HFT Book Builder

Implemented on DE1-SOC FPGA board

Shivam Shekhar (ss6960@columbia.edu)
Choka Thenappan (ct3185@columbia.edu)
Ameya Keshava Mallya (am6024@columbia.edu)

Guided by
Prof. Stephen Edwards
Vasileios Panousopoulos
Department of Computer Engineering,
Columbia University,
New York

May 12th, 2024

Contents

1 Abstract 3

2 Introduction 3

3 System Design 4

3.1 ITCH NASDAQ Data Structure 4

3.2 Market Simulator 5

3.3 Hardware 6

3.3.1 Top-Level Module 7

3.3.2 Avalon Module 9

3.3.3 Orderbook Module 11

3.3.4 Parser Module 12

3.3.5 Top Module 14

3.3.6 Data Structures 15

3.3.7 State Transitions 15

3.3.8 Handshake Mechanism 17

3.4 Software 18

4 Results 19

5 Conclusion 21

6 Lessons Learnt 21

7 References 21

8 Code Pieces 22

8.1 Market Simulator 22

8.2 Software 25

8.2.1 Driver File 25

8.2.2 Header file 31

8.2.3 User Space Program 32

8.3 Hardware Module 39

8.3.1 Top Module 39

8.3.2 Parser module 52

8.3.3 Order Book Module 56

1 Abstract

Our project focuses the implementation of a Book Builder system on a DE1-SOC FPGA board. We start by transmitting simulated market data to the FPGA board, which then efficiently stores this data in its onboard memory registers. Leveraging this stored data, the FPGA actively processes and analyzes it to generate books for the known stocks in the form of a hardware linked-list. Our paper delves into the architecture and implementation details of this FPGA-based Book Builder, highlighting its efficiency and its potential impact on financial market analysis.

2 Introduction

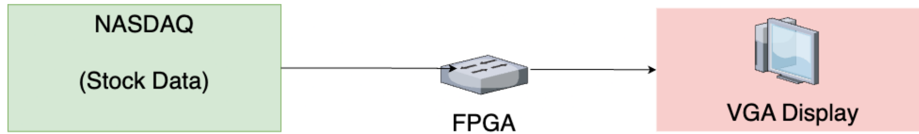

Figure 1: Overview

The Book Builder system tailored for real-time order book generation for specific sets of stocks, primarily aimed at financial institutions like banks. This system efficiently tracks the most recent bidding prices in real-time, catering to the dynamic nature of modern electronic markets. By consuming substantial volumes of market data, the Book Builder outputs an order book that filters out large amounts of information to focus solely on the most important details.

Given the high packet rate flow of modern electronic markets, ensuring minimal latency and deterministic behavior is crucial to mitigate the risk of data loss. Software implementations fall short in this regard due to it being susceptible to packet loss and latency issues.

To address these concerns, leveraging an Ethernet-enabled FPGA proves to be a more suitable approach. By harnessing the parallel processing capabilities inherent in FPGA architecture, we aim to develop a Book Builder system that excels in terms of low latency, determinism, and reliability, offering a robust solution for real-time order book generation in dynamic financial markets.

3 System Design

3.1 ITCH NASDAQ Data Structure

The ITCH protocol, developed by NASDAQ, provides real-time information about market events, such as order additions, modifications, executions, and cancellations. The ITCH data structure is designed to be compact and efficient, enabling fast dissemination and processing of market data. The ITCH data structure consists of a series of binary messages, where each message represents a specific market event. Each message is composed of a fixed-length header followed by a variable-length body. The header contains information such as the message type, timestamp, and message length, while the body contains the actual data payload specific to the message type. Table 1 presents the key data types and their corresponding byte sizes used in the ITCH data structure. The ITCH data structure

Data Type	Byte Size	Type	Value Meaning
Message	1	8'hA	Add order
Timestamp	4	32'h0300	Time that order happened
Order number	4	32'h03BA	Unique value to distinguish order
Buy or sell	1/8	1'b1	A Buy order
Shares	4	32'h01BB	The total number of shares
Stock Symbol	8	64''h0AAB_2341	Which stock the order concerns
Price	4	32'hBABB	The price offered to buy

Table 1: ITCH Data Structure

employs a binary format to minimize the message size and reduce network bandwidth usage. Each field in the message is represented using a specific data type and allocated a fixed number of bytes. Let's discuss each data type in detail:

- **Message Type**: A single byte is used to identify the type of market event associated with the message. Different message types are assigned unique code values. For example, "A" may represent an order addition, "E" may represent an order execution, and "D" may represent an order deletion.

- **Timestamp**: The timestamp field occupies 8 bytes and represents the time at which the market event occurred. It is typically expressed in nanoseconds since midnight.

- **Order Reference Number**: An 8-byte field is used to uniquely identify each order in the system. It allows tracking the lifecycle of an order from its submission to its execution or cancellation.

- **Buy/Sell Indicator**: A single byte is used to indicate whether an order is a buy (0) or sell (1) order.
- **Shares**: The number of shares associated with an order is represented using 4 bytes. It specifies the quantity of the security being bought or sold.
- **Stock Symbol**: The stock symbol field occupies 8 bytes and uniquely identifies the security instrument associated with the order.
- **Price**: The price field is represented using 4 bytes and specifies the price at which the order is to be executed. It is typically expressed in a fixed-point format with a specific number of decimals.
- **Match Number**: An 8-byte match number is assigned to each executed order, allowing the tracking of trade activities.

The ITCH data structure is designed to be parsed and processed efficiently by trading systems. The fixed-length fields allow for quick extraction of relevant information from the messages. The compact binary representation reduces the overall message size, enabling faster transmission and lower latency in real-time trading environments.

3.2 Market Simulator

The Python code developed for market simulation utilizes socket programming to establish a connection between a client and a server. The code begins by importing essential libraries, including the `socket` library, which enables network communication. The IP address and port number of the FPGA server to which the client will connect are defined. The `socket.socket()` function is used to create a socket object, specifying the address family (AF_INET for IPv4) and socket type (SOCK_STREAM for TCP). The `connect()` method is then called on the socket object to establish a connection with the server using the provided IP address and port number.

To keep track of the order reference numbers of existing orders, a list called `existing_orders` is maintained. The `generate_message()` function plays a crucial role in generating random messages representing market orders. It takes an `order_book_id_int` parameter to determine the stock symbol for the order. Inside the `generate_message()` function, various fields of the order message are randomly generated, such as the message type (0x53 for new order, 0x44 for delete order, 0x45 for execute order), order reference number, transaction ID, side, quantity, price, and yield value. The stock symbol is selected based on the

order_book_id_int parameter.

If the message type is 0x44 (delete order), an existing order reference number is randomly chosen from the existing_orders list and removed from the list. If there are no existing orders, the message type is changed to 0x53 (new order). For new orders (0x53), a unique order reference number is generated and added to the existing_orders list. The generated order message is then packed into a binary format using the struct.pack() function, specifying the format string and the corresponding values for each field.

To simulate different stocks, the code iterates over a range of order book IDs (0 to 3). For each order book ID, an initial message is generated and sent using the generate_message() function and the sendall() method of the socket object. A loop is then executed 1000 times for each order book ID. In each iteration, a new message is generated using generate_message(), and the binary representation of the message is printed to the console. The message is sent to the server using sendall().

After sending all the messages, the close() method is called on the socket object to gracefully close the connection.

Sample market data sent:

```
Order Book ID: 0
53 12 34 56 78 B7 57 40 AE C4 90 C7 06 00 00 00 00 42 00 00 00 64 00 00 00 00
53 12 34 56 78 7C DC DF F4 AF 52 34 FF 00 00 00 00 42 00 00 00 64 00 00 00 00
53 12 34 56 78 4F 08 D1 F8 FC 36 40 74 00 00 00 00 42 00 00 00 64 00 00 00 00
44 12 34 56 78 7C DC DF F4 3C 68 E1 1A 00 00 00 00 42 00 00 00 64 00 00 00 00
53 12 34 56 78 94 67 20 B7 A1 4E 57 67 00 00 00 00 42 00 00 00 64 00 00 00 00
45 12 34 56 78 00 00 00 00 F4 9E 39 FF 00 00 00 00 42 00 00 00 64 00 00 00 00
```

3.3 Hardware

The order book system follows a modular design approach, with each module having a specific responsibility. The top-level modules handle the overall system integration and provide the necessary interfaces.

The parser module is responsible for parsing the input buffer and extracting the relevant information. It generates the control signals and activates the appropriate stock based on the parsed data.

The order book module is the core component of the system, managing the orders for a single stock. It maintains the order memory, handles the addition, deletion, and decrease

of orders, and keeps track of the maximum order.

The top module acts as an intermediary, connecting the parser module to the order book instances. It also generates the `system_free` signal based on the ready signals from the order book instances.

