

University of Toronto
ECE 532H1 Digital Systems Design

Final Project Group Report:
Virtual Vaccine Passport Scanner for Remote Entry Approval

Group #3:
Mustafa Kanchwala
Guoxian Wu
Eduardo Stecca Ortenblad
Xuening Dong

April 14, 2022

Table of Contents

1.	Project Overview	4
1.1.	Background and Motivation	4
1.2.	Goals	4
1.3.	Technical Background	4
1.3.1.	Data Matrix	4
1.3.2.	TLV	5
1.4.	Design Overview	5
1.5.	Block Diagrams.....	9
1.6.	Brief IP Description	12
1.6.1.	VGA Display IP	12
1.6.2.	Data Matrix Decoding IP	12
1.6.3.	Database Parsing IP.....	12
2.	Outcome	12
2.1.	Changes Made	12
2.1.1.	System Changes	12
2.1.2.	Requirement Changes.....	13
2.2.	Results.....	13
2.3.	Future Work.....	15
2.3.1.	Improving utilization of the Server Database:	15
2.3.2.	Improving speed of GET and POST requests:.....	15
2.3.3.	Implementing QR code decoding:.....	16
2.3.4.	Camera input:.....	16
3.	Project Schedule.....	16
3.1.	Milestone #1: Research and initiation	16
3.2.	Milestone #2: Basic functionality.....	17
3.3.	Milestone #3: Finish most units.....	18
3.4.	Milestone #4: Initial integration and debugging	19
3.5.	Milestone #5: Final integration and debugging.....	19
3.6.	Milestone #6: Final Testing	20
4.	Description of the Blocks	21
4.1.	Custom Hardware IP	21
4.1.1.	VGA IP.....	21

4.1.2.	Data Matrix Decoding IP	22
4.2.	Database Parsing IP.....	24
4.3.	Important IP Blocks Used.....	26
4.4.	Integration of Hardware System.....	27
4.4.1.	Integration of Data Matrix and VGA display	27
4.5.	MicroBlaze Software	30
4.5.1.	FPGA Client (FPGA 1).....	30
4.5.2.	FPGA Database Server (FPGA 2).....	30
4.5.3.	TLV Encoding/Decoding (FPGA 1 and FPGA 2).....	31
4.6.	Python Script (PC 1)	32
4.6.1.	Python TLV Encoding/Decoding.....	34
5.	Description of Design Tree.....	35
5.1.	How to Use.....	35
5.2.	Repository Structure (only major components included)	37
6.	Tips and Tricks.....	38
6.1.	Tips for the design process	38
6.2.	Tips for making block designs	39
6.3.	Tips for working with Vivado and Xilinx SDK on DESL machines	39
7.	References	40
8.	Appendices	41
8.1.	Appendix A: TLV Communication Protocol.....	41
8.2.	Appendix B: Database Entry Information	43
8.3.	Appendix C: Proposed Functional Requirements	44
8.4.	Appendix D: Utilization Reports.....	45
8.5.	Appendix E: Collection of Milestone Progress Reports	46

1. Project Overview

1.1. Background and Motivation

It is 2022 and COVID is already 2 years old! Even though this is not the case anymore, your vaccine passport used to be your key to entering a restaurant, school, office building, airports and almost everywhere else you wanted to go. Our project implements an exciting solution where a user can scan a user's vaccine passport at a remote client terminal near an entry point. Once scanned their information is verified by a server in the cloud using a database, and they are approved to enter if they are vaccinated.

The key advantage to our approach is that no more human interaction is going to be needed throughout the entire process of vaccine checks. Although speedup in the vaccine passport scanning process is not guaranteed for our current implementation on the Nexys 4 DDR FPGA, this project can be further improved with more powerful CPU cores that guarantees a speedup in the entire process. This is because, unlike the current vaccine passport, our project will be implementing custom cores/IP in hardware for both the Data Matrix decoding and the server database implementation.

Finally, it is worth mentioning that Data Matrices and QR codes are widely used for a variety of different purposes. Our project can be easily modified to be used in different application such as warehouse managing, ticket scanning, and product manufacturing. We chose a Vaccine passport solution as it is very relevant to our current scenario.

1.2. Goals

The Goal of our project is to successfully implement a digital, FPGA cloud-based database system that combines a Data Matrix decoder with a remote FPGA server located in a data center, to mimic and try to improve upon the current vaccination passport system that is being widely used throughout Canada and the world by removing the need for a person to be present to scan the vaccine passport and allow or deny entry.

1.3. Technical Background

1.3.1. Data Matrix

Similar to a QR Code, a Data Matrix is a square or rectangular two-dimensional code consisting of black and white cells that represent bits. It is composed of two solid adjacent "L" shape borders (finder pattern) and the two other borders consists of alternatively dark and light cells (timing pattern) [1]. Apart from the data blocks, it also consists of error correction blocks in case some blocks are not readable. Figure 1 shows an example of a Data Matrix.



Figure 1: Data Matrix Example (Left). An annotated example of a Data Matrix code showing data (green), padding (yellow), error correction (red), finder and timing (magenta) and unused (orange) [1]

1.3.2. TLV

Whenever a TCP message is sent over our network, this message needs to be encoded into packets. We decided to use the TLV (type-length-value) encoding scheme to encode our messages as this is a widely used encoding scheme in the embedded industry.

TLV messages can contain a variable number of elements of variable length. In our case, each element of the TLV messages followed this convention:

Type	Length	Value
1 Byte	1 Byte	Length Bytes

By using the TLV encoding scheme, messages flowing through our network can have variable lengths. We designed our TLVs to mimic RESTful APIs: that way we could send GET and POST requests to our database through the TCP layer at a fraction of the data size and resources required by HTTP RESTful JSON requests - our messages are encoded in no more than 128 bytes. Refer to Appendix A for a clear description of our TLV protocol design.

1.4. Design Overview

Our project consists of a standalone TCP server that hosts a database on an FPGA (FPGA2). This database stores our users' personal identification data and vaccination status; refer to Appendix B for a detailed description of the type and format of the data that is stored in the database. The current implementation uses a 64MB dedicated region in the Nexys 4 DDR Boards' DDR2 memory. This allows us to store 16384 entries where each user can have up to a maximum of 4kB of information associated with their user ID. Our vaccine passport

implementation only requires a maximum of 128 Bytes. The additional storage capability demonstrates our ability to scale our implementation to beyond the vaccine passport use case.

Additionally, our database can accept requests from a multitude of simultaneous clients accessing and modifying its data. These can be both PC clients and FPGA clients. However, for simplicity purposes, in the following sections we will be describing our system as having a single PC client (PC 1) and a single FPGA client (FPGA 1). Refer to *Figure 2* for the network connection within our system.

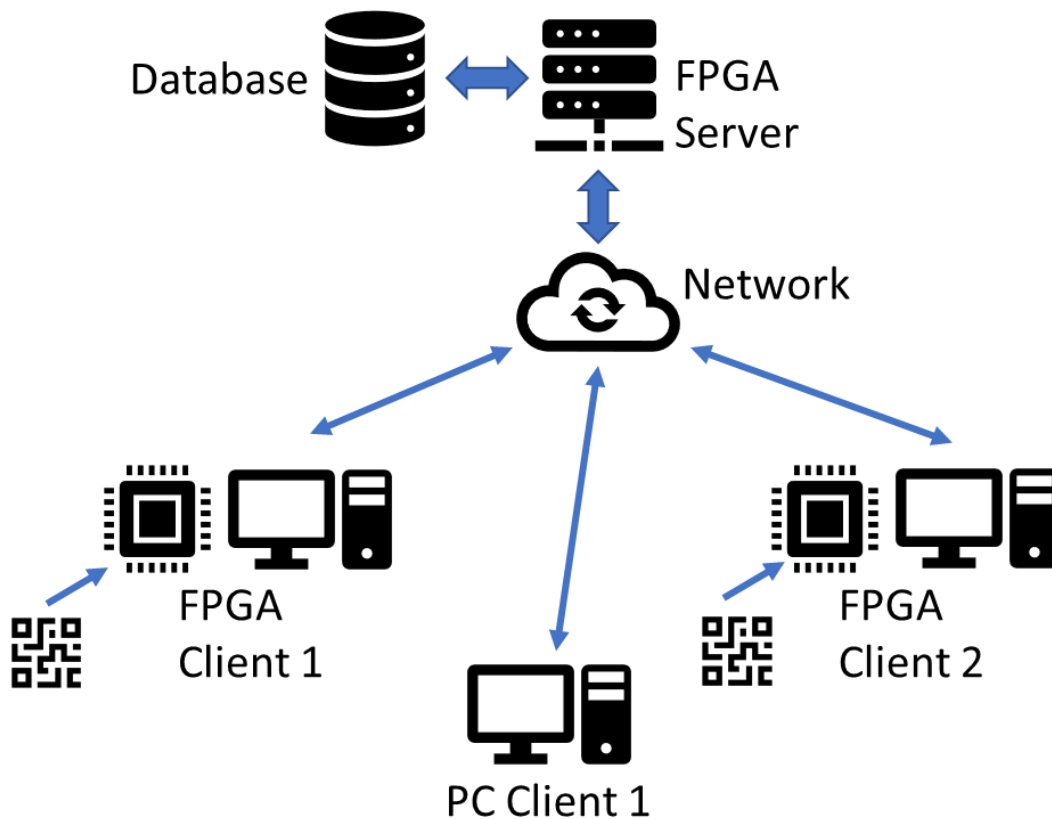


Figure 2: System Network Connections

Our system can be divided into three main components as follows (Refer to *Figure 3* for a system overview diagram):

FPGA Server (FPGA2): This is the FPGA containing our database server. It receives GET and POST requests from clients in the network and handles these requests appropriately by fetching (GET) or modifying/adding (POST) entries in the database.

FPGA Client (FPGA1): This can be any FPGA connected to the FPGA-NET network. This FPGA receives a Data Matrix encoded with a user's UID and the first 10 characters of their name. The Data Matrix is sent to FPGA 1 through any PC in the FPGA network over TCP/IP. FPGA 1 then proceeds to decode the Data Matrix and sends a GET request to the database for that specific user. If the UID is found within the database, the server returns the information associated with that UID to the client. Otherwise, the server returns an error message to the client. FPGA1 then

proceeds to parse the server's response. Finally, the result of whether a user is allowed or denied entry based on their vaccination status is displayed on the VGA display.

PC Client (PC 1 or Admin PC): This can be any PC in the FPGA Network. We created a single python script that allows this PC client to communicate with both FPGA 1 and FPGA 2 depending on the user's needs.

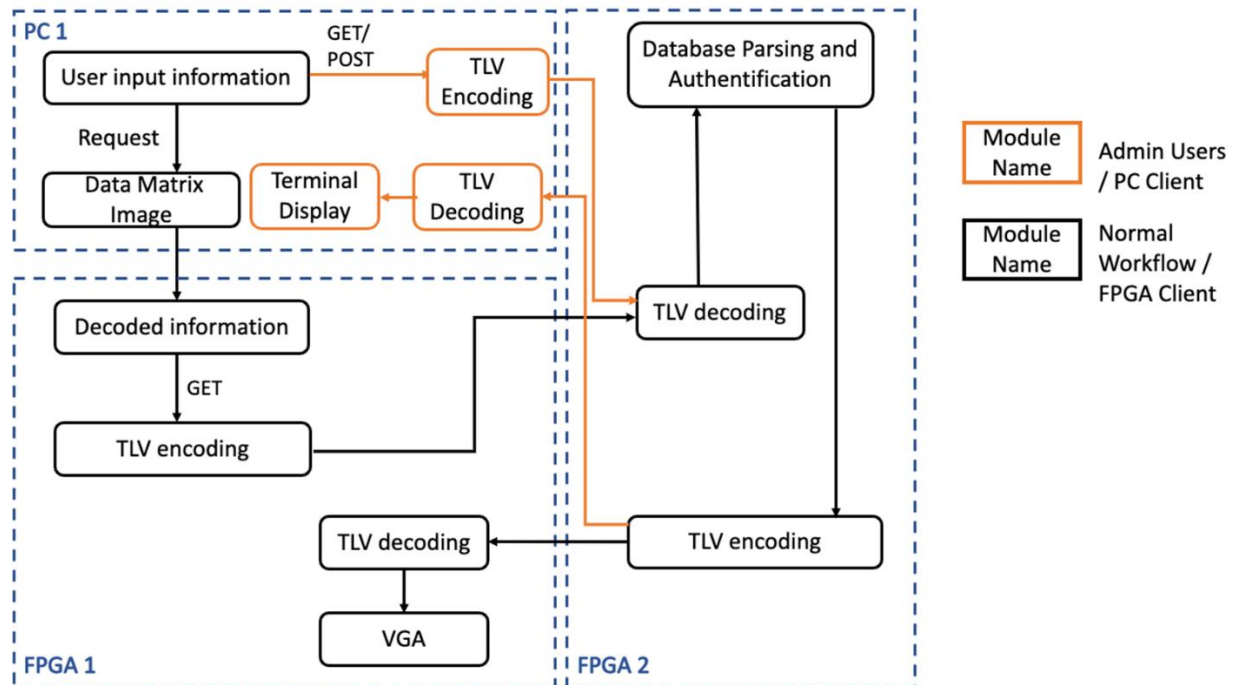


Figure 3: System Overview Diagram

There are two workflow modes for the system:

1. Administrator Mode
2. Normal (Data Matrix Scanning) Mode

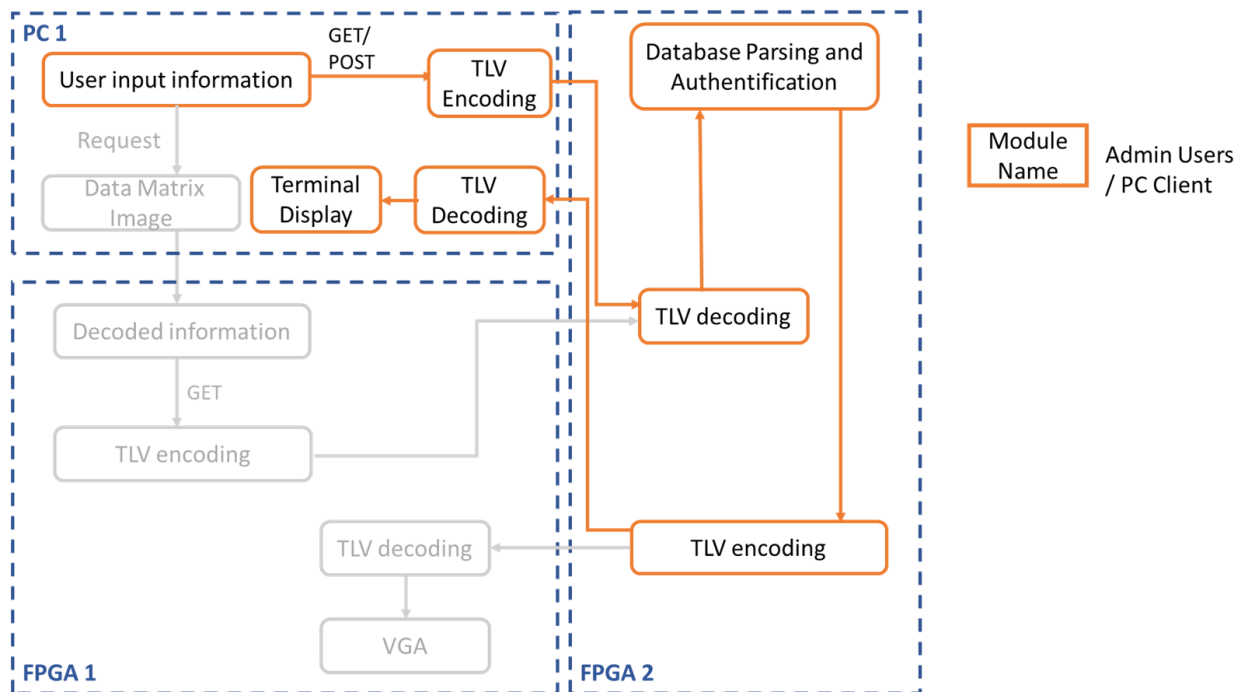


Figure 4: Administrator Mode Workflow

Workflow 1 (Administrator): This is supposed to simulate an Admin user to our database; refer to *Figure 4*. Through this workflow, our python script allows users to:

