Information Processing Project 2024

Group 6

Sriyesh Bogadapati, Archisha Garg, Ronit Ravi, Sne Samal, Meric Song, Yannis Zioulis

1. Purpose of the System

The system is a Morse code chat application which uses a DE10 lite board FPGA, a computer and a server. There can be any number of participants chatting on the application in one of four rooms at any point in time. Participants can use the switches to submit messages and switch chat rooms. In addition, buttons, switches and gestures via the accelerometer can be used to craft messages in Morse code.

FPGA functionality requirements include gesture recognition, switch and button reading and formatting the seven-segment display. Users can press either button for dots and dashes, they can toggle a switch to send their message, alter other switches to change their chat room, and use tilt gestures to backspace, start new characters and words.

The Python GUI client application interprets Morse code input from the Nios II terminal, converting it into readable text. The TCP server handles multi-way chatroom functionality and relays messages between clients. Currently, chat history is stored unencrypted in DynamoDB, accessible to new clients.

Future enhancements will focus on implementing robust encryption for secure communication and data storage.

This proof-of-concept demonstrates the successful integration of the FPGA board as a Morse code input device and reliable chat communication between the Python client and EC2 server. It sets the stage for future improvements in security, performance, and user experience.

2. Overall Architecture of the System

2a. FPGA

The main requirements of the FPGA's functionality were:

- Read accelerometer tilt data, whose gestures provide additional functionality such as backspacing, new characters, new words.
- Read button presses that translate to dots and dashes.
- Read switch data that affect room changes, punctuation and sending of messages.

There needed to be a balance between sampling rate and transmission of data to the server. I.e. it is necessary to sample the accelerometer quick enough that buttons and switches are read as well, and there is minimal lagging, but at the same time, it is important to not sample too fast such that the participants can tilt the board for the desired gesture for the correct time so that only one gesture is sent to the server and not too many due to frequent sampling. To find this balance, we came up with a rigorous testing procedure that measured the average time taken for tilts for different people, using the onboard NIOS interval timer.

Team Member	**Average Tilt Time (μs)**[1]
Sriyesh	1,873,345
Archisha	2,243,989
Ronit	2,093,459
Sne	1,834,619
Meric	2,197,103
Yannis	1,717,141
Average	**1,993,276**

It became clear that a delay around 2 seconds was required to effectively capture, only one time, that the board was tilted. However, a delay of 2 seconds is too long and caused inevitable lagging in the recording of button and switch pressing, since a participant can press a button multiple times in a 2 second gap. Using a similar experimental procedure, we found that that a satisfactory sampling interval is around 0.45s.

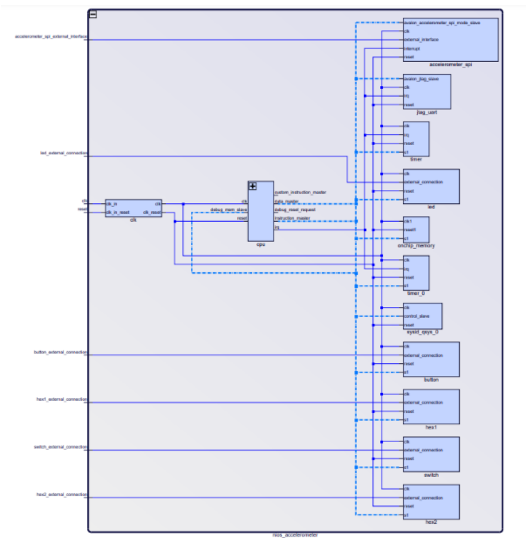
Since the time differences are relatively large, we came up with a solution to stop the NIOS II from detecting gestures for around 2s, without affecting the button and switch sampling rate. In our code, the condition for recognising gestures (flags) is set to false for several iterations after a gesture has been detected, effectively stalling the recognition of gestures after the first recognition. In the meantime, button and switch data is gathered as normal.

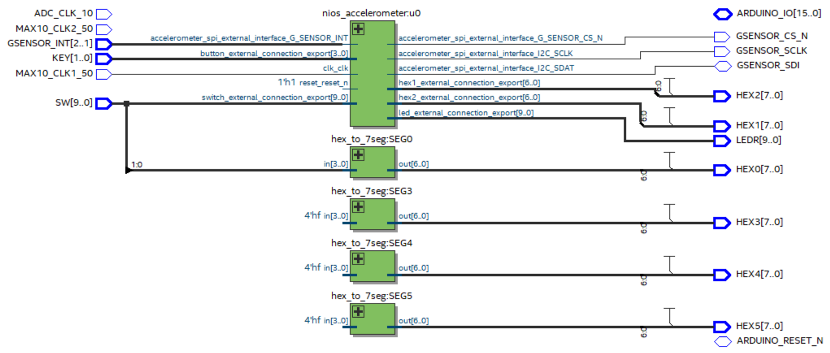
In the C code, it was necessary to know the sensor values when the board is tilted. This was done by keeping record of both x and y accelerometer data whilst the board was held at an ideal tilt level. Following this, upper and lower bounds were set to provide a range of acceptable values to determine whether the board has been tilted in a certain direction.