The system uses a memory-based approach to store the orders, allowing for efficient insertion, deletion, and searching operations. The use of separate instances of the order book module for each stock enables parallel processing and independent management of orders for different stocks.

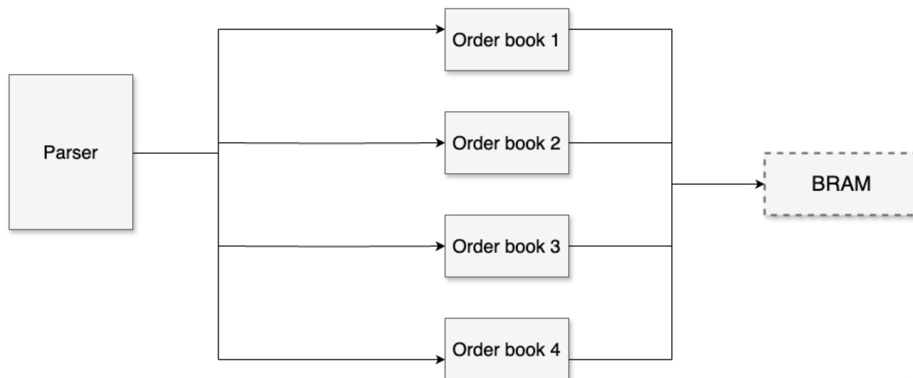

Figure 2: Block Diagram

3.3.1 Top-Level Module

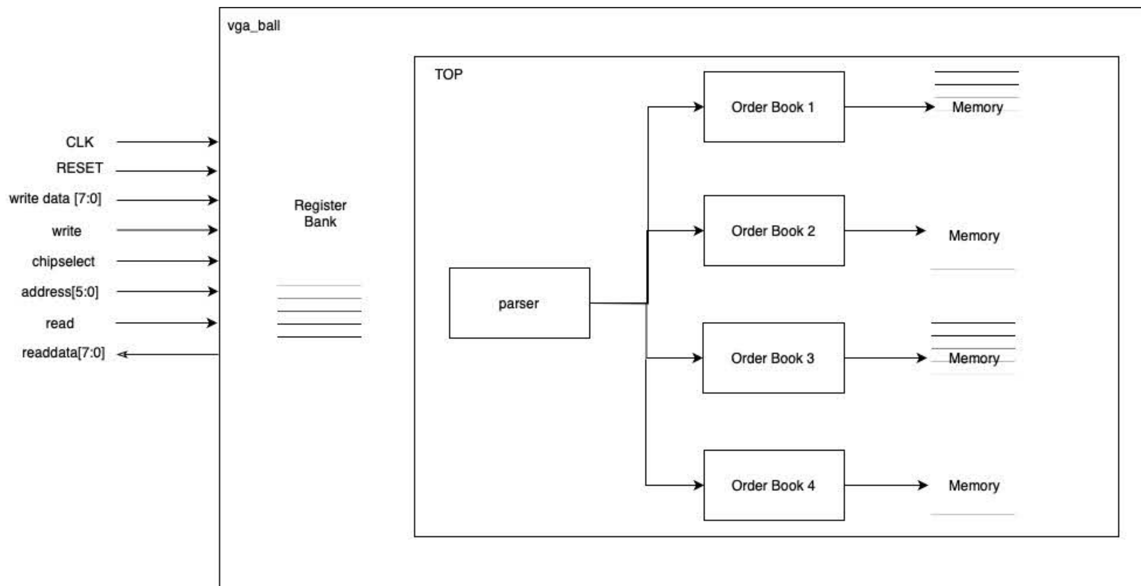

Figure 3: Top Module

The full_top module serves as the top-level entity of the order book system. It instan-

tiates two instances of the top module, one for the bid side and one for the ask side. The module takes in a clock signal (clk), a reset signal (resetn), a `buffer_not_empty` signal indicating the presence of data in the input buffer, and a 320-bit wide input buffer (ff_buffer).

The full_top module assigns the `bid_buffer_not_empty` and `ask_buffer_not_empty` signals based on the content of the `ff_buffer`. It checks if the bits [151:144] of the `ff_buffer` match specific values (8'h42 for bid and 8'h44 for ask) and if the `buffer_not_empty signal` is asserted.
The module outputs the `system_free signal`, which indicates whether the system is ready to accept new orders, and various output registers for the maximum order IDs, quantities, and prices for both the bid and ask sides.

Module interface:

```
module full_top(
input clk,
input resetn,
input buffer_not_empty,
input [319:0] ff_buffer,
output reg system_free,
output reg [31:0] bid_max_order_id_1,
output reg [31:0] bid_max_quantity_1,
output reg [63:0] bid_max_price_1,
output reg [31:0] bid_max_order_id_2,
output reg [31:0] bid_max_quantity_2,
output reg [63:0] bid_max_price_2,
output reg [31:0] bid_max_order_id_3,
output reg [31:0] bid_max_quantity_3,
output reg [63:0] bid_max_price_3,
output reg [31:0] bid_max_order_id_4,
output reg [31:0] bid_max_quantity_4,
output reg [63:0] bid_max_price_4,
output reg [31:0] ask_max_order_id_1,
output reg [31:0] ask_max_quantity_1,
```

```
output reg [63:0] ask_max_price_1 ,
output reg [31:0] ask_max_order_id_2 ,
output reg [31:0] ask_max_quantity_2 ,
output reg [63:0] ask_max_price_2 ,
output reg [31:0] ask_max_order_id_3 ,
output reg [31:0] ask_max_quantity_3 ,
output reg [63:0] ask_max_price_3 ,
output reg [31:0] ask_max_order_id_4 ,
output reg [31:0] ask_max_quantity_4 ,
output reg [63:0] ask_max_price_4 ,
);
```

3.3.2 Avalon Module

The avalon module is a new top-level module that instantiates the full_top module. It provides an interface with individual byte-wide inputs for the `ff_buffer`, making it easier to connect to other modules or systems.
The avalon module also performs some signal conversions. It inverts the reset signal to match the active-low `resetn` signal used in the full_top module. The output signals from the full_top module, such as `max_order_id`, `max_quantity`, and `max_price`, are connected to the corresponding output ports of the avalon module.
Module interface:

```
module avalon(
    input clk,
    input reset,
    input buffer_not_empty,
    input [7:0] ff_buffer_0 ,
    input [7:0] ff_buffer_1 ,
    input [7:0] ff_buffer_2 ,
    input [7:0] ff_buffer_3 ,
    input [7:0] ff_buffer_4 ,
    input [7:0] ff_buffer_5 ,
    input [7:0] ff_buffer_6 ,
    input [7:0] ff_buffer_7 ,
```

```
input [7:0] ff_buffer_8 ,
input [7:0] ff_buffer_9 ,
input [7:0] ff_buffer_10 ,
input [7:0] ff_buffer_11 ,
input [7:0] ff_buffer_12 ,
input [7:0] ff_buffer_13 ,
input [7:0] ff_buffer_14 ,
input [7:0] ff_buffer_15 ,
input [7:0] ff_buffer_16 ,
input [7:0] ff_buffer_17 ,
input [7:0] ff_buffer_18 ,
input [7:0] ff_buffer_19 ,
input [7:0] ff_buffer_20 ,
input [7:0] ff_buffer_21 ,
input [7:0] ff_buffer_22 ,
input [7:0] ff_buffer_23 ,
input [7:0] ff_buffer_24 ,
input [7:0] ff_buffer_25 ,
input [7:0] ff_buffer_26 ,
input [7:0] ff_buffer_27 ,
input [7:0] ff_buffer_28 ,
input [7:0] ff_buffer_29 ,
input [7:0] ff_buffer_30 ,
input [7:0] ff_buffer_31 ,
input [7:0] ff_buffer_32 ,
input [7:0] ff_buffer_33 ,
input [7:0] ff_buffer_34 ,
input [7:0] ff_buffer_35 ,
input [7:0] ff_buffer_36 ,
input [7:0] ff_buffer_37 ,
input [7:0] ff_buffer_38 ,
input [7:0] ff_buffer_39 ,
input [7:0] ff_buffer_40 ,
```

```
    output logic system_free,
    output logic [7:0] max_order_id_1,
    output logic [7:0] max_quantity_1,
    output logic [7:0] max_price_1,
    output logic [7:0] max_order_id_2,
    output logic [7:0] max_quantity_2,
    output logic [7:0] max_price_2,
    output logic [7:0] max_order_id_3,
    output logic [7:0] max_quantity_3,
    output logic [7:0] max_price_3,
    output logic [7:0] max_order_id_4,
    output logic [7:0] max_quantity_4,
    output logic [7:0] max_price_4
);
```

3.3.3 Orderbook Module

The order book module implements the core functionality of managing orders for a single stock. It maintains an internal memory to store the orders and provides an interface to add, delete, and decrease orders.

