Design and Implementation of FPGA-Based High-Frequency Trading System

Alamuru Pavan Kumar Reddy
M.Tech ESE, 2024–26
24942
kpavan@iisc.ac.in

Shubham Lanjewar
M.Tech MVLSI, 2024–26
24055
shubhaml@iisc.ac.in

Andavarapu Rakesh
M.Tech MVLSI, 2024–26
24003
rakesha@iisc.ac.in

Jaideep M
M.Tech ESE, 2024–26
24954
jaideepm@iisc.ac.in

***Abstract*—High-Frequency Trading (HFT) has revolutionized financial markets by executing large volumes of trades at extremely high speeds, capitalizing on small price discrepancies that exist for mere microseconds. This paper presents the design and implementation of an FPGA-based HFT system that achieves microsecond-level response times, significantly outperforming traditional CPU-based solutions. Our system includes a complete trading pipeline with UART communication, market data parsing, order book management, and multiple trading strategy implementations. We demonstrate that even with the resource constraints of an Artix-7 FPGA, the system can process and respond to market data with deterministic, ultra-low latency. Performance evaluations show consistent processing latencies of approximately 3.2 µs, with the system capable of handling 400 orders per second. This work showcases the potential of FPGA technology to transform high-frequency trading through hardware acceleration and parallelism.**

***Index Terms*—FPGA, High-Frequency Trading, Low Latency, Hardware Acceleration, Order Book, Financial Systems**

I. Introduction and Motivation

A. Introduction

High-Frequency Trading (HFT) is an advanced form of automated trading that relies on sophisticated algorithms to analyze multiple markets and execute large volumes of orders at extremely high speeds. The time frames involved in HFT are measured in microseconds or even nanoseconds, with trading firms constantly seeking ways to reduce latency to gain competitive advantage. In this race for speed, Field-Programmable Gate Arrays (FPGAs) have emerged as crucial technology enablers, offering deterministic execution times and ultra-low latency through hardware-level implementation of trading functions.

The trading process begins when an exchange transmits market data over a network connection, providing real-time information about stock prices available for buying and selling. The HFT engine must receive this data, interpret it using the exchange's protocol, update its internal view of the market, and promptly respond by submitting trades back to the exchange. Companies such as Tower Research, Optiver, and Goldman Sachs are at the forefront of HFT technology, with competition centered on who can execute orders fastest once market data is received.

Department of Electronic Systems Engineering, Indian Institute of Science, Bangalore

For an effective HFT system, several capabilities are essential:

- Parsing incoming market data from the exchange with minimal latency
- Continuously updating the internal market state (order book)
- Executing trades swiftly based on the latest market conditions
- Implementing trading strategies that can identify profitable opportunities

This paper describes our implementation of an FPGA-based HFT system designed to provide these capabilities while maximizing both speed and efficiency. Our design targets the Basys3 Artix-7 FPGA platform and implements a complete trading pipeline with UART communication, market data parsing, order book management, and trading logic.

The primary contributions of this work are:

- A complete FPGA-based HFT system architecture with demonstrated microsecond-level latency
- An efficient order book implementation optimized for minimal search and update times
- Multiple trading strategy implementations including equal-weight allocation and momentum-based approaches
- A Python-based market simulator for realistic system testing

The rest of this paper is organized as follows: Section II discusses the motivation for using FPGAs in HFT and related work. Section III provides an overview of our system architecture. Sections IV through VIII detail each major component of our design. Section IX presents implementation results, and Section X concludes the paper with discussion of challenges and potential future enhancements.

B. Motivation

Traditional software-based HFT systems face several limitations. Market data arriving at a network interface card (NIC) must traverse the PCIe bus to reach the CPU and memory, adding latency in the hundreds of nanoseconds even with the latest PCIe generations. Even after data arrives at the CPU, software-based processing introduces additional latency and, more problematically, variance in execution times due

to operating system overhead, cache misses, and multitasking effects.

FPGAs offer several key advantages for HFT applications:

- **Deterministic Latency**: Hardware implementations provide consistent execution times, eliminating the unpredictability of software systems.
- **Ultra-Low Latency**: Direct hardware processing eliminates multiple layers of software abstraction, reducing response times to nanoseconds.
- **Parallelism**: FPGAs can process multiple operations simultaneously, enhancing throughput for market data handling.
- **Customized Architecture**: Trading algorithms can be implemented directly in hardware, optimized for specific strategies.
- **Reduced Jitter**: Timing variations are minimized, ensuring consistent response times.
- **Direct Market Access**: FPGAs can interface directly with exchange feeds, further reducing latency.

These advantages make FPGAs the platform of choice for cutting-edge HFT implementations, capable of responding to market changes orders of magnitude faster than software-based alternatives.

II. Related Work

FPGA-based trading systems have attracted significant research attention. Leber et al. [2] demonstrated one of the early FPGA implementations for HFT, showing latency improvements of up to 25× compared to software implementations. Lockwood et al. [3] developed specialized network processing on FPGAs for financial applications, focusing on market data handling.

Kahssay et al. [1] implemented a proof-of-concept HFT accelerator on FPGA, showing that even with modest hardware, significant performance improvements are possible. Commercial offerings from companies like Exablaze (now Cisco) and Xilinx provide specialized FPGA solutions for financial trading, advertising sub-microsecond latencies.

Our work builds on these foundations while developing a complete end-to-end system targeting a modest FPGA platform, demonstrating that the benefits of hardware acceleration for trading applications can be achieved even with entry-level FPGA technology.

III. System Architecture

Our FPGA-based HFT system implements a complete trading pipeline consisting of several key components. Fig. 1 shows the high-level architecture of the system.

The system consists of the following main modules:

- **UART Communication**: Handles serial communication with a host computer, which acts as the market simulator in our implementation.
- **Parser**: Decodes incoming market data messages in NASDAQ ITCH format and extracts relevant fields.

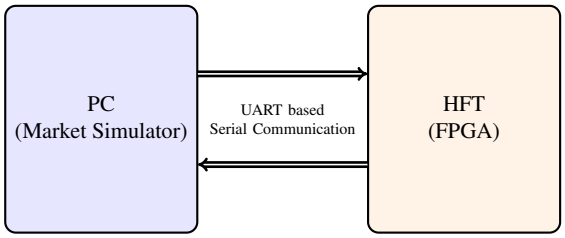

Fig. 1. System architecture of the FPGA-based HFT system-Level 0

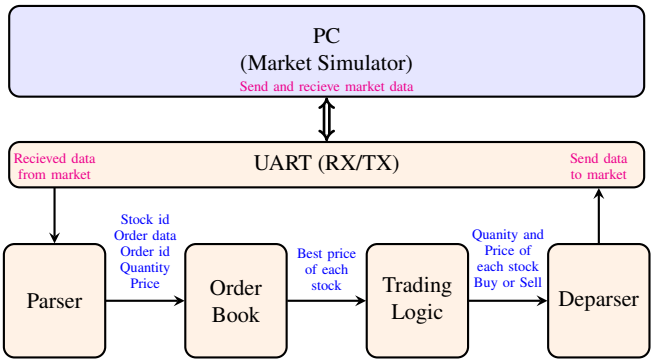

Fig. 2. System architecture of the FPGA-based HFT system-Level 1

- **Order Book**: Maintains a record of current market state, tracking all active orders for multiple stocks and providing best price information.
- **Trading Logic**: Implements multiple trading strategies and makes buy/sell decisions based on current market conditions.
- **Deparser**: Formats trading decisions into messages to be sent back to the market simulator.

