Digital System Design with FPGAs (E3-231) Course Project

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

Team Members

Andavarapu Rakesh, Shubham Lanjewar, Alamuru Pavan Kumar Reddy, Jaideep M

Guided by: Prof. Debayan Das

April 30, 2025

Department of Electronic Systems Engineering

INDIAN INSTITUTE OF SCIENCE

भारतीय विज्ञान संस्थान

Outline

Motivation

What is HFT?

- High-Frequency Trading involves executing large volume of orders at extremely high speeds
- Leverages algorithms to analyze multiple markets
- **Timeframes**: naoseconds to microseconds
- Capitalizes on small price discrepancies

Why FPGA for HFT?

- **Deterministic Latency**: Consistent execution times
- **Ultra-Low Latency**: Hardware-level implementation
- **Parallelism**: Multiple simultaneous operations
- **Customization**: Tailored architecture
- **Reduced Jitter**: Less timing variation
- **Direct Market Access**: Interface with exchange feeds

Network System Architecture

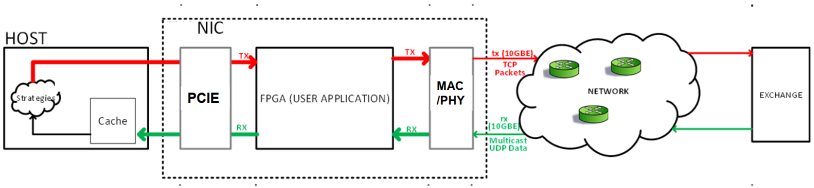


Figure: HFT Network Communication Flow

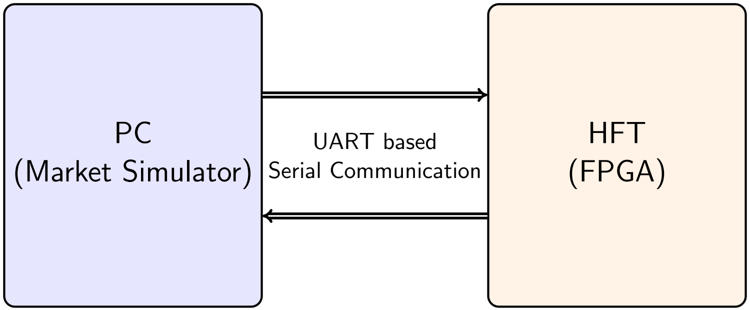
- **Data Flow:** TCP packets for orders (TX), Multicast UDP for market data (RX)
- **Hardware:** FPGA user application connected via PCIe to host with 10GbE MAC/PHY

System Level Assumptions

- **Order Processing:**
  - Buy orders only (no sell orders)
  - Supporting only four stock IDs
- **System Design:**
  - No internal feedback mechanism
  - Fixed capital investment model
- **Implementation Scope:**
  - Application-level functionality
  - Order book limited to 256 entries per stock
  - No order cancellation handling

