Hardware Accelerated High-Frequency Trading System

Design Document

TA:
Mingjia Huo

Team Members:
Siyi Yu(siyiyu2)
Kevin Lim (wzlim2)
Richard Deng (ruichao4)

September 29, 2022

Contents

1 Introduction

1.1 Problem

Modern electronic markets have been characterized by a relentless drive towards faster decision making. Significant technological investments have led to dramatic improvements in latency, the delay between a trading decision and the resulting trade execution. We describe a theoretical model for the quantitative valuation of latency. A low latency for the communication, which refers to sending the order to exchange and receiving the order from exchange, is critical since sometimes a delay of milliseconds could result in a loss of millions dollars.

“Low latency” in a contemporary electronic market would be qualified as under 10 milliseconds, “ultra low latency” as under 1 millisecond. This change represents a dramatic reduction by five orders of magnitude. To put this in perspective, human reaction time is thought to be in the hundreds of milliseconds.

In the financial market today there is a lot of need to optimize the trading/execution latency to support automated quantitative trading systems. While most of the computer software runs on generic operating systems and the CPU executing the logic has many parts of unnecessary instructions/procedures, the automated trading strategy can be highly optimized with hardware to achieve low-latency and high-frequency.

1.2 Solution

The purpose of this project is to prototype a hardware based trading tool that can perform such a high-frequency and low-latency trading. while being able to communicate with the exchange using basic network protocol.

We plan to build a trading system that uses one PCB to connect to a fake exchange and get/send binary market data and use highly optimized FPGA to consume this market data and make decisions based on the data. Since we will not be able to directly connect to the exchanges without approval, we are also going to simulate an exchange using software. Most of the existing low-latency trading strategy will be implemented using FPGA solely. However, our project will utilize an ESP8266 chip for networking purposes.

The ESP8266 is a low-cost Wi-Fi microchip, with built-in TCP/IP networking software, and microcontroller capability. FPGAs which could accomplish this task are pretty high end and relatively expensive. FPGAs tend to offer vastly greater flexibility which we do not need for this project. It will help us bring down the cost while maintaining a relatively good performance of the system.

1.3 Visualization

Figure 1 illustrates the overview of our project design.