The module has input signals as such, the clock (clk), reset (resetn), a valid signal indicating the presence of a new request, an order ID (order_id), a quantity (quantity), a price (price), and a request type (req_type). The request type indicates the operation to be performed (add, delete, or decrease order).

The order book module uses a memory array (memory) to store the orders. Each entry in the memory consists of 128 bits, divided into the order ID (32 bits), quantity (32 bits), and price (64 bits). The module also maintains a pointer (pointer) to keep track of the current position in the memory.

The module has different states (defined using parameters) to control its operation. The states include `IDLE`, `ADD_ORDER`, `DELETE_ORDER`, `SHIFT_BOOK`, `FIND_MAX`, and `DECREASE_ORDER`. The module transitions between these states based on the input signals and the current state.

When an order is added (ADD_ORDER state), the module appends the order to the memory and updates the pointer. If the added order has a price higher than the current maxi-

mum price, the module updates the `max_order_id`, `max_quantity`, and `max_price` registers.

When an order is deleted (DELETE_ORDER state), the module searches for the order in the memory based on the order ID. If found, it shifts the subsequent orders to fill the gap and updates the pointer accordingly. If the deleted order was the maximum order, the module transitions to the FIND_MAX state to find the new maximum order.

When an order is decreased (DECREASE_ORDER state), the module searches for the order in the memory based on the order ID and reduces its quantity by the specified amount. If the quantity becomes zero or less, the order is effectively deleted.

The order book module outputs the maximum order ID (max_order_id), quantity (max_quantity), and price (max_price) for the stock it manages. It also provides a ready signal to indicate when it is ready to accept new requests.

Module interface:

```
module order_book (
input clk,
input resetn,
input valid,
input [31:0] order_id,
input [31:0] quantity,
input [63:0] price,
input [2:0] req_type,
output reg [31:0] max_order_id,
output reg [31:0] max_quantity,
output reg [63:0] max_price,
output reg ready
);
```

3.3.4 Parser Module

The parser module is responsible for parsing the input buffer (ff_buffer) and extracting the relevant information, such as the order ID, quantity, price, and stock ID. It generates the appropriate control signals and activates the corresponding stock based on the parsed information.

The parser module takes the 320-bit wide ff_buffer as input and outputs the parsed order

ID (out_order_id), quantity (out_quantity), price (out_price), and a 12-bit stock activation signal (stock_activate).

Inside the module, the input buffer is split into various fields using assign statements. The `req_type` (request type) is extracted from bits [319:312], the `stock_id` from bits [183:152], the `order_id` from bits [247:216], the quantity from bits [143:112], the price from bits [111:48], and the side (bid or ask) from bits [151:144].

The parser module generates the `stock_activate` signal based on the `stock_id` and the request type. It uses a combination of if statements and predefined macros to set the appropriate bits in the `stock_activate` signal. For example, if the `stock_id` matches STOCK1 and the request type is REQ_TYPE_ADD, the module sets the corresponding bit in `stock_activate` to activate the addition operation for STOCK1.

The parsed information (order_id, quantity, price) and the `stock_activate` signal are outputted from the parser module to be used by other modules in the system.

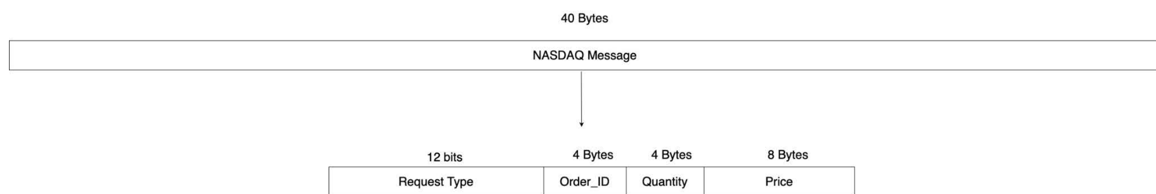

Figure 4: Message Parser

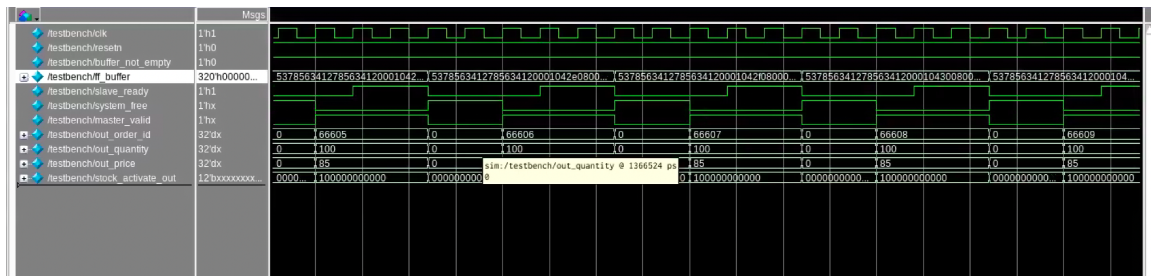

Figure 5: Parser Wave

Module interface:

```
module parser(
input [319:0] ff_buffer,
output reg [31:0] out_order_id,
output reg [31:0] out_quantity,
output reg [63:0] out_price,
output reg [11:0] stock_activate
```

```
);
```

3.3.5 Top Module

The top module instantiates the parser module and four instances of the order book module, one for each stock. It serves as an intermediate layer between the parser and the order book modules.

The top module takes in the clock (clk), reset (resetn), `buffer_not_empty` signal, and the input buffer (ff_buffer). It outputs the `system_free` signal and the maximum order information (order ID, quantity, price) for each stock.

The module connects the outputs of the parser module (order_id_p, quantity_p, price_p, stock_activate_out) to the inputs of the order book instances (order_id, quantity, price, req_type). The req_type input of each order book instance is derived from the corresponding bits of the stock_activate_out signal.

The `system_free` signal is generated by performing a logical AND operation on the ready signals (ready_stock1, ready_stock2, ready_stock3, ready_stock4) from all the order book instances. This ensures that the system is considered free only when all the order book instances are ready to accept new requests.

Module interface:

```
module top(
input clk,
input resetn,
input buffer_not_empty,
input [319:0] ff_buffer,
output reg system_free,
output reg [31:0] max_order_id_1,
output reg [31:0] max_quantity_1,
output reg [63:0] max_price_1,
output reg [31:0] max_order_id_2,
output reg [31:0] max_quantity_2,
output reg [63:0] max_price_2,
output reg [31:0] max_order_id_3,
output reg [31:0] max_quantity_3,
output reg [63:0] max_price_3,
```

```
output reg [31:0] max_order_id_4,
output reg [31:0] max_quantity_4,
output reg [63:0] max_price_4
);
```

3.3.6 Data Structures

The main data structure used in the order book system is the memory array in the order book module. It is a 2D array of 128-bit entries, where each entry represents an order and consists of the order ID (32 bits), quantity (32 bits), and price (64 bits). The memory array has a depth of 1024 entries, allowing for a maximum of 1024 orders per stock.
Another important data structure is the command_out struct defined in the parser module. It is used to store the parsed information and control signals. The struct contains fields for the order ID, quantity, price, stock activation signal, master valid signal, and system free signal.

3.3.7 State Transitions

The order book module has multiple states that control its operation. The state transitions occur based on the input signals and the current state of the module.
The module starts in the `IDLE` state, waiting for a valid request. When a valid request arrives, it transitions to the appropriate state based on the request type (`ADD_ORDER`, `DELETE_ORDER`, or `DECREASE_ORDER`).
In the `ADD_ORDER` state, the module adds the new order to the memory, updates the pointer, and checks if the new order becomes the maximum order. It then transitions back to the `IDLE` state.

In the `DELETE_ORDER` state, the module searches for the order to be deleted. If found, it shifts the subsequent orders and updates the pointer. If the deleted order was the maximum order, it transitions to the `FIND_MAX` state to find the new maximum order.

Otherwise, it transitions back to the `IDLE` state. In the `DECREASE_ORDER` state, the module searches for the order to be decreased and reduces its quantity accordingly. It then transitions back to the `IDLE` state.

The `FIND_MAX` state is entered when the maximum order is deleted, and a new maximum order needs to be found. The module traverses the memory to find the new maxi-

mum order and updates the corresponding registers. It then transitions back to the IDLE state.