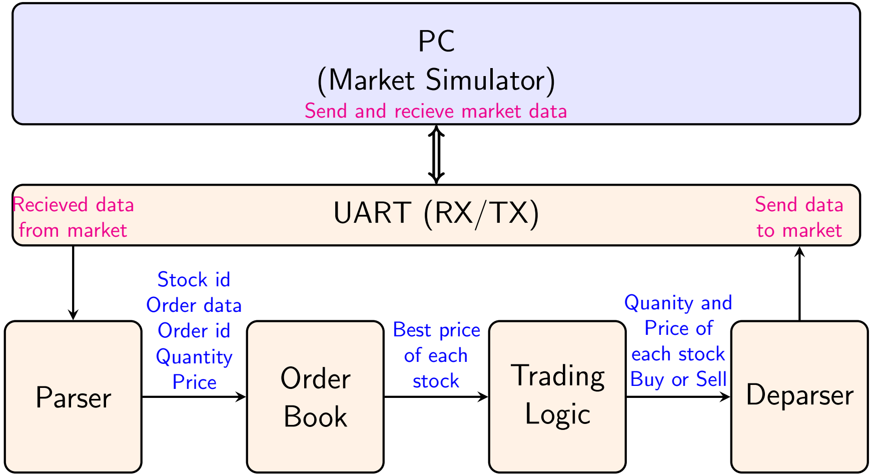
System Architecture

Level 0



System Architecture

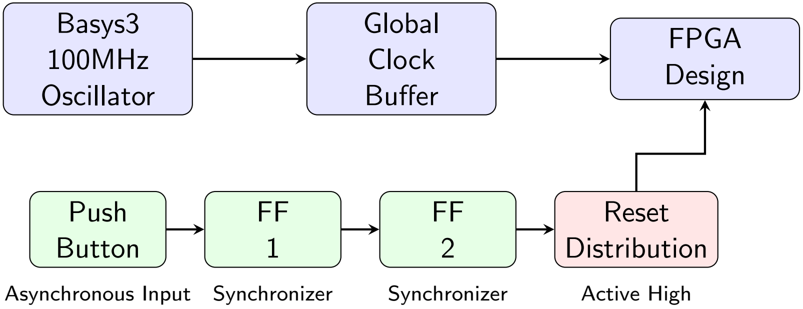
Level 1



Target Specifications

Parameter	**Target Specification**
Target Platform	Basys3 (Artix-7 35 T -1 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

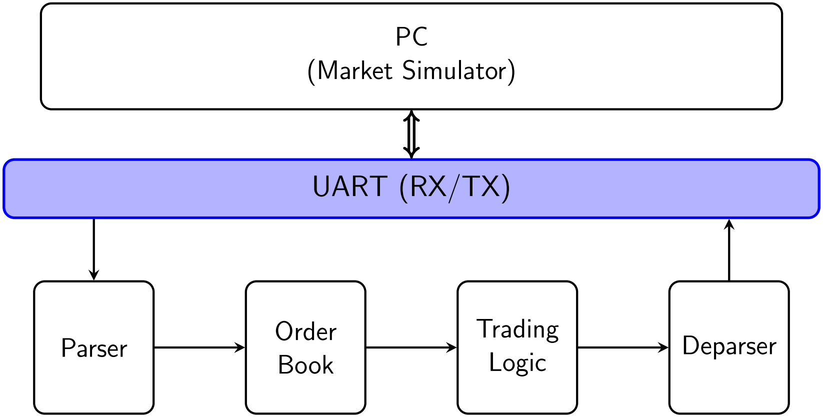
Clock and Reset Scheme



Key Features:

- Global synchronous reset (active-high)
- Two-stage synchronization for async inputs

UART


PC
(Market Simulator)
UART (RX/TX)
Parser
Order Book
Trading Logic
Deparser

UART Communication Module - Overview

Function:

- Serial communication interface
- Handles data reception and transmission
- Deserializes incoming data
- Serializes outgoing data

Technical Details:

- 8-N-1 configuration
- Baud rate: 115200 bps
- 100 MHz system clock
- Parameterized design

Common Features:

- One-hot state encoding
- Maximum throughput: 11.5 KB/s
- Two-process state machine model

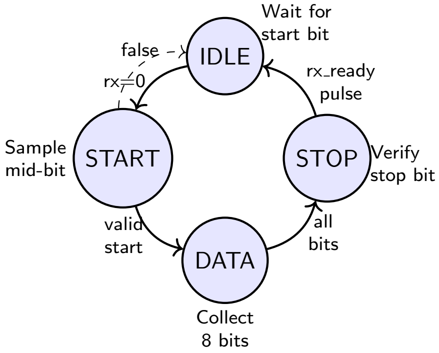
UART Receiver Module

RX Module Features:

- Active high reset signal
- Handshake signals:
  - rx_ready: Pulses when byte received
  - rx_data: 8-bit received data
- Triple-register synchronization
- Majority filter for noise immunity

RX FSM States:

- IDLE, START, DATA, STOP
- Middle-of-bit sampling
- False start detection



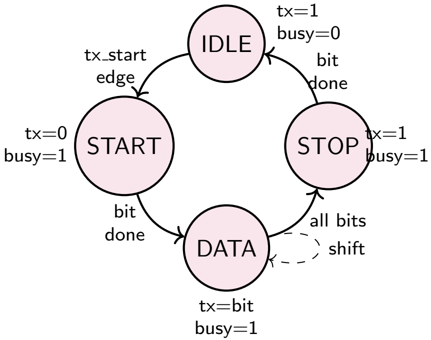
UART Transmitter Module

TX Module Features:

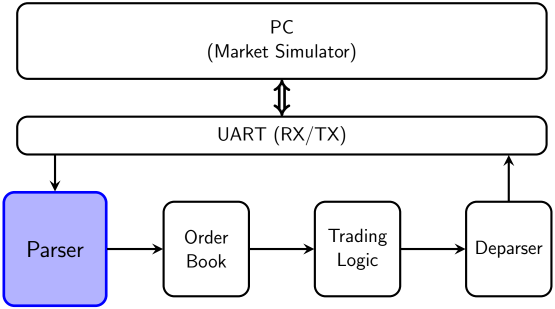
- Active high reset signal
- Handshake signals:
  - tx_start: Initiates transmission
  - tx_busy: Indicates ongoing transmission
- Edge detection for transmission control
- IOB register for timing closure

TX FSM States:

- IDLE, START, DATA, STOP
- LSB-first data transmission



Parser



- The Parser Module extracts and processes NASDAQ ITCH protocol messages
- Since we are using UART (slow) this module is bottleneck in the pipelining.

Goal: To extract relevant fields from NASDAQ ITCH protocol messages for order book construction and trading decisions.

Parser Module - Functions

Primary Functions:

- Receive byte data from UART RX
- Process data & extract fields
- Send data to Order Book module

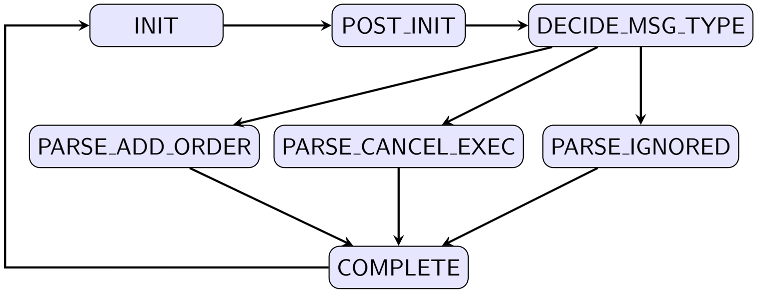
Internal Working:

- Receive bytes when UART ready
- Identify message type
- Read data for required length
- Buffer data in registers
- Extract fields when complete
- Signal Order Book to process

Supported NASDAQ ITCH Messages:

- ADD_ORDER
- EXECUTE_ORDER
- CANCEL_ORDER

Parser Module - Finite State Machine



- Parser implements a 7-state FSM for message processing
- State transitions depend on message type identification
- Complete state signals the Order Book when parsing is finished

Parser Module - ADD_ORDER Message Structure

ADD_ORDER Register Structure (37 bytes total = 296 bits)

Price (4B)	STOCK_SYM (8B)	SHARES (4B)	B/S (1B)	ORDER_REF (8B)	TIMESTAMP (6B)	TRACK (2B)	STOCK (2B)	MSG (1B)	L (1B)

Parameters: LENGTH=1, MESSAGE_TYPE=1, STOCK_LOCATE=2, TRACKING_NUMBER=2, TIMESTAMP=6, ORDER_REF_NUM=8, BUY_SELL_IND=1, SHARES=4, STOCK=8, PRICE=4

- ADD_ORDER message requires 37 bytes $\times$ 8 = 296 clock cycles to process
- Parser extracts all fields necessary for order book construction
- Fields include price, quantity, order id, stock identifier

Parser Module - EXECUTE & CANCEL Message Structure

CANCEL_ORDER (24 bytes) **EXECUTE_ORDER (24 bytes)**

EXECUTED_SHARES (4B)	ORDER_REF_NUM (8B)	TIMESTAMP (6B)	TRACK (2B)	STOCK (2B)	MSG (1B)	L (1B)

Parameters: LENGTH=1, MESSAGE_TYPE=1, STOCK_LOCATE=2, TRACKING_NUMBER=2,
TIMESTAMP=6, ORDER_REF_NUM=8, SHARES=4

- Both EXECUTE_ORDER and CANCEL_ORDER messages require 24 bytes $\times$ 8 $=$ 192 clock cycles
- Similar structure allows for efficient parsing in the same state

Parser Module - Performance Metrics

Processing Time:

- ADD_ORDER: 37 bytes $\times$ 8 = 296 clock cycles
- EXECUTE_ORDER: 24 bytes $\times$ 8 = 192 clock cycles
- CANCEL_ORDER: 24 bytes $\times$ 8 = 192 clock cycles