1. Create new database entries and generate the appropriate Data Matrix
2. Preload users from a .csv file into the database and generate their Data Matrix accordingly
3. Read entries from the database and display them on terminal
4. Update entries on the database by fetching that entry, displaying its current stored data, and allowing the user to modify it

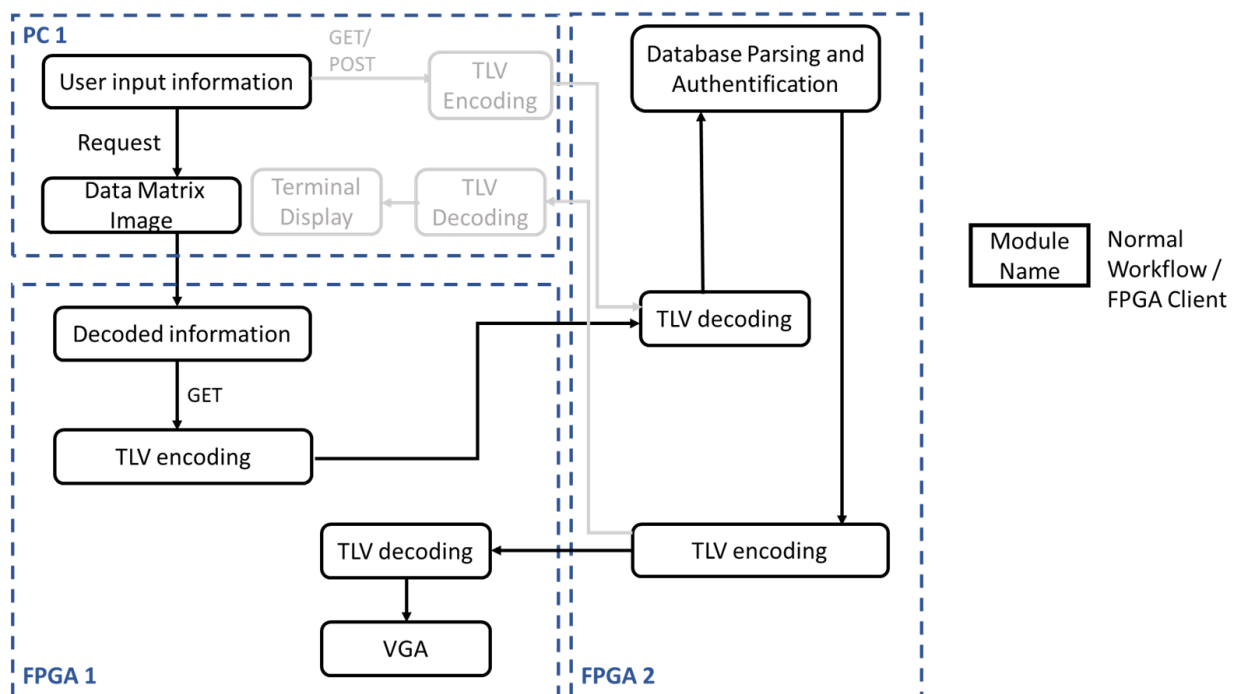


Figure 5: Data Matrix Scanning Mode Workflow

Workflow 2 (Data Matrix Scanning): This is supposed to represent a Data Matrix being scanned in order to allow/deny entry of that user. The PC sends the appropriate Data Matrix to FPGA 1, initiating the FPGA 1 workflow; refer to Figure 5. Through this workflow, our python script allows users to:

1. Input a Data Matrix image (.png file) and send this Data Matrix as an input to FPGA 1
2. Input a user's UID and Name, generate the appropriate Data Matrix and send it as an input to FPGA 1

1.5. Block Diagrams

The overall system block diagram is shown in Figure 6. It shows the connections of the Client FPGA, Server FPGA, and the Admin PC on the FPGA-Net interface. The Client FPGA also has a VGA display connected to it to display the results of a user's request.

Figure 7 shows the internal details and a system level block diagram of the FPGA Client (FPGA 1). 2 custom IP blocks are used for Data Matrix Decoding and VGA Interfacing, while several existing IP blocks are used for the overall system. A MicroBlaze soft-core processor IP is used as a microprocessor for the system to run the software code.

Figure 8 shows the internal details and system level block diagram of the FPGA Server (FPGA 2). 1 custom IP block is used for the overall system. Once again, a MicroBlaze soft-core processor IP is used as a microprocessor for the system to run the software code.

Figure 9 shows the software workflow of the complete system, which includes both the client and server FPGAs and the Admin/Client PC. The software is a combination of C programs run on

MicroBlaze for ethernet connectivity, TLV encoding/decoding and providing control signals to the different peripheral IPs, and Python scripts run on the PC terminal to control Data Matrix push requests to the Client FPGA, or pre-loading / adding new values to the database on the Server FPGA.

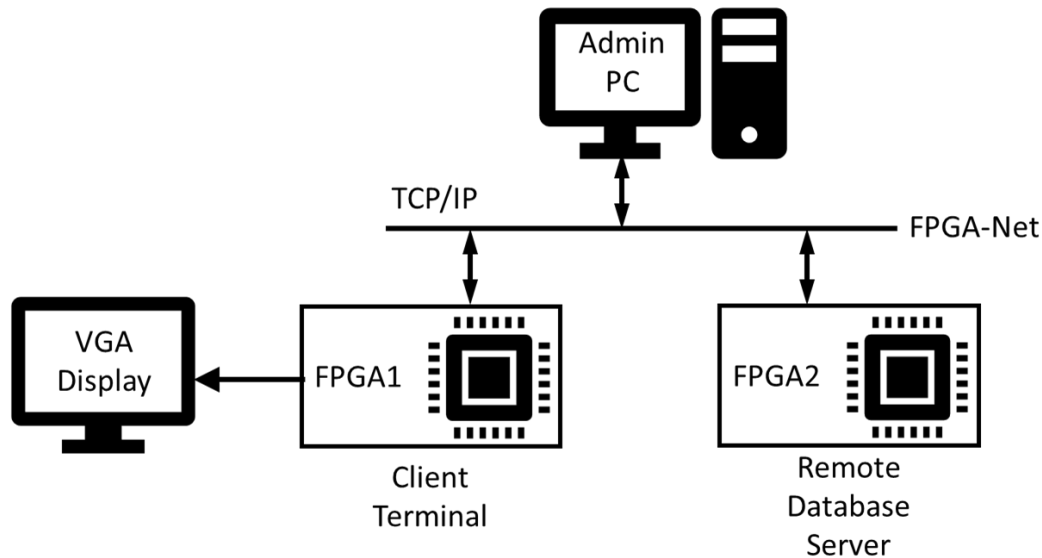


Figure 6: Overall System Block Diagram

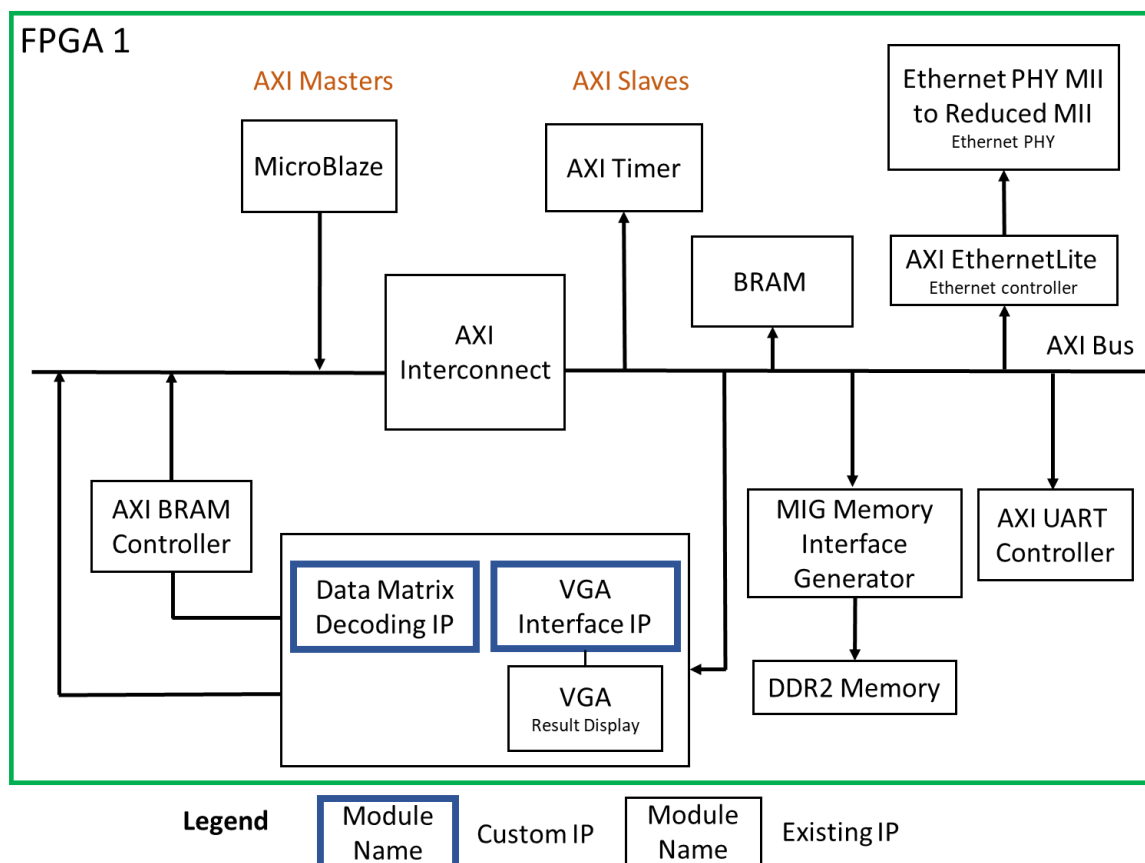


Figure 7: Client (FPGA 1) System Block Diagram

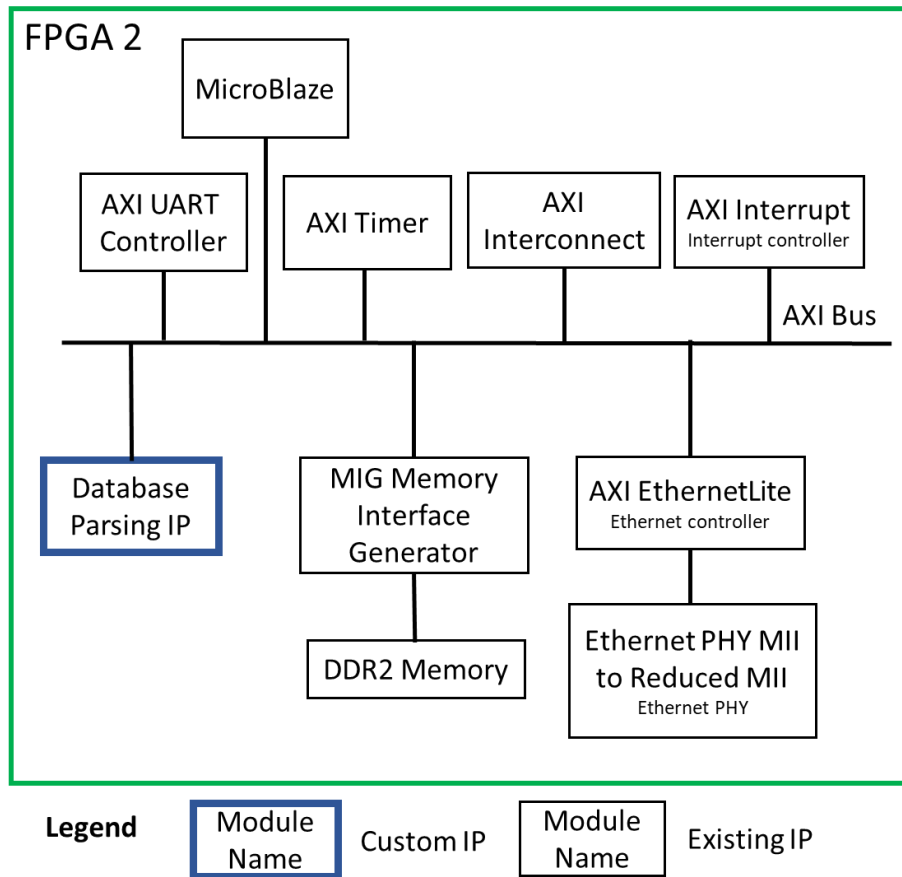


Figure 8: Server (FPGA 2) System Block Diagram

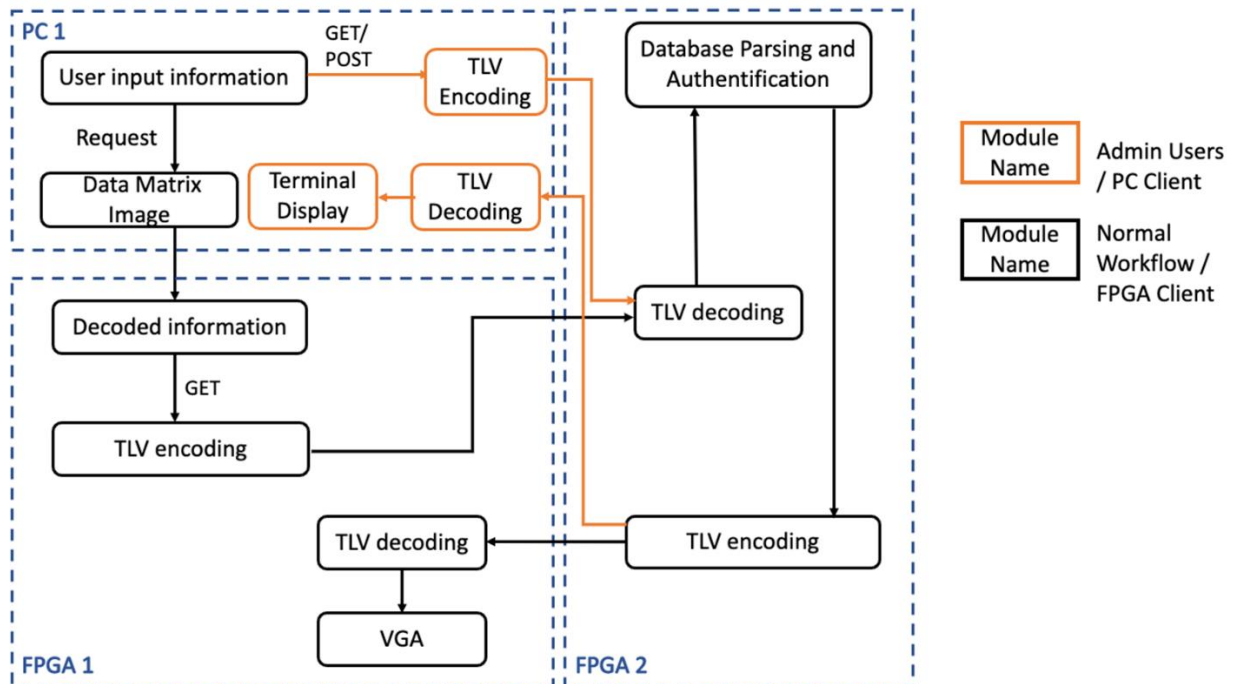


Figure 9: Software Workflow Block Diagram

1.6. Brief IP Description

1.6.1. VGA Display IP

The VGA display IP is a custom IP for outputting information of the user on the monitor screen. It directly writes the RGB information to pixels on the screen according to the user's name, the number of doses that person received, and if permission has been granted to the person. This allows the user to know if he/she is permitted to enter the place without the presence of an actual person collecting such information at the entrance.

1.6.2. Data Matrix Decoding IP

The Data Matrix Decoding IP is a custom IP that decodes the bitstream of Data Matrix into personal information including names and UIDs.

The integrated IP of Data Matrix Decoder and VGA consists of AXI-Lite Slave and Master Registers to for data and control signal transmission between hardware and MicroBlaze, as well as a BRAM controller to store Data Matrix.