The switches were used allow participants to change rooms by binary encoding the room number they want to join. The most significant switch was used as a toggle to send complete messages to be viewed by everyone. This was done by reading values at `SWITCH_BASE` and sending appropriate responses to the server.

Finally, seven-segment displays were used to display the room number and an abbreviation for the gesture detected from the accelerometer. The former was not performed via the NIOS, instead, we used the `hex_to_7seg.v` file and altered Verilog code in the top level to directly show room numbers from the switches on hardware rather than software. This decision was made so that we can poll one less item in software. The later was implemented through software since the accelerometer was also on software and the hardware implementation would be extremely challenging.

[1] Average taken for forward, backward, left and right tilts, taken for 20 readings in total, 5 for each tilt.




ADC_CLK_10
MAX10_CLK2_50
GSENSOR_INT[2..1]
KEY[1..0]
MAX10_CLK1_50
SW[9..0]
nios_accelerometer:u0
accelerometer_spi_external_interface_G_SENSOR_INT
button_external_connection_export[3..0]
clk_clk
1'h1 reset_reset_n
switch_external_connection_export[9..0]
accelerometer_spi_external_interface_G_SENSOR_CS_N
accelerometer_spi_external_interface_I2C_SCLK
accelerometer_spi_external_interface_I2C_SDAT
hex1_external_connection_export[6..0]
hex2_external_connection_export[6..0]
led_external_connection_export[9..0]
hex_to_7seg:SEG0
hex_to_7seg:SEG3
hex_to_7seg:SEG4
hex_to_7seg:SEG5
in[3..0]
out[6..0]
4'hf
ARDUINO_IO[15..0]
GSENSOR_CS_N
GSENSOR_SCLK
GSENSOR_SDI
HEX2[7..0]
HEX1[7..0]
LEDR[9..0]
HEX0[7..0]
HEX3[7..0]
HEX4[7..0]
HEX5[7..0]
ARDUINO_RESET_N

The Platform Designer schematic below shows the peripherals used alongside the NIOS II soft processor and the final Verilog top-level module summarises how the embedded hardware was combined with the processor:[2]

2b. Server and Client Side

Splitting the processing across a server and client nodes allows us to have a centralised system which handles all the clients' connections and is responsible for relaying information between clients, acting as a router or hub, and it allows the clients to handle all the localised processing, distributed the total cost of computation. The tasks can be seen in the table below:

Server	Client
• Handle all client connections and information. • Process client requests. • Store chat logs in database	• Process input data from FPGA. • Maintain server connection. • Translate morse to English characters. • Display all relevant information on a front-end GUI. • On room connection, pull chat logs from database.

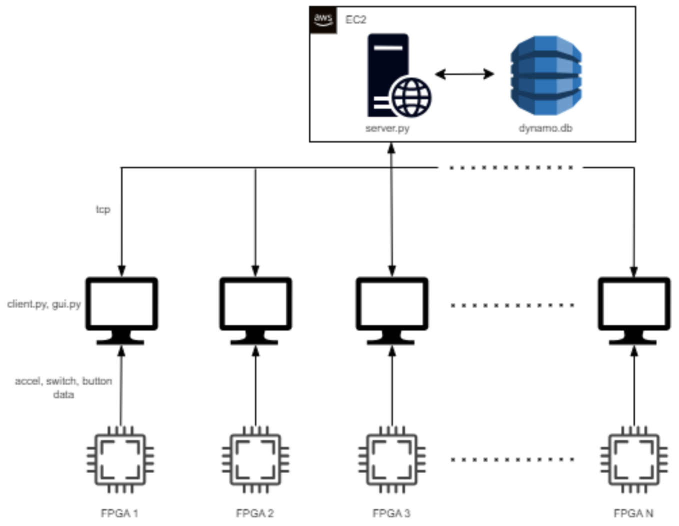
To handle multiple clients, the server must listen to client connection requests on starting up and continue to do so while running and handling other clients. Due to this, we have decided to start a new thread for each client. This allows parallel processing of multiple requests coming in from multiple clients at the same time.