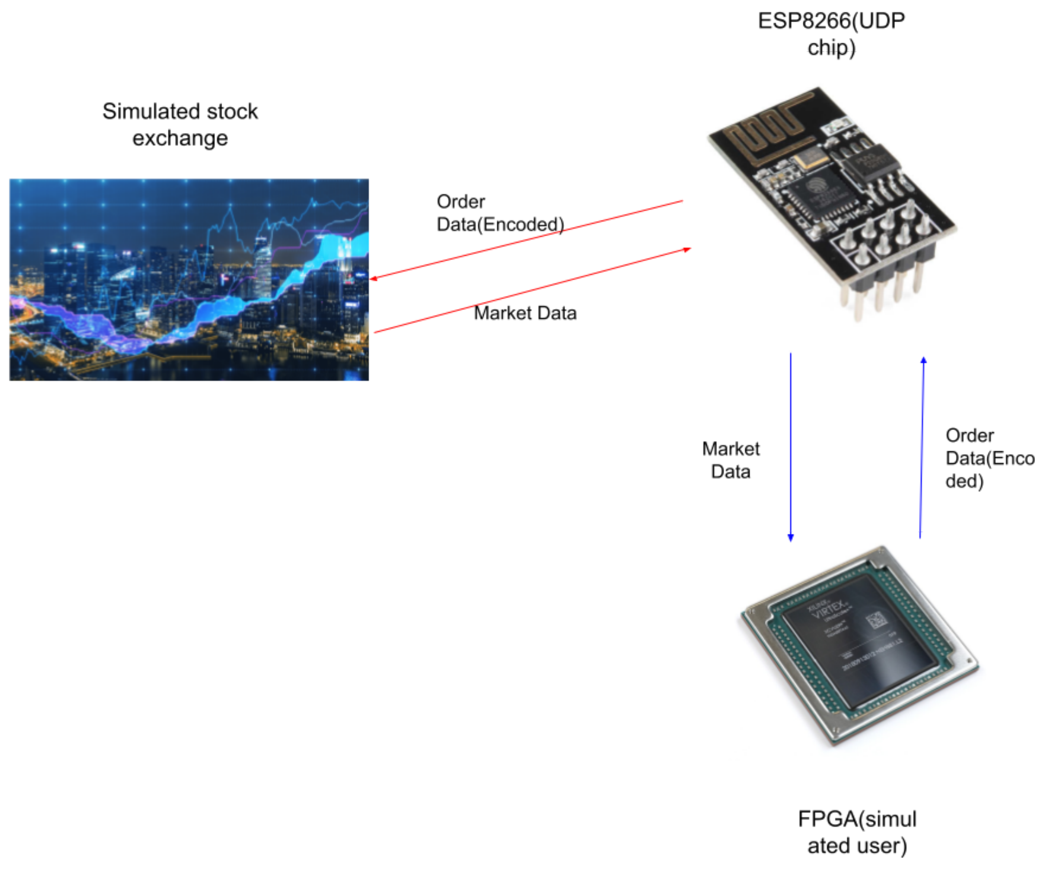


Figure 1: User diagram

- Simulated stock exchange will be implemented fully using software using C++ code

- We are planning on to use USB for the data transmission between FPGA and ESP8266 chip

- We will use UDP protocol for the communication between ESP8266 and simulated stock exchange