Figure 6: Add Order Waveforms

Figure 7: Delete Order Waveforms

Figure 8: Decrease Order Waveforms

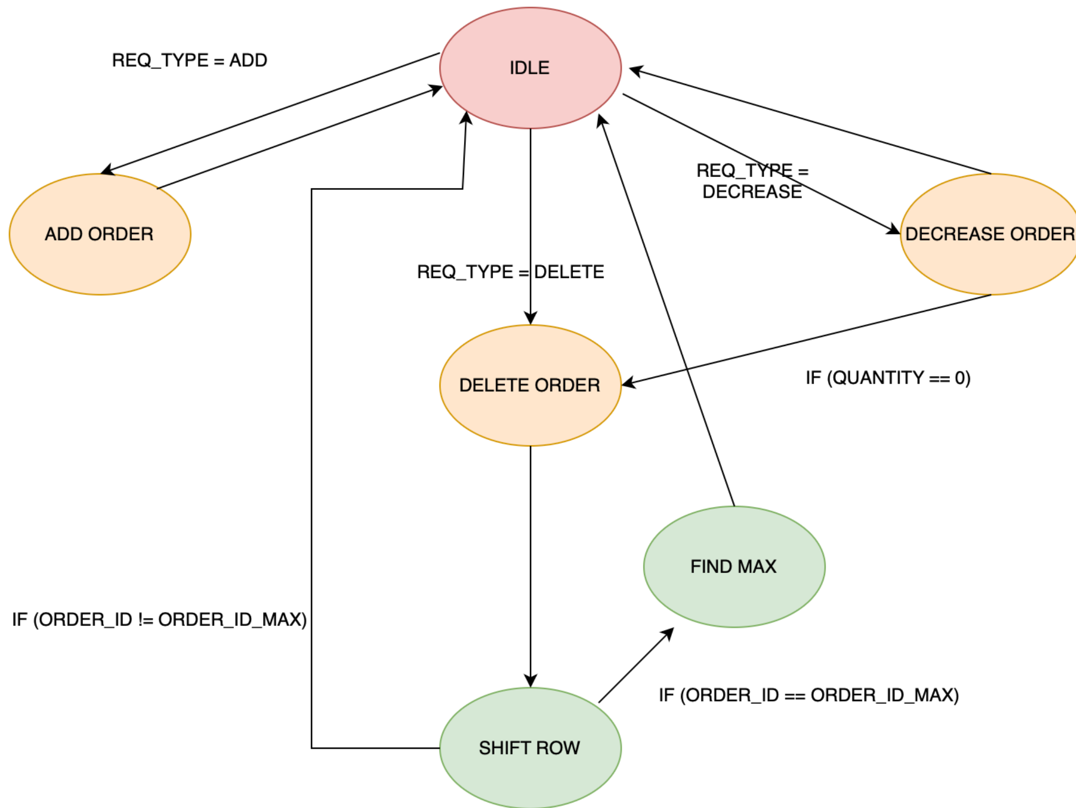

Figure 9: State Transitions

3.3.8 Handshake Mechanism

The order book system uses a handshake mechanism to coordinate the communication between modules and ensure data integrity. The handshake mechanism involves the use of valid and ready signals to control the flow of data and indicate when modules are ready to receive or send data. Handshake mechanism is implemented in the following places:

1. Between the `top` module and the `order_book` instances:

   - The `top` module passes the `buffer_not_empty` signal as the `valid` signal to each `order_book` instance. This signal indicates the presence of a new request.
   - Each `order_book` instance has a `ready` output signal that indicates whether it is ready to accept new requests.
   - The `top` module generates the `system_free` signal by performing a logical AND operation on the `ready` signals from all the `order_book` instances. This ensures that the system is considered free only when all the `order_book` instances are ready to accept new requests.

2. Between the order_book module and the internal states:

   - The order_book module uses a state machine approach to handle different operations (add, delete, decrease) on the orders.
   - The valid signal is used to indicate the presence of a new request to the order_book module.
   - The order_book module transitions to different states based on the valid signal and the req_type input.
   - In each state, the order_book module performs the necessary operations and updates the ready signal to indicate its availability for new requests.

3. Between the parser module and the top module:

   - The parser module takes in the ff_buffer input and parses the relevant information.
   - It generates the order_id, quantity, price, and stock_activate signals based on the parsed data.
   - These signals are connected to the inputs of the order_book instances in the top module.
   - The parser module operates combinatorially and does not have explicit handshake signals. However, the data flow from the parser module to the top module is controlled by the buffer_not_empty signal, which acts as an implicit valid signal.

This handshake mechanism ensures that requests are processed sequentially and that the system maintains data integrity. It prevents data overwrite or loss by ensuring that modules are ready to receive and process data before new data is sent.

3.4 Software

The software module is required because we are simulating the market data using a script instead of taking it from a Hardware IP such as Xilinx which is majorly because of the Hardware Configurations.

The Software modules first establishes a socket connection with the Market simulator, which relays the data to the FPGA Software using the TCP connection. The software then transfers the incoming data to the Hardware to parse the data and process the data such

as Add Order, Delete Order, Execute Order, etc. The trasnfer of data from software to hardware is done using the defined registers in the software which does the memory read and write using the 2-D array memory in the hardware.

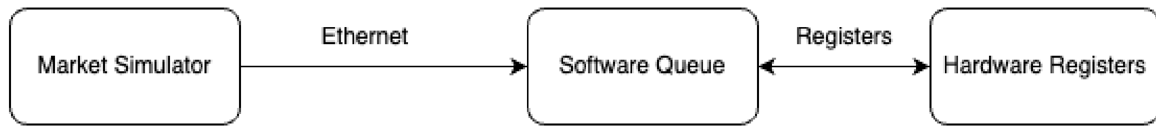

Figure 10: Software Data Transfer

The data transfer from software to hardware isn't a simple transmission; it involves a synchronization mechanism. The software must wait for the hardware to finish processing one order before sending the next data packet. To achieve this, the software initially queues the market data and engages in a handshake with the hardware. When the hardware signals that it's ready (via the READDPORTT signal), indicating that an order has been processed, the software proceeds to transmit the next set of data to the hardware.

This approach ensures efficient coordination between the software and hardware components, facilitating smooth processing of market data within the simulation environment.

4 Results

In the testing, we tested two types of parsing the data in the hardware module: basic add, delete, and decrease order and an optimized version of the same.

First, we used a basic data structure. In this case, add order was working fine as it was taking 1 cycle to add the order in the order book. For Delete Order, it was taking N cycles as it had to parse the entire order book to look for the reference number and then perform the delete operation on that order. In the case where we're deleting the Max value, it was taking N + N - 1 cycles, as it would look for the order, delete it and shift everything up to the new indexes in the order book. Cycles for decrease order was variable as it depends on the position of the order in the order book.

In the optimized version, instead of using 1 way memory, we are using 4 way memory where the data gets divided and individual operations take place and cached to get the final output of the operation. This helps to improve the clock cycles as well. Add order remains the same as 1 cycle, Delete order becomes N/4, Delete order where the order is Max Price becomes N/4 + 3 cycles and Decrease still remains variable but is in 1/4th of the previous implementation.

Request Type	Number of Cycle
ADD ORDER	1 cycle
DELETE ORDER	N cycles
DELETE ORDER is MAX	N + N - 1
DECREASE ORDER	~Depends on the position

Figure 11: Result with basic data structure

Request Type	Number of Cycle
ADD ORDER	1 cycle
DELETE ORDER	~N/4 cycles
DELETE ORDER is MAX	N/4 + 3 Cycles
DECREASE ORDER	~Depends on the position (/4 the previous time)

Here N is the number of valid entries in the Book

Figure 12: Result with the optimized data structure

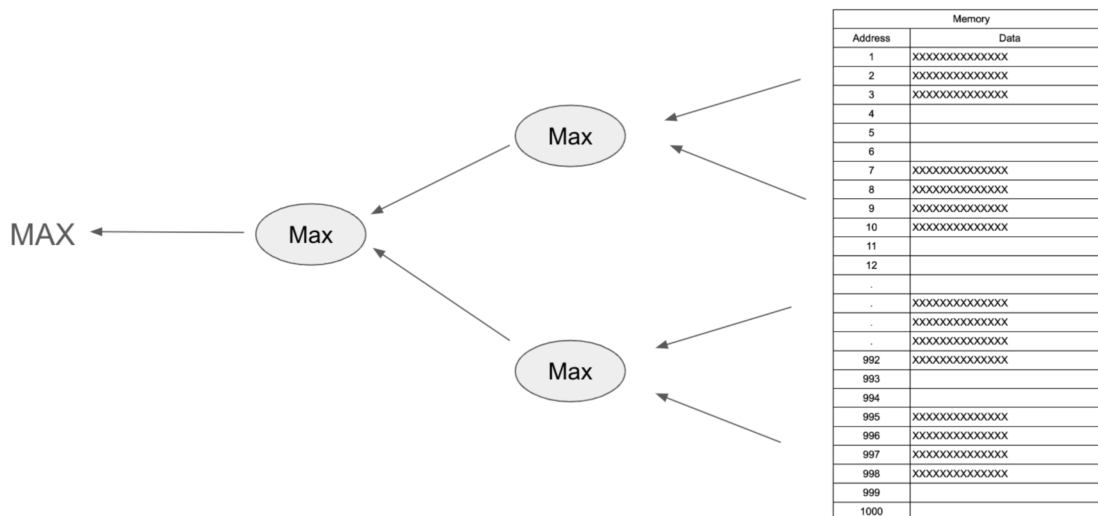

Figure 13: Optimization

5 Conclusion

The project successfully implemented a Book Builder system on an FPGA platform, showcasing its potential for real-time order book generation in dynamic financial markets. By harnessing the parallel processing capabilities of FPGAs, the system achieved low latency and deterministic behavior, crucial for mitigating data loss in high-frequency trading environments. The modular design approach facilitated efficient parsing, processing, and management of market data, enabling the system to maintain accurate order books for multiple stocks simultaneously.