Upon connection, the client will be queried for their alias and room that they would like to be connected to. This information is then used alongside the socket to create a client object. This is then stored in a nested array, where the first dimension represents the room the client is connected to.

2c. GUI and DynamoDB

Our user interface consists of an individual client window for each corresponding connected FPGA. Using this window as a chat/communication function we can toggle switches and press buttons to send a morse letter to the "Display Box", this acts as an intermediary buffer between the user and the chatroom where other FPGA clients can communicate. Using the send gesture will allow the intermediary buffer to then be cleared and its contents sent to the chatroom and stored in our communal database in DynamoDB. There are four chatrooms instantiated with its own corresponding chat history stored, that can be quickly switched from the toggle of FPGA switch [9]. When messages are sent to the room the user ID is tagged alongside the actual converted morse message and corresponding time stamps, the user ID is randomly generated for security purposes.

The overall architecture of the system can be seen in this system diagram:



[2] Some displays were hard coded to be turned off, resulted in minor modifications to `hex_to_7seg.v`

3. Performance Evaluation; Performance Metrics of the System

In terms of network latency, we measured an average RTT time of 35ms, a very acceptable delay given the time necessary to form a response to any received message.

On the server side, a provisioned throughput of 5 capacity units was configured both for reading and writing into the DynamoDB database. This implies a maximum of 5 reads of size 4 kB per second, and 5 writes of size 1 kB per second. This configuration acts as the main limiting factor to the number of users that can be connected at once. (We have not been able to hit the limit of reads/writes while testing with 6 users communicating simultaneously)

4. Summary of Important Design Decisions

5a. FPGA

Hardware (FPGA) decisions: Our design utilised all four tilts, both buttons, and 7 of the switches to ensure ease for the user. We displayed letters for the tilts such as E for forward to include an English space, L for backward to backspace English letter, B for left to backspace morse symbol, S for right to send entire morse array and convert it to an English letter.

A decision was made to make use of switches to enable punctuation shortcuts. Switches 7 to 4 inclusive we used for full-stops, commas, exclamation points and question marks respectively. This added additional functionality from the FPGA perspective and was a more efficient method as opposed to typing Morse for punctuation.

5b. Server and Client Side

When choosing how to store the client information on the server, we initially had two methods:

The first being to hold two 2D arrays, for the client socket and their alias where the first dimension represents the room they're in. The benefit of this approach meant that when broadcasting messages we can iterate through the required room. However, it meant we couldn't access the room number from the client but would rather require the opposite.

A potential workaround for this we thought of was that we could have the client send their room number along with every message they sent, as a header to the message being the payload. But this meant that we were adding extra network traffic and would be bad scalability practice. Furthermore, it meant we would require further computation for when the client undergoes a room change, as we would need to keep track of the previous and current rooms of each client.

The other method we had come up with was to store a single object for the client, with attributes holding its alias, room and socket information and store all active clients in a single list. Since python doesn't have any pointers, we could not do two-way graphs, the benefit of this meant that we could access the rooms from each client. However, the downside of this was if we were to broadcast the information, we would have to iterate through every client and check their room attribute to see if it was the right one. This would be very bad for scalability once again.

The solution we settled on incorporated both methods in one. With the introduction of static redundancy by storing the room number in the client object in a list of rooms which themselves were lists, we minimise the unnecessary traffic and the elements in the iterations.

6. Testing Approach

6a. FPGA

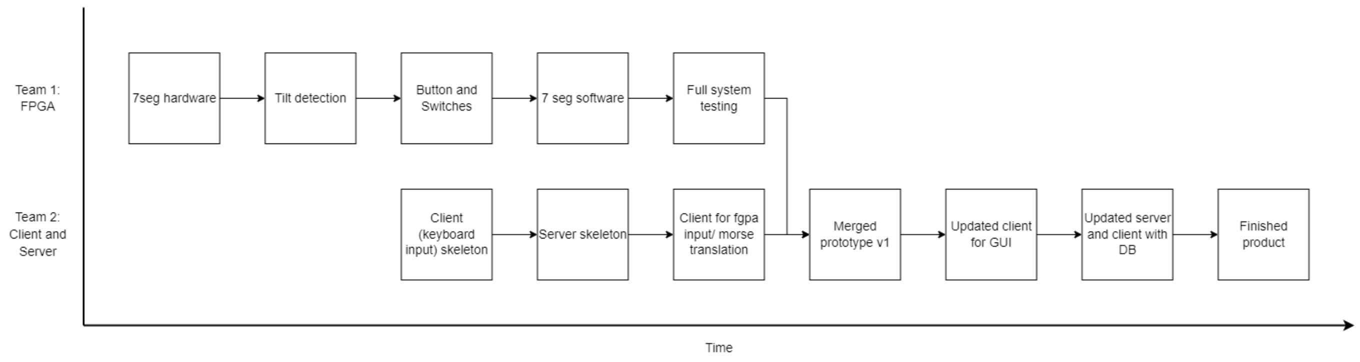
The code on the FPGA was built up bit by bit. At first, `hex_to_7seg.v` was tested. Following the implementation of the NIOS processor, testing began on detecting button and switch inputs., then accelerometer tilt data in accordance with the method outlined in section 2a. Finally, the seven-segment display used to show tilt abbreviations was tested before merging with the client and server.