Our design is pipelined, with each module processing data and passing results to the next stage. This architecture allows for efficient processing and scalability, enabling the system to handle multiple stocks simultaneously.

A. Target Specifications

The system was designed to meet the following target specifications:

TABLE I
Target Specifications for HFT System

Parameter	Target Specification
Target Platform	Basys3 (Artix-7 35T FPGA)
Target Orders/Second	400 orders/second
Message Format	NASDAQ ITCH and Custom format
Message Types	ADD / EXECUTE messages
UART Baud Rate	115200 bps
Processing Latency	≈ 3 μs per order

B. Clock and Reset Scheme

The system employs a synchronous design with a global 100 MHz clock derived from the on-board oscillator. To ensure

reliable operation, we implemented a robust reset scheme with the following features:

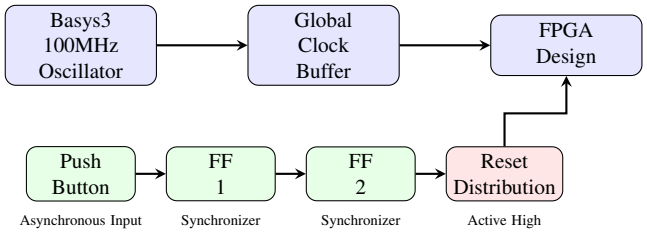

Fig. 3. Clock and Reset Scheme

- Global synchronous reset (active-high)
- Two-stage synchronization for asynchronous inputs
- Reset distribution network to ensure consistent reset across all modules

This approach minimizes the risk of metastability while providing deterministic startup behavior, which is critical for financial applications.

IV. UART Communication Module

The UART (Universal Asynchronous Receiver/Transmitter) module serves as the communication interface between the FPGA and the host computer running the market simulator. Given the critical nature of data transmission in HFT, the UART module was carefully designed for reliability and optimal timing.

A. Design Overview

The UART module consists of two main components:

- **UART Receiver (RX)**: Deserializes incoming market data from the host
- **UART Transmitter (TX)**: Serializes trading decisions to send back to the host

Both components are parameterized to support different baud rates and clock frequencies, though our implementation uses a fixed 115200 bps baud rate with the 100 MHz system clock.

B. UART Receiver

The receiver module implements a state machine with the following states:

- **IDLE**: Waits for start bit (falling edge on RX line)
- **START**: Verifies start bit validity at mid-bit position
- **DATA**: Samples 8 data bits sequentially
- **STOP**: Verifies stop bit and signals data readiness

To improve noise immunity, the receiver implements:

- Triple-register synchronization for the RX input
- Majority filter to prevent spurious transitions
- Mid-bit sampling for optimal signal detection
- False start detection to recover from line noise

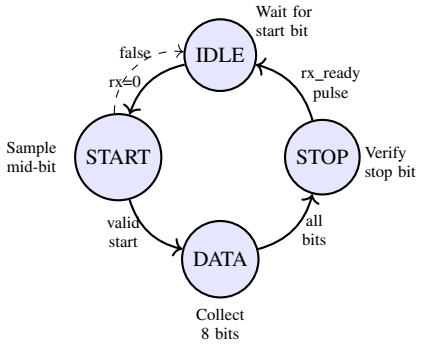

Fig. 4. UART Receiver FSM

C. UART Transmitter

The transmitter module implements a similar state machine:

- **IDLE**: Waits for transmission request (TX line high)
- **START**: Sends start bit (TX line low)
- **DATA**: Transmits 8 data bits LSB-first
- **STOP**: Sends stop bit (TX line high)

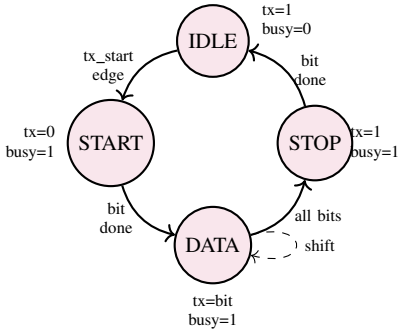

Fig. 5. UART Transmitter FSM

For timing optimization, the transmitter features:

- Edge detection for accurate transmission timing
- IOB (Input/Output Block) register placement for the TX signal
- Busy signal to prevent data overrun

D. Performance Considerations

The UART interface, running at 115200 bps, represents a significant bottleneck in our system's overall latency. With each byte requiring approximately 87 µs to transmit, message processing time is dominated by communication overhead rather than the FPGA's processing capabilities.

In a production HFT system, the UART would be replaced with a high-speed Ethernet interface, reducing communication latency by orders of magnitude. However, for our prototype implementation, UART provides a convenient and reliable interface for system testing and validation.

V. Parser and Deparser Modules

The Parser and Deparser modules handle the conversion between serialized communication protocols and the internal

data formats used by the trading system. These modules are critical for extracting relevant market information and formatting trading decisions.

A. Parser Module

The Parser receives byte-stream data from the UART RX module and reconstructs NASDAQ ITCH protocol messages. It implements a state machine that processes incoming bytes, identifies message types, and extracts relevant fields.

1) Parser State Machine: The Parser implements a 7-state finite state machine:

- **INIT**: Initial state awaiting first byte (message length)
- **POST_INIT**: Receives message type byte
- **DECIDE_MSG_TYPE**: Determines message type and selects appropriate parsing path
- **PARSE_ADD_ORDER**: Processes ADD order messages
- **PARSE_CANCEL_EXEC_ORDER**: Processes CANCEL or EXECUTE order messages
- **PARSE_IGNORED_MSG**: Handles unsupported message types
- **COMPLETE**: Signals completion and output data validity

2) Message Formats: The Parser handles multiple NASDAQ ITCH message types, each with its own format:

- **ADD_ORDER**: 37 bytes, contains information for new order creation
- **EXECUTE_ORDER**: 24 bytes, indicates execution of an existing order
- **CANCEL_ORDER**: 24 bytes, indicates cancellation of an existing order

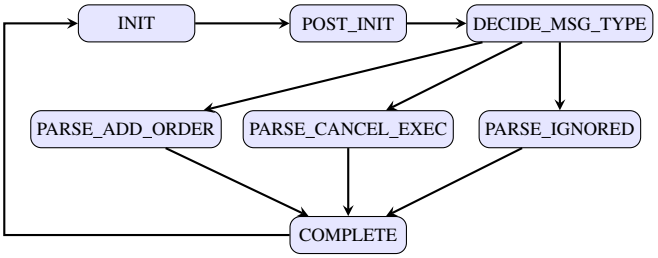

Fig. 6. Parser FSM

For each message, the Parser extracts key fields including:

- Stock identifier
- Order ID
- Price
- Quantity
- Buy/Sell indicator

The extracted data is formatted for the Order Book module and accompanied by a request type signal (ADD, CANCEL, or EXECUTE) to indicate the required operation.

B. Deparser Module

The Deparser receives trading decisions from the Trading Logic module and formats them into messages for transmission back to the market simulator. Unlike the Parser, the Deparser uses a simplified custom message format rather than the NASDAQ ITCH protocol.

1) Custom Message Format: The custom format uses 7 bytes per message:

- Length (1 byte)
- Stock ID (1 byte)
- Buy/Sell indicator (1 byte)
- Quantity (2 bytes)
- Price (2 bytes)

This compact format reduces transmission time while providing all necessary information for the market simulator to process the trading decision.