1.6.3. Database Parsing IP

The Database Parsing IP is a custom IP created with a goal to accelerate the task of reading and writing from the database. It consists of an AXI-Lite Slave with 5 registers and an AXI-Full Master to perform the read and write to the memory. This allows MicroBlaze to become available to perform other tasks such as receiving and processing new requests and communicating to other clients and peripherals.

2. Outcome

Even though we had to make some adjustments from our original proposed design, we were all very satisfied the outcome of our final project. The next sections will describe the changes made from our original proposal, the outcome results of our final project, and future improvements that we would like to see.

2.1. Changes Made

2.1.1. System Changes

As our project matured, we realized that some of our proposed ideas needed to be changed. This was either because of challenges faced, because of initial misconception, or simply for practicality purposes.

1. Switching to Data Matrix from QR code decoding:

After the first milestone discussion, we decided to switch to using Data Matrix instead of QR code due to challenges in implementing a QR code decoder. After doing some research, we realized that the QR code protocol is extremely complex, and we found that decoding a QR code in hardware would be very challenging.

2. Performing TLV encoding and decoding in Software:

TLV implementation in hardware was an initial stretch goal, and after the initial implementation, we realized that TLV encoding and decoding could be performed very efficiently in software, while allowing the freedom to make changes to the structure to add more data. So, we decided to keep TLV as a software module.

3. Transmission of Data Matrix to FPGA 1 through TCP instead of UART:

Initially, we failed to realize that FPGA 1 could simultaneously act as a TCP client (to send GET requests to FPGA 2) and as a TCP server (to receive a Data Matrix from PC 1). Once we figured out that was possible, since we already had experience with setting up a TCP server, we opted to transmit the Data Matrix through TCP rather than UART.

2.1.2. Requirement Changes

With changes made to our overall system, requirements that we had initially proposed for our project had to change accordingly. Hence, we either updated the requirements or merged multiple requirements together to ensure that our project requirements would be coherent with our system changes. Below is a table that identifies the requirements that were updated, and summarizes the changes made; refer to Appendix C for our initial proposed requirements as presented in our Project Proposal. Section 2.2 will then describe the requirements that have successfully been met.

No	<u>Initial</u> Requirement/ Feature	Priority	<u>Updated</u> Requirement/ Feature	Changes Made
1	QR Code data should be correctly decoded by the FPGA	High	Data Matrix data should be correctly decoded by the FPGA	Switched to using Data Matrix instead of QR code due to initial challenges with implementation
7	UART for data transmission between PC1 and FPGA1	High	TCP/IP for data transmission between PC1 and FPGA1	Switched to using TCP/IP instead of UART to send and receive data. TCP/IP was more efficient and faster
12	AXI GPIO connected to LEDs showing the access status	Low	Merged with requirement #8 – “VGA Display can show results received from the server, and display visually if access is granted or denied”	Requirement 8 effectively does the same task and displays results visually. VGA display shows red for denied and green for permitted.

2.2. Results

Our project was a general success, and all of our set goals were achieved! Below is a table summarizing our updated project requirements. A new column showing the final outcome has been added. If interested, the reader is invited to refer to Appendix D for Utilization Reports on both our FPGA Server and FPGA Client Hardware.

No	Requirement/ Features	Priority	Acceptance Criteria	Final Outcome
1	Data Matrix data should be correctly decoded by the FPGA	High	Verified that information on Data Matrix is correctly decoded by checking with source information.	Requirement met successfully
2	Database Parsing can be performed on the server using information received from the client	High	Server can successfully read the database using Unique Identifier (UID) sent from the client	Requirement met successfully
3	Information can be fetched from the Database when a match is found	High	Once a match of UID is found, server can successfully perform a fetch action and retrieve the correct result.	Requirement met successfully
4	Server can transfer the fetched data to the client via TCP/IP	High	Database Parsing IP retrieves the entire data entry from the database and sends it to the client for TLV decoding	Requirement met successfully
5	Client can receive the data sent from the server using TCP/IP	High	Correct data can be successfully transferred from the Server to Client	Requirement met successfully
6	Data received from the Server can be displayed on the Client	High	Data displayed matches the requested data from the server	Requirement met successfully
7	TCP/IP for data transmission between PC1 and FPGA1	High	Display correct information in the terminal	Requirement met successfully
8	VGA Display can show results received from the server, and display visually if access is granted or denied	Medium	Verified that information displayed matches the pre-defined user information and is in the correct position on the screen.	Requirement met successfully
9	TLV encoding is performed on the decoded data before sending via TCP/IP	Medium	TLV encoding is successfully performed on the data received from the Data Matrix	Requirement met successfully

No	Requirement/ Features	Priority	Acceptance Criteria	Final Outcome
10	Client can perform TLV decoding on the received data received from server	Medium	Decoded data matches encoded data by the server	Requirement met successfully
11	TLV: encoding and decoding of the information to be sent between FPGAs is consistent	High	Verified that the decoded message from the encoded message matches the original one.	Requirement met successfully

2.3. Future Work

2.3.1. Improving utilization of the Server Database:

While we have relied on a fixed addressing offset scheme for the server database, there are several alternatives that could be explored based on the needs of the implementation. Below is a breakdown of the different possible modes of operation for the Server Database with a dedicated 64MB space in the DDR2 memory. Current implementation is highlighted in the table below.

Address base (Hex)	Address Offset (Hex)	Addressing Scheme (Hex)	User data size	max no of users
80000000	100000	80100000	1MB	64
80000000	10000	80010000	64kB	1024
80000000	1000	80001000	4kB	16384
80000000	100	80000100	256B	262144

We selected the 4kB user data size implementation to demonstrate the possible uses of this system beyond the vaccine passport use case. However, if we were to use this system primarily as a vaccine passport implementation, then we could reduce the user data size to 256Bytes, which would comfortably fit the vaccine passport details of 128Bytes with some additional user data space. This would allow us to store the information for 262,144 users within a space of just 64MB!

2.3.2. Improving speed of GET and POST requests:

Currently the system uses a 4kB section of the DDR2 as a scratchpad memory block (starting at the base address with 0 offset). This is where MicroBlaze would write the received values or read from when the result has been fetched by the Database Parsing IP. However, the time to access the DDR2 memory is much longer compared to accessing a BRAM block. In our current implementation, due to challenges in the connectivity of the Database Parsing IP block, we were unable to have both a BRAM and the DDR memory be simultaneously connected to the AXI-Full Master. In the future, by adding another AXI-Full Master to the

Database Parsing IP, we could allow it to access a BRAM based scratchpad memory block. When writing new values or reading the fetched results, MicroBlaze would be able to read from this scratchpad location much faster as compared to reading from the DDR2 block.

2.3.3. Implementing QR code decoding:

For this implementation we chose to perform Data Matrix decoding on the client. While this sufficiently demonstrates the capability of our hardware, we could enhance this by moving on to performing QR code decoding, as QR code is much more popular and would allow this project to be useful in several other applications that already use QR codes.

2.3.4. Camera input:

Currently the image is pre-processed by PC1, and FPGA1 only receives the bitstream of the Data Matrix. However, if we have access to a camera, we can make it a real-time system that scans the Data Matrix without any involvement of the PC. We will need a camera interface to transfer the video data and a VDMA core to store it in memory. Also, the image processing can be implemented in MicroBlaze instead of PC1.

3. Project Schedule

A high-level overview of task allocation:

PC1/ Admin PC	
Data Matrix Generation (python script):	Xuening
TCP/IP data transfer (python script):	Xuening, Mustafa, Eduardo
Client FPGA (FPGA 1)	
Data Matrix Decode IP:	Guoxian
VGA Display IP:	Xuening
Block Integration:	Guoxian
TCP Client (FPGA-FPGA) & Server (PC-FPGA) (MicroBlaze)	Xuening
TLV Encoding & Decoding (MicroBlaze)	Eduardo
Server FPGA (FPGA 2)	
Database Parsing IP:	Mustafa
Block Integration:	Mustafa
TCP Server and Request handling (MicroBlaze)	Eduardo
TLV Encoding & Decoding (MicroBlaze)	Eduardo

3.1. Milestone #1: Research and initiation

Original Plan:

M1Task1: Eduardo: Design and document data encoding and decoding for TCP Server and Clients using TLVs

M1Task2: Xuening: Start on the VGA, finish at least displaying static images on the screen. Write a python script for generating QR codes for future test purposes.

M1Task3: Guoxian: Research on QR code decoding, test C/Python code for QR code decoding and write a document including the functional requirements, interfaces and the algorithm used in RTL design

M1Task4: Mustafa: Start by identifying a framework for the Database retrieval IP. Write a sample dataset to the DDR2 memory and confirm information is being stored appropriately (static database).

Actual Accomplishment (Details included in Appendix E):

In the first week of the project, we focused on initiating the project with background research, finalizing the communication format between the client and server, and finishing some basic implementation of the blocks. A comprehensive document was compiled to define the TLV encoding and decoding format and more research into QR code decoding was done. Framework for the database parsing and VGA display IPs were initiated.

After conducting exhaustive research into the QR code decoding methodology, we realized that it would be very complicated to implement it on FPGA within a short time. We switched from QR code to Data Matrix, which has a simpler format but similar functionality in storing information.

The challenges we faced during this milestone were mostly concerns regarding the implementation of the blocks we had proposed, whether we could finish them in time or if they would be too difficult to implement by ourselves.

3.2. Milestone #2: Basic functionality

Original Plan:

M2Task1: Eduardo: Python scripts for TLV encoding and decoding – integrate with TCP Server and client. This will be used to test FPGAs' client and server, and to modify the database.

M2Task2: Xuening: Continue working on the VGA implementation, including showing some texts (e.g. permission, personal information) on the screen once received response.

M2Task3: Guoxian: Design RTL code the QR code decoder implemented on FPGA.

M2Task4: Mustafa: Implement a mechanism to parse the data stored in the DDR2 when a request is received by FPGA2 and perform fetch if matching data is found.

Actual Accomplishment (Details included in Appendix E):

In the second milestone, we were mainly focussed on creating the IPs, including database parsing, VGA display, Data Matrix decoder and TLV implementation in MicroBlaze.

For client FPGA 1, we researched on the Data Matrix encode/decode library in python and started to rewrite a decoder on hardware. Additionally, we used VIOs to test if we could store some simple contents in BRAM and display it using VGA. For the server FPGA 2, simple database address parsing and data storing in DDR2 was completed using MicroBlaze. Furthermore, a python script/library to encode TLVs was implemented on PC.

The challenges of Data Matrix decoder partly came from dealing with the turning point of the characters in Data Matrix and some special blocks in it. Extra logic was added to handle these situations.

It is also worth mentioning that after we had done some research on TLV, we decided to move the TLV IP to a software implementation, since it is complicated to implement this in hardware, and it would be much more efficient to do it in software.

3.3. Milestone #3: Finish most units

Original Plan:

M3Task1: Eduardo: Write TLV encoding and decoding in C for the MicroBlaze, this will be used to test the database and as a backup resource in case we cannot get an IP to do it.

M3Task2: Xuening: Finalize the VGA part, including showing the information in an appealing manner. Start on PC - FPGA data transfer script for sending images to FPGA 1 (starting from using MicroBlaze and python script).

M3Task3: Guoxian: Build test environment of QR code decoding with software decoding as a reference model and start testing the design code. Start on RTL simulation and debugging

M3Task4: Mustafa: Perform testing and improve the parsing and fetching of information from the database. If possible, attempt to write new data that's received from client which is not currently present in the database (dynamic database)

Actual Accomplishment (Details included in Appendix E):

In Milestone 3, most of the individual work was close to being finalized. On the client side, we were close to finalizing the individual work on FPGA 1 such as Data Matrix decoder and VGA display IP. The Data Matrix IP was tested with more cases and modified to fit the FPGA1 design. Names and vaccine doses, as dynamic texts, can be displayed on the VGA. Besides, we finished a script for transforming Data Matrix images into binary matrices. Then we started with a preliminary integration of the hardware system on FPGA 1. The overall block diagram of FPGA 1 was discussed in this milestone. To ensure the hardware blocks work well to get information from MicroBlaze, we made a detailed plan on the FSM and AXI masters/ slaves control to make sure transactions were sent in correct order.

On the server side, TLV decoding was completed in python script, and we started to work on the encoding/decoding in C and implementing it in MicroBlaze. For the database, Initial tests at storing data at a sample UID offset were successful and we were working on creating an AXI compatible module for Memory Parsing that included both Master and Slave configurations. One challenge for the VGA display was that the background is too large to store on a BRAM, so we decided to remove the background.

3.4. Milestone #4: Initial integration and debugging

Original Plan:

M4Task1: Eduardo: Start IPs for TLV encoding/decoding

M4Task2: Xuening: Test the data transfer script, draft on a version of PC-FPGA communication only with the UART. Work together with Guoxian to complete the workflow in FPGA 1. Catch up with any delayed previous milestone.

M4Task3: Guoxian: Continue to debug and test the custom IP. Connect PC1 and FPGA1 with UART. Work together with Xuening to build up preliminary system on FPGA1.

M4Task4: Mustafa: Interface with MicroBlaze to ensure data fetch requests can be received and processed effectively with several use cases. Implement testbenches to validate IP functionality. Work with Eduardo to ensure requests coming to FPGA 2 can be fulfilled appropriately by database parsing IP core.

Actual Accomplishment (Details included in Appendix E):

This week was the busiest time of the term where all team members had several assignments and mid term tests. At this milestone, the FPGA 1 (client) side team had finalized the design for VGA display IP and Data Matrix decoding IP modules. We had also developed a FSM for controlling the states and signal transmission within FPGA 1 and were able to link all these components together into a block design. On the FPGA 2 (server) side, we had the TLV decoding module fully functional, and the Database Parsing IP was almost finalized. The database parsing IP contains both AXI Lite and AXI Full interfaces, with registers to write values and read status, a BRAM block implemented as scratchpad to store the user data and read and write data to a BRAM based database.

The major challenges encountered by the team were related to the time management and how to make best use of the AXI buses. Questions were raised on how to synchronize the signals and use the AXI Full buses for database parsing purposes, as there were challenges in reading from and writing to DDR.

3.5. Milestone #5: Final integration and debugging

Original Plan:

M5Task1: Xuening/Guoxian: Perform system level testing on FPGA 1 including checking the quality of communication between PC1 and FPGA 1, correctness of information decoded from the QR code and correctness of information displayed on VGA.

M5Task2: Eduardo/Mustafa: Begin system integration and testing of hardware and software on FPGA 2. Conduct extensive debugging activity during this week

Actual Accomplishment (Details included in Appendix E):

By this milestone, the client and the server side teams worked on integrating the individual blocks to work together along with the software code. For the FPGA 1 (client) side, the team fixed the problem of mixing clocks at different frequencies and reset signals for peripherals and interconnects. The AXI master and slave communications were established and tested to be working well in the first simulation with direct BRAM access from the software testing script. The PC could encode a user-input string with username and UID and send the generated data matrix in the format of bitstream to FPGA 1 to be written into the BRAM. On the FPGA 2 (server) side, the TLV encoding/decoding block worked with the Database Parsing IP and was tested by performing POST and GET requests. A TCP client script was finalized for testing the connection and communication between the Database and the python based TLV script run on the PC. The database stored the user information in the order of UID and successfully fetched the information as requested by the TCP client after performing TLV encoding/decoding during the communication.

As the FPGA-Net was down during the weekend of this milestone, the team had a hard time in testing whether the TCP/IP modules were functioning correctly. There was another challenge related to accessing DDR memory for the database. The instruction set memory sections of the TCP server program were occupying the DDR memory space, which was causing conflicts when the Database Parsing IP was trying to read/write to the DDR memory. Instead of using the DDR, a large BRAM was used to store the Database to avoid this issue. For the FPGA 1 (client) side, the team had some confusion relating to the use of ASCII numbers for the data matrix as the encoder automatically added 1 to the ASCII of all characters in the input string and this led to an offset in username and UID after decoding.

3.6. Milestone #6: Final Testing

Original Plan:

M6Task1: Eduardo/Xuening/Guoxian/Mustafa: Finalize full system integration, test overall functionality and connections PC-FPGA or FPGA-FPGA. Adjust functions and fix problems as needed for optimal system performance.