6 Lessons Learnt

- *Leveraging FPGAs for financial applications:* The project demonstrated the suitability of FPGAs for low-latency and deterministic processing of high-frequency market data, highlighting their potential in financial applications.
- *Modular design approach:* The modular design of the system, with separate modules for parsing, order book management, and top-level integration, facilitated easier development, testing, and maintenance.
- *Efficient data structures:* The use of compact binary data structures, such as the ITCH NASDAQ format, and hardware-optimized data structures, contributed to efficient memory utilization and processing performance.
- *Parallel processing:* The ability to process multiple order books in parallel by instantiating multiple order book modules showcased the inherent parallelism of FPGAs, enabling scalability and improved throughput.

7 References

- https://web.mit.edu/6.111/volume2/www/f2019/projects/endrias_Project_Design_Presentation.pd
  Tony, Natnael, ”An HFT (High Frequency Trading) Accelerator”
- https://web.mit.edu/6.111/volume2/www/f2019/projects/endrias_Project_Proposal_Revision.pdf”
  Accelerator”

8 Code Pieces

8.1 Market Simulator

```
import random
import struct
import time
import socket

# Change teh IP address to the IP address of the FPGA
FPGA_IP = "128.59.19.114"
FPGA_PORT = 42000

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

sock.connect((FPGA_IP, FPGA_PORT))

# List to store existing orders
existing_orders = []

# Function to generate random message
def generate_message(order_book_id_int):
    order_reference_number = 0

    msg_type = random.choice([0x53, 0x44, 0x45])



    if msg_type == 0x44:
        if existing_orders:
            order_reference_number = random.choice(existing_orders)
            existing_orders.remove(order_reference_number)
        else:
            msg_type = 0x53
    elif msg_type == 0x53:
```

```
        order_reference_number = random.randint(1, 0xFFFFFFFF)
        existing_orders.append(order_reference_number)

    if order_book_id_int == 0:
        stock = b"AAPLE\0\0\0"
    elif order_book_id_int == 1:
        stock = b"META\0\0\0\0"
    elif order_book_id_int == 2:
        stock = b"NETFLIX"
    else:
        stock = b"ARM\0\0\0\0\0"

    time_stamp = 0x12345678  #
    transaction_id = random.randint(1, 0xFFFFFFFF)
    side = 0x42
    quantity = 0x64
    price = random.randint(1, 1000)
    order_book_id = order_book_id_int
    yield_value = 0x0000000500000005

    message = struct.pack(
        "!BIIIIBIQQ8s",
        msg_type,
        time_stamp,
        order_reference_number,
        transaction_id,
        order_book_id,
        side,
        quantity,
        price,
        yield_value,
        stock
    )
```

```
    return message


for order_book_id_int in range(4):
    print(f"Order Book ID: {order_book_id_int}")
    sock.sendall(generate_message(order_book_id_int))
    for _ in range(1000):
        message = generate_message(order_book_id_int)
        print(" ".join(f"{b:02X}" for b in message))
        sock.sendall(message)
        time.sleep(0.1)

sock.close()
```

8.2 Software

8.2.1 Driver File

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_ball.h"

#define DRIVER_NAME "vga_ball"

/* Device Registers */
#define MESSAGE_TYPE(x) (x)
#define TIMESTAMP(x) ((x)+1)
#define ORDER_REF_NUMBER(x) ((x)+2)
#define TRANS_ID(x) ((x)+3)
#define ORDER_BOOK_ID(x) ((x)+4)
#define SIDE(x) ((x)+5)
#define QTY(x) ((x)+6)
#define PRICE(x) ((x)+7)
#define YIELD(x) ((x)+8)
#define BUFFER_NOT_EMPTY(x) ((x)+9)
#define READPORTT(x) ((x)+10)

/*
```

```
 * Information about our device
 */
struct vga_ball_dev {
        struct resource res;
        void __iomem *virtbase;
        vga_ball_color_t message;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_data(vga_ball_color_t *message)
{
        iowrite8(message->msg_type, MESSAGE_TYPE(dev.virtbase) );
        iowrite8(message->timestamp, TIMESTAMP(dev.virtbase) );
        iowrite8(message->order_ref_number, ORDER_REF_NUMBER(dev.virtbase) );
        iowrite8(message->trans_id, TRANS_ID(dev.virtbase) );
        iowrite8(message->order_book_id, ORDER_BOOK_ID(dev.virtbase) );
        iowrite8(message->side, SIDE(dev.virtbase) );
        iowrite8(message->qty, QTY(dev.virtbase) );
        iowrite8(message->price, PRICE(dev.virtbase) );
        iowrite8(message->yield, YIELD(dev.virtbase) );
        iowrite8(1, BUFFER_NOT_EMPTY(dev.virtbase));
        dev.message = *message;
}


/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
```

```
static long vga_ball_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
        vga_ball_arg_t vla;
        switch (cmd) {
        case VGA_BALL_WRITE_DATA:
                if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                                   sizeof(vga_ball_arg_t)))
                        return -EACCES;
                write_data(&vla.message);
                break;

        case VGA_BALL_READ_DATA:
                vla.message = dev.message;
                if (copy_to_user((vga_ball_arg_t *) arg, &vla,
                                 sizeof(vga_ball_arg_t)))
                        return -EACCES;
                break;
        default:
                return -EINVAL;
        }

        return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_ball_fops = {
        .owner          = THIS_MODULE,
        .unlocked_ioctl = vga_ball_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice vga_ball_misc_device = {
        .minor          = MISC_DYNAMIC_MINOR,
```

```
        .name           = DRIVER_NAME,
        .fops           = &vga_ball_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_ball_probe(struct platform_device *pdev)
{
        vga_ball_color_t message = { 0x41000000000000000000003ea000000000000500
        int ret;

        /* Register ourselves as a misc device: creates /dev/vga_ball */
        ret = misc_register(&vga_ball_misc_device);

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
                ret = -ENOENT;
                goto out_deregister;
        }

        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
                ret = -EBUSY;
                goto out_deregister;
        }

        /* Arrange access to our registers */
        dev.virtbase = of_iomap(pdev->dev.of_node, 0);
        if (dev.virtbase == NULL) {
```

```
                ret = -ENOMEM;
                goto out_release_mem_region;
        }

        /* Set an message */
        write_data(&message);

        return 0;

out_release_mem_region:
        release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
        misc_deregister(&vga_ball_misc_device);
        return ret;
}

/* Clean-up code: release resources */
static int vga_ball_remove(struct platform_device *pdev)
{
        iounmap(dev.virtbase);
        release_mem_region(dev.res.start, resource_size(&dev.res));
        misc_deregister(&vga_ball_misc_device);
        return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_ball_of_match[] = {
        { .compatible = "csee4840,vga_ball-1.0" },
        {},
};
MODULE_DEVICE_TABLE(of, vga_ball_of_match);
#endif
```

```
/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_ball_driver = {
        .driver = {
                .name    = DRIVER_NAME,
                .owner   = THIS_MODULE,
                .of_match_table = of_match_ptr(vga_ball_of_match),
        },
        .remove = __exit_p(vga_ball_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_ball_init(void)
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&vga_ball_driver, vga_ball_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit vga_ball_exit(void)
{
        platform_driver_unregister(&vga_ball_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_ball_init);
module_exit(vga_ball_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("HFT Book Builder Team, Columbia University");
MODULE_DESCRIPTION("VGA ball driver");
```

8.2.2 Header file

```
#ifndef _VGA_BALL_H
#define _VGA_BALL_H

#include <linux/ioctl.h>

typedef struct {
        unsigned char msg_type;
  uint32_t timestamp;
  uint32_t order_ref_number;
  uint32_t trans_id;
  uint32_t order_book_id;
  unsigned char side;
  uint32_t qty;
  uint64_t price;
  uint32_t yield;
  unsigned char buffer_not_empty;
  unsigned char readportt;
} vga_ball_color_t;


typedef struct {
  vga_ball_color_t message;
} vga_ball_arg_t;

#define VGA_BALL_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_BALL_WRITE_DATA _IOW(VGA_BALL_MAGIC, 1, vga_ball_arg_t *)
#define VGA_BALL_READ_DATA  _IOR(VGA_BALL_MAGIC, 2, vga_ball_arg_t *)