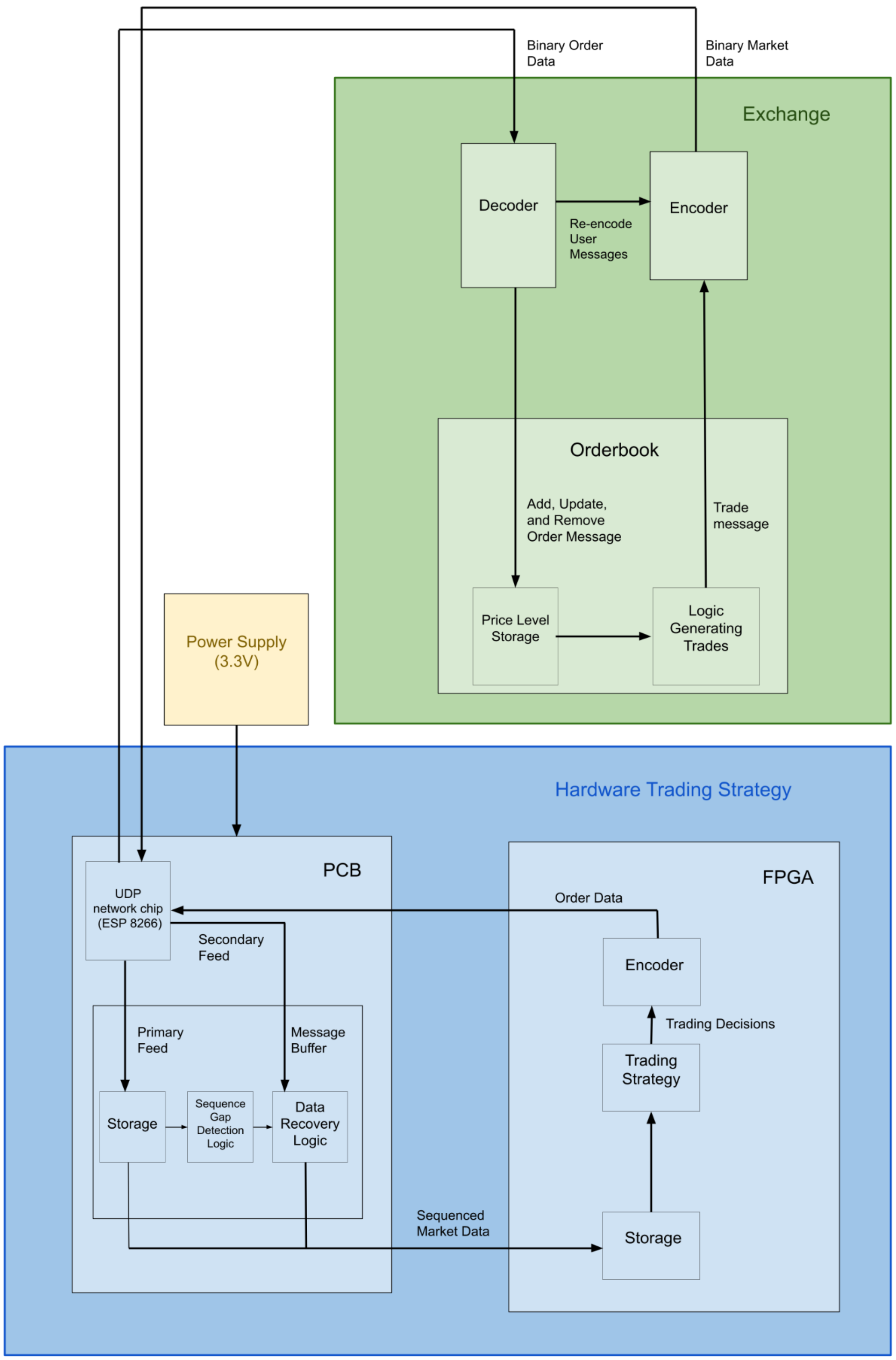
1.4 High-Level Requirements

For this project, we have 3 high-level requirements that we aim to achieve:

1. 80% orders being successfully sent to exchange and received by users
2. 30% faster than using purely software implementation of this system
3. 100% orders being correctly processed by the exchange

2 Design

2.1 Block Diagram



2.2 Subsystem Overview

Our project consists of three major systems, simulated exchange, networking hardware, and FPGA trading strategy.

2.2.1 Simulated Exchange

This is a simulation of real-world exchange. It can receive orders from market participants and automatically generate trade messages and broadcast it back to market participants. This simulated exchange consists of three parts:

I. Order Data Decoder. This decoder will decode the binary order data sent by market participants into internal, more organized, human-readable data and feed it to the orderbook.
II. Orderbook. This is where all the orders from different market participants are stored, and it will generate a trade if the bid side and ask side meets.
III. Market Data Encoder. This encoder will consume the data from the orderbook and encode it into binary market data and broadcast it to market participants.

The simulated exchange will be a piece of software that simulates an exchange in real-time. This simulated exchange will only have one security to trade and will be receiving three types of binary-encoded messages (add order, cancel order, update order) from the market participants using a certain protocol, building a full-depth limit order book based on the order received, automatically trade two orders when the bid side and ask side meet, and re-broadcasting the trade message with the other three client messages in binary encoded form as market data messages. The message protocol we designed are presented below:

Client Message Header (shared among all client messages)

Field Name	Offset (in bytes)	Length (in bytes)	Description
Message Type	0	1	Type of the message (zero padded)
authentication Key	1	4	Client authentication for sending requests to the exchange

Client Add Order Message

Field Name	Offset (in bytes)	Length (in bytes)	Description
Message Type	0	1	Add Order message type (00)
Authentication Key	1	4	Client authentication for sending requests to the exchange
Order Type	5	1	0 for Bid Order, 1 for Ask Order
Price	6	4	Integer value of the price, should be scaled by 10^-4 to get the real price
Quantity	10	4	Integer value of the quantity

Client Cancel Order Message

Field Name	Offset (in bytes)	Length (in bytes)	Description
Message Type	0	1	Cancel Order message type (01)
Authentication Key	1	4	Client authentication for sending requests to the exchange
Order Reference Number	5	4	32 bit unsigned integer for the order intend to delete

Client Update Order Message

Field Name	Offset (in bytes)	Length (in bytes)	Description
Message Type	0	1	Update Order message type (10)
Authentication	1	4	Client authentication for sending requests to the