Actual Accomplishment (Details included in Appendix E):

At the final milestone of the project, the team had finalized all individual blocks and successfully completed the first pass of full system integration. A python script on the PC was able to accept an input image of a data matrix, transform it into a bitstream and send to FPGA 1. FPGA 1 could

then store the data matrix in BRAM and decode it. The decoded UID was sent to FPGA 2 to be searched in the database and the corresponding vaccination information was fetched. The fetched information was sent back to FPGA 1 to be displayed on a VGA display.

One of two major challenges during this milestone involved a synchronization problem on the FPGA 1 side where the last line of the data matrix in BRAM was not being read before the decoder started. This did not affect the correctness of decoding but lead to a missing line when the data matrix was displayed. We solved this problem by changing the initialization condition of the decoder enable signal. On the FPGA 2 side, we were still stuck on DDR access issue when the memory was shared for instruction set storage. This problem was later resolved by correcting issues with address mapping on the Hardware and Software side. We successfully partitioned the DDR memory space to be used by the database and to store runtime instruction set from MicroBlaze. Once the address mapping was corrected, we were able to access DDR2 to add entries to the database and read from it even while using all other functionalities of the system.

4. Description of the Blocks

4.1. Custom Hardware IP

4.1.1. VGA IP

The VGA is responsible for displaying all the essential information of the user, including the data matrix of the user (encoded by the name and the UID), the username, the number of doses that the person received and the permission on whether this person is allowed to access the place (i.e., whether this person has been fully vaccinated)

The design of the VGA block contains various frames or title to be displayed on the screen in pixels. The fixed titles, such as “name:”, “doses:”, are directly drawn and stored as coefficient files and stored in the BRAM. Other varying information, such as the name and the number of doses (the actual “number”) are displayed in terms of characters and digits. The 26 English characters (and some special characters) and 10 digits are stored in one coefficient file, and we used a lookup table to fetch the corresponding position of the character in memory according to the ASCII number. We also recorded the size of each character since some of the characters such as “m” and “n” takes more pixels along the x-axis than the rest. Keeping the size of the character allows us to calculate a dynamic length of pixels character strings during runtime such that the characters do not overlap or space unevenly.

The changing between different titles and different users’ names are controlled by an FSM which is also connected to the later data matrix decoding IP. There are four stages in total, corresponding to four different frames to be displayed on VGA, one for waiting for the user to scan, one totally blank page for decoding, one with the data matrix and username directly after the information has been decoded and one for showing all the essential information of the user’s entry permission. The VGA top module fetches the corresponding pixels from the memory according to the control signals.

The functionality of this block is verified by some simulation script for checking the correct synchronization signals for vertical and horizontal axis are raised (*Figure 10*) and an operation test with VIO to see if various information is correctly printed on the screen (*Figure 11*).

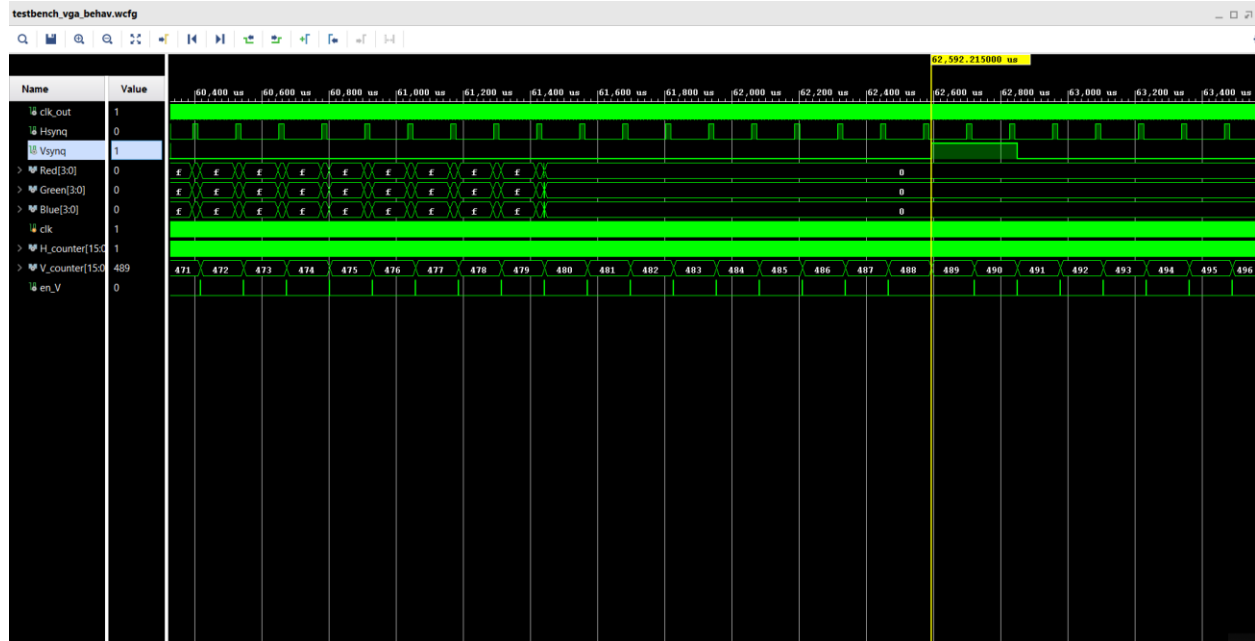


Figure 10: Waveform simulating the VGA operation at the moment when Vsync is raised



Figure 11: VGA screens at each stage (the decoding stage changed too fast to be captured)

4.1.2. Data Matrix Decoding IP

The Data Matrix Decoding IP in our project is mainly responsible for decoding the Name and UID from the Data Matrix from bitstream of Data Matrix. Then, we can use the UID to access the database for vaccine status.

The hardware of Data Matrix decoder uses the Python Data Matrix libraries [4] as references. The placement of message in Data Matrix is arranged in a complicated diagonal pattern. The starting point of the Data Matrix is (4,0) and the order of messages is shown in the red line in the figure. In the hardware IP, the cursor of the red line keeps moving through the red line throughout the entire decode process. Typically, it moves two blocks towards the bottom-left or the top-right. Whenever it goes outside the boundary, the cursor is disabled since it out-of-boundary blocks do not store information. Also, the direction of the red line is stored in a register. When the cursor hits the boundary, the direction changes, and the cursor moves to the next legal coordinate.

Typically, ASCII data is stored in the Data Matrix in L-shape tiles except for the one on the top-right corner. Finally, in the Data Matrix IP, we get the coordinate of each bit of a character and concatenate each bit as the decoded data.

To fit the Data Matrix into our design with less complexity, we decided to make a little change on the protocol. In the standard Data Matrix Decoder protocol, the ASCII data is encoded as ASCII value + 1. However, in our project, it is encoded as the actual ASCII value instead.

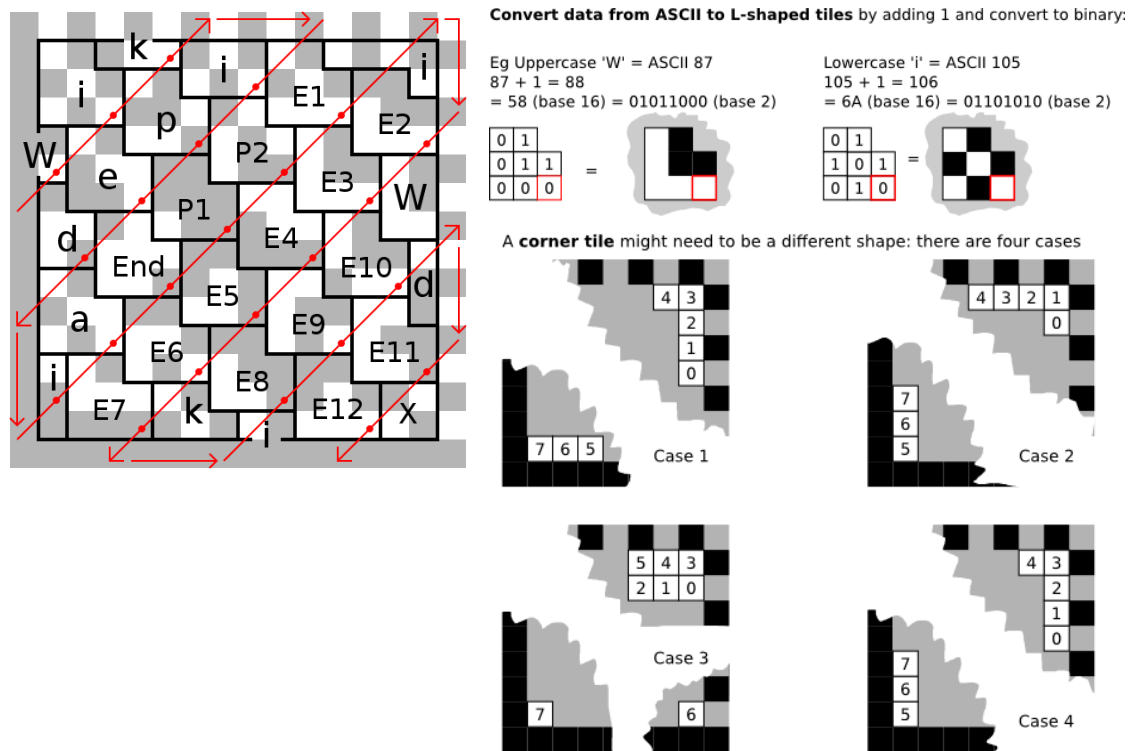


Figure 12: Placement of the message data in Data Matrix [1]

The functionality of the Data Matrix Decoder is verified by testbench. Encoded Data Matrices are used as the test vector. Apart from checking the waveform, we compare the decoded data with the encoded information. After the decoder is integrated with the VGA, decoded information can be shown on the screen for further testing.



Figure 13: Waveform simulating the Data Matrix Decoder

4.2. Database Parsing IP

The Database Parsing IP is implemented with a goal to accelerate the task of reading from and writing to the database. By using this IP, we can free up MicroBlaze to perform other tasks such as receiving and processing new requests, communicating to other clients via TCP/IP etc.

This IP consists of an AXI-Lite Slave with 5 x 32-bit registers and an AXI-Full Master. The first 3 slave registers are command registers and are written to by MicroBlaze, while the 4th and 5th registers are status registers, useful for determining the state of the IP.

Figures 14 and 15 provide a snapshot of the inner workings of this IP.

There are 2 basic modes of operations – POST (ADD) and GET (FETCH) that can be performed by this IP. A combination of these two can allow us to perform 2 more secondary operations UPDATE (FETCH -> modify -> ADD) and DELETE (zeros -> ADD)

Let us look at these operations in more detail.

1. POST (ADD) operation:

During the POST operation (See Figure 14), MicroBlaze performs the following tasks:

- i. Writes the information to be stored in the database onto the scratchpad location
- ii. Writes 0x5 to the CMD register,
- iii. Writes the UID to the UID register,
- iv. Writes 0x1 to the EXEC register to begin the transaction.

Once Microblaze has trigger the execution by writing to the EXEC register, the database parsing IP instructs the AXI-Full Master to read the data stored in the scratchpad location and then write it to the specific memory location offset that is determined by the UID.

Once this task is completed, the value of DONE register switches to 1, which lets MicroBlaze now that the Database Parsing IP has successfully moved the data onto the appropriate block in DDR memory.

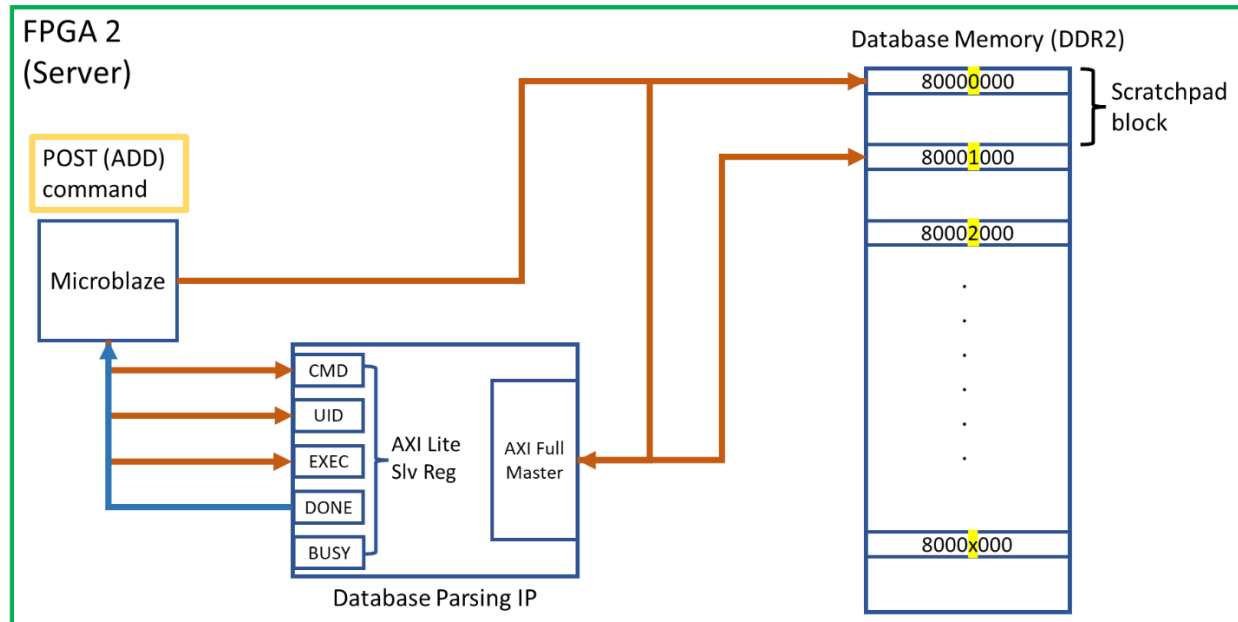


Figure 14: Database IP performing POST command

2. GET (FETCH) operation:

During the GET operation (See figure 15), MicroBlaze performs the following tasks:

- i. Writes 0x4 to the CMD register,
- ii. Writes the UID to the UID register,
- iii. Writes 0x1 to the EXEC register to begin the transaction.

Once Microblaze has trigger the execution by writing to the EXEC register, the database parsing IP instructs the AXI-Full Master to read the data stored in the specific memory location offset that is determined by the UID, and then write it to the scratchpad location.

Once this task is completed, the value of DONE register switches to 1, which lets MicroBlaze now that the Database Parsing IP has successfully moved the data onto the scratchpad location in DDR memory.

MicroBlaze then reads the information stored at the scratchpad location.

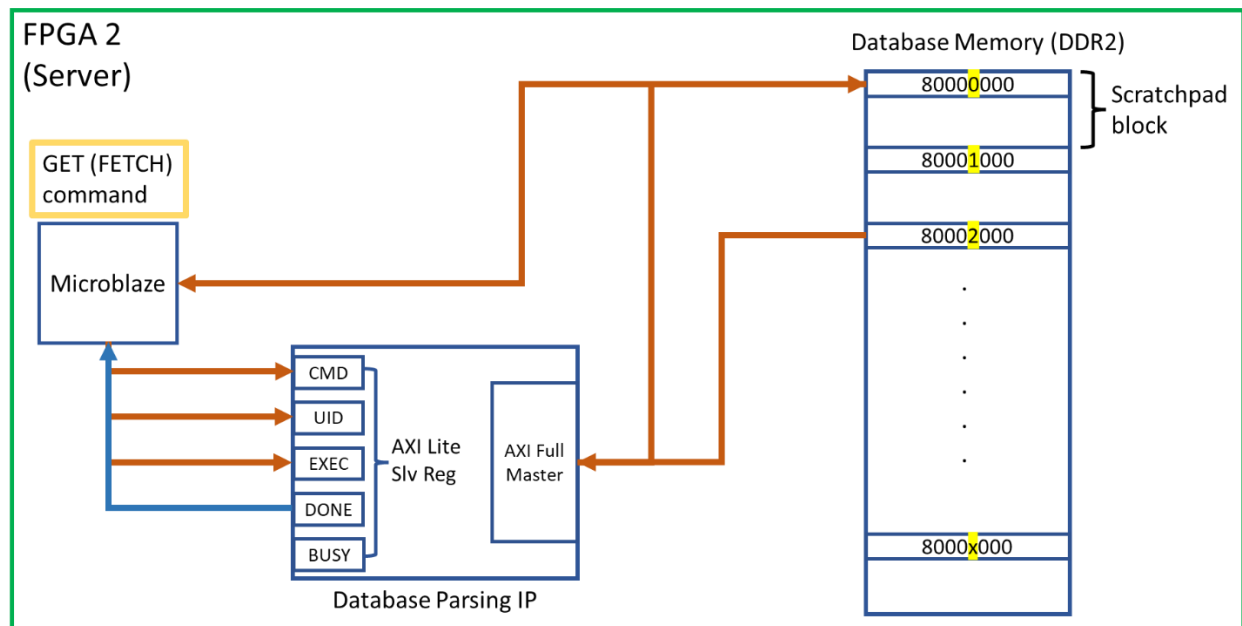


Figure 15: Database IP performing GET command

The Database Parsing IP was tested using AXI VIP (during simulation) and System ILA (during run-time). Several iterations were made to improve the overall functionality as a result of this debugging effort.