#endif
```

8.2.3 User Space Program

```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdint.h>
#include <stdio.h>
#include "vga_ball.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define SERVER_HOST "128.59.64.144"
#define SERVER_PORT 42001
#define BUFFER_SIZE 1024
#define QUEUE_SIZE 100
int vga_ball_fd;
typedef struct {
    char data[BUFFER_SIZE];
} QueueData;
typedef struct {
    QueueData items[QUEUE_SIZE];
    int front, rear;
    pthread_mutex_t mutex;
    pthread_cond_t not_empty;
} Queue;
Queue dataQueue;
void initializeQueue(Queue *q) {
    q->front = -1;
```

```
    q->rear = -1;
    pthread_mutex_init(&q->mutex, NULL);
    pthread_cond_init(&q->not_empty, NULL);
}
int isQueueEmpty(Queue *q) {
    return (q->front == -1 && q->rear == -1);
}
int isQueueFull(Queue *q) {
    return ((q->rear + 1) % QUEUE_SIZE == q->front);
}
void enqueue(Queue *q, QueueData item) {
    pthread_mutex_lock(&q->mutex);
    if (isQueueFull(q)) {
        pthread_cond_wait(&q->not_empty, &q->mutex);
    }
    if (isQueueEmpty(q)) {
        q->front = q->rear = 0;
    } else {
        q->rear = (q->rear + 1) % QUEUE_SIZE;
    }
    q->items[q->rear] = item;
    pthread_cond_signal(&q->not_empty);
    pthread_mutex_unlock(&q->mutex);
}
QueueData dequeue(Queue *q) {
    pthread_mutex_lock(&q->mutex);
    while (isQueueEmpty(q)) {
        pthread_cond_wait(&q->not_empty, &q->mutex);
    }
    QueueData item = q->items[q->front];
    if (q->front == q->rear) {
        q->front = q->rear = -1;
    } else {
```

```
        q->front = (q->front + 1) % QUEUE_SIZE;
    }
    pthread_mutex_unlock(&q->mutex);
    return item;
}
void read_message() {
    vga_ball_arg_t vla;
    if (ioctl(vga_ball_fd, VGA_BALL_READ_DATA, &vla)) {
        perror("ioctl(VGA_BALL_READ_DATA) failed");
        return;
    }
    // printf("%02x %02x %02x %02x %02x %02x %02x %02x %02x\n",
    //        vla.message.msg_type, vla.message.timestamp, vla.message.order_re
    //        vla.message.trans_id, vla.message.order_book_id, vla.message.side
    //        vla.message.qty, vla.message.price, vla.message.yield);
}

uint64_t current_timestamp = 0;

uint64_t generate_increasing_timestamp() {
    return current_timestamp++;
}

uint32_t generate_random_32bit() {
    return rand();
}

uint64_t generate_random_64bit() {
    return ((uint64_t)rand() << 32) | rand();
}

void write_message(const vga_ball_color_t *c) {
    vga_ball_arg_t vla;
```

```
vla.message = *c;
unsigned char bufferNotEmpty = ioctl(vga_ball_fd, VGA_BALL_READ_DATA, &vla)
unsigned char readPort = ioctl(vga_ball_fd, VGA_BALL_READ_DATA, &vla);
if (bufferNotEmpty && readPort) {

if (ioctl(vga_ball_fd, VGA_BALL_WRITE_DATA, &vla)) {
    perror("ioctl(VGA_BALL_WRITE_DATA)␣failed");
    return;
}
printf("Data␣written␣to␣device:\n");
printf("Msg␣Type:␣%02x,␣Timestamp:␣%02x,␣Order␣Ref␣Number:␣%02x,␣Trans␣ID:␣
       vla.message.msg_type, vla.message.timestamp, vla.message.order_ref_nu
       vla.message.trans_id, vla.message.order_book_id, vla.message.side,
       vla.message.qty, vla.message.price, vla.message.yield);

}
else {



    srand(time(NULL));

    uint8_t msg_type = rand() % 3 == 0 ? 0x53 : (rand() % 2 == 0 ? 0x44 : 0
    uint64_t timestamp = generate_increasing_timestamp();
    uint32_t order_ref_number = generate_random_32bit();
    uint32_t order_book_id = rand() % 4;
    uint32_t qty = generate_random_32bit();
    uint64_t price = generate_random_64bit();

    printf("Msg␣Type:␣0x%02x,␣Timestamp:␣0x%016llx,␣Order␣Ref␣Number:␣0x%08
           msg_type, timestamp, order_ref_number, order_book_id, qty, price

    printf("Done!");
```

```
        printf("\n");

        printf("Waiting for ready...\n");
        // Sleep based on the message type
        if (msg_type == 0x44) {
            sleep(4);
        } else if (msg_type == 0x45) {
            sleep(2);
        }
    }
}

void *network_thread_f(void *arg) {
    int sockfd = *(int *)arg;
    char recvBuf[BUFFER_SIZE];
    while (1) {
        int n = read(sockfd, recvBuf, BUFFER_SIZE);
        if (n < 0) {
            perror("Error reading from socket");
            exit(1);
        } else if (n == 0) {
            // printf("Connection closed by client\n");
            close(sockfd);
            break;
        }
        // printf("Received data:\n");
        // for (int i = 0; i < n; i++) {
        //     printf("%02X ", (unsigned char)recvBuf[i]);
        // }
        // printf("\n");
    }
    return NULL;
}
```

```
int main() {
    const char *device_path = "/dev/vga_ball";
    if ((vga_ball_fd = open(device_path, O_RDWR)) < 0) {
        perror("Failed to open vga_ball device");
        exit(1);
    }
    int sockfd, newsockfd, client_len;
    struct sockaddr_in serv_addr, client_addr;
    pthread_t network_thread;
    initializeQueue(&dataQueue);
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Error: Could not create socket");
        exit(1);
    }
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(SERVER_PORT);
    if (bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        perror("Error on binding");
        exit(1);
    }
    listen(sockfd, 5);
    client_len = sizeof(client_addr);
    newsockfd = accept(sockfd, (struct sockaddr *)&client_addr, &client_len);
    if (newsockfd < 0) {
        perror("Error on accept");
        exit(1);
    }
    if (pthread_create(&network_thread, NULL, network_thread_f, (void *)&newso
        perror("Error creating network thread");
        exit(1);
    }
```

```
    while (1) {
        char data[BUFFER_SIZE] = "Data from FPGA";
        QueueData newData;
        strcpy(newData.data, data);
        enqueue(&dataQueue, newData);
        QueueData dataItem = dequeue(&dataQueue);
        vga_ball_color_t vla;
        memcpy(&vla, dataItem.data, sizeof(vga_ball_color_t));
        write_message(&vla);
        sleep(2);
    }
    close(sockfd);
    close(vga_ball_fd);
    return 0;
}
```

8.3 Hardware Module

8.3.1 Top Module

```
/* Order book Marcos */

`define IDLE 3'd0
`define ADD_ORDER 3'd1
`define DELETE_ORDER 3'd2
`define SHIFT_BOOK 3'd3
`define FIND_MAX 3'd4
`define DECREASE_ORDER 3'd5

/* Parser Macros */

`define STOCK1 8'h0
`define STOCK2 8'h20
`define STOCK3 8'h30
`define STOCK4 8'h40

`define REQ_TYPE_ADD 8'h53
`define REQ_TYPE_DELETE 8'h44
`define REQ_TYPE_DECREASE 8'h45

`define ADD1 12'b100000000000
`define DELETE1 12'b010000000000
`define DECREASE1 12'b001000000000
`define ADD2 12'b000100000000
`define DELETE2 12'b000010000000
`define DECREASE2 12'b000001000000
`define ADD3 12'b000000100000
`define DELETE3 12'b000000010000
`define DECREASE3 12'b000000001000
`define ADD4 12'b000000000100
`define DELETE4 12'b000000000010
```

```
`define DECREASE4 12'b000000000001

//`define IDLE 2'b00
//`define SEND_COMMAND 2'b01
//`define WAIT_FOR_RESPONSE 2'b10

typedef struct{
        logic [31:0] order_id;
        logic [31:0] quantity;
        logic [63:0] price;
        logic [11:0] stock_activate;
        logic master_valid;
        logic system_free;
} command_out;

module vga_ball(
/* Shivam changes */
        input logic             clk,
        input logic             reset,
        input logic      [7:0]  writedata,
        input logic             write,
        input logic             chipselect,
        input logic      [5:0]  address
);

logic [7:0] message [63:0];
assign VGA_R = 8'd45;
integer i;

always_ff @(posedge clk) begin
        if (reset) begin
                for (i=0; i<64; i=i+1) message[i] <= 8'd0;
        end else if (write) begin
```