6b. Client and Server

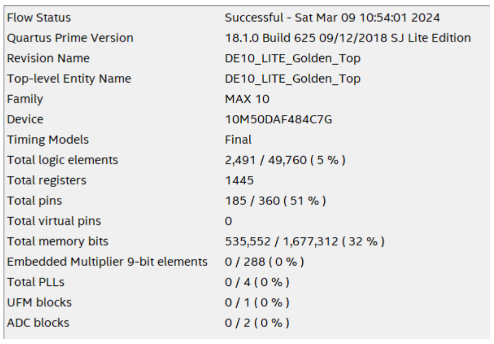
Client and server testing was conducted by using our prototype versions of client.py and server.py which had keyboard input (as opposed to input formed by using the FPGA). We tested a set of scenarios such as communication between multiple users and joining and exiting chatrooms, both empty chatrooms and ones that are already occupied by other users.

6c. Combined Testing

Once both teams finished, FPGA, client and server were combined and tested together. Following successful testing of sending and receiving messages in different rooms, a GUI was built to visualise activity in different rooms. Following this successful test and integration, a dynamo DB was included to store message history and display previous messages to the user when the join a room. The overall testing flow is encapsulated in the diagram below:



7. Resources utilised for the DE10-Lite from Quartus



Flow Status	Successful - Sat Mar 09 10:54:01 2024
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	DE10_LITE_Golden_Top
Top-level Entity Name	DE10_LITE_Golden_Top
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	2,491 / 49,760 (5 %)
Total registers	1445
Total pins	185 / 360 (51 %)
Total virtual pins	0
Total memory bits	535,552 / 1,677,312 (32 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

From the compilation report, our design is not constrained by memory requirements, only 32% is used. Seeing that this a relatively small design, it's not surprising. This compilation report shows statistics like that of the labs., since in both cases, the maximum potential of the processor was not used.

8 Extension – E2E Encryption

End-to-end encryption (E2EE) is a crucial technique employed by popular chat applications like WhatsApp, Telegram, and Signal to ensure secure communication over untrusted networks. In our extension, we made the assumption that client-side security is robust, and therefore, a shared private key encrypted message relay is sufficiently secure. However, it's important to recognize that this design is susceptible to man-in-the-middle (MiTM) attacks in real-world scenarios.

In a non-ideal world outside the scope of this project, one can significantly reduce the risk of MiTM attacks, a digital signature check such as OpenPGP can be implemented during the public key exchange process. While borderline unnecessary for most commercial applications, this extra step prevents a malicious actor from generating two fraudulent private keys to intercept and relay the communication. Additionally, employing an ephemeral key exchange system that changes private key for every session, provides forward secrecy; ensuring that even if a private key is compromised, past communication records are safe from attack as they were encrypted using different keys.

In our implementation, when clients connect to the server, they receive unique userIDs that other clients can use to initiate a secure communication channel. The process begins when one client requests a secure session. The server then notifies both clients that it is ready to facilitate the ECDH key exchange by broadcasting a "/ready" message.

Upon receiving the "/ready" message, each client computes their public key and sends it to the server, which relays it to the intended recipient. The clients save the incoming "/ecdh_key" message locally. Once both clients have exchanged their public keys, they perform elliptic point multiplication using their private key and the shared public key to generate a shared private key.

After the private key is computed, the clients notify the server with a "/secure" broadcast, indicating their readiness for secure communication. The server considers the session secure only after receiving this confirmation and starts relaying incoming encrypted messages between the clients.

The secure session check is a robust mechanism to prevent users from communicating unencrypted messages and does not introduce vulnerabilities to the E2EE scheme, as all encryption and decryption operations are performed on the client side.