4.3. Important IP Blocks Used

Unless specifically indicated, we used the default settings for configuring the following IP blocks.

IP Used	Ver No.	Used By	Purpose
MicroBlaze	v11.0	Client & Server	MicroBlaze is a soft-core processor provided by Xilinx. All software for this project is run on MicroBlaze. The implementation is similar to the one created for course tutorial 5 and the Group Demo. It runs the TCP/IP connection code as well as TLV encoding and decoding, in addition to providing control signals for the custom IPs and other hardware peripherals.
Memory Interface Generator (MIG)	v4.2	Client & Server	Memory Interface Generator is a controller and a physical layer for interfacing to the DDR2 SDRAM on the Nexys 4 DDR board. This allows us to interface our custom IP with the DDR2 memory and read/write to it using the AXI Bus following AXI protocol
AXI EthernetLite	v3.0	Client & Server	AXI EthernetLite core provides an interface to Ethernet PHY-MII block. And allows a processor to communicate to the ethernet Physical Layer using an AXI protocol. For our project we use the AXI4LITE protocol and operating at speeds of 100Mbps. This IP is clocked at 100MHz.

IP Used	Ver No.	Used By	Purpose
AXI Interrupt Controller	v4.1	Client & Server	AXI Interrupt controller manages multiple interrupts from peripheral devices and combines them to be processed by the MicroBlaze core. It has an AXI-Lite interface and is useful to interrupt the MicroBlaze to service requests from Ethernet or UART peripherals
AXI Uartlite	v2.0	Client & Server	AXI Uartlite module allows MicroBlaze to communicate via UART by using the AXI interface. The UART module used in this project is similar to the one created for course tutorial 5 and the Group Demo. For this project it is used to print debug statements, messages received or sent via TLV as well as to identify when server or client connections have been established. This IP is clocked at 100MHz.
AXI BRAM Controller	v4.1	Client	AXI BRAM Controller is used to interface with a BRAM block and allow other AXI masters to access the memory location. For this project it is used to store the Data Matrix bitstream as well as the decoded UID.
AXI Interconnect	v2.1	Client & Server	AXI Interconnect IP allows connecting multiple AXI Masters to multiple AXI Slaves. In this project, for FPGA 1, we had 6 slaves and 7 masters and for FPGA 2, we had 6 slaves and 3 masters. Strategic use of AXI interconnect also allows us to separate different BUS connections to optimize performance. In FPGA 2, a dedicated interconnect was used to improve access performance to DDR2 memory
System ILA	v6.2	Client & Server	System ILA stands for Integrated Logic Analyzer. This IP allows us to monitor internal signals at runtime. It was used throughout this project for debugging purposes.

4.4. Integration of Hardware System

4.4.1. Integration of Data Matrix and VGA display

Apart from the Data Matrix Decoder and VGA display, the integrated IP includes an AXI slave registers to get control signals from the server FPGA2, an AXI master to send decoded information to the server FPGA2 and an AXI BRAM controller to store bitstream of Data Matrix.

The following figure shows the FSM and control flow of FPGA 1.

At IDLE state, the hardware system waits for the MicroBlaze in FPGA 1 to store the Data Matrix in BRAM. After that, it sends a decode_start signal to trigger the DECODE state and enable the Data Matrix decoder.

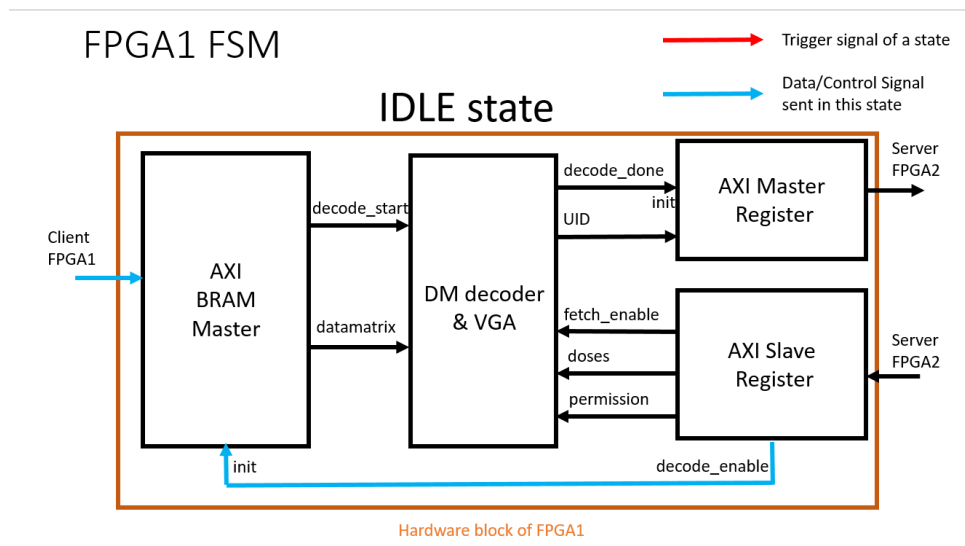


Figure 16: Data and control flow at IDLE state

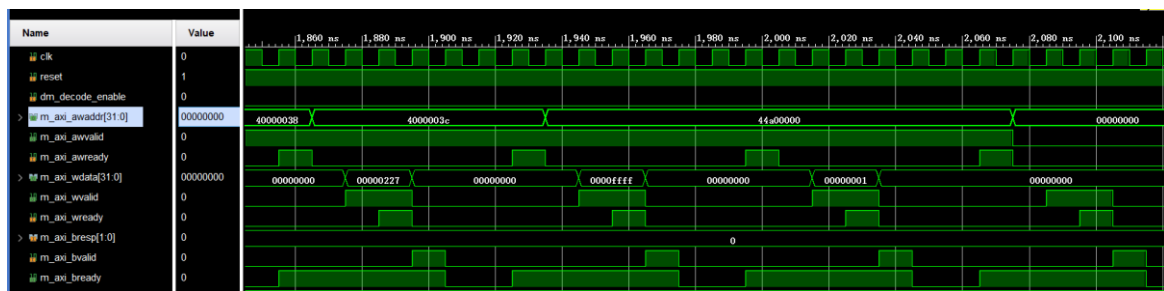


Figure 17: Simulation waveform of writing Data Matrix to BRAM

At DECODE state, Data Matrix is read out from BRAM and the Data Matrix Decoder decodes the name and UID from the input bitstream of Data Matrix.

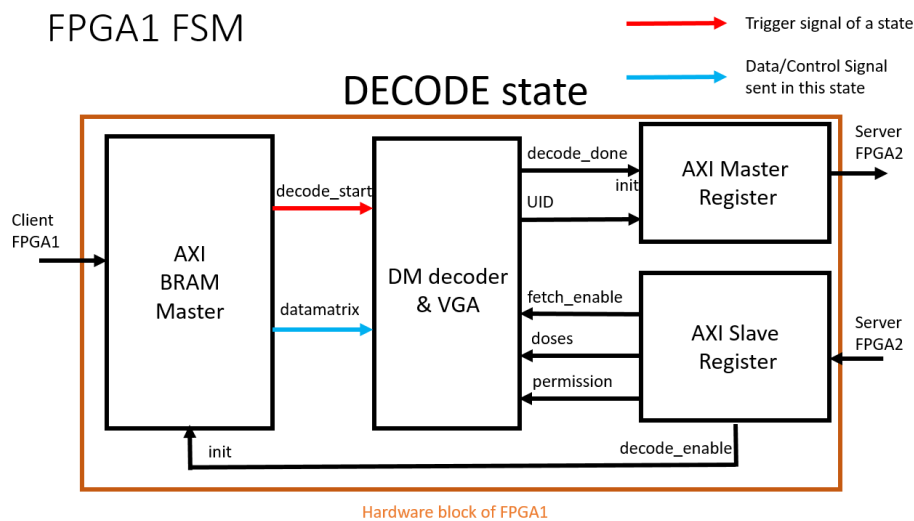


Figure 18: Data and control flow at DECODE state

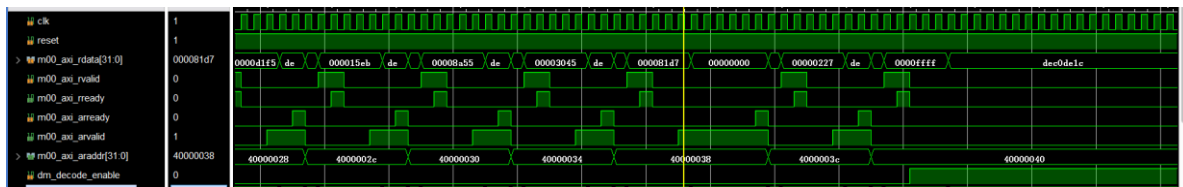


Figure 19: Simulation waveform of reading Data Matrix from BRAM

When the personal information is decoded, the Data Matrix Decoder IP sends decode_done signal and triggers SEND state. In this state, the MicroBlaze of FPGA 1 gets the UID and perform a FETCH operation to the server FPGA2 database.

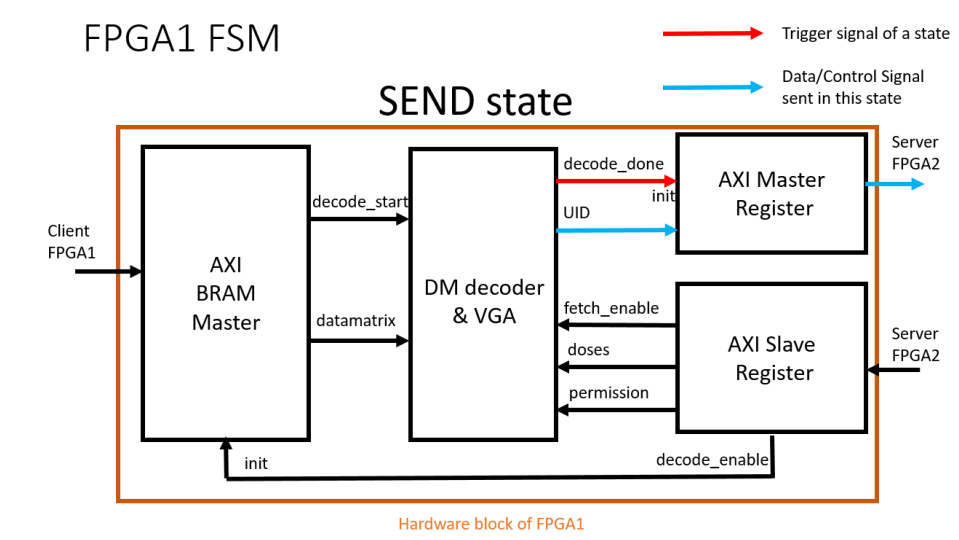


Figure 20: Data and control flow at SEND state

Finally, when the vaccine information is fetched from the server FPGA2, the system enters FETCH state and displays doses and permission information on VGA.

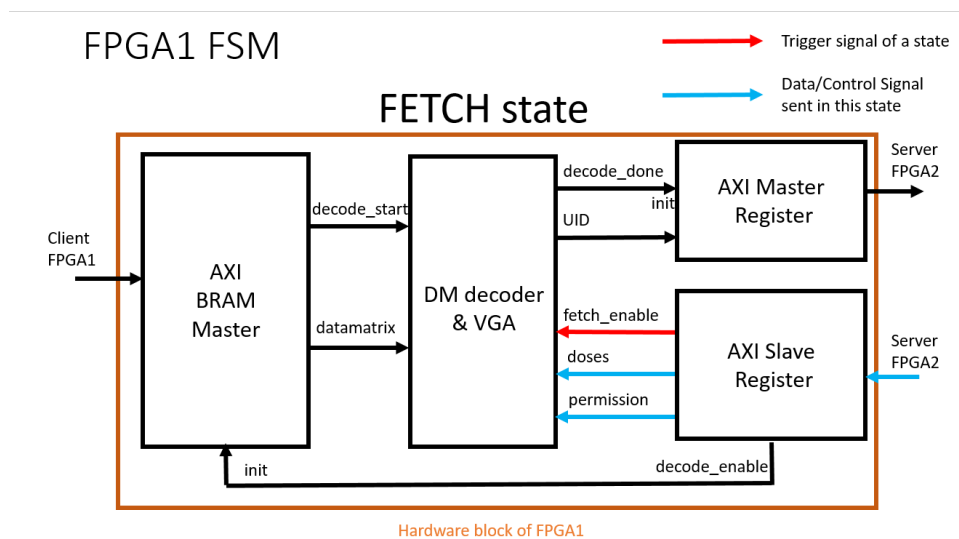


Figure 21: Data and control flow at FETCH state

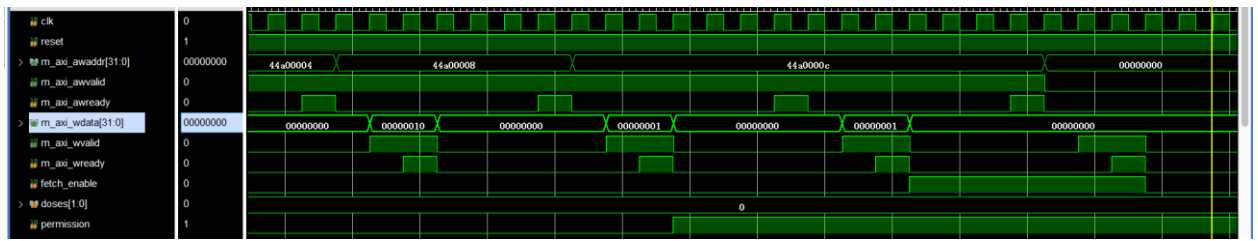


Figure 22: Simulation waveform of fetching Information from data matrix decoder

In order to simulate the integrated FPGA1 system, we build a system level testbench. AXI VIP is used to simulate the AXI bus. To verify the functionality of the whole system, we carefully check the AXI bus and signals mentioned in the control flow. Additionally, system ILA is applied to monitor these signals. Checking the behavior of these signals is an efficient way to debug the entire system.

4.5. MicroBlaze Software

4.5.1. FPGA Client (FPGA 1)

FPGA 1 (client) firstly acts as a server to receive the data matrix bitstream sent from the TCP client on PC 1. Before any transaction has taken place, the program resets all the numbers, including the non-zero UID, number of doses and permission, to zero. It then starts a server that is constantly waiting for input.

For a data matrix of 256 (16x16) pixels, the PC client sends them in a 16-bit pattern for 16 times. Each 16-bit pattern is then stored in the BRAM with increasing offset. The server echoes back acknowledgement to tell whether the correct information has been received. Upon receiving all 16 patterns, the program also initiates a signal telling the BRAM control module in the FPGA 1 top module to fetch the information in BRAM and start decoding.

The program then enters a loop polling over the BRAM area which contains the UID until it is non-zero. The UID is then transformed from ASCII to a binary number and encoded into a TLV GET request. At this moment, the program starts acting as a client and connecting with the TCP server that is held on FPGA 2 (server). A GET request is then sent across this channel. The client then receives a response containing the user data stored in the database. FPGA 1 decodes the TLV response and identifies the field that contains the user's vaccination status and number of vaccine doses received. Permission information (access permitted or denied) and number of vaccine doses is then sent through the bus to be displayed on the VGA. The system resets again when the next user data matrix is inputted and the same procedure is repeated.