Throughput Limitations:

- Limited by UART baud rate
- One byte per 8 clock cycles
- Same module structure can be used to process other message types by adding one state

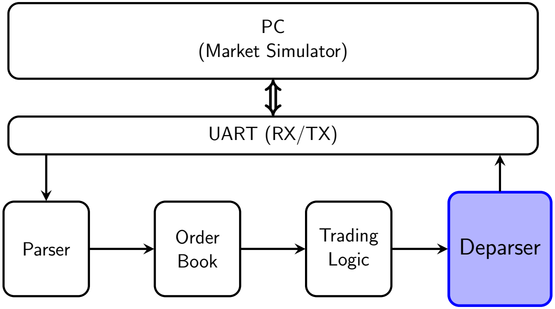
Future Optimizations:

- Replace UART with Ethernet interface
- Reducing number of clock cycles since this module is a bottleneck in the pipelining
- Supporting all other message types

Parser Module - Summary

- Parser module successfully processes NASDAQ ITCH protocol messages
- Supports ADD_ORDER, EXECUTE_ORDER, and CANCEL_ORDER message types
- Same module can be used to process other messages
- Current implementation limited by UART interface
- Future work includes Ethernet integration for improved performance

Deparser Module



- The Deparser Module frames custom messages to send back to the Python simulator
- Creates output messages based on trading logic commands
- Uses a custom message format rather than ITCH protocol

Goal: To format trading decisions into transmittable messages for the python market simulator.

Deparser Module - Functions

Primary Functions:

- Receive trading commands
- Format data into custom messages
- Prepare byte data for UART TX
- Control message flow to simulator

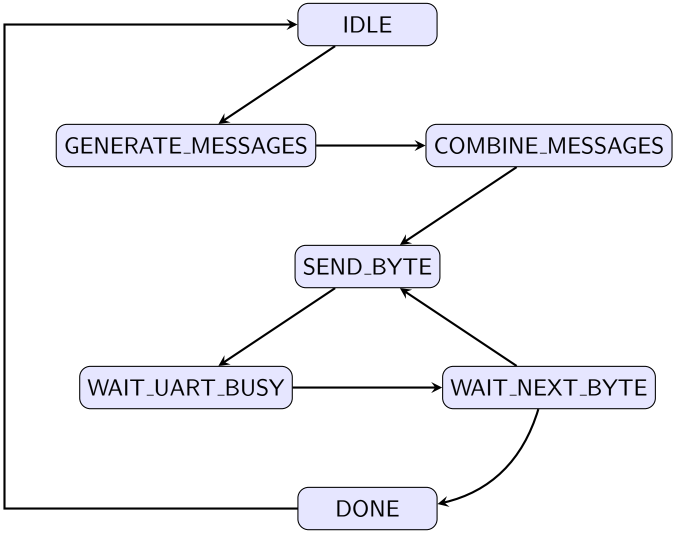
Message Design Rationale:

- Custom format avoids message avalanche
- Ensures complete message transmission before next input processing

Key Consideration:

Using the ITCH protocol would create an avalanche of messages (4 ADD_ORDERS triggered for each received ADD_ORDER)

Deparser Module - Finite State Machine



Deparser Module - Custom Message Structure

Custom Message Structure (7 bytes total)

LENGTH	STOCK_LOCATE	BUY/SELL_IND	SHARES	PRICE
(1B)	(1B)	(1B)	(2B)	(2B)

- Custom message format designed for simplicity and efficiency
- Total message size: 7 bytes (compared to 24-37 bytes for ITCH messages)
- Contains essential order information (buy/sell indicator, shares, price, etc.)