2) Deparser State Machine: The Deparser implements a state machine with the following states:

- **IDLE**: Awaits trading decision signals
- **GENERATE_MESSAGES**: Creates messages for each stock
- **COMBINE_MESSAGES**: Combines messages for efficient transmission
- **SEND_BYTE**: Initiates transmission of current byte
- **WAIT_UART_BUSY**: Waits for UART TX to accept byte
- **WAIT_NEXT_BYTE**: Waits for current byte transmission to complete
- **DONE**: Signals completion of message transmission

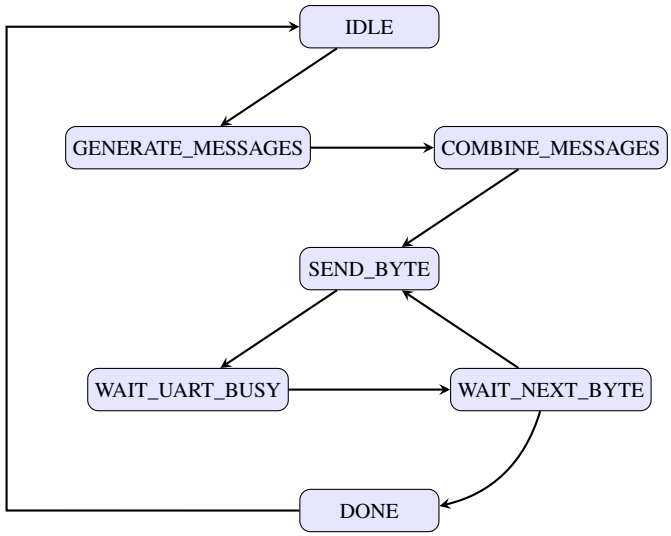

Fig. 7. Deparser FSM

C. Design Rationale

The use of a custom message format in the Deparser, rather than the NASDAQ ITCH protocol, was a deliberate design choice for several reasons:

- **Avoiding Message Avalanche**: Using the ITCH protocol would create a feedback loop where each received message could trigger multiple outgoing messages
- **UART Bandwidth Constraints**: The simplified format reduces transmission time over the limited-bandwidth UART connection

- **System Stability**: Custom formatting ensures complete message transmission before processing the next input

In a production environment with higher-bandwidth communication, standard protocols would likely be preferred for compatibility with exchange systems.

VI. Order Book Module

The Order Book is a central component of any trading system, responsible for maintaining an accurate record of the current market state. It tracks all active buy and sell orders for multiple stocks and provides essential information to the trading logic.

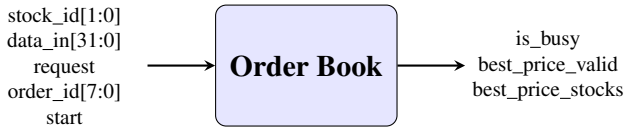

Fig. 8. Order Book inputs and outputs

A. Design Goals

The Order Book module was designed with the following objectives:

- Low-latency updates when new orders are received
- Efficient access to the best available price for each stock
- Support for multiple stocks with independent order tracking
- Compact memory footprint to fit within FPGA BRAM resources

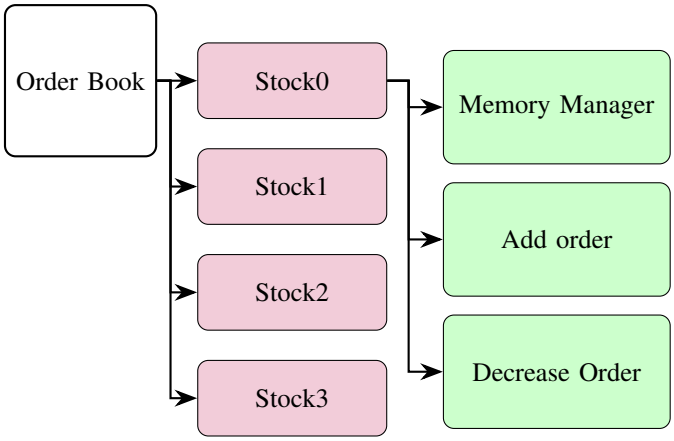

Fig. 9. Order book top level structure

B. Data Structure

The Order Book uses a 32-bit data format for each order entry:

- Quantity (8 bits): Number of shares in the order
- Order ID (8 bits): Unique identifier for the order
- Price (16 bits): Price per share in fixed-point format

Orders are stored in BRAM (Block RAM), with 256 entries available per stock. The system supports up to 4 different stocks, using a total of 4 KB of BRAM.

C. Key Operations

The Order Book supports three primary operations:

- **Add Order**: Inserts a new order into the book
- **Cancel Order**: Removes an existing order
- **Execute Order**: Partially or fully fills an existing order

For each stock, the Order Book maintains a record of the current best price (highest bid), which is updated whenever orders are added, cancelled, or executed.

D. Implementation Details

The Order Book is implemented as a set of sub-modules:

- **Memory Manager**: Controls access to BRAM storage
- **Add Order**: Handles addition of new orders
- **Decrease Order**: Manages cancellation and execution
- **Order Book Wrapper**: Coordinates operations across multiple stocks

1) Memory Manager: The Memory Manager serves as an abstraction layer for BRAM access, handling read and write operations with appropriate timing. It implements a simple state machine:

- **WAITING**: Idle state, awaiting operation request
- **STARTED**: Active state during memory operation

The module includes parameterized BRAM latency management and provides valid signals to indicate operation completion.

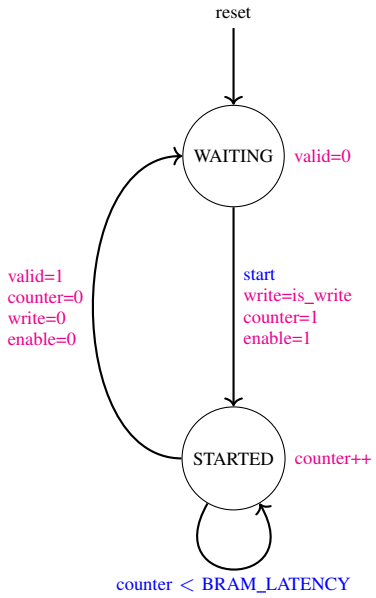

Fig. 10. Memory Manager FSM

2) Add Order: The Add Order module implements a state machine for inserting new orders:

- **START**: Initial state, processes order insertion request
- **PROGRESS**: Waits for memory operation completion

When adding an order, the module updates the best price if the new order has a better price than the current best.

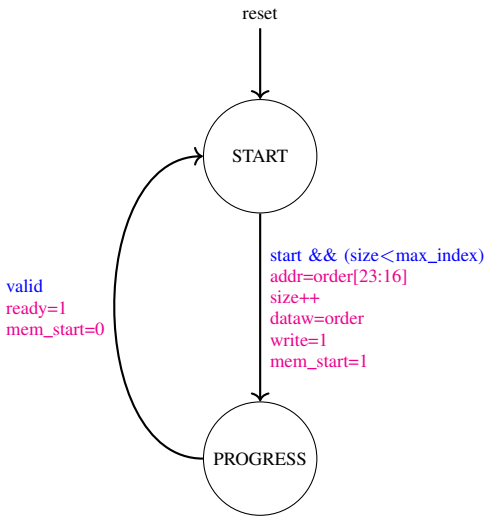

Fig. 11. Add Order FSM

3) Decrease Order: The Decrease Order module implements a more complex state machine for handling order cancellation and execution:

- **WAITING**: Idle state, awaiting operation request
- **FIND**: Locates the target order in memory
- **UPDATE**: Modifies order quantity for partial execution
- **DELETE**: Removes order for cancellation or complete execution
- **BESTPRICE**: Updates best price if necessary

When the best-priced order is cancelled or executed, the module scans the remaining orders to find the new best price.