4.5.2. FPGA Database Server (FPGA 2)

FPGA 2 acts as a standalone TCP server and handles incoming requests independently, regardless of the platform sending the request (i.e., if it is an FPGA client or a PC client). Once FPGA 2 receives a TLV formatted request, it decodes the request, parses it, and then handles the request appropriately.

If the received request is of type POST, the MicroBlaze software will write the encoded TLV user data to the scratchpad and indicate to the Database Parsing IP that it wishes to add a new Database entry with the given UID. A response TLV is then generated and sent back to the client, indicating the status of the request (i.e., if it was successful or not).

If the received request is of type GET, the MicroBlaze software indicates to the Database Parsing IP that it wishes to fetch the data corresponding to the request's UID. Once that entry is fetched and written to the scratchpad, the MicroBlaze software will read the data from the scratchpad and try to decode it to determine if it is a valid entry. A response TLV is then generated with the fetched user data and sent back to the client, indicating the status of the request (i.e., if it was successful or not).

During any point of this workflow, if an error occurs, the MicroBlaze software will gracefully handle that error and send a response to the client with the appropriate status code. Supported status codes can indicate if the request was successful, if the database entry was not found, or if the request data is invalid. Refer to Appendix A for TLV Communication Protocol.

4.5.3. TLV Encoding/Decoding (FPGA 1 and FPGA 2)

The codebase to encode and decode TLVs is shared between FPGA 1 and FPGA 2. This code builds upon the open-source EasyTLV library v1.0.0 [2]. EasyTLV is used to read TLV data from raw byte-arrays when decoding, and to write TLV data to raw byte-arrays when encoding.

The decoding process goes as follow: once the data has been fetched from the raw TLV buffer using EasyTLV, it is parsed appropriately and written to structs that represent either a Request, or a User (see *Figure 23*). Once the TLV data is stored in the appropriate struct, reading and modifying this data becomes very simple.

The TLV encoding process follows the opposite workflow. Once we have a correctly formatted DBUser struct or a DBRequest struct (see *Figure 23*), we convert it to the native format accepted by EasyTLV and then use EasyTLV to generate the TLV byte-array to be sent over TCP.

```

typedef struct {
    uint32_t UID;
    char firstName[MAX_NAME_LEN];
    char lastName[MAX_NAME_LEN];
    uint32_t DoB; //Date encoded in UNIX timestamp
    VaxStatus vaxStatus;

    //Stores the info on each vax dose
    int num_vax_doses;
    vaxDose_t vaxDoses[MAX_NUM_VAX_DOSES];
} DBUser;

typedef struct {
    RequestMethod method;
    RequestStatusCode sc;
    DBUser userData;
} DBRequest;

```

Figure 23: Structs to store TLV data in C

4.6. Python Script (PC 1)

```

*****
*           Data Matrix Client           *
*****
* Mode 1 image: input image              *
* Mode 2 input: input string             *
* Mode 3 encode: new datamatrix          *
* Mode 4 encode: Preload users           *
* Mode 5 encode: Read data from DB       *
* Mode 6 encode: Update entry in DB      *
*****

```

Figure 24: Screenshot of the user interface

The whole workflow of our project was controller through a single python script on a PC. The python script contains 6 modes as shown in Figure 24 with the following functions each:

- (1) **Image input:** This part of the code takes in a data matrix image of format “.png” (as generated by the encoder), translates it into pixels and eventually into a 256-bit bitstream. Each pixel is represented in RGB as read from the Pillow library [3] and since there are only black and white tiles in the data matrix, we simply transform them into bit 1 representing the black tile and bit 0 representing the white tiles. The long bitstream is then sliced into 16 16-bit patterns and sent across the TCP/IP connection to the server side. The information echoed from the server is checked to ensure that the correct information is delivered, otherwise the connection is abandoned and the bitstream needs to be resent (Figure 25).


```

def send_data_matrix(datas, shutdown_=_True):
    # send the patterns
    host_ip = input("Input client number: ")
    HOST = "1.1."+host_ip+'.2'
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((HOST, PORT))

        for idx, d in enumerate(datas):

            # send the message
            s.sendall(str.encode(d))
            data = s.recv(1024)

            # determine if the message has been sent
            if str(repr(data)) != "b'ACK'":
                print("Unsuccessful transmission")
                s.shutdown(1)
                s.close()

            # the exact output you're looking for:
            print("[%s] %d%% \r" % ('*idx, 100*idx/len(datas)), end_='\\r')

        if shutdown_:
            s.shutdown(1)
            s.close()
    print("[%s] %d%% Done" % ('*len(datas), 100*len(datas)/len(datas)))
    print()

```

Figure 25 Code snippet for sending the bitstream of data matrix

- (2) **String input:** this is a debugging and edge-case only function in our design. When users hold an invalid UID, they usually do not have a data matrix in hand. We ask the user to input their name and UID manually and then generate a data matrix at runtime to be sent to the server and checked. This function can be further developed to fit into the scenario when users forget to bring their data matrix, but more verification and data checking needs to be implemented to ensure the correctness of information fetched.

```

Input your first name: ECESStudent
Input your last name: ECE532
Input your birth date: (YYYY-MM-DD)2002-02-02
Input your id here: 5
Input number of doses you have received: 2
Input your vaccinated date: (YYYY-MM-DD) 2021-09-10
***** Vaccine Types *****
* 0: Unknown *
* 1: AstraZenca *
* 2: Bharat Biotech *
* 3: Janssen *
* 4: Moderna *
* 5: Pfizer-BioNTech *
* 6: Sinopharm BIBP *
* 7: Sinovac *
* 8: Other *
*****
Input your vaccine type: 4
Input your vaccinated date: (YYYY-MM-DD) 2021-10-10
***** Vaccine Types *****
* 0: Unknown *
* 1: AstraZenca *
* 2: Bharat Biotech *
* 3: Janssen *
* 4: Moderna *
* 5: Pfizer-BioNTech *
* 6: Sinopharm BIBP *
* 7: Sinovac *
* 8: Other *
*****
Input your vaccine type: 4

```

Figure 26: Information collected in the program

- (3) **New data matrix:** this is where we create a new user, store the person's information in the database and generate a new data matrix for him/her. It asks for a variety of information (as shown in Figure 26) from the user to ensure that all the information necessary for deciding the person's permission is collected. All the information is then encoded into a POST TLV request and sent to FPGA server to be stored in the database. A data matrix is also generated according to the name given and the UID with the image saved in .png format using the pyStrich library [4].
- (4) **Preload users:** in case an admin user would like to load a batch of user information into the database, we provide this preload function. The program reads from a .csv file with all the essential information listed. All the information is then compiled into a queue, sent to the database through POST requests and encoded into data matrix images.
- (5) **Read data from database:** the admin user may also wish to view if a person's information has been correctly stored in the database. This function provides a shortcut for the admin to view the user information without inputting a data matrix or having information sent to FPGA client. A TLV GET request is sent to the Database and the response is decoded and displayed on the terminal window.
- (6) **Update entry in database:** the user information needs to be updated when either the user received another vaccine shot or had incorrect information stored in the database. This function allows the admin user to check what has been stored in the database and change the entry if needed. This is done by sending a TLV GET request to the Database server, displaying the user's info on the terminal window, which is then modified by the admin user, and finally sending a POST request to the Database with the new user data. Notice that we only allow the admin user to change the number of doses, the dose types and the date of birth of the person. Information such as name and UID are not allowed to be changed as this will also alter the shape of the data matrix generated.

Overall, the python script in PC 1 is where we demonstrated the portability of our project. Knowing the IP address of the FPGA clients (which can be inputted at runtime) and the database FPGA, this script can be running at any place as needed.

4.6.1. Python TLV Encoding/Decoding

Every TCP message between our python script and the FPGA server (FPGA 2) was encoded with the designed TLV protocol (see Appendix A). The code to encode and decode TLVs builds upon the open-source uttlv library v0.6.0 [5]. uttlv is TLV parser library, and it was used within our program to read TLV data from raw byte-arrays when decoding, and to write TLV data to raw byte-arrays when encoding.

TLV decoding and encoding all happens within instances of classes representing either a Database Request or a Database User. Figure 27 illustrates all the members and methods of these classes.

The decoding process goes as follows: Instantiate an instance of the class of which your TLV belongs (DBUser or DBRequest) (see *Figure 27*) and call the TlvDecode() method with the raw

TLV byte-array and an input. The raw TLV byte-array will be parsed using uttlv and the members of the class will be set accordingly. Once the TLV data is stored in the class, reading, modifying, and printing this data becomes very simple.

The TLV encoding process follows the opposite workflow. Once we have a correctly formatted instance of the DBUser or a DBRequest class (see *Figure 27*), simply call the TlvEncode() method and the appropriate byte-array is formatted using uttlv and returned to the user.

```
class DBUser:
    def __init__(self, firstName = "", lastName = "", DoB = datetime.datetime(1971, 1, 1), vaxStatus = 0, UID = 0, vaxDoses = []):
        self.UID = UID
        self.firstName = firstName
        self.lastName = lastName
        self.DoB = DoB
        self.vaxStatus = vaxStatus
        self.vaxDoses = vaxDoses

    def TlvEncode(self): ...

    def TlvDecode(self, tlv_array): ...

    def print(self): ...

class DBRequest:
    def __init__(self, requestType = requestType.GET, statusCode = 0, user = DBUser()):
        self.requestType = requestType
        self.statusCode = statusCode
        self.userData = user

    def TlvEncode(self): ...

    def TlvDecode(self, tlv_array): ...

    def print(self): ...
```

Figure 27: Python Classes Used to Encode and Decode TLVs

5. Description of Design Tree

5.1. How to Use

(All files are accessible on GitHub: <https://github.com/dudublad/ECE532>)

Before opening any of the files below, go to the command prompt and:

- (1) Enter “ipconfig” to get the IP address, especially the station number, suppose that you are at machine X for the FPGA client and machine Y for the FPGA server
- (2) Install the following python libraries using pip:
 - PyStrich (pip install PyStrich)
 - Uttlv (pip install uttlv)
 - Numpy (pip install numpy)
 - pillow(pip install pillow)

FPGA client:

- (1) Download the file named Vivado_Projects/FPGA_1, make sure that the ip_repo folder is in the same folder as the dm_server folder, also include the sdk file in XilinxSDK/FPGA1 in the dm_server file.
- (2) Go to the dm_server folder and find demo_group.xpr, double-click to open it.
- (3) Once Vivado is opened, choose File -> Launch SDK
- (4) Choose Xilinx -> Program FPGA -> Program
- (5) Open the dm_server folder and go to main.c
- (6) In main.c, change the following information:
 - a) SRC_MAC_ADDR {0x00, 0x0a, 0x35, 0x00, 0x00, 0xX},
 - b) SRC_IP4_ADDR "1.1.X.2"
- (7) Also in main.c, change DEST_IP4_ADDR to "1.1.Y.2", where Y is the station number of the FPGA server
- (8) Go to SDK terminal -> the green "+" -> connect to COM6 at Baud Rate 9600
- (9) Choose Run -> Run Configurations -> Run
- (10) If you are physically at the lab, switch the monitor to VGA mode, otherwise use a webcam test website to view what is displayed on VGA.

FPGA server:

1. Download all the files in the XilinxSDK/FPGA2 directory
2. Open all the three projects in that directory Xilinx SDK 2018.3
3. Open the file Db_Test_15_SW/src/setup.h
 - a. Modify the definition MACHINE_NUM to reflect the number of your station in the FPGA-Net
 - b. Modify the definition MACHINE_NUM_HEX to reflect the number of your station in hex format. Number should just match the decimal version. E.g., if your machine number is 11, set MACHINE_NUM_HEX to 0x11.
4. Choose Xilinx -> Program FPGA -> Program
 - a. Program the FPGA with the bitstream T5_wrapper.bit
5. To enable logs, go to SDK terminal -> the green "+" -> connect to COM6 at Baud Rate 9600
6. Choose Run -> Run Configurations -> Run

PC:

- (1) Go to the Scripts folder and make sure there is a folder called dm_images inside
- (2) Right click on the file "definitions.py" -> Edit with IDLE -> Edit with IDLE 3.7 (64-bit)
- (3) Change the first line to fpga2_ip_addr = "1.1.Y.2"
- (4) Close IDLE and double click on the "main_script.py" file
- (5) Follow the instructions shown on the terminal

5.2. Repository Structure (only major components included)

Scripts

- |-- **RequestTLV.py**: Classes used to encode/decode TLV requests and user data
- |-- **TLV_Decode.py**: TLV decoding example/test in Python
- |-- **TLV_Encode.py**: TLV encoding example/test in Python
- |-- **CSV_convert.py**: Take .csv file as an input to preload users
- |-- **database_cheatsheet.csv**: Example .csv format
- |-- **definitions.py**: Database IP address setup
- |-- **main_script.py**: Main user interface
- |-- **dm_images**: Folder for storing data matrix images
 - | |-- **dm_0.png**: Example image with naming convention: "dm_"+UID+".png"

Vivado_Projects

- |-- **FPGA_1**: FPGA client in the project
 - | |-- **dm_server**: hardware block design file for the FPGA client (server in terms of receiving data matrix)
 - | | |-- **bitstream**
 - | | | |-- **design_1_wrapper.bit**
 - | | | |-- **design_1_wrapper.ltx**
 - | | |-- **demo_group.cache**
 - | | |-- **demo_group.hw**
 - | | |-- **demo_group.runs**
 - | | |-- **demo_group.srcs**
 - | | |-- **demo_group.xpr**
 - | |-- **ip_repo**
 - | | |-- **fpga1_top_1.0**: integration of two custom IPs: VGA display and Data Matrix Decoder
 - | | | |-- **src**:
 - | | | | |-- **data_matrix_decoder.v**: Custom IP of Data Matrix Decoder
 - | | | | |-- **vga_control.v**: Custom IP of VGA display
 - | | | | |-- **fsm_control.v**: FSM of FPGA 1
 - | | |-- **fpga1_top_master_1.0**: Data and control signals transferred from Microblaze to hardware
 - | | |-- **fpga1_top_master_bram_1.0**: BRAM controller in FPGA 1 (store Data Matrix)
 - | | |-- **fpga1_top_slv_1.0**: Data and control signals transferred from hardware to Microblaze
- |-- **FPGA2**
 - | |-- **bitstreams**
 - | | |-- **DB_Test7_DDR_4.bit**

```

| | |-- T5_wrapper.hdf
| |-- DB_Server_Final
| | |-- DB_Test7_DDR_4.cache
| | |-- DB_Test7_DDR_4.hw
| | |-- DB_Test7_DDR_4.ip_user_files
| | |-- DB_Test7_DDR_4.ipdefs
| | |-- DB_Test7_DDR_4.srscs
| | |-- DB_Test7_DDR_4.xpr
| |-- ip_repo
| | |--DB_Parsing_IP_v1_0
|-- FPGA1_tb: System level testbench of FPGA1
| |-- proj_p3.xpr
XilinxSDK
|-- FPGA1
| |-- demo_group.sdk
| | |-- design_1_wrapper_hw_platform_0
| | |-- dm_server: containing TCP server & client with communication encoded by TLV
| | |-- dm_server_bsp
| | |-- design_1_wrapper.hdf
|-- FPGA2
| |-- DB_Test15_HW Xilinx SDK Hardware project for FPGA 2
| |-- DB_Test_15_SW Xilinx SDK Software project for FPGA 2
| | |-- setup.h Define server's station number and port
| | |-- server.c Main server functionality. This gets called directly from main.c
| | |-- server_helper.h Helper function for the server. Gets called by server.c
| | |-- server_helper.c
| | |-- request_handler.h Handles incoming TLV requests appropriately
| | |-- request_handler.c
| | |-- requestTLV.h Encoding/decoding of TLV requests and user data
| | |-- requestTLV.c
| | |-- easytlv.h EasyTLV library header [2]
| | |-- easytlv.c EasyTLV library source code [2]
| |-- DB_Test_15_SW_bsp Xilinx SDK Software bsp project for FPGA 2
docs
|-- Final_Demo_Presentation.pdf
|-- Final_Proposal_Presentation.pdf
|-- Final_Proposal_Writeup.pdf
|-- Mid_Project_Presentation.pdf

```

6. Tips and Tricks

6.1. Tips for the design process

Using DDR2 memory on Nexys 4 DDR board: This board has 128MB of DDR2 SDRAM memory available, when larger projects are implemented such as those using ethernet and other software functionality, MicroBlaze instruction set needed to run the program may become large enough that it won't fit in any BRAM block. When using DDR2 to store the instruction set,

heap and stack, make sure to use appropriate addressing scheme on any Hardware IP blocks that will ensure they do not overlap with memory space required by MicroBlaze. This will avoid a need to perform frustrating debugging to detect and correct this.

