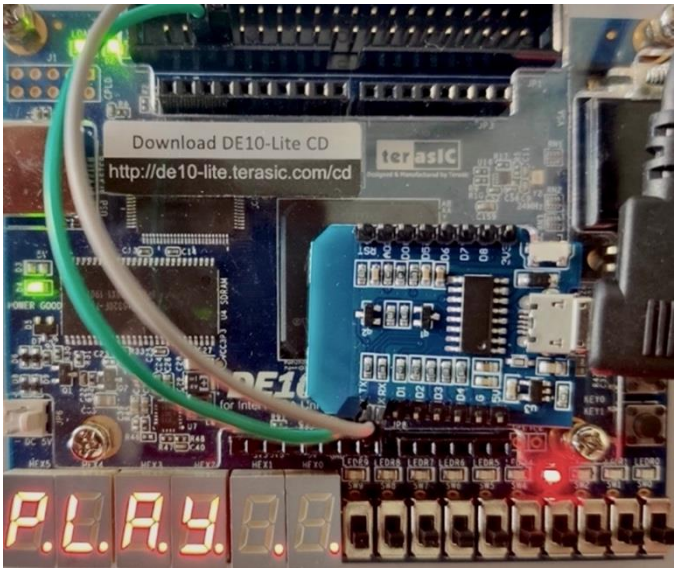
- This game is a race against users to correctly input the displayed code (a sequence of five movements) and send it to the server.
- This is a quick fire round to test the users' agility and accurate use of the FPGA controller.
- The user interface displays the winner and the server updates the leader board.

The existing communication infrastructure between nodes allows this platform to be easily expandable for more multiplayer games.

Design

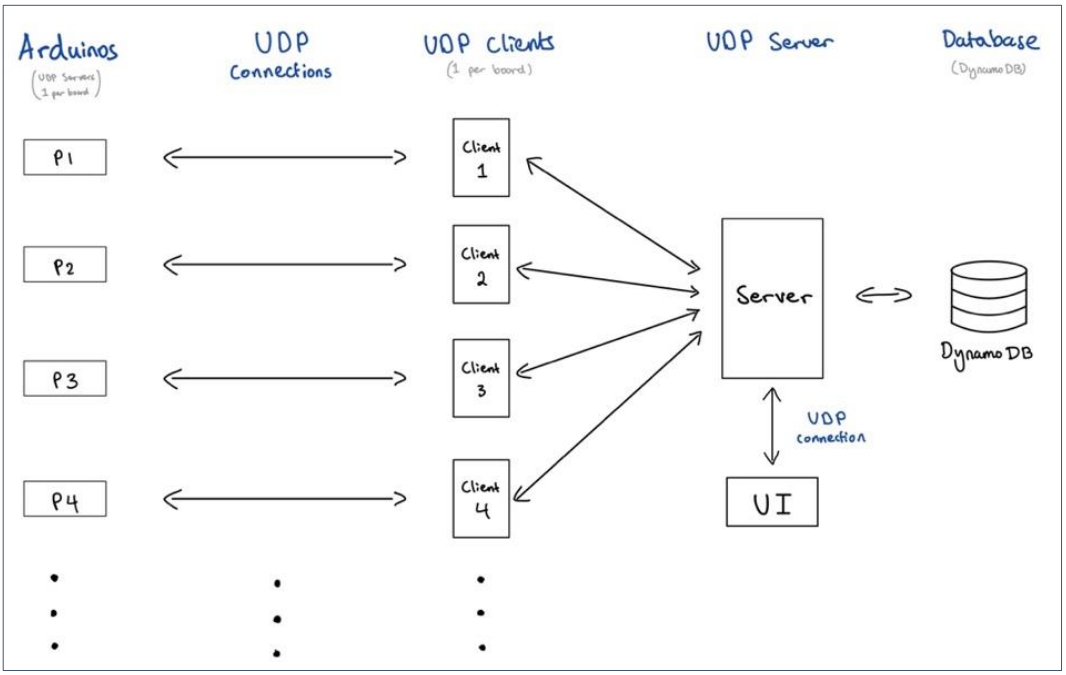
1. FPGA Hardware and Software





The FPGA is connected to an Arduino which has a built-in ESP8266 Wi-Fi module to communicate to the Python UDP client (see communication section). The reasoning for the design decision to include the Arduino is to enable Wi-Fi connectivity for each board, rather than having each board connected through a cable as done with the JTAG-UART. This allows greater portability and improves user experience, whilst avoiding JTAG-UART communication between the FPGA and the local computer host. Local processing on the FPGA happens depending on the state of the game. For example, if it is a user's turn to play, the data is received by the Arduino and sent via UART to the FPGA to change the seven-segment display to indicated "PLAY." If the player is in the "main menu" and is the "admin," i.e., decides which game to play, "ADMIN" is displayed on the seven-segment display. Otherwise, the standard display is the player number – e.g. "P1" for player 1. UART communication between the FPGA and the Arduino is enabled using the general purpose input/output (GPIO) pins. Two wires are used (indicated in figure above), for receive and transmission. In order to program these features onto the FGPA, we added the ***UART (RS-232 Serial Port) Intel FPGA IP*** to our .qsys file. In addition to this we added more on-chip memory to support the codebase for the software to be uploaded.