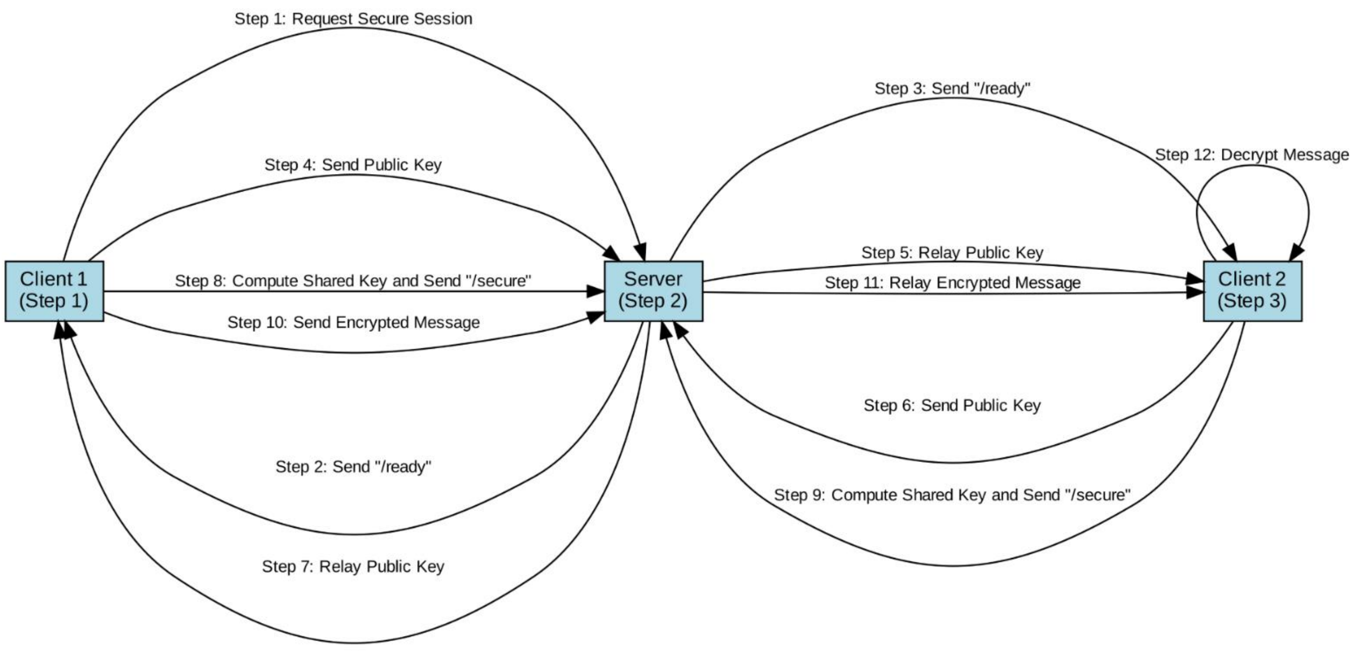
When a client initiates a new session with a different user, the current session is terminated, and the entire process restarts from the beginning. This entire process is visualised in the diagram below.

Throughout the development of this extension, we gained a deep understanding of the underlying principles and practical implementation of end-to-end encryption. Studying the intricacies of the Diffie-Hellman key exchange, particularly the Elliptic Curve Diffie-Hellman (ECDH) variant, provided valuable insights into the mathematical foundations of secure communication protocols.

By carefully designing and implementing the key exchange process, session management, and message relay, we have created a secure chat extension that prioritizes user privacy and data protection. While our assumptions about client-side security simplify the design, it's essential to acknowledge and address potential vulnerabilities in real-world deployments.

Moving forward, incorporating digital signature checks and ephemeral key exchange mechanisms would further enhance the security of the system, providing robust protection against MiTM attacks and ensuring forward secrecy.

Overall, this extension has been an enlightening experience, allowing me to apply theoretical knowledge to a practical implementation of end-to-end encryption. It has deepened my appreciation for the importance of secure communication and the challenges involved in designing resilient systems.


Step 1: Request Secure Session
Step 4: Send Public Key
Step 3: Send "/ready"
Step 12: Decrypt Message
Client 1 (Step 1)
Step 8: Compute Shared Key and Send "/secure"
Server (Step 2)
Step 5: Relay Public Key
Step 11: Relay Encrypted Message
Client 2 (Step 3)
Step 10: Send Encrypted Message
Step 6: Send Public Key
Step 2: Send "/ready"
Step 9: Compute Shared Key and Send "/secure"
Step 7: Relay Public Key