Vivado Synthesis and Implementation may 'optimize away' logic blocks: There is a possibility that once simulation shows your logic blocks to be working as expected, you may not see the correct operation during runtime. While it is not very common, if there are multiple always blocks where the same logic signal is changing, then Vivado may not show any errors regarding that during implementation but will simply 'optimize away' one of those always blocks. This could break your logic and cause issues at runtime. Again, its better to make sure this is corrected rather than spending a long time trying to debug this issue.

6.2. Tips for making block designs

Connection Automation feature: Pay attention to the drop-down options when using the Connection Automation feature in Vivado. When working with different clock signals or reset signals, there is a possibility that Connection Automation may not correctly identify where the signals should be connected. Sometimes the feature messes up the different clock signals and the interconnect_aresetn with peripheral_aresetn especially after the memory interface generator is added (then you have two Processor System Reset blocks). Make sure to manually trace the connections to confirm they are correct.

6.3. Tips for working with Vivado and Xilinx SDK on DESL machines

System may lag due to poor network connectivity: In essence, Be Patient! Remote desktop connections can be slow depending on your network bandwidth. However, sometimes connection issues in the lab could also be the culprit, make sure to reach out to the Instructor, TAs to keep them informed.

System reassignment at 12 midnight: Remember to save a copy of whatever you have finished to the My Document (W:) drive from your working (C:) drive when it is close to 12 am. Or you may lose any progress that you have made. A new machine is assigned every time you login, so files may not be accessible if they were stored in the C: drive of the local machine.

Use Linker script to partition memory space in SDK: During the software implementation, use the Linker script to manage the sizes of the Heap and Stack. Additionally, if you are using DDR2 memory for your project, then use the Linker script to partition the addressing so that locations accessed by Hardware IPs do not overlap with those accessed by MicroBlaze to store software instruction set. Remember to refresh/ regenerate linker script when any changes have been made to the hardware.

7. References

[1] "Data matrix," Wikipedia. [Online].

Available: https://en.wikipedia.org/wiki/Data_Matrix. [Accessed: 12-April-2022].

[2] phreaknik, "Phreaknik/easytlv: A Lightweight and simple to use TLV serializer/deserializer library.," *GitHub*, 2020. [Online]. Available: <https://github.com/phreaknik/easytlv>. [Accessed: Feb-2022].

[3] A. Clark, "Pillow," *readthedocs.io*. [Online]. Available: <https://pillow.readthedocs.io/en/stable/#>. [Accessed: Feb-2022].

[4] mmulqueen, "Mmulqueen/pystrich: Pystrich is a python 3 module to generate 1D and 2D barcodes," *GitHub*, 26-Jul-2015. [Online]. Available: <https://github.com/mmulqueen/pyStrich>. [Accessed: Mar-2022].

[5] ustropo, "USTROPO/UTTLV: Python Library for TLV objects," *GitHub*, 10-Jul-2021. [Online]. Available: <https://github.com/ustropo/uttlv>. [Accessed: Feb-2022].

8. Appendices

8.1. Appendix A: TLV Communication Protocol

TLVs use the following convention:

Type	Length	Value
1 Byte	1 Byte	Length Bytes

Message Header

The header is used to indicate the intent and status of the request. It has type **0x01**. The header TLV is formatted as follows

Type	Length	Value	
0x01	2 (1 Byte Status Code, 1 Byte Request Type)	Status Code (1B)	Request Type (1B)

Request Types

Request Type	Encoding	Description
GET	1	Get data entry from the database
POST	2	Create a new data entry
Response	5	Server Response message

Status Codes

Status Code	Status
0	Success
1	Unauthorized
2	Not Found
3	Invalid

Body

The body is used to pass in the message's payload data and has type **0x02**. The body TLV is formatted as follows

Type	Length	Value
0x02	Body length bytes	Body TLV (length bytes)

Body TLV

Type	Length (Bytes)	Value
0x00	8	UID
0x01	Length of First Name	First Name
0x02	Length of Last Name	Last Name
0x03	4	Date of Birth
0x04	1	Vaccination Status
0x05	1	First Dose Vaccine Type
0x06	4	First Dose Date
0x07	1	Second Dose Vaccine Type
0x08	4	Second Dose Date
0x09	1	Third Dose Vaccine Type
0x0a	4	Third Dose Date

Example**Successful POST Request**

Say we want to add the following entry to the database:

	First Name	Last Name	DoB	Vaccination Status
Value (Native)	Novak	Djokovic	22/05/1987	1 (Unvaccinated)
Value (HEX)	4E 6F76 6163	446A 6F6B 6F76 6963	B2D7 0x20B3CD40	01

Request:

01 02 0002 02 1a 01 05 4E6F766163 02 08 446A6F6B6F766963 03 04 20B3CD40 04 01 01

The request gets processed, and an entry is added to the database with **UID 2**

Response:

01 02 0005 02 06 00 04 00000002

8.2. Appendix B: Database Entry Information

For each user (entry) in our database, we will store the following information

Field	Data Type	Size (Bytes)	Description
UID	Unsigned Integer (unique)	4	A unique identifier for the database entry. This might not be stored directly into the database and just be calculated based on a hash function.
First Name	String	1-32	User's First Name
Last Name	String	1-32	User's Last Name
Date of Birth	Date*	4	User's Date of Birth
Vaccination Status	Enum	1	User's vaccination status encoded as an Enum with the following format: 1. Unvaccinated 2. Partially vaccinated 3. Fully vaccinated
First Dose Vaccine Type (optional)	Enum	1	User's first dose vaccine type encoded as an Enum with the following format: 1. AstraZeneca 2. Bharat Biotech 3. Janssen/Johnson & Johnson 4. Moderna 5. Pfizer-BioNTech 6. Sinopharm BIBP 7. Sinovac 8. Other
First Dose Date (optional)	Date*	4	Date of first vaccine
Second Dose Vaccine Type (optional)	Enum	1	
Second Dose Date (optional)	Date*	4	
Third Dose Vaccine Type (optional)	Enum	1	
Third Dose Date (optional)	Date*	4	

*: Dates are encoded using POSIX timestamps

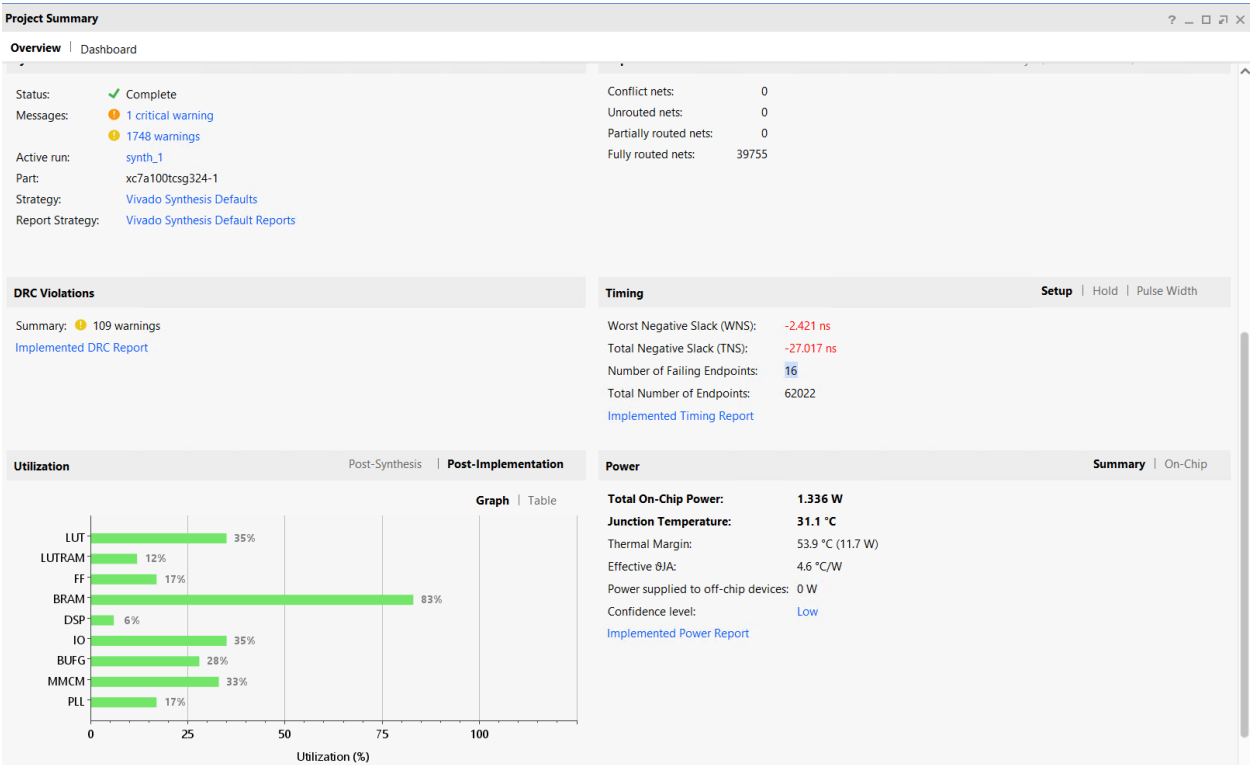
8.3. Appendix C: Proposed Functional Requirements

The table below describes the functional requirements for our project as presented in our Project Proposal:

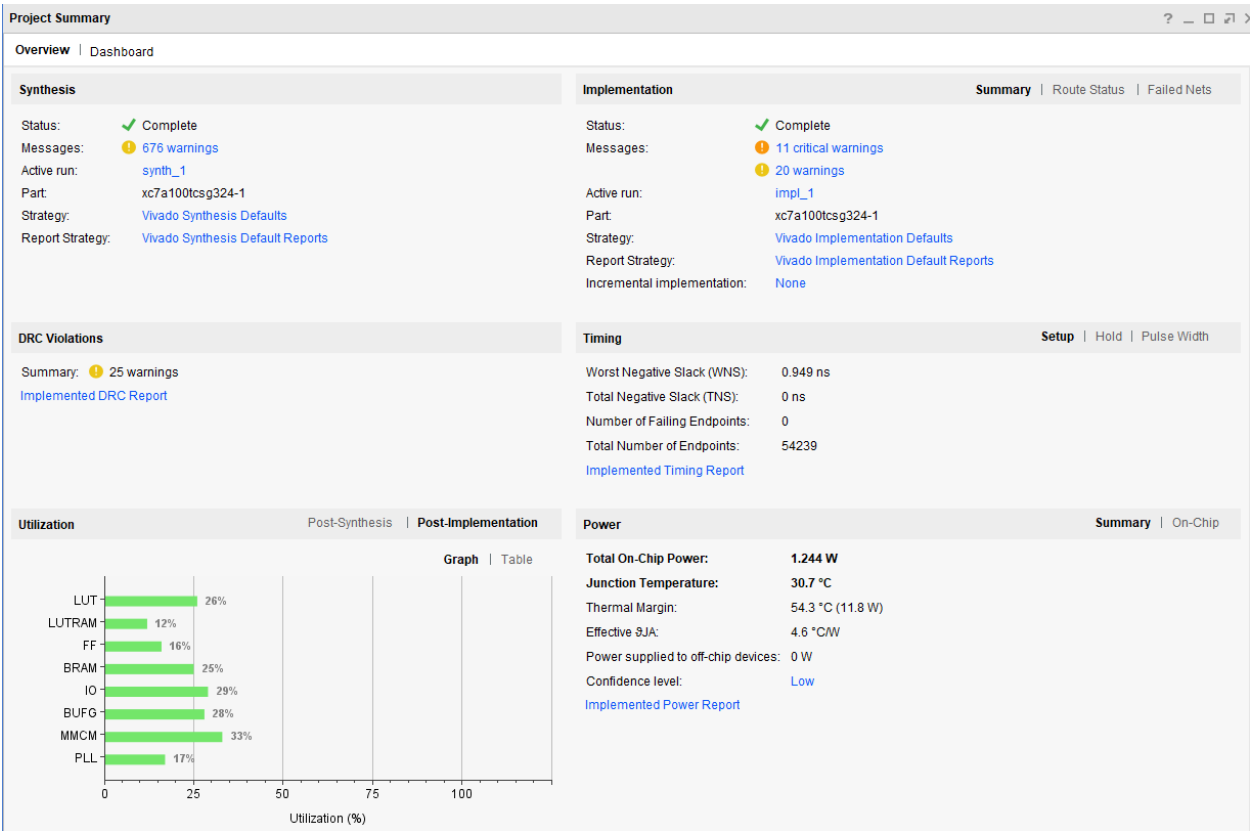
Sl no	Requirement/ Features	Priority	Acceptance Criteria
1	QR Code data should be correctly decoded by the FPGA	High	Verified that information on QR code is correctly decoded by checking with source information.
2	Database Parsing can be performed on the server using information received from the client	High	Server can successfully read the database using Unique Identifier (UID) sent from the client
3	Information can be fetched from the Database when a match is found	High	Once a match of UID is found, server can successfully perform a fetch action and retrieve the correct result.
4	Server can transfer the fetched data to the client via TCP/IP	High	Database Parsing IP retrieves the entire data entry from the database and sends it to the client for TLV decoding
5	Client can receive the data sent from the server using TCP/IP	High	Correct data can be successfully transferred from the Server to Client
6	Data received from the Server can be displayed on the Client	High	Data displayed matches the requested data from the server
7	UART for data transmission between PC1 and FPGA1	High	Display correct information in the terminal
8	VGA Display can show results received from the server, and display visually if access is granted or denied	Medium	Verified that information displayed matches the pre-defined user information and is in the correct position on the screen.
9	TLV encoding is performed on the decoded data before sending via TCP/IP	Medium	TLV encoding is successfully performed on the data received from the QR code
10	Client can perform TLV decoding on the received data received from server	Medium	Decoded data matches encoded data by the server
11	TLV: encoding and decoding of the information to be sent between FPGAs is consistent	If (9) is implemented: High, otherwise, medium	Verified that the decoded message from the encoded message matches the original one.
12	AXI GPIO connected to LEDs showing the access status	Low	Verify that the LED shows the correct vaccine status

8.4. Appendix D: Utilization Reports

FPGA 1 (Client)



FPGA 2 (Server)



8.5. Appendix E: Collection of Milestone Progress Reports

Milestone 1:

Name	Proposed Milestone	Progress Made	Challenges	Questions/Comments
Eduardo	Design and document data encoding and decoding for TCP Server and Clients using TLVs	Designed and documented the encoding of messages sent through our network. Work can be found here .	Manually writing TLVs is tedious Doing TLV encoding/decoding in Hardware may be tough. Will most likely only be done in software.	Any recommendation on how to make a better protocol? We could have messages of fixed size, but these would be way larger. Should we value simplicity over space?
Guoxian	Research on QR code decoding, test C/Python code for QR code decoding and write a document including the functional requirements, interfaces and the algorithm used in RTL design	1. Research on QR code decode algorithms Image acquisition: camera interfaces capturing image signals from camera Image processing: image-greying, enhancement (denoise, Contrast stretch), binarization (Otsu thresholding), tilt correction (Hough transform) Image translation: Localization, decoding and error-correction. 2. Find a QR code decode library (quirc) that is suitable to be implemented in Microblaze. Write a test demo to verify the source library code for Microblaze. It can scan QR code from the camera of the laptop and correctly decode it. Github library: https://github.com/dlbeer/quirc	Decoding QR code in hardware is rather complicated. It is unlikely that I can make it in approximately three weeks.	The decode part should be initially implemented in Microblaze software. Image-greying, median filter and thresholding in hardware while leaving other algorithms in Microblaze.
Xuening	Start on the VGA, finish at least displaying static images on the screen. Write a python script for generating	VGA: <ul style="list-style-type: none"> - Able to display static images on the screen - Verified by simulation Python Script: <ul style="list-style-type: none"> - Able to generate QR code with input from terminal or read from file - Able to recognize an uploaded QR code and display information Others: Created a document for planning how to display text on VGA	For the text display part in the next milestone, it would involve more memory and is expected to take more time to implement.	The current VGA version has resolution 480x640 with clock speed of 25MHz. I am thinking of updating this to 1080x1024 with clock speed of 108MHz if I have time on this.