Message Processing:

- Messages are generated in the GENERATE_MESSAGES state based on trading logic output
- Combined and prepared for transmission in COMBINE_MESSAGES state
- Transmitted byte-by-byte through the UART interface

Deparser Module - Message Flow and Optimization

Message Flow Control:

- Wait for trading logic trigger
- Generate appropriate message response
- Handle UART TX handshaking
- Ensure complete message transmission
- Return to idle for next command

Design Optimizations:

- Custom protocol vs. ITCH protocol
- Simplified message format
- State-based transmission control
- Efficient UART utilization

Key Challenge Addressed:

- Using standard ITCH protocol would create message avalanche
- Custom format ensures one complete response per input message
- Critical for system stability with UART bandwidth limitations

Deparser Module - Performance Considerations

Current Limitations:

- UART transmission speed bottleneck
- One byte per multiple clock cycles
- Must complete message before next input
- Limited by message size and format

Real-World Scenario Differences:

- In production, Ethernet would be used
- Higher bandwidth would allow standard protocols
- Message avalanche less problematic
- Lower latency transmission

Future Optimizations:

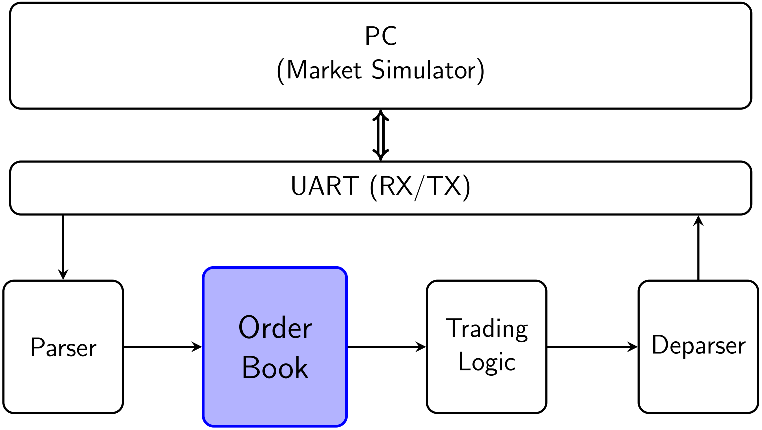
- Replace UART with Ethernet interface
- Potentially adopt standard protocols
- Improve message throughput and latency
- Add support for more complex order types

Deparser Module - Summary

- Deparser module successfully formats trading decisions into transmittable messages
- Uses custom message format rather than ITCH protocol to avoid message avalanche
- Implements 7-state FSM for efficient message generation and transmission
- Handles UART interface with proper handshaking
- Design optimized for testbench environment with UART limitations

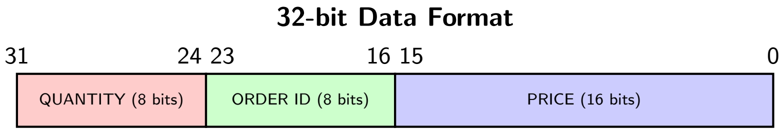
Key Design Decision: Custom protocol prioritizes system stability and prevents feedback loops in the constrained UART environment.

Order Book

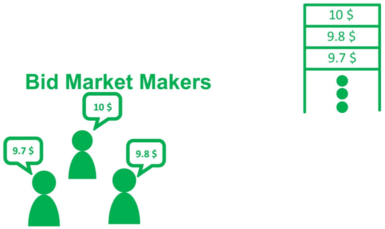

PC
(Market Simulator)
UART (RX/TX)
Parser
Order Book
Trading Logic
Deparser

Order Book

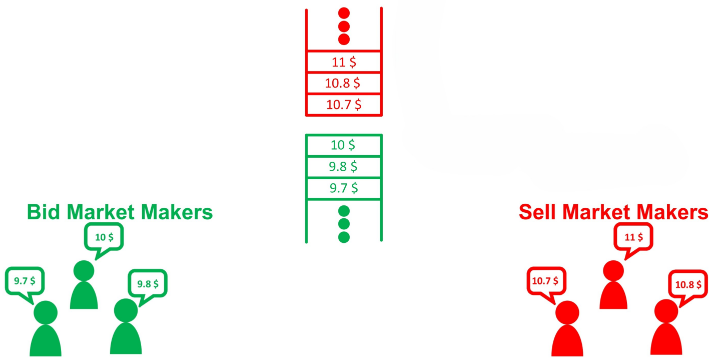
- Want FPGA to maintain the current state of the market – a list of best prices for buying (bid) / selling (asks) for multiple stocks.
- Receives updates from parser about new orders, and cancels.
- Continuously provide the best available stock price and generate a valid trigger signal for the trading logic.
- Needs to be low latency and have low space storage (<500 kb) to fit in the constraints of the nexus FPGA.