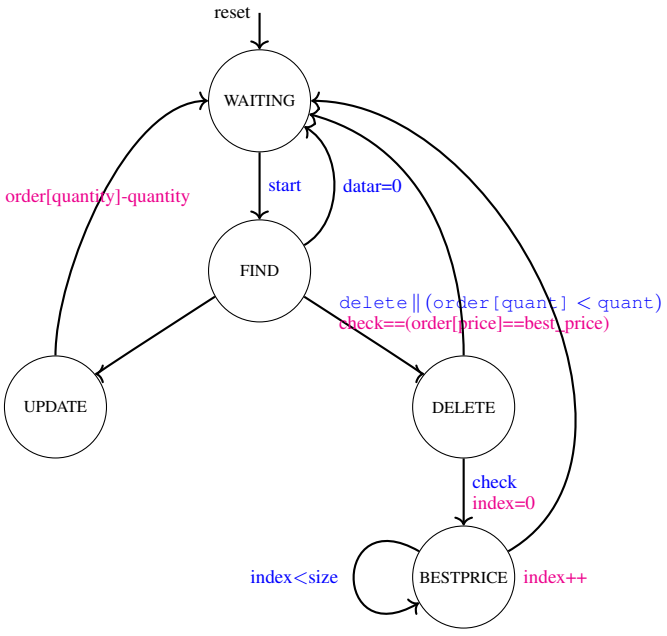

Fig. 12. Decrease Order FSM

4) Order Book Wrapper: The wrapper module coordinates operations across multiple stocks and provides a unified interface to the rest of the system:

- Routes requests to the appropriate stock's Order Book
- Aggregates best prices and valid signals from all stocks
- Manages busy signals to prevent concurrent operations

E. Performance Analysis

The latency of Order Book operations varies based on the operation type:

- **Add Order**: 8 clock cycles (O(1) complexity)
- **Decrease Order**: Up to 260 clock cycles (O(n) complexity)

The O(n) complexity for Decrease Order operations arises from the need to scan all orders to find the new best price when the current best is removed. While more complex data structures like binary heaps could provide O(log n) performance, our implementation prioritizes simplicity and deterministic behavior, which are critical for financial applications.

VII. Trading Logic Module

The Trading Logic module represents the core decision-making component of our HFT system. It analyzes the current market state provided by the Order Book and determines when and how to execute trades based on implemented strategies.

A. Design Overview

The Trading Logic takes best price information from the Order Book for each stock and applies one of several trading strategies to determine:

- Which stocks to buy or sell
- Quantity of shares to trade
- Price points for the trades

The module is implemented as a parameterized design with configurable strategies that can be selected at runtime. It supports multiple assets (in our case, four stocks) and uses fixed-point arithmetic for precise calculations.

B. Trading Module Interface

The Trading Logic module exposes the following interface:

- **Inputs:**
  - Clock and reset signals
  - Best price data for each stock
  - Best price valid signal
  - Strategy selection (2-bit control)
- **Outputs:**
  - Price for each trade (16-bit fixed-point)
  - Quantity for each trade (16-bit fixed-point)
  - Buy/Sell indicator for each stock
  - Valid signal for trade decisions

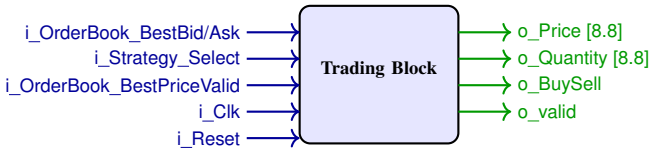

Fig. 13. Trading logic input and outputs

C. Fixed-Point Representation

The trading module uses 16-bit fixed-point representation with 8 bits for the integer part and 8 bits for the fractional part (8.8 format). This format allows for:

- Price range from 0 to 255.996 with 0.004 precision
- Sufficient precision for financial calculations
- Efficient implementation on the FPGA

TABLE II
SIGNAL DESCRIPTION FOR TRADING DATA INTERFACE

Signal	Width	Format	Description
Price	16 bits	8.8 fixed point	Trading price
Quantity	16 bits	8.8 fixed point	Trading volume
BuySell	1 bit	Boolean	1: Buy, 0: Sell

D. Implemented Trading Strategies

We implemented multiple trading strategies with varying complexity:

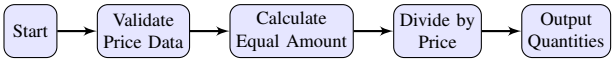

Fig. 14. Trading Strategy - 1

1) Strategy 1: Equal-Weight Allocation: The simplest strategy divides available capital equally among all stocks:

- Allocates funds equally among N stocks
- Total capital (fixed at $1024) divided by N stocks
- Quantity = (Capital per stock / Best Price)
- Executes only sell orders in our implementation

This strategy provides a baseline for testing and benchmarking, with minimal computational requirements.

2) Strategy 2: Momentum-Based (Moving Average): This more sophisticated strategy uses historical price data to make trading decisions:

- Maintains a moving average of depth 8 for each stock
- Determines trend direction by comparing current price to moving average
- Buy when current price is below moving average (undervalued)
- Sell when current price is above moving average (overvalued)
- Uses bit-shifting for efficient division (right shift by 3 for division by 8)

This strategy demonstrates how historical data can be maintained and processed on the FPGA for making more informed trading decisions.

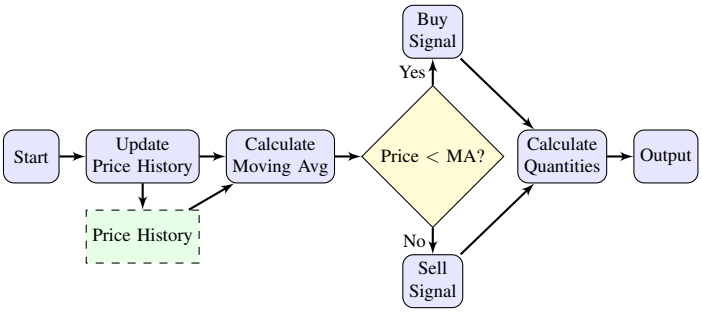

Fig. 15. Trading Strategy - 2

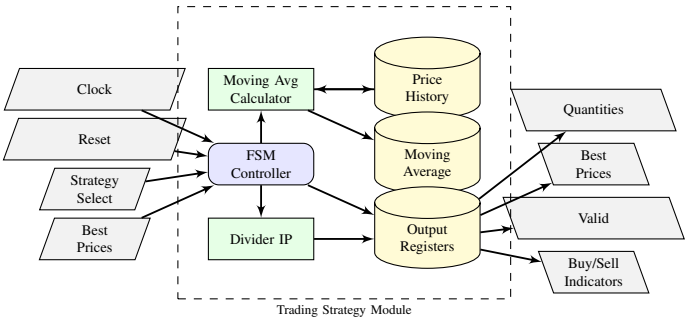

Fig. 16. Integrating Strategy 1 and Strategy 2

3) Strategy 3: Minimum Variance Portfolio (MVP): The most complex strategy implemented aims to minimize overall portfolio risk:

- Forms a covariance matrix from historical stock returns
- Solves matrix equation to find optimal portfolio weights
- Adjusts positions based on risk-adjusted weight allocations
- Requires matrix inversion and linear equation solving