```
                message [address] <= writedata;
        end
end


/*top new_top (
    .clk(clk),
    .resetn(~reset),
    .buffer_not_empty(buffer_not_empty),
    .ff_buffer({ff_buffer_0 , ff_buffer_1 , ff_buffer_2 , ff_buffer_3 , ff_buffer_4
    .system_free(system_free),
    .max_order_id_1(max_order_id_1),
    .max_quantity_1(max_quantity_1),
    .max_price_1(max_price_1),
    .max_order_id_2(max_order_id_2),
    .max_quantity_2(max_quantity_2),
    .max_price_2(max_price_2),
    .max_order_id_3(max_order_id_3),
    .max_quantity_3(max_quantity_3),
    .max_price_3(max_price_3),
    .max_order_id_4(max_order_id_4),
    .max_quantity_4(max_quantity_4),
    .max_price_4(max_price_4)
);*/


endmodule

module top(
input logic         clk,
input logic         reset,
input logic [319:0]  ff_buffer,
input logic         buffer_not_empty,
```

```
output logic system_free,
output logic [31:0] max_order_id_1,
output logic [31:0] max_quantity_1,
output logic [63:0] max_price_1,
output logic [31:0] max_order_id_2,
output logic [31:0] max_quantity_2,
output logic [63:0] max_price_2,
output logic [31:0] max_order_id_3,
output logic [31:0] max_quantity_3,
output logic [63:0] max_price_3,
output logic [31:0] max_order_id_4,
output logic [31:0] max_quantity_4,
output logic [63:0] max_price_4
);

logic ready_stock1;
logic ready_stock2;
logic ready_stock3;
logic ready_stock4;

logic [31:0] order_id_p;
logic [31:0] quantity_p;
logic [63:0] price_p;
logic [11:0] stock_activate_out;

assign system_free = ready_stock1 && ready_stock2 && ready_stock3 && ready_stoc

parser p_block(
.ff_buffer(ff_buffer),
.out_order_id(order_id_p),
.out_quantity(quantity_p),
.out_price(price_p),
.stock_activate(stock_activate_out));
```

```
order_book stock1(
.clk(clk),
.reset(reset),
.valid(buffer_not_empty),
.order_id(order_id_p),
.quantity(quantity_p),
.price(price_p),
.req_type(stock_activate_out[11:9]),
.max_order_id(max_order_id_1),
.max_quantity(max_quantity_1),
.max_price(max_price_1),
.ready(ready_stock1));

order_book stock2(
.clk(clk),
.reset(reset),
.valid(buffer_not_empty),
.order_id(order_id_p),
.quantity(quantity_p),
.price(price_p),
.req_type(stock_activate_out[8:6]),
.max_order_id(max_order_id_2),
.max_quantity(max_quantity_2),
.max_price(max_price_2),
.ready(ready_stock2));

order_book stock3(
.clk(clk),
.reset(reset),
.valid(buffer_not_empty),
.order_id(order_id_p),
.quantity(quantity_p),
```

```
.price(price_p),
.req_type(stock_activate_out[5:3]),
.max_order_id(max_order_id_3),
.max_quantity(max_quantity_3),
.max_price(max_price_3),
.ready(ready_stock3));

order_book stock4(
.clk(clk),
.reset(reset),
.valid(buffer_not_empty),
.order_id(order_id_p),
.quantity(quantity_p),
.price(price_p),
.req_type(stock_activate_out[2:0]),
.max_order_id(max_order_id_4),
.max_quantity(max_quantity_4),
.max_price(max_price_4),
.ready(ready_stock4));


endmodule



module order_book (
input clk,
input resetn,
input valid,
input [31:0] order_id,
input [31:0] quantity,
input [63:0] price,
input [2:0] req_type,
output logic [31:0] max_order_id,
```

```
output logic [31:0] max_quantity,
output logic [63:0] max_price,
output logic ready
);

logic [127:0] memory [1023:0];
logic [9:0] pointer;
logic [2:0] current_state;
logic [9:0] search_pointer;
logic [63:0] temp_max;
logic [63:0] temp_max_price;
logic [31:0] temp_max_quantity;
logic [31:0] temp_max_order_id;

always_ff@(posedge clk)begin
    case (current_state)
        `IDLE: begin
            //ready <= 1'b1;
            search_pointer <= 10'd0;
            if (ready == 1'b1 && valid == 1'b1 && req_type != 3'b000 && resetn
                if (req_type == 3'b100)begin
                    current_state <= `ADD_ORDER;
                    ready <= 1'b0;
                end
                else if (req_type == 3'b010)begin
                    current_state <= `DELETE_ORDER;
                    ready <= 1'b0;
                end
                else if (req_type == 3'b001)begin
                    current_state <= `DECREASE_ORDER;
                    ready <= 1'b0;
                end
            end
```

```
end
`ADD_ORDER: begin
    ready <= 1'b1;
    current_state <= `IDLE;
    pointer <= pointer + 1;
    memory [pointer] <= { order_id , quantity , price };
    if (price > max_price)begin
        max_order_id <= order_id;
        max_quantity <= quantity;
        max_price <= price;
    end
end
`DELETE_ORDER: begin
    //ready <= 1'b0;
    if (memory[search_pointer][127:96] == order_id)begin
        pointer <= pointer - 1;
        current_state <= `SHIFT_BOOK;
    end
    else begin
        search_pointer = search_pointer + 1'b1;
        if (search_pointer == pointer) begin
                current_state <= `IDLE;
                ready <= 1'b1;
        end
    end
end
`SHIFT_BOOK: begin
    if (search_pointer == (pointer + 1)) begin
        if (order_id == max_order_id) begin
            current_state <= `FIND_MAX;
            search_pointer <= 10'd0;
            temp_max <= 64'd0;
        end
```

```
          else begin
                current_state <= `IDLE;
                ready <= 1'b1;
          end
      end
      else begin
          memory [search_pointer] <= memory [search_pointer+1];
          search_pointer <= search_pointer + 1;
      end
end
`FIND_MAX: begin
      if (memory[search_pointer][63:0] > temp_max )begin
          temp_max_order_id <= memory [search_pointer][127:96];
          temp_max_quantity <= memory [search_pointer][95:64];
          temp_max_price <= memory [search_pointer][63:0];
      end
      search_pointer <= search_pointer + 1'b1;
      if (search_pointer == pointer) begin
          current_state <= `IDLE;
          ready <= 1'b1;
          max_order_id <= temp_max_order_id;
          max_price <= temp_max_price;
          max_quantity <= temp_max_quantity;
      end
end
`DECREASE_ORDER: begin
      ready <= 1'b0;
      if (memory[search_pointer][127:96] == order_id)begin
          if (memory[search_pointer][95:64] > quantity)
              memory[search_pointer][95:64] <= memory[search_pointer][95:
          else memory[search_pointer][95:64] <= 32'd0;
          current_state <= `IDLE;
          ready <= 1'b1;
```

```
            end
            search_pointer <= search_pointer + 1;
            if (search_pointer == pointer) begin
                    current_state <= 'IDLE;
                    ready <= 1'b1;
                end
        end
        default: current_state <= 'IDLE;
    endcase
end

always_comb begin
    if (!resetn)begin
        ready <= 1'b1;
        pointer <= 10'd0;
        current_state <= 'IDLE;
        max_price <= 64'd0;
    end
end

endmodule

module parser(
input [319:0] ff_buffer,
output logic [31:0] out_order_id,
output logic [31:0] out_quantity,
output logic [63:0] out_price,
output logic [11:0] stock_activate
);

logic [7:0] req_type;
logic [31:0] order_id;
logic [31:0] stock_id;
```

```
logic [31:0] quantity;
logic [63:0] price;
logic [7:0] side;
logic [1:0] current_state;
logic [1:0] next_state;
command_out ns_command;
command_out cs_command;
command_out stock_command;
assign req_type = ff_buffer[319:312];
assign stock_id = ff_buffer[183:152];
assign order_id = ff_buffer[247:216];
assign quantity = ff_buffer[143:112];
assign price = ff_buffer[111:48];
assign side = ff_buffer[151:144];
assign stock_command = cs_command;
assign stock_activate_out = cs_command.stock_activate;
assign master_valid = cs_command.master_valid;
assign system_free = cs_command.system_free;
assign out_order_id = order_id;
assign out_quantity = quantity;
assign out_price = price;

always_comb begin
                if (stock_id == `STOCK1) begin
                        if (req_type == `REQ_TYPE_ADD) begin
                                stock_activate <= `ADD1;
                        end
                        else if (req_type == `REQ_TYPE_DELETE) begin
                                stock_activate <= `DELETE1;
                        end
                        else if (req_type == `REQ_TYPE_DECREASE) begin
                                stock_activate <= `DECREASE1;
                        end
```