Key			exchange
Order Reference Number	5	4	32 bit unsigned integer for the order intend to update
Price	9	4	Integer value for the price willing to be updated to, should be scaled by 10^-4, 0 for unchange
Quantity	13	4	Integer value for the quantity willing to be updated to, 0 for unchange

Market Data Message Header (shared among all exchange-sent market data messages)

Field Name	Offset (in bytes)	Length (in bytes)	Description
Message Type	0	1	Type of the message (zero padded)
Sequence Number	1	4	Sequence number of the message

Market Data Add Order Message

Field Name	Offset (in bytes)	Length (in bytes)	Description
Message Type	0	1	Add Order message type (00)
Sequence Number	1	4	Sequence number of the message
Order Type	5	1	0 for Bid Order, 1 for Ask Order
Price	6	4	Integer value of the price, should be scaled by 10^-4 to get the real price
Quantity	10	4	Integer value of the quantity

Market Data Cancel Order Message

Field Name	Offset (in bytes)	Length (in bytes)	Description
Message Type	0	1	Cancel Order message type (01)
Sequence Number	1	4	Sequence number of the message
Order Reference Number	5	4	32 bit unsigned integer for the order intend to delete

Market Data Update Order Message

Field Name	Offset (in bytes)	Length (in bytes)	Description
Message Type	0	1	Update Order message type (10)
Sequence Number	1	4	Sequence number of the message
Order Reference Number	5	4	32 bit unsigned integer for the order intend to update
Price	9	4	Integer value for the price willing to be updated to, should be scaled by 10^-4, 0 for unchange
Quantity	13	4	Integer value for the quantity willing to be updated to, 0 for unchange

Trade Message

Field Name	Offset (in bytes)	Length (in bytes)	Description

Message Type	0	1	Trade message type (11)
Sequence Number	1	4	Sequence number of the message
Order Reference Number Bid Side	5	4	32 bit unsigned integer for the order that is filled to trade on the bid side
Order Reference Number Ask Side	9	4	32 bit unsigned integer for the order that is filled to trade on the ask side
Price	13	4	Trade price integer value, should be scaled by 10^-4
Quantity	17	4	Trade quantity integer value

Below are some examples of the messages:

Client Example Messages

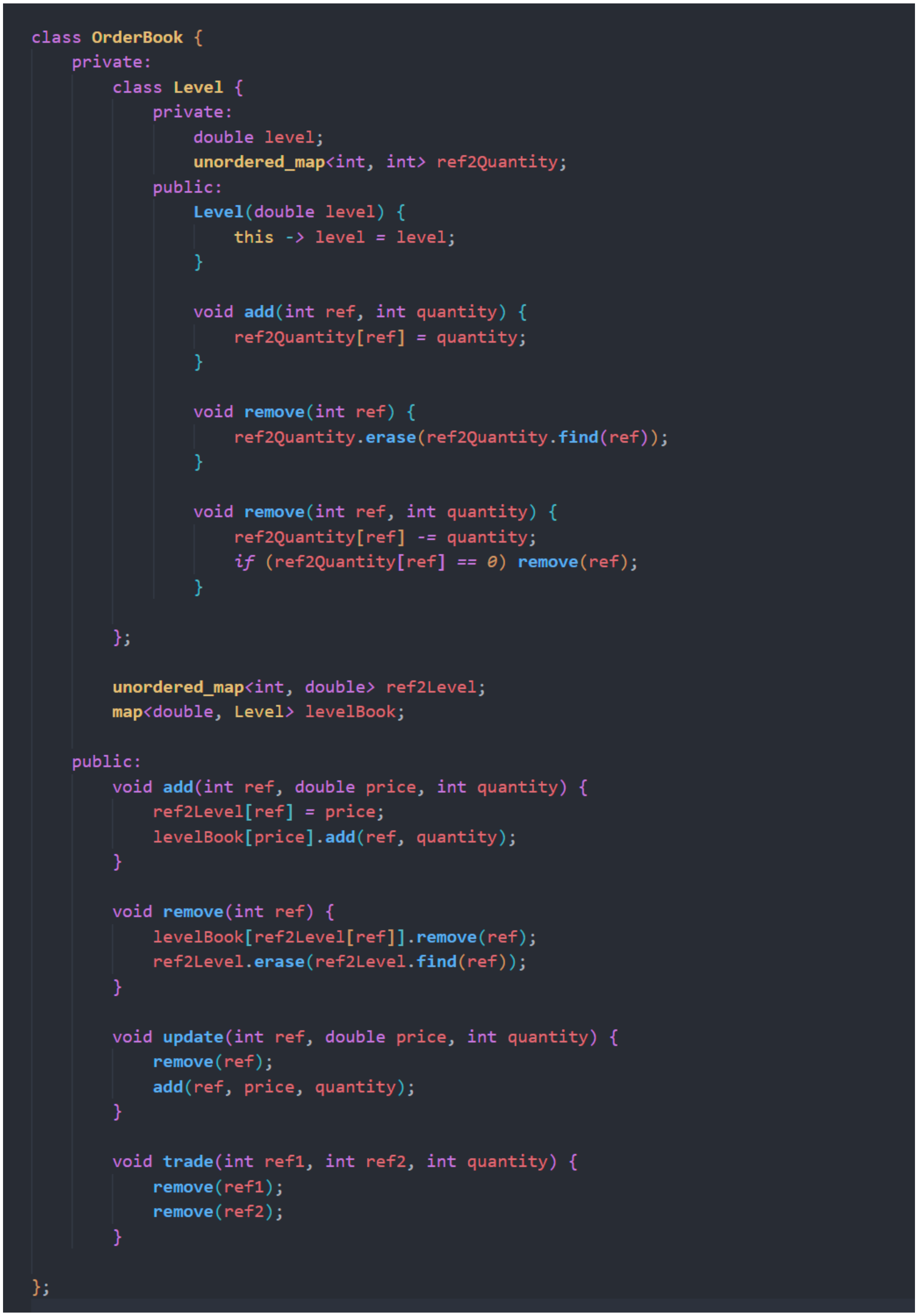
Message in Hex	Message interpretation
00 76F43B25 00 00126308 0000000A	Client with key 0x76F43B25 sent an limit bid order with price of $120.5 and quantity of 10 shares
10 76F43B25 00000010 00124F80 00000000	Client with key 0x76F43B25 send a request to change the price of order 0x10 to $120 while keep quantity unchanged

Market Data Example Messages

Message in Hex	Message interpretation
01 00000001 00000010	Exchange-sent market data message which has a sequence number 1 indicating someone is canceling his order with order

	reference number 0x10
11 00000002 00000010 00000011 00124F80 00000005	Exchange-sent market data message which has a sequence number 2 indicating there is a trade with agreed price $120 and quantity of 5 (could possibly only have partial filled on one side). The order reference number on the bid side is 0x10, and the order reference number on the ask side is 0x11

We will also need to implement an order book so that we could keep track of all the sell and buy orders. Our implementation of it is shown in the figure below.



```cpp
class OrderBook {
    private:
        class Level {
            private:
                double level;
                unordered_map<int, int> ref2Quantity;
            public:
                Level(double level) {
                    this -> level = level;
                }

                void add(int ref, int quantity) {
                    ref2Quantity[ref] = quantity;
                }

                void remove(int ref) {
                    ref2Quantity.erase(ref2Quantity.find(ref));
                }

                void remove(int ref, int quantity) {
                    ref2Quantity[ref] -= quantity;
                    if (ref2Quantity[ref] == 0) remove(ref);
                }

        };

        unordered_map<int, double> ref2Level;
        map<double, Level> levelBook;

    public:
        void add(int ref, double price, int quantity) {
            ref2Level[ref] = price;
            levelBook[price].add(ref, quantity);
        }

        void remove(int ref) {
            levelBook[ref2Level[ref]].remove(ref);
            ref2Level.erase(ref2Level.find(ref));
        }

        void update(int ref, double price, int quantity) {
            remove(ref);
            add(ref, price, quantity);
        }

        void trade(int ref1, int ref2, int quantity) {
            remove(ref1);
            remove(ref2);
        }

};
```

To test our strategy we will simulate the market data received from market participants, constantly adding limit ask/bid at the same price level in high frequency. We will expect 100% of the orders to be processed correctly. This means an ask order and a bid order with the same price will not be existing in the order book, since it will generate a trade. This part of the project is designed to be run completely by software so there is no power supply unit involved in the subsystem.