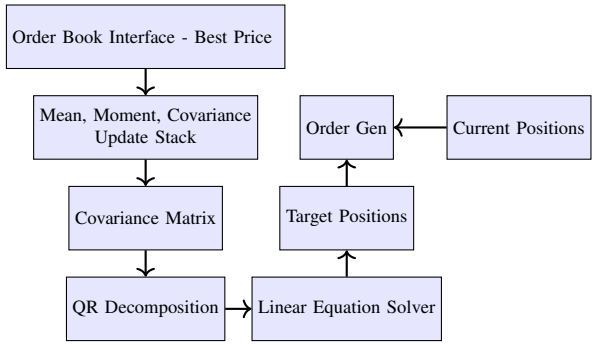

Fig. 17. Trading Strategy - 3

a) Mathematical Formulation: The Minimum Variance Portfolio strategy is mathematically formulated as follows:

- **Setup:** For a universe of N stocks, let $\mathbf{\Sigma}$ be the $N \times N$ covariance matrix of stock returns, and $\mathbf{w}$ the $N \times 1$ portfolio weight vector.
- **Objective:** Minimize total portfolio variance:

$$\min_{\mathbf{w}} \ \mathbf{w}^T \mathbf{\Sigma} \mathbf{w} \tag{1}$$

- **Constraint:** Full investment (weights sum to 1):

$$\mathbf{w}^T \mathbf{1} = 1 \tag{2}$$

- **Closed-form Solution:**

$$\mathbf{w}_{\text{opt}} = \frac{\mathbf{\Sigma}^{-1}\mathbf{1}}{(\mathbf{\Sigma}^{-1}\mathbf{1})^T \mathbf{1}} \tag{3}$$

where $\mathbf{1}$ is a vector of all ones.

b) Covariance Matrix Computation: For each stock, return is computed as:

$$r = \frac{q}{p} \tag{4}$$

where p is the latest price and q is the previous price.

The covariance matrix $\mathbf{K}$ is estimated as:

$$\mathbf{K} = \mathbb{E}[XX^T] - \mathbb{E}[X]\mathbb{E}[X]^T \tag{5}$$

where X is the random vector of stock returns.

c) Recursive Update: The covariance matrix is updated recursively for computational efficiency. At step N, expectations are calculated as:

$$\mathbb{E}_N[XX^T] = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^T \tag{6}$$

$$\mathbb{E}_N[X]\mathbb{E}_N[X]^T = \left(\frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_i\right)\left(\frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_i^T\right) \tag{7}$$

For the online update at step $N+1$:

$$\mathbf{K}_{N+1} = \mathbb{E}_{N+1}[XX^T] - \mathbb{E}_{N+1}[X]\,\mathbb{E}_{N+1}[X]^T \tag{8}$$

$$\mathbb{E}_{N+1}[XX^T] = \frac{N\,\mathbb{E}_N[XX^T] + \mathbf{x}_{N+1}\mathbf{x}_{N+1}^T}{N+1} \tag{9}$$

$$\mathbb{E}_{N+1}[X] = \frac{N\,\mathbb{E}_N[X] + \mathbf{x}_{N+1}}{N+1} \tag{10}$$

d) Matrix Inversion via QR Decomposition: Rather than explicitly computing $\mathbf{K}^{-1}$, which is computationally expensive and numerically unstable, we solve the linear system:

$$\mathbf{v} = \mathbf{K}^{-1}\mathbf{1} \Rightarrow \mathbf{K}\mathbf{v} = \mathbf{1} \tag{11}$$

Then normalize to get $\mathbf{w}$:

$$\mathbf{w} = \frac{\mathbf{v}}{\mathbf{v}^T\mathbf{1}} = \frac{\mathbf{v}}{\sum_i v_i} \tag{12}$$

For improved numerical stability, we use QR decomposition:

$$\mathbf{K} = \mathbf{Q}\mathbf{R} \tag{13}$$

where $\mathbf{Q}$ is an orthogonal matrix ($\mathbf{Q}^{-1} = \mathbf{Q}^T$) and $\mathbf{R}$ is an upper triangular matrix.

The linear system becomes:

$$\mathbf{K}\mathbf{v} = \mathbf{1} \tag{14}$$

$$\mathbf{Q}\mathbf{R}\mathbf{v} = \mathbf{1} \tag{15}$$

$$\mathbf{R}\mathbf{v} = \mathbf{Q}^{-1}\mathbf{1} = \mathbf{Q}^T\mathbf{1} \tag{16}$$

e) Givens Rotations for QR Decomposition: To implement QR decomposition efficiently on FPGA, we use Givens rotations, which operate on two rows at a time to zero out elements below the diagonal. For a 4×4 matrix, we transform:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \rightarrow \begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & 0 & a'_{44} \end{bmatrix} \tag{17}$$

For a 2×2 example, to convert $\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ to upper triangular form, we compute $\theta = \arctan\left(\frac{a_{21}}{a_{11}}\right)$ and apply the rotation:

$$\begin{bmatrix} a'_{11} \\ a'_{21} \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix} = \begin{bmatrix} \sqrt{a_{11}^2 + a_{21}^2} \\ 0 \end{bmatrix} \tag{18}$$

Algorithm 1 QR Decomposition with Givens Rotations

```
Augment K with a column of ones: [K|1]
for k = 1 to N − 1 do {Process each column}
    for j = k + 1 to N do {Process each row below diagonal}
        Compute θ = arctan(A[j, k], A[k, k])
        for i = k to N + 1 do {Include augmented column}
            Apply rotation to (A[k, i], A[j, i]) using θ
        end for
    end for
end for
R ← upper triangular part of transformed matrix
Q^T 1 ← transformed augmented column
Solve Rv = Q^T 1 by back-substitution
w ← v/∑_i v_i =0
```

f) Order Generation: Once optimal weights are determined, orders are generated as follows:

$$\Delta_i = \text{target}_i - \text{position}_i \tag{19}$$

$$\text{shares}_i = \frac{\Delta_i}{\text{price}_i} \tag{20}$$

$$\text{direction}_i = \begin{cases} \text{BUY}, & \text{if shares}_i > 0 \\ \text{SELL}, & \text{otherwise} \end{cases} \tag{21}$$

$$\text{quantity}_i = |\text{shares}_i| \times \text{account\_value} \tag{22}$$

g) FPGA Implementation: The hardware implementation uses:

- CORDIC (COordinate Rotation DIgital Computer) modules for computing trigonometric functions and performing rotations
- Pipelined matrix operations for covariance updates
- Fixed-point arithmetic with sufficient precision for financial calculations
- Parallel processing across all assets for O(1) execution time regardless of portfolio size

The complete algorithm leverages QR decomposition followed by back-substitution, avoiding the need for explicit matrix inversion while maintaining numerical stability.

E. Trading Strategy Implementation Status

Strategy	Post-Synth Sim	Post-Impl Timing	FPGA Impl
Strategy 1 (Equal Weight)	✓	✓	✓
Strategy 2 (Momentum-Based)	✓	✓	✓
Strategy 3 (Minimum Variance)	✓	×	×

TABLE III
TRADING STRATEGY IMPLEMENTATION STATUS

This table shows implementation status of the three trading strategies on the Basys3 (Artix-7 35T FPGA) board. While all strategies were successfully simulated post-synthesis, only Strategy 1 and Strategy 2 achieved successful post-implementation timing closure and FPGA hardware implementation. Achieving post implementation and FPGA implementation of strategy 3 [Minimum Variance Portfolio (MVP)] is one of the future scope of this project.