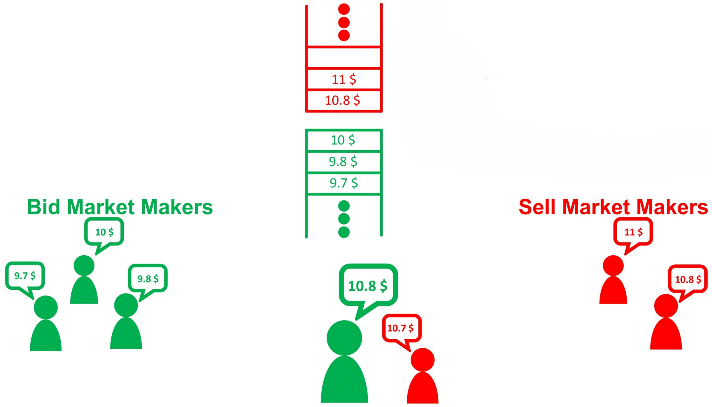
Order Book



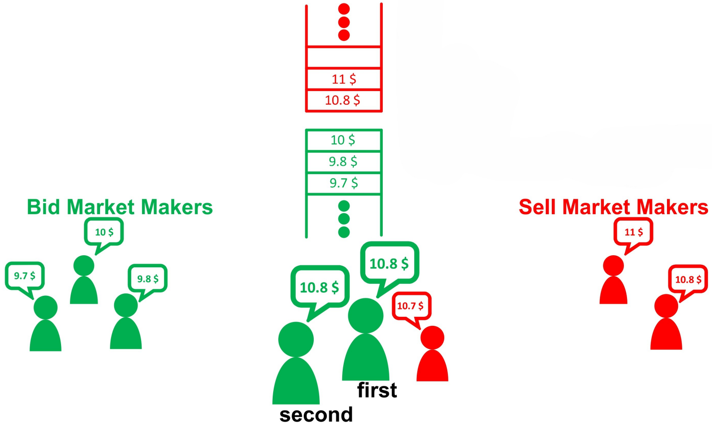
Order Book


11 $
10.8 $
10.7 $
10 $
9.8 $
9.7 $
Bid Market Makers
10 $
9.7 $
9.8 $
Sell Market Makers
11 $
10.7 $
10.8 $

Order Book

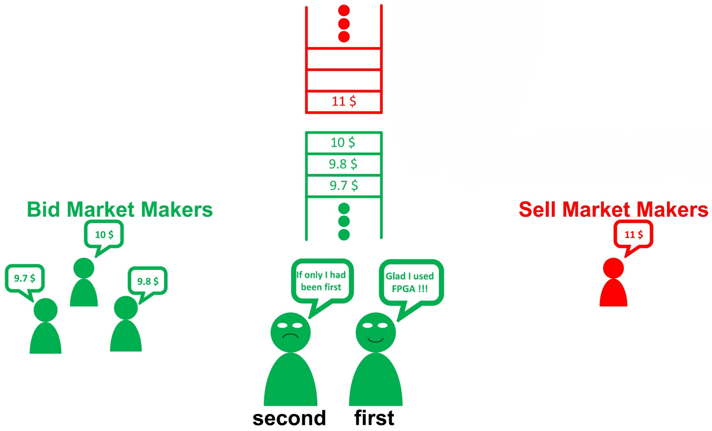

11 $
10.8 $
10 $
9.8 $
9.7 $
Bid Market Makers
10 $
9.7 $
9.8 $
Sell Market Makers
11 $
10.8 $
10.8 $
10.7 $

Order Book

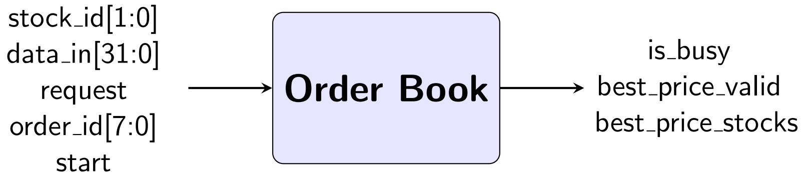


11 $
10.8 $
10 $
9.8 $
9.7 $
Bid Market Makers
10 $
9.7 $
9.8 $
10.8 $
10.8 $
10.7 $
first
second
Sell Market Makers
11 $
10.8 $