2.2.2 Networking Hardware

This part is generally a PCB that handles networking between the simulated exchange and the FPGA trading strategy. This system consists of two parts:

I. UDP Networking Chip. This is a ESP8266 chip that processes network packets using UDP protocol.
II. Sequence Gap Detector. This part is responsible for detecting the sequence gap when packets are dropped given the unreliable nature of UDP protocol, and it is also responsible for recovering the lost packets from backup data feed.

Networking hardware will be responsible for the communication between users and the stock exchange. It will be made of a PCB with ESP8266 chip on it and other parts/ports to communicate with the simulated exchange through the network. We plan to use UDP since it is what most exchanges will use. This part will be optimized to reduce the connectivity latency between the fake exchange and the trading system and feed the data to the FPGA.
We would utilize ESP8266WiFi.h and WiFiUdp.h these two libraries to enable the UDP communication. The first library ESP8266WiFi.h is required by default if we are using ESP8266's Wi-Fi. The second one WiFiUdp.h is needed specifically for programming of UDP routines. A complete sketch of code is provided below.

```
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>

const char* ssid = "********";
const char* password = "********";

WiFiUDP Udp;
unsigned int localUdpPort = 4210;  // local port to listen on
char incomingPacket[255];  // buffer for incoming packets
char  replyPacket[] = "Hi there! Got the message :-)";  // a reply string to send back
```

```
void setup()
{
  Serial.begin(115200);
  Serial.println();

  Serial.printf("Connecting to %s ", ssid);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println(" connected");

  Udp.begin(localUdpPort);
  Serial.printf("Now listening at IP %s, UDP port %d\n",
WiFi.localIP().toString().c_str(), localUdpPort);
}


void loop()
{
  int packetSize = Udp.parsePacket();
  if (packetSize)
  {
    // receive incoming UDP packets
    Serial.printf("Received %d bytes from %s, port %d\n", packetSize,
Udp.remoteIP().toString().c_str(), Udp.remotePort());
    int len = Udp.read(incomingPacket, 255);
    if (len > 0)
    {
      incomingPacket[len] = 0;
    }
    Serial.printf("UDP packet contents: %s\n", incomingPacket);

    // send back a reply, to the IP address and port we got the packet from
    Udp.beginPacket(Udp.remoteIP(), Udp.remotePort());
    Udp.write(replyPacket);
    Udp.endPacket();
  }
}
```

Without this component, there will be no way for users and stock exchanges to communicate with each other. Since ESP8266 requires a power supply for it to be functional, we would need a power supply for this subsystem. The ESP8266 requires a 3.3V power supply and 3.3V logic levels for communication and a maximum of 170mA current.

2.2.3 FPGA Trading Strategy

This part is an FPGA board that consumes market data from the networking hardware and generates decisions based on the market data. This consists of two parts:

I. Decision maker (Strategy). This part is the core trading strategy that takes in market data in real-time and makes decisions based on some logic that directly manipulates the binary market data to optimize speed.
II. Order Encoder. This part will take in the decisions that have been made from our strategy, and encode it to binary and send it back to the networking chip, which will eventually be sent to the exchange.

In order to simplify the process(since trading algorithm is not the key part of this project), we plan to implement two simple strategies:

1. High-frequency market-making: This is a liquidity-providing trading strategy that simultaneously generates many bids and asks for a security at ultra-low latency while maintaining a relatively neutral position. When put into implementation, it will make a spread of \$a ~ \$b by sending limit bid at \$a and limit ask at \$b, adjust the spread based on Best Bid Offer (BBO) market data. This part is to test how fast our trading system can be adjusted based on real-time information changes.
2. High-filling rate limit order: A limit order is used to buy or sell a security at a predetermined price and will not execute unless the security's price meets those qualifications. A High-filling rate limit order trading strategy sends a limit bid order whenever there is a limit ask order at \$a. This part is designed to test the latency of our trading system and is extremely useful in options trading where the bid-ask spread is very large and has a low order filling rate.