F. Hardware Implementation Challenges

1) Division Implementation: Division is a complex operation in hardware that typically requires significant resources and multiple clock cycles. For the equal-weight strategy, which requires division to calculate quantities, we used an AXI-based division IP core with the following characteristics:

- Pipelined implementation with fixed latency
- Handshaking signals for data flow control
- 35 cycle latency for 16-bit division

2) Matrix Operations for Strategy 3: The minimum variance portfolio strategy requires matrix operations including:

- Covariance matrix computation
- QR decomposition using Givens rotations
- Back-substitution for solving linear equations

These operations were implemented using CORDIC (COordinate Rotation DIgital Computer) modules for trigonometric functions and vector rotations, though timing closure challenges limited full implementation.

G. Trading Logic FSM

The Trading Logic implements a finite state machine to coordinate the application of the selected strategy:

- **IDLE**: Waits for valid price data from the Order Book
- **PREP**: Prepares data structures for calculation
- **UPDATE**: Updates historical data if needed
- **MA_INIT/MA_SUM/MA_DIV**: States for moving average calculation
- **SETUP/WAIT**: States for division operation coordination
- **OUTPREP/OUT**: States for preparing and outputting results

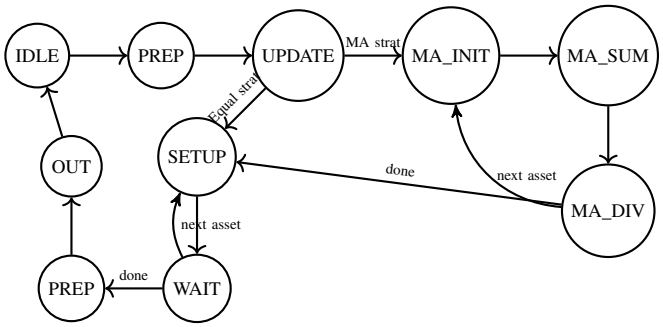

Fig. 18. Trading logic FSM-Strategy 1 and 2

H. Performance Analysis

The latency of the Trading Logic varies based on the selected strategy:

- Strategy 1 (Equal-Weight): Approximately 40 cycles
- Strategy 2 (Momentum): Approximately 160 cycles
- Strategy 3 (Minimum Variance): Theoretical ¿500 cycles (not fully implemented)

For all strategies, the throughput is limited by the need to wait for complete processing of each set of market data before accepting new inputs, resulting in a sequential processing model.

VIII. PYTHON-BASED EXCHANGE SIMULATOR

To test and validate our FPGA-based HFT system, we developed a comprehensive Python-based exchange simulator that mimics a real-world market environment.

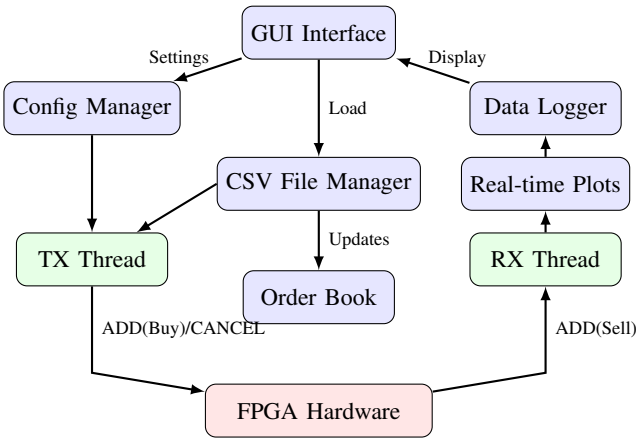

Fig. 19. Software Architecture

A. Simulator Architecture

The simulator consists of several key components:

- **GUI Interface**: Provides visualization of market data and system state
- **Config Manager**: Handles configuration settings for the simulation
- **CSV File Manager**: Imports/exports market data scenarios

- **TX/RX Threads**: Handle concurrent communication with the FPGA
- **Order Book**: Maintains a software mirror of the FPGA's order book
- **Real-time Plots**: Visualize price movements and trading activity
- **Data Logger**: Records system performance for analysis

B. Market Data Generation

The simulator can generate realistic market data in two ways:

- **Synthetic Generation**: Creates order flow based on configurable parameters
- **CSV Import**: Loads pre-recorded or historical market scenarios

For synthetic generation, parameters such as price range, quantity distribution, and order type probabilities can be adjusted to create various market conditions.

C. UART Communication

The simulator communicates with the FPGA through a serial connection:

- Sends market data formatted according to the NASDAQ ITCH protocol
- Receives and decodes trading decisions in the custom message format
- Manages transmission timing to prevent buffer overrun

Multi-threading is employed to ensure responsive operation while maintaining continuous data exchange with the FPGA.

D. Visualization Features

The simulator provides rich visualization capabilities:

- Real-time price charts for each stock
- Order book depth visualization
- Buy/sell activity tracking
- Latency measurement and display
- Message log for protocol debugging

These visualizations enable quick assessment of the HFT system's behavior and performance characteristics.

E. Integration with FPGA System

The simulator was designed to facilitate development and debugging of the FPGA implementation:

- Configurable message rates to match FPGA processing capabilities
- Protocol verification to ensure correct message parsing
- Order tracking to verify order book consistency
- Performance metrics to quantify system latency

This tight integration between the simulator and FPGA system allowed for rapid development iterations and thorough validation of the hardware implementation.

IX. Challenges and Lessons Learned

A. Technical Challenges

During the development of our FPGA-based HFT system, we encountered several significant challenges:

TABLE IV
Challenges and Solutions

Challenge	Solution
Timing problems in trading block	Used IP cores to optimize critical paths
Large latency for cancelling stock	Improved using Order ID based Addressing
Interface problems between parser and order book	Implemented ready-busy handshake protocol
Handshaking issues between processing modules	Corrected based on Integrated Simulation
Big-endian vs. little-endian format handling	Created dedicated conversion modules
Bit-level debugging while integrating with Python simulator	Utilized ILA (Integrated Logic Analyzer) with 16K depth with Trigger Control

B. Key Lessons

The development process yielded several valuable lessons for FPGA-based financial system design:

- **Interface Definition**: Clearly documented interface control, timing diagrams, and dataflow proved critical between modules
- **Handshaking Protocols**: Careful testing and verification of handshaking mechanisms ensured reliable data transfer between modules
- **Individual Testbenches**: Creating separate testbenches for each module significantly reduced debugging time
- **Data Format Conversion**: Fixed-point precision required careful handling at each processing step
- **Endianness**: Big Endian/Little Endian conversions needed explicit management at each layer
- **Resource Optimization**: FPGA resource vs. performance tradeoffs required iterative optimization

These lessons highlight the importance of systematic design practices and thorough verification for FPGA-based financial systems.

C. Overview of Debugging and Design Features

The diagram presents a structured summary of key debugging and design components utilized in the system. It includes functional modules like ILA, BRAM, and VIO, as well as design methodologies such as pipelining, modularity, and parametric design. Features supporting robustness, like reset synchronization, exception handling, and message rejection, are also highlighted. The inclusion of fixed-point precision, AXI interface integration, and GUI-based simulation ensures an efficient and testable hardware design workflow.

X. Implementation Results

This section presents the implementation results of our FPGA-based HFT system, including resource utilization, performance metrics, and latency analysis.