Name	Proposed Milestone	Progress Made	Challenges	Questions/ Comments
	QR codes for future test purposes.			
Mustafa	Start by identifying a framework for the Database retrieval IP. Write a sample dataset to the DDR2 memory and confirm information is being stored appropriately (static database).	<p>Framework:</p> <ul style="list-style-type: none"> - 2 approaches were identified for Initial data load. Either via UART (1) or Ethernet (2) - for client requests use Database Retrieval IP <p>Sample Database Implementation: Working on Tutorial. Facing Bugs in accessing MIG in Xilinx (see challenges)</p>	<p>Memory Retrieval in Hardware will be a challenge when using TLV encoding. Once Static Database is ready, testing will be done in S/W before H/W implementation</p> <p>Xilinx Crashes when using MIG IP. No resolution found yet.</p>	<p>For pre-loading data into DDR2, should a DMA be used between Ethernet/UART and MIG? I am working with a tutorial to learn about this.</p> <p>Are there any examples of previous simple database implementations in FPGA that can be used for guidance?</p>

Milestone 2:

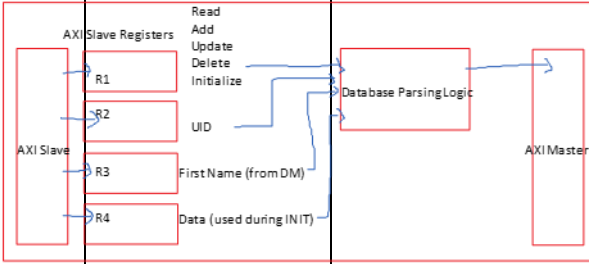
Name	Proposed Milestone	Progress Made	Challenges	Questions/ Comments
Eduardo	Python scripts for TLV encoding and decoding – integrate with TCP Server and client. This will be used to test FPGAs' client and server, and to modify the database.	<p>Made a python script/library that encodes TLVs.</p> <ul style="list-style-type: none"> • Script user can create TLVs for any type of request through a DBRequest class • Script user can create TLVs to represent database users through a DBUser class <p>Decoding is not yet complete but should follow soon. This should not take too long, as all classes are already implemented.</p>	<ul style="list-style-type: none"> - Required quite a bit of time to integrate all classes. This was not due to difficulty, but rather a lot "tedious" work, e.g. had to "hardcode" many constant values. - Busy with other schoolwork like presentation, assignment 1 and workload from other courses 	<p>I am using a python library to handle TLVs. Do I need to reference this anywhere? If so, how/where?</p> <p>I've decided to use POSIX timestamps to record dates, instead of our own method. This requires 4 bytes per date, instead of 2, but makes handling dates easier.</p>

Name	Proposed Milestone	Progress Made	Challenges	Questions/ Comments
Guoxian	Design RTL code the QR code decoder implemented on FPGA.	<ul style="list-style-type: none"> - Research on DataMatrix(DM) and run the DM encode/decode Python library in PC. - Design and debug custom IP of DM Decoding and write a document about its interfaces. - Write a testbench of the DM Decoding IP in one special case (12 characters, 14 x 14 DM) and verify the correctness of data. 	How are the DM decode IP connected in the system? Where to store the DM and how to control DM access?	Changing our plan from decoding QR code into decoding DataMatrix.
Xuening	Continue working on the VGA implementation, including showing some texts on the screen once received response.	<ul style="list-style-type: none"> - Designed VGA background screens to show either waiting for a data matrix input or displaying some personal information - Stored all the background or text components in the BRAM and allow buffering of contents to ensure smooth transitioning between frames - Able to display some simple dynamic contents (e.g., number of doses) subject to input signal (currently controlled by VIO) 	The VGA backgrounds and frames take up a large portion of the BRAM memory space. Attention needs to be paid during the integration of FPGA 1 to avoid exceeding memory limit.	The current visualization contains only the name, number of doses taken and the access permission. More contents can be added to the screen, according to what is sent over the two FPGAs.
Mustafa	Implement a mechanism to parse the data stored in the DDR2 when a request is received by FPGA2 and perform fetch if matching data is found.	<ul style="list-style-type: none"> - Storing sample data in DDR2 and performing a loopback was successfully completed using MicroBlaze late last week - This week, parsing through more than 1 entries and performing fetch data is work in progress - Based on last week discussions, I spent time this week looking into a simple hash table implementation to perform this - Plan is to start by using a Block RAM before eventually moving onto DDR2 	<ul style="list-style-type: none"> - Reviewing Hash Table examples on GitHub. Hardware Implementation is not trivial - Multiple due dates this week. Milestone 2, Presentation and Assignment 1 + other courses	- Possible Risk: If Hash Table H/W IP implementation is not successful, what alternate solutions can we use?

Milestone 3:

Name	Proposed Milestone	Progress Made	Challenges	Questions/ Comments
Eduardo	Write TLV encoding and decoding in C for the MicroBlaze, this will be used	<p>Completed TLV decoding in python script</p> <p>Stated implementing TLV encoding/decoding methods</p>	Testing C code in the Microblaze through the VM will be tedious. I will test the TLV	Q: Is it viable to have full Vivado/SDK projects added to our git repo? Or should we just add source files?

Name	Proposed Milestone	Progress Made	Challenges	Questions/ Comments
	to test the database and as a backup resource in case we cannot get an IP to do it.	in C that will run on the Microblaze Setup github repository for the project	functionality on my personal computer and then pass that code onto the Microblaze. Might have formatting issues.	Comment: More work will be done on Wednesday morning in the lab before the TA meeting. I want to start integrating TLV functionality into the Microblaze server and client.
Guoxian	Build test environment of QR code decoding with software decoding as a reference model and start testing the design code. Start on RTL simulation and debugging	<ul style="list-style-type: none"> - Test Datamatrix decoding IP with more cases. - Modify Datamatrix IP to fit FPGA1 design - Start working on VGA display of Datamatrix 	Datamatrix decode IP might not work for UID other than 12 bytes.	Comment: We use 12 bytes Datamatrix for UID in our project, the first 10 bytes for personal information (eg. names) and last 2 bytes for Database Address. Maybe Datamatrix image can be generated in Verilog so that we can save RAM resources
Xuening	Finalize the VGA part, including showing the information in an appealing manner. Start on PC - FPGA data transfer script for sending images to FPGA 1 (starting from using Microblaze and python script).	<ul style="list-style-type: none"> -Finished a script for transforming data matrix images into binary matrices and use TCP/IP) to send the matrix bitstreams to the FPGA. -Added names as dynamic texts to the VGA and changed the original background 	Sometimes it could be hard to find an available DESL A station. The background I prepared for VGA is too large to store on BRAM and I constantly hit the limit of the board. I removed the background now.	We need to add some python libraries to the DESL machines. Are we allowed to do so? Otherwise, I should look for some other ways to run the script on PC, such as Google Colab.

Name	Proposed Milestone	Progress Made	Challenges	Questions/ Comments
Mustafa	Perform testing and improve the parsing and fetching of information from the database. If possible, attempt to write new data that's received from client which is not currently present in the database (dynamic database)	<p>Based on last week's discussion, I have determined that we can use 1GB of space on DDR2 when using offset of 8000_0000. Initial tests at storing data at this offset are successful (used modified version of Tutorial 4).</p> <p>Currently working on creating an AXI Module (Master port and slave port) for Memory Parsing. See below Image</p>	<p>Following the methodology from Assignment 1. No major challenges yet. Real test will be when I complete the logic and start testing. Possibility of encountering errors.</p>  <p>The diagram shows an AXI Slave interface with four registers: R1, R2, R3, and R4. R1 is labeled 'AXI Slave Registers'. R2 is labeled 'AXI Slave'. R3 is labeled 'First Name (from DM)'. R4 is labeled 'Data (used during INIT)'. The registers are connected to a 'Database Parsing Logic' block. The logic block has inputs for 'Read', 'Add', 'Update', 'Delete', and 'Initialize'. The logic block is connected to an 'AXI Master' block. The logic block also has a 'UID' input and a 'First Name (from DM)' input. The logic block outputs to the 'AXI Master' block.</p>	<p>Q: Does the logic presented here in the diagram look correct?</p> <p>If not, any advice on how I could do this more efficiently? or possible issues that might come up due to this approach?</p>

Milestone 4:

Name	Proposed Milestone	Progress Made	Challenges	Questions/ Comments
Eduardo	Start IPs for TLV encoding/decoding	<p>Finished implementing TLV decoding in C, integration into microblaze should be easy.</p> <p>TLV encoding will also follow naturally from the current progress.</p>	<p>Very busy week with midterms and assignments</p> <p>TLV encoding and decoding is still not completed in software, I find it unlikely that we will have time to implement it in hardware.</p> <p>Implementing in hardware was a best-case scenario for us so we are ok with that change.</p>	<p>More work will be done on Wednesday morning prior to our meeting time. Want to start integrating the TLV messages with Microblaze and python clients and servers.</p>
Guoxian	Continue to debug and test the custom IP. Connect PC1 and FPGA1 with UART. Work together with Xuening to build up preliminary system on FPGA1.	<p>Integrate VGA, DM decoder and FSM to the HDL top level of FPGA1. Started working on the block design of FPGA1.</p>	<p>A little busy with some assignments and project due this week.</p> <p>A little confused about how to connect out top level to block design.</p>	<p>Use AXI master for monitoring the enable signal of HDL top and BRAM for storing datamatrix. Synchronization might be a problem.</p>
Xuening	Test the data transfer script, draft on a version	Finalized the VGA displays and managed the changing	It has been busy weeks for me, with	The client side of the project is functioning with

Name	Proposed Milestone	Progress Made	Challenges	Questions/ Comments
	of PC-FPGA communication only with the UART. Work together with Guoxian to complete the workflow in FPGA 1. Catch up with any delayed previous milestone.	of pages with an FSM controlled by signals from Microblaze and the decoder. Finalized TCP/IP connection between PC1 and FPGA 1 with data matrix stored into BRAM. Worked with Guoxian to finalize design of FPGA 1, PC 1	assignments and midterms clustered. I started on searching for how to establish connection between PC and FPGA with UART but have not had a script ready.	TCP/IP connections. There are some modifications needed to make the currently manually controlled signals to automatically generated ones.
Mustafa	Interface with MicroBlaze to ensure data fetch requests can be received and processed effectively with several use cases. Implement testbenches to validate IP functionality. Work with Eduardo to ensure requests coming to FPGA 2 can be fulfilled appropriately by database parsing IP core.	Working on implementing the Database Parsing IP in a custom package (1 AXI S-Lite, 1 AXI S-Full, 1 AXI M-Full) AXI S-Lite provides 6 registers to write values to and read status from it. AXI S-Full provides a BRAM block to store the 128 Byte data. It also allows transfer in Burst Mode. AXI M-Full interfaces with the Interconnect to write to and read from the DDR block in Burst mode. Still working on: (1) connection b/w sub blocks, (2) signalling mechanism using registers, (3) state machine for different modes of operation	AXI is tricky to work with. Attempted to do the 32 separate registers in AXI-S Lite, couldn't figure out easy way to run through consecutive writes/ reads without breaking the handshaking. Explored AXI Full Burst mode using online resources. Attempting to use AXI Full to read/write the entire block from/to DDR. Busy week – Other course assignment and project.	Plan is to have something prepared and tested to show by Milestone meeting and get live feedback. Any thoughts or feedback about creating AXI Full based IPs that could help me overcome the steep learning curve would be helpful!

Milestone 5:

Name	Proposed Milestone	Progress Made	Challenges	Questions/ Comments
Eduardo	Begin system integration and testing of hardware and software on FPGA 2. Conduct extensive debugging activity during this week	Integrated Mustafa's database IP control into my TCP server (see below).	FPGA network was down over the weekend, so couldn't run any tests from my side. Had issues with memory mapping, for the microblaze (see below)	Comment: Currently only POST and GET requests work, we will be implementing DELETE and PUT requests in software towards the final demo, if time allows.
Mustafa		First round of integration is completed. Hardware IP is added to the TCP/IP Server Project. A TCP client python script is used to emulate an actual FPGA client.	Tried to implement the Database in DDR (this works well as a standalone project) However, when the Project includes Ethernet for a TCP Server, the memory sections of the program must be moved to DDR, as they	Is there a way to write to/ access DDR when using it for memory mapped regions of the code? If no alternate option exists, can we proceed with using the BRAM as

Name	Proposed Milestone	Progress Made	Challenges	Questions/ Comments
		POST (Add) and GET (Fetch) requests are sent by the client, Server stores information in the database at the provided UID offset, and then successfully fetches it during a GET request.	don't fit in any other blocks. This led to an issue with storing the database in DDR, when attempting writing to DDR, the SDK would hang up and not write to DDR Currently using a large BRAM as workaround to resolve this	our final Database memory block?
Xuening	Perform system level testing on FPGA 1 including checking the quality of communication	Fixed the issue in FPGA 1 block diagram, added the TCP server to Microblaze. Finished a version of TCP client that is combined with the TCP server on Microblaze. Updated the python script in PC 1 to allow encoding of input message at runtime.	The auto connection function sometimes messes up the clock signals between different blocks.	Need to confirm with FPGA 2 on the format of messages sent. The data matrix encoder encodes each character with value (ascii + 1). Need to be careful with the database address in the data matrix message.
Guoxian	between PC1 and FPGA 1, correctness of information decoded from the QR code and correctness of information displayed on VGA.	Design the block diagram of FPGA1. Design the AXI master and slaves for communication between custom IP and Microblaze. Work together with Xuening on debugging and integrating FPGA1.	Debugging AXI bus with many masters and slaves is a challenge. Use AXI VIP to debug it and find out if the AXI bus transactions come in correct order. Also, manually connecting AXI interconnects can ensure that the master can communicate correctly with the slave.	The encoded value of the character in the Datamatrix should be the ascii for the character in our project. (eg. 65 for 'A') instead of (ascii + 1) (eg. 66 for 'A') in the Datamatrix protocol

Milestone 6:

Name	Proposed Milestone	Progress Made	Challenges	Questions/ Comments
Eduardo	Finalize full system integration, test overall functionality and connections PC-FPGA or FPGA-FPGA. Adjust functions and fix problems as	Connected client FPGA with FPGA server, allowing them to swap TLV messages Improvements and bug fixes on the server interface and control python scripts.	Had some formatting issues with the input coming in from the Data Matrix script. This has now been fixed and whole system integration testing will happen tomorrow.	
Mustafa		Tested the DDR access issue using guidance	Database IP is writing to the correct DDR	IP works well with the client, no further

Name	Proposed Milestone	Progress Made	Challenges	Questions/ Comments
	needed for optimal system performance.	from Piazza post response. System still doesn't write to DDR after following that approach Plan is to continue to use BRAM block for the database as it works well with the client. Confirmed this by hardware tests	address location – confirmed by using ILA However, when checked using Memory monitor, no data was written to the DDR. Also, Ethernet connection breaks when writing to DDR is attempted. My guess is since both Microblaze and Database IP are trying to access the DDR via MIG, there is a conflict of some sorts that causes the issue	changes will be made. BRAM will be used as the database memory
Xuening		Worked together with Guoxian to solve a problem in displaying data matrix on FPGA 1. Tested connection with FPGA 2. Updated the PC1 python script to allow inputting of an image.	The TCP/IP connection sometimes behaves weirdly. I can send packets from FPGA (client) to PC (server) but not in the reversed direction.	FPGA 1 now works well with the FPGA 2 python script. Need some final testing on the DESL machines.
Guoxian		Fix bugs in FPGA hardware design. Work together in system integration		The input database address is added by an offset to be encoded by DataMatrix Library.