```
end
if (stock_id == `STOCK2) begin
        if (req_type == `REQ_TYPE_ADD) begin
                stock_activate <= `ADD2;
        end
        else if (req_type == `REQ_TYPE_DELETE) begin
                stock_activate <= `DELETE2;
        end
        else if (req_type == `REQ_TYPE_DECREASE) begin
                stock_activate <= `DECREASE2;
        end
end
if (stock_id == `STOCK3) begin
        if (req_type == `REQ_TYPE_ADD) begin
                stock_activate <= `ADD3;
        end
        else if (req_type == `REQ_TYPE_DELETE) begin
                stock_activate <= `DELETE3;
        end
        else if (req_type == `REQ_TYPE_DECREASE) begin
                stock_activate <= `DECREASE3;
        end
end
if (stock_id == `STOCK4) begin
        if (req_type == `REQ_TYPE_ADD) begin
                stock_activate <= `ADD4;
        end
        else if (req_type == `REQ_TYPE_DELETE) begin
                stock_activate <= `DELETE4;
        end
        else if (req_type == `REQ_TYPE_DECREASE) begin
                stock_activate <= `DECREASE4;
        end
```

```
        end
end
endmodule
```

8.3.2 Parser module

```
`define STOCK1 8'h0
`define STOCK2 8'h20
`define STOCK3 8'h30
`define STOCK4 8'h40

`define REQ_TYPE_ADD 8'h53
`define REQ_TYPE_DELETE 8'h44
`define REQ_TYPE_DECREASE 8'h45

`define ADD1 12'b100000000000
`define DELETE1 12'b010000000000
`define DECREASE1 12'b001000000000
`define ADD2 12'b000100000000
`define DELETE2 12'b000010000000
`define DECREASE2 12'b000001000000
`define ADD3 12'b000000100000
`define DELETE3 12'b000000010000
`define DECREASE3 12'b000000001000
`define ADD4 12'b000000000100
`define DELETE4 12'b000000000010
`define DECREASE4 12'b000000000001

`define IDLE 2'b00
`define SEND_COMMAND 2'b01
`define WAIT_FOR_RESPONSE 2'b10

typedef struct{
        reg [31:0] order_id;
        reg [31:0] quantity;
        reg [63:0] price;
        reg [11:0] stock_activate;
        reg master_valid;
```

```
        reg system_free;
} command_out;

module parser(
input [319:0] ff_buffer,
output reg [31:0] out_order_id,
output reg [31:0] out_quantity,
output reg [63:0] out_price,
output reg [11:0] stock_activate
);

reg [7:0] req_type;
reg [31:0] order_id;
reg [31:0] stock_id;
reg [31:0] quantity;
reg [63:0] price;
reg [7:0] side;
reg [1:0] current_state;
reg [1:0] next_state;
command_out ns_command;
command_out cs_command;
command_out stock_command;
assign req_type = ff_buffer[319:312];
assign stock_id = ff_buffer[183:152];
assign order_id = ff_buffer[247:216];
assign quantity = ff_buffer[143:112];
assign price = ff_buffer[111:48];
assign side = ff_buffer[151:144];
assign stock_command = cs_command;
assign stock_activate_out = cs_command.stock_activate;
assign master_valid = cs_command.master_valid;
assign system_free = cs_command.system_free;
assign out_order_id = order_id;
```

```
assign out_quantity = quantity;
assign out_price = price;

always@(*) begin
        if (stock_id == `STOCK1)begin
                if (req_type == `REQ_TYPE_ADD)begin
                        stock_activate <= `ADD1;
                end
                else if (req_type == `REQ_TYPE_DELETE)begin
                        stock_activate <= `DELETE1;
                end
                else if (req_type == `REQ_TYPE_DECREASE)begin
                        stock_activate <= `DECREASE1;
                end
        end
        if (stock_id == `STOCK2)begin
                if (req_type == `REQ_TYPE_ADD)begin
                        stock_activate <= `ADD2;
                end
                else if (req_type == `REQ_TYPE_DELETE)begin
                        stock_activate <= `DELETE2;
                end
                else if (req_type == `REQ_TYPE_DECREASE)begin
                        stock_activate <= `DECREASE2;
                end
        end
        if (stock_id == `STOCK3)begin
                if (req_type == `REQ_TYPE_ADD)begin
                        stock_activate <= `ADD3;
                end
                else if (req_type == `REQ_TYPE_DELETE)begin
                        stock_activate <= `DELETE3;
                end
```

```
                else if (req_type == `REQ_TYPE_DECREASE) begin
                    stock_activate <= `DECREASE3;
                end
            end
            if (stock_id == `STOCK4) begin
                if (req_type == `REQ_TYPE_ADD) begin
                    stock_activate <= `ADD4;
                end
                else if (req_type == `REQ_TYPE_DELETE) begin
                    stock_activate <= `DELETE4;
                end
                else if (req_type == `REQ_TYPE_DECREASE) begin
                    stock_activate <= `DECREASE4;
                end
            end
end

endmodule
```

8.3.3 Order Book Module

```
    `define IDLE 3'd0
`define ADD_ORDER 3'd1
`define DELETE_ORDER 3'd2
`define SHIFT_BOOK 3'd3
`define FIND_MAX 3'd4
`define DECREASE_ORDER 3'd5
module order_book (
input clk,
input resetn,
input valid,
input [31:0] order_id,
input [31:0] quantity,
input [63:0] price,
input [2:0] req_type,
output reg [31:0] max_order_id,
output reg [31:0] max_quantity,
output reg [63:0] max_price,
output reg ready
);

reg [127:0] memory [1023:0];
reg [9:0] pointer;
reg [2:0] current_state;
reg [9:0] search_pointer;
reg [63:0] temp_max;
reg [63:0] temp_max_price;
reg [31:0] temp_max_quantity;
reg [31:0] temp_max_order_id;

always@(posedge clk)begin
    case (current_state)
        `IDLE: begin
```

```
        //ready <= 1'b1;
        search_pointer <= 10'd0;
        if (ready == 1'b1 && valid == 1'b1 && req_type != 3'b000 && resetn
            if (req_type == 3'b100)begin
                current_state <= `ADD_ORDER;
                ready <= 1'b0;
            end
            else if (req_type == 3'b010)begin
                current_state <= `DELETE_ORDER;
                ready <= 1'b0;
            end
            else if (req_type == 3'b001)begin
                current_state <= `DECREASE_ORDER;
                ready <= 1'b0;
            end
        end
    end
    `ADD_ORDER: begin
        ready <= 1'b1;
        current_state <= `IDLE;
        pointer <= pointer + 1;
        memory [pointer] <= { order_id, quantity, price};
        if (price > max_price)begin
            max_order_id <= order_id;
            max_quantity <= quantity;
            max_price <= price;
        end
    end
    `DELETE_ORDER: begin
        //ready <= 1'b0;
        if (memory[search_pointer][127:96] == order_id)begin
            pointer <= pointer - 1;
            current_state <= `SHIFT_BOOK;
```

```
        end
        else begin
            search_pointer = search_pointer + 1'b1;
            if (search_pointer == pointer) begin
                    current_state <= `IDLE;
                    ready <= 1'b1;
            end
        end
    end
    `SHIFT_BOOK: begin
        if (search_pointer == (pointer + 1)) begin
            if (order_id == max_order_id) begin
                current_state <= `FIND_MAX;
                search_pointer <= 10'd0;
                temp_max <= 64'd0;
            end
            else begin
                    current_state <= `IDLE;
                    ready <= 1'b1;
            end
        end
        else begin
            memory [search_pointer] <= memory [search_pointer+1];
            search_pointer <= search_pointer + 1;
        end
    end
    `FIND_MAX: begin
        if (memory[search_pointer][63:0] > temp_max )begin
            temp_max_order_id <= memory [search_pointer][127:96];
            temp_max_quantity <= memory [search_pointer][95:64];
            temp_max_price <= memory [search_pointer][63:0];
        end
        search_pointer <= search_pointer + 1'b1;
```

```
            if (search_pointer == pointer) begin
                current_state <= `IDLE;
                ready <= 1'b1;
                max_order_id <= temp_max_order_id;
                max_price <= temp_max_price;
                max_quantity <= temp_max_quantity;
            end
        end
        `DECREASE_ORDER: begin
            ready <= 1'b0;
            if (memory[search_pointer][127:96] == order_id)begin
                if (memory[search_pointer][95:64] > quantity)
                    memory[search_pointer][95:64] <= memory[search_pointer][95:
                else memory[search_pointer][95:64] <= 32'd0;
                current_state <= `IDLE;
                ready <= 1'b1;
            end
            search_pointer <= search_pointer + 1;
            if (search_pointer == pointer) begin
                        current_state <= `IDLE;
                        ready <= 1'b1;
                end
        end
        default: current_state <= `IDLE;
    endcase
end

always@(*)begin
    if (!resetn)begin
        ready <= 1'b1;
        pointer <= 10'd0;
        current_state <= `IDLE;
        max_price <= 64'd0;
```

```
    end
end

endmodule
```