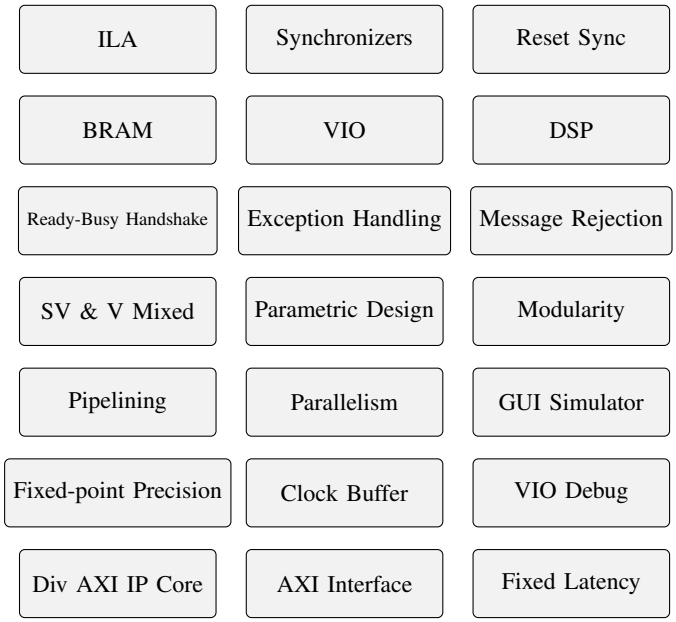

Fig. 20. Overview of Debugging and Design Features

A. Resource Utilization

The complete system was implemented on a Basys3 board featuring an Artix-7 35T FPGA. Table V summarizes the resource utilization.

TABLE V
BASYS3 FPGA RESOURCE UTILIZATION

Resource	Used	Utilization (%)
LUTs	5,440	26%
LUTRAM	366	4%
Flip-Flops	8,159	20%
BRAM	3	6%
IO	4	4%

The resource utilization indicates that the design comfortably fits within the constraints of the target FPGA, with significant room for additional functionality if required.

B. Performance Metrics

Table VI summarizes the key performance metrics of the implemented system.

The maximum clock frequency of 104 MHz exceeds our design target of 100 MHz, indicating that

The maximum clock frequency of 104 MHz exceeds our design target of 100 MHz, indicating that timing closure was successfully achieved. The average processing latency of 3.2 μs meets our target specification of approximately 3 μs per order.

TABLE VI
PERFORMANCE METRICS

Metric	Value
F_{max}	104 MHz
Throughput	400 orders/sec
Latency (avg)	3.2 μs
Power Consumption	0.135 W
UART Baud Rate	115200 bps

C. Latency Analysis

Fig. 21 shows the latency breakdown by module. This analysis highlights the relative contribution of each component to the overall system latency.

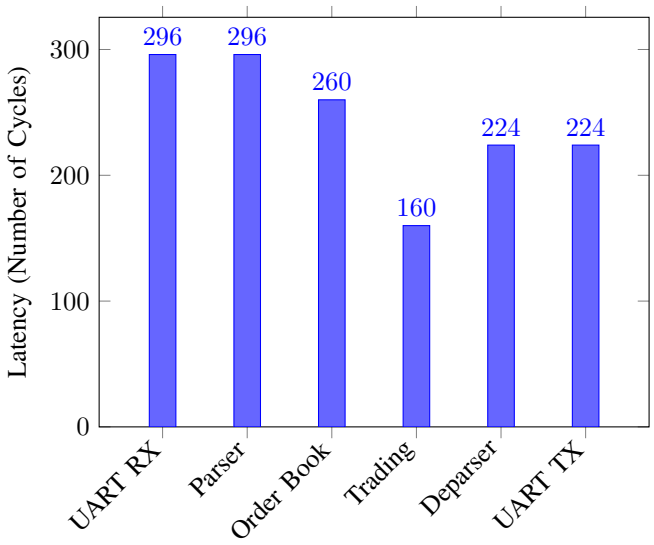

Fig. 21. Latency breakdown by module (clock cycles)

The UART RX and Parser modules contribute the most to the overall latency, primarily due to the serial nature of communication and the need to process each byte sequentially. In a production environment with parallel communication interfaces, these bottlenecks would be significantly reduced.

D. Trading Strategy Performance

Different trading strategies showed varying performance characteristics:

- **Strategy 1 (Equal Weight)**: Successfully implemented with post-synthesis and post-implementation verification, and confirmed in FPGA hardware testing.
- **Strategy 2 (Momentum-Based)**: Successfully implemented with complete verification and hardware testing.
- **Strategy 3 (Minimum Variance)**: Successfully verified in post-synthesis simulation but faced timing issues in post-implementation, preventing full hardware deployment.

The equal-weight and momentum-based strategies proved robust and reliable in testing, while the minimum variance strategy demonstrated the limits of what can be achieved on the targeted FPGA platform.

E. System Optimizations

Several optimizations were implemented to enhance system performance:

- **Parallel Order Book Updates**: Multiple order books updated simultaneously
- **Pipelined Market Data Processing**: Parser designed with pipelined stages
- **Optimized Fixed-Point Calculations**: Arithmetic operations tuned for FPGA implementation
- **Memory Addressing**: Tuned for typical trading patterns to minimize latency

These optimizations collectively contributed to the achievement of our performance targets despite the inherent limitations of the UART interface and the modest FPGA platform.

F. Screenshots

Figures from Fig. 14 to Fig. 21 are showing screenshots of the working demo

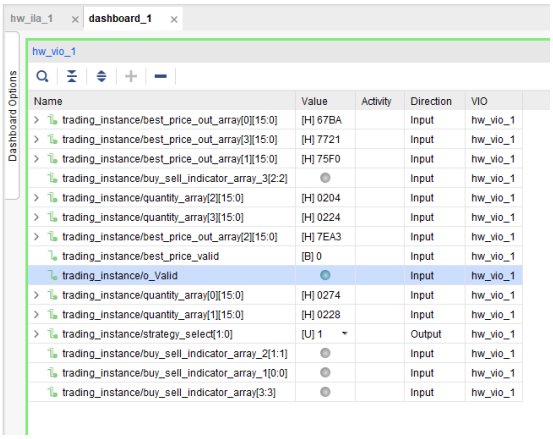

Fig. 22. VIO in Vivado

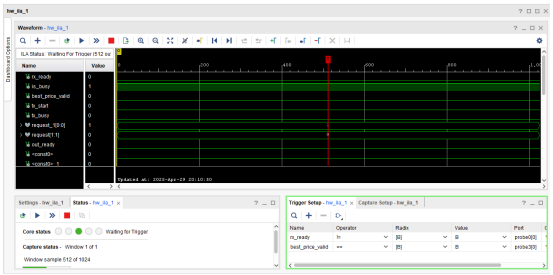

Fig. 23. ILA in Vivado

XI. Applied Concepts and System-Level Features

Our implementation leveraged numerous advanced FPGA design techniques and system-level features to achieve the required performance and reliability:

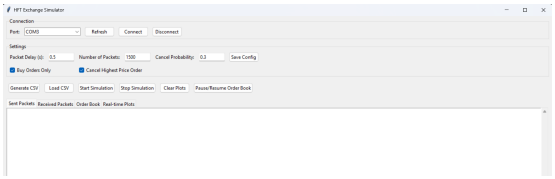

Fig. 24. Python GUI initial page

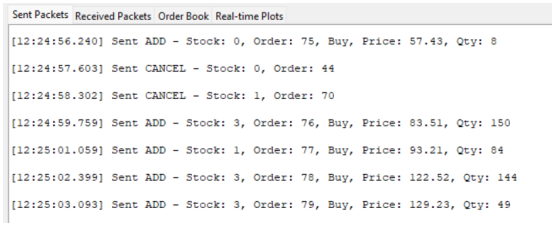

Fig. 25. Sent messages to FPGA from market Simulator (Python GUI)