These two trading algorithms will be sufficient enough for us to mock the real world trading cases and allow us to test the system.

2.2.4 Power Supply

We will need a 3.3V power supply for our PCB board.

2.3 Subsystem Requirements

2.3.1 Simulated Exchange

This simulated exchange will be purely software that is optimized for its speed and correctness. For speed optimization, it will be written in C++ and optimized as much as possible.

I. Order Data Decoder. The decoder should be able to decode the order data sent by market participants in 100% accuracy, and be able to reject invalid orders. We will design a specific protocol for this decoder, and reject any orders that don't follow this protocol. We will also build in an authentication system by forcing the user to send a key as part of the binary message, and our system should be able to verify whether this user is authorized to trade in our exchange.

Requirements	Verification
80% accuracy for order data decoding	Encode several messages according to the protocol, send it to the exchange and check whether the data we sent is in the system
Reject binary data that doesn't follow the protocol	Send some random binary data to see whether it's rejected by the exchange
Reject unauthorized users	Send order data using random authentication key to see whether it's rejected by the exchange

II. Orderbook. The orderbook should be able to store valid order data in our system into two sides, the ask(sell) side and bid(buy) side. To make sure to update the same order later, we will generate an Order Reference Number that associates with each individual order, and store this relation into a map. It will also automatically generate trade when the bid side and ask side meets, i.e. whenever there exists some ask order with price $x with quantity a and some bid order with price $y with quantity b that satisfies (x < = y), a trade of quantity min(a, b) will be generated.

Requirements	Verification

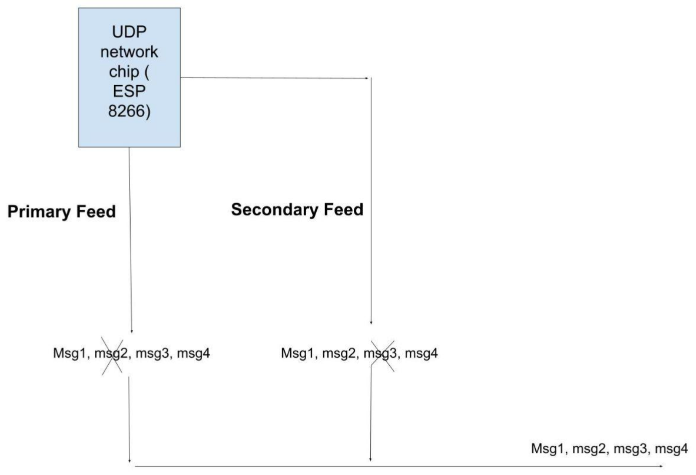
orders are being stored in orderbook and be able to modify	Send orders to the exchange to see whether it's in our system. Modify it later and see whether it's modified in our system.
Trade is generated automatically and properly	Send random orders and make sure best bid price and best ask price never meets

III. Market Data Encoder. The encoder should be able to encode the market data in 100% accuracy and send it back to market participants. To make sure users can fully replicate everything happening in exchange in a time-sequenced manner and also enable users to build sequence gap detection, there will be a sequence number in each market data message. This encoder will also send data in two feeds through different ports/hosts to enable lost packet recovery.

Requirements	Verification
80% accuracy for market data encoding and sequenced	Generate several artificial events in the exchange, and using another test software to receive and decode this data, making sure it's what's happening in the exchange and sequenced correctly
Being able to recover data from backup feed	Receive and check whether both feeds are sending the same data

2.4 Tolerance Analysis

Since we are using UDP as our network communication protocol, it might fail to deliver some of the packets. It is very critical for us that every single packet is received, otherwise we cannot process the market data correctly. However, we designed a sequence gap detection and recovery method built in the PCB, only when both primary data feed and secondary data feed fails on the same packet we will experience a failure, thus our system has high tolerance for network failure. A diagram shown below demonstrates when message 2 drops on primary feed and message 3 drops on secondary feed, we can still recover all 4 messages after combining them.