2. Communication



There are 4 independent communication streams:

- UART communication between each FPGA boards and their associated Arduino microcontroller.
- UDP communication protocol between each Arduino and its allocated Python UDP client.
- UDP communication protocol between the Python UDP clients and a UDP server running on a Ubuntu EC2 instance (controlling a DynamoDB database).
- UDP connection between the UDP server on the EC2 and a User Interface (UI) python script.

The EC2 instance runs a Python UDP server script which also controls a DynamoDB database. This UDP server script is run as a daemon so that it runs uninterruptedly and continuously in the background of the EC2. The central EC2 UDP server communicates with multiple Python UDP client scripts, where each UDP client is associated with a player and an Arduino board. This UDP client acts as a client to both the Arduino UDP server and the EC2 UDP server, and purposes in offloading processing from the Arduinos and EC2 server . It also enforces strict order control for the packets sent between the two servers, which compensates for the loss of reliability compared to using TCP. When testing the communication flow with and without the 'middle-man' client, we concluded that the DynamoDB database was accessed and modified faster when utilizing the clients. This allowed for faster leaderboard information retrieval between the EC2 server and UI due to less processing done on the EC2 server. The UI acts as another UDP client to the server, and constantly waits for messages to update its state in real time.

The design choice of a UDP protocol rather than a TCP protocol for communication was used for several reasons. Due to the nature of a multiplayer game, we require a fast means of sending and receiving packets for each player. UDP provides this necessary speed as it does not maintain a connection and packets are not routed over a fixed path. Furthermore, by not using TCP we eliminate a large amount of overhead associated with the data transmitted.

3. EC2 Instance and Database

The EC2 Instance acts primarily as interface which links all the physical devices into one system. All communication between different devices goes through the EC2 instance using UDP.

The most prominent features of the EC2 instance and the database are as follows:

1) Whenever there is a game update (e.g. a game starts or ends, a player makes a turn, a player is eliminated etc.) the server informs the UI of the change, so that the screen can be updated.

2) When in the menu screen, the EC2 instance communicates with the "admin board", allowing the admin user to select a board.

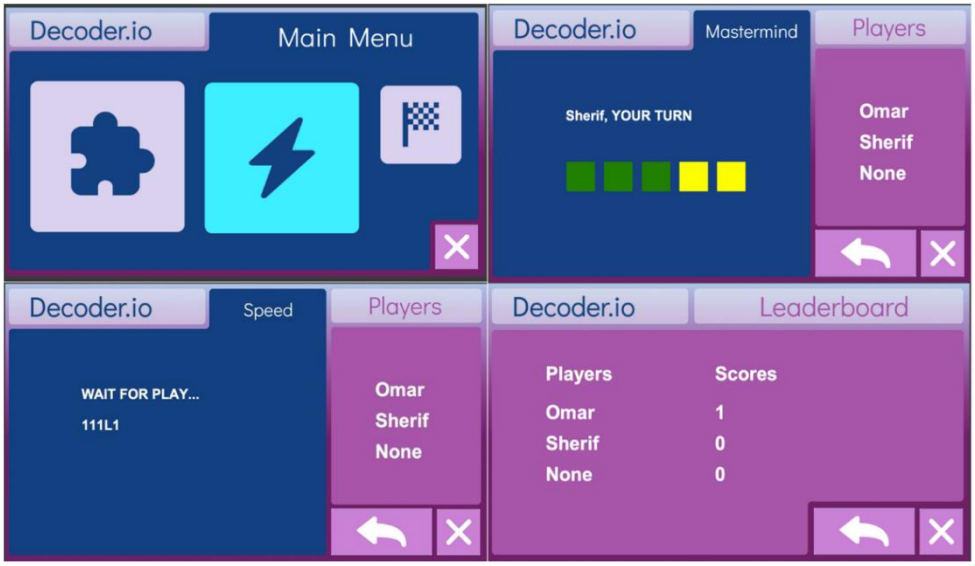
3) During games, the EC2 instance maintains the game (generates the random pattern in the game mastermind/keeps track of the current increasing pattern in the game memory) and informs users when it is their turn and receives the user's guesses.

4) At the end of every game, the server informs the UI of the score of each player so that the UI can update the leaderboard.

A DynamoDB database is used to keep track of the score of each user and is updated whenever a new board joins the game system (a new player is added to the database), or whenever a game end (the number of points of the winner is updated). The use of the database allows the number of points of each player to be saved, so that if the system is closed and reopened, the leaderboard is preserved. The database was chosen to be a DynamoDB database because of the unstructured nature of DynamoDB. This would allow more features to be easily added and data to the server in the future to store more user data. If the group had more time to continue developing the project, data such as the number of games played, and the win loss ratio of each player would be implemented into the database.