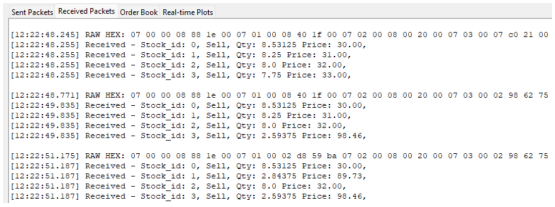

Fig. 26. Received messages from FPGA to market Simulator (Python GUI)

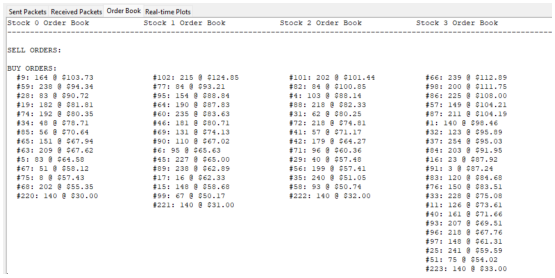

Fig. 27. Live order book status (Python GUI)

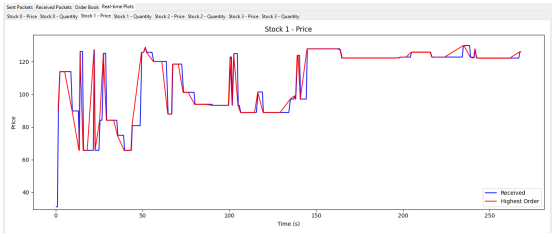

Fig. 28. Live price plots (Python GUI)

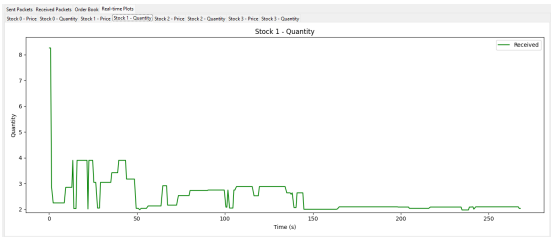

Fig. 29. Live quantity plots (Python GUI)

A. Verification and Debugging Features

- **ILA (Integrated Logic Analyzer)**: Enabled real-time signal monitoring inside the FPGA
- **VIO (Virtual Input/Output)**: Allowed dynamic parameter adjustment without reconfiguration in runtime
- **Mixed-Language Simulation**: Combined SystemVerilog and Verilog for optimal verification

B. Reliability Features

- **Synchronizers**: Implemented for all clock domain crossings
- **Reset Synchronization**: Ensured reliable system initialization
- **Exception Handling**: Detected and managed invalid message cases
- **Message Rejection**: Properly handled malformed input data

C. Performance Features

- **Pipelining**: Used throughout the design to enhance throughput
- **Parallelism**: Implemented for multi-stock processing
- **BRAM Optimization**: Efficiently utilized block RAM resources
- **DSP Blocks**: Leveraged for accelerated computation

D. Design Methodology Features

- **Parametric Design**: Enabled scaling to different numbers of stocks
- **Modular Architecture**: Facilitated independent testing and integration
- **Ready-Busy Handshaking**: Ensured reliable data transfer between modules
- **Fixed-Point Precision**: Used different precision across modules as appropriate

These features collectively contributed to a robust, high-performance system capable of meeting the demanding requirements of high-frequency trading applications.

XII. Future Work

While our current implementation demonstrates the viability of FPGA-based HFT systems, several enhancements could further improve performance and functionality:

1) **Ethernet Interface**: Implementing a full network stack including the Network Layer would significantly reduce communication latency
2) **Buy/Sell Support**: Adding capability to handle both buy and sell orders would enable more sophisticated trading strategies
3) **Exchange Format Support**: Extending the parser to support additional exchange data formats (e.g., NSE/BSE) would increase versatility
4) **Additional Message Types**: Supporting more complex order types including cancellations, modifications, and other actions
5) **Scaling to More Stocks**: Increasing the number of simultaneously tracked securities
6) **Advanced Trading Strategies**: Implementing additional algorithmic approaches, including the fully optimized Strategy 3
7) **Latency Reduction**: Further optimizing critical paths in the hardware design
8) **Adaptive Strategy Adjustments**: Adding AI-based mechanisms for strategy parameter tuning

These enhancements would build upon the foundation established in this work to create an even more capable and competitive trading platform which will be much closer to the deployable model in the live network.

XIII. Conclusion

This paper has presented the design and implementation of an FPGA-based High-Frequency Trading system capable of processing market data and executing trades with microsecond-level latency. Our implementation on a modest Artix-7 FPGA demonstrates that significant performance advantages can be achieved even with entry-level hardware.

Key achievements of our work include:

- A complete trading pipeline from market data reception to trade execution
- Efficient order book implementation supporting multiple stocks
- Multiple trading strategies with varying levels of sophistication
- Processing latency of approximately 3.2 μs per order
- Comprehensive market simulation environment for testing

The performance results validate the advantages of FPGA technology for financial applications, showing deterministic, ultra-low latency response times that would be difficult to achieve with traditional software implementations. Despite using UART as the communication interface, which introduces significant bottlenecks, the core processing components demonstrated exceptional performance.

Furthermore, our implementation highlights both the potential and challenges of implementing sophisticated financial algorithms in hardware. While simpler strategies were successfully deployed, more complex approaches like the minimum variance portfolio strategy exposed the limits of what can be achieved on modest FPGA platforms without specialized optimization.

Future work will focus on communication interface improvements, support for additional order types and exchanges,

and implementation of more advanced trading strategies. As FPGA technology continues to advance and become more accessible, we anticipate growing adoption of hardware-accelerated solutions in the financial industry and beyond.

REFERENCES

[1] N. Kahssay, E. Kahssay, and Z. Wang, "An HFT (High Frequency Trading) Accelerator," *IEEE Transactions on FPGA Implementation*, 2019.

[2] C. Leber, B. Geib, and H. Litz, "High frequency trading acceleration using FPGAs," in *2011 21st International Conference on Field Programmable Logic and Applications*, 2011, pp. 317-322.

[3] J. W. Lockwood et al., "Implementing Ultra Low Latency Data Center Services with Programmable Logic," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 68-77.

[4] S. Jain, "High Frequency Trading: A Cutting-Edge Application of FPGA," *IEEE Talk on HFT*, 2025.

[5] NASDAQ, "NASDAQ TotalView-ITCH 5.0 Specification," Technical Document, 2020.

[6] Xilinx Inc., "Artix-7 FPGA Family Data Sheet," DS181 (v1.10), 2019.

[7] Digilent Inc., "Basys 3 FPGA Board Reference Manual," 2019.

INDIVIDUAL CONTRIBUTIONS

The project was executed through collaborative effort, with each team member focusing on distinct modules. Jaideep was responsible for implementing the UART RX and TX communication modules and developed a Python-based Market Exchange Simulator for functional testing. Shubham contributed by designing and verifying the Parser and Deparser modules, which handled command encoding and decoding. Rakesh developed the core Order Book logic to manage and match buy/sell orders efficiently. Pavan implemented the Trading Algorithms, including decision-making strategies for order placement. Each contribution was integral to achieving a modular, functional, and verifiable trading system.