3 Cost & Schedule

3.1 Cost Analysis

3.1.1 Labor

Name	Hourly Rate($)	Total Hours	Total
Ricahrd Deng	33	144	4752
Kevin Lim	33	144	4752
Siyi Yu	33	144	4752
Total	99	432	14,256

3.1.2 Parts

Parts	Part#	Quantity	Unit Cost($)	Cost($)
FPGA		1		
PCB	-	1	0	0

3.2 Schedule

Week	Task	Responsibility
9/26	Complete Design Document	Siyi Kevin Richard
	FPGA Trading Algorithm Research	Siyi Kevin Richard
10/3	ESP8266 UDP design	Siyi Kevin Richard

	Build Power Supply	Siyi Kevin Richard
10/10	PCB design	Siyi Kevin Richard
	Trading Algorithm Implementation	Siyi Kevin Richard
10/17	Trading Algorithm Implementation	Siyi Kevin Richard
	Testing Trading Algorithm	Siyi Kevin Richard
	PCB design	Siyi Kevin Richard
	Exchange Implementation	Siyi Kevin Richard
10/24	Exchange Implementation and testing	Siyi Kevin Richard
10/31	Integrate PCB to the system	Siyi Kevin Richard
11/7	Testing the system	Siyi Kevin Richard
11/14	Debug and fix any errors which may occur	Siyi Kevin Richard
11/21	Final test for the whole system	Siyi

		Kevin Richard
11/28	Prepare for final presentation	Siyi Kevin Richard
12/5	Final presentation	Siyi Kevin Richard

4 Ethics & Safety

4.1 Ethics

We have identified 3 major concerns of ethics from the Association for Computing Machinery (ACM) Code of Ethics and Professional Conduct for our project[3].

Respect privacy (ACM 1.6)

- Our design may enable users to collect and trade personal information, which violates the privacy of the user. We shall protect the privacy of our users and make sure that no one is able to access personal information.

Honor Confidentiality (ACM 1.7)

- Our design may process highly valuable information such as trade secrets, financial information, and client data. We must not share this data with anyone, and we must not access the information ourselves.

Design and Implement Systems that are Robust and Usably Secure (ACM 2.9)

- A fragile or easily breakable design will likely lead to data leakage among other problems. We should make sure that our design is robust and all information is secure.

The Code of Ethics from the Institute of Electrical and Electronic Engineers (IEEE) also provides important guidelines for our project. We shall properly credit the contribution of others to our project (IEEE #5), uphold safety standards and disclose any safety concerns to users (IEEE #1), and to not tolerate any form of discrimination with our project (IEEE #7).[4]

Our design aims to reduce the latency of trading and make financial operations more efficient. As a team, our goal is to make this design safe and accessible while respecting the code of ethics and other moral concerns.

4.2 Safety

One concern regarding safety within our project is the power supply. If not handled properly, there is a risk of overcharging which could damage the PCB. We will make sure to test our power supply to not exceed 3.3V.

Our team takes ethical and safety concerns seriously and we will strictly follow all ethics and safety guidelines, including those listed above and other concerns that are not covered by these topics.

5 References

[1] Ultra Low Latency Networking with FPGA, 2020.
https://www.youtube.com/watch?v=32cK_yDcouQ

[2] UDP - ESP8266 Arduino Core, 2017.
https://arduino-esp8266.readthedocs.io/en/latest/esp8266wifi/udp-examples.html

[3] Association for Computing Machinery, “ACM Code of Ethics and Professional Conduct”, 2018. [Online]. Available: https://www.acm.org/code-of-ethics.

[4] Institute of Electrical and Electronic Engineers, “IEEE Code of Ethics”, 2017. [Online]. Available: http://www.ieee.org/about/corporate/governance/p7-8.html.