4. User Interface

The user interface is written using the *Tkinter* library for python. It consists of 4 key tabs: the main menu, *MasterMind*, *Speed* and the leaderboard. Navigating the menu is done by receiving commands from the server – these commands have a set format of [keyword + parameters] or [keyword] so they can all be processed similarly. For example: *leaderboard Jack 10*, would add the player *Jack* to the leaderboard, with a score of 10.

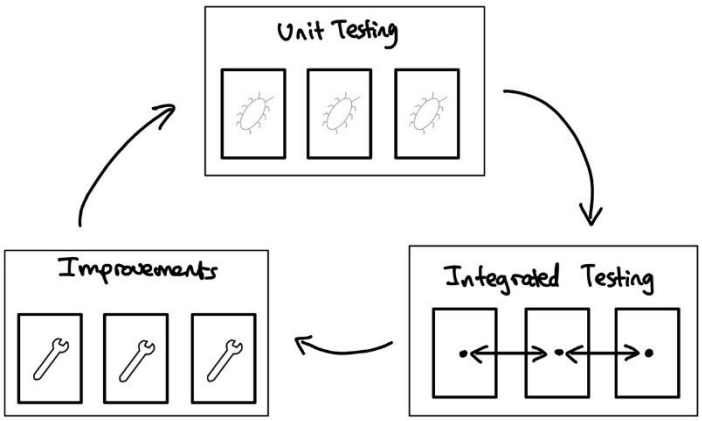


Threading was a key factor when programming the UI. While constantly listening for commands from the server, the script also needs to enter a *tkinter.mainloop(),* to provide functionality to the interface. We came to realisation that the two games also require separate threads to allow us to update the UI in real time while still listening to commands. Therefore, we created separate threads for the input listener and games.

Testing/Performance

We conducted methodical testing in three stages in order to achieve a reliable and robust system:

1. Testing subsections individually.
2. Testing connections between each subsection, individually.
3. Testing the complete system with edge cases.



1. Testing individual components

FPGA/Arduino:

- All switches, buttons, and LEDs pin assignment were to verified to see if they were correctly assigned.
- The software implementation on the hardware after the DE10-Lite had the .sof/.pof programmed to its EEPROM required continuous debugging. Paired programming proved effective here.
- FPGA Software testing was conducted using the command window in Eclipse to observe what outputs the NIOS II was generating and amend our code accordingly.
- The codebase was minimized as much as possible to allow efficient local processing on the FPGA.

EC2/Database

- The cloud server database functionality was tested on its own by using terminal inputs to replace the UDP connection between each node. The game structure and communication structure were implemented separately from the database, with the two parts of the cloud server being combined after each was tested thoroughly.

User Interface:

- To test the UI without a server running, we enabled secondary input through command line interface.

- The UI also recognised mouse input to navigate, allowing us to solidify commands and identify errors relating to faulty codes.
- Various print statements were added to indicate the state that the interface is in: “waiting for message,” “message received” and [received message]. These outputs also proved helpful when testing between various stages, as this allows problems to be better identified.

2. Unit Testing

UART communication/Arduino:

- An Arduino test script was implemented to read from the UART ports between FPGA and Arduino. This ensured that all data from the FPGA could be sent and read by the Arduino. Using the serial monitor, we could analyse the data being sent. After we confirmed complete functionality between the FPGA and Arduino, we proceeded to integrate the UDP communication.

User Interface to server:

- Testing this connection was as simple as replacing the terminal input lines of code with a socket.recvfrom() command and establishing a UDP socket to enable message transfer. Having such a simple connection on the UI side meant that errors here were exceedingly rare.

UDP Server/Client Protocol :

- Before implementing any game functionality, communication with the clients and the user interface. The key was ensuring that all client data is saved once a client first sends a message to the server, in a way that makes it easy for the server to access client data and send all of the clients a message.

3. Testing the complete system with edge cases

Testing of the complete system comprised of two parts:

1) Testing different game-flow scenarios:

- We ran the system having players input specific inputs to generate certain game results. In doing this we were able to debug the system covering all possible game scenarios so that it would reliably work when played.
    - For “Speed” this could be:
        - Player 1 gets the answer wrong, but sends it first.
        - Player 1 gets the answer right and sends it first, and player 2 then gets it right.
        - Player 1 gets the answer wrong, and player 2 gets the answer wrong.
    - For “Mastermind” this could be:
        - A player inputs a guess sequence and clicks the send button when it is not their turn.
        - The player inputs a sequence longer/shorter than 5 inputs long.
    - We also tested what happened in the game when the UI and the 7-Segment displays would show the expected outputs.

2) Testing the impact of parts of the system failing:

- To test this we would begin a game session and unplug one or both FPGAs or disconnect the Wi-Fi connection to the UI, simulating possible failures the users would potentially have during playing.
    - When the UI is disconnected from Wi-Fi, it is able to update when the reconnection occurs and so it remains up to date with the running of the game – importantly the leaderboard is kept up to date even if the UI does not display when disconnected from Wi-Fi.
    - When an FPGA is disconnected during game play, the UI and other players continue to receive the expected data from the server, and the game continues with one fewer player.
    - Unless there are only two players in the game, the game continues with one fewer player, much in the same way as before.