Order Book

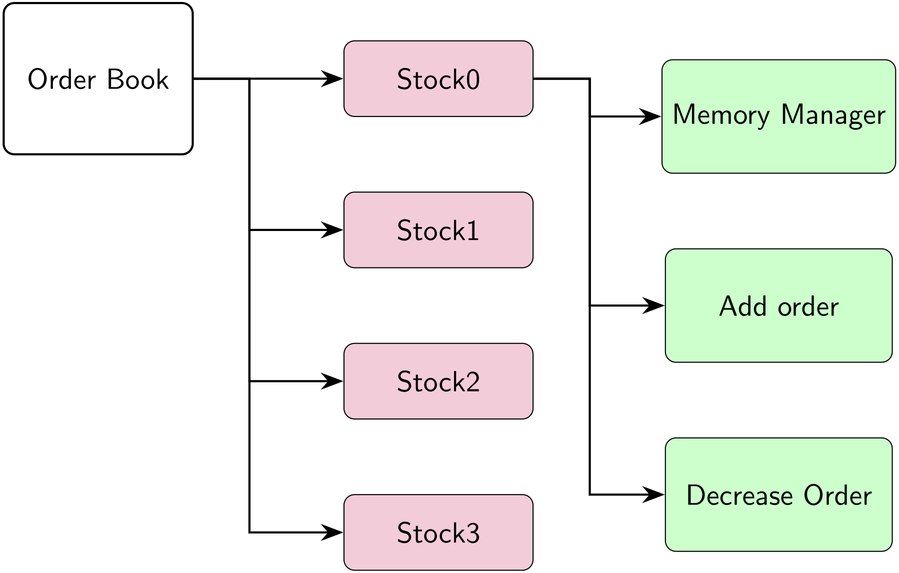

11 $
10 $
9.8 $
9.7 $
Bid Market Makers
10 $
9.7 $
9.8 $
If only I had been first
Glad I used FPGA !!!
second
first
Sell Market Makers
11 $

Order Book

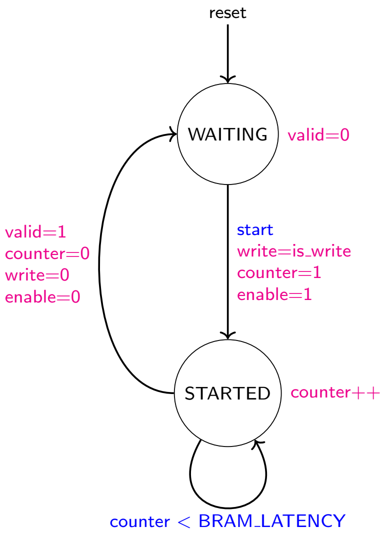


- stock_id[1:0] : stock to be operation performed
- data_in[31:0] : 32 bit data to be added in the mentioned format
- request : request to add or cancel
- order_id[7:0] : should be provided for cancel operation
- start : trigerring signal
- is_busy : indicates that the order book is busy
- best_price_valid : will be high when non zero best_price is there in all stocks
- best_price_stocks[3:0][15:0] : best_price of the individual stocks

Order Book


Order Book
Stock0
Stock1
Stock2
Stock3
Memory Manager
Add order
Decrease Order

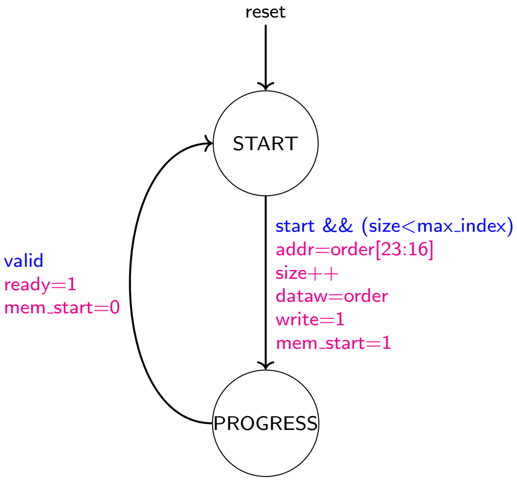
Order Book



Memory Manager FSM

- Initiates read/write operations when start signal activates
- Operation type determined by is_write signal
- Waits BRAM_LATENCY cycles for operation completion
- Asserts valid signal when done, providing handshaking between BRAM and user
- Latency : BRAM LATENCY - 2 clock cycles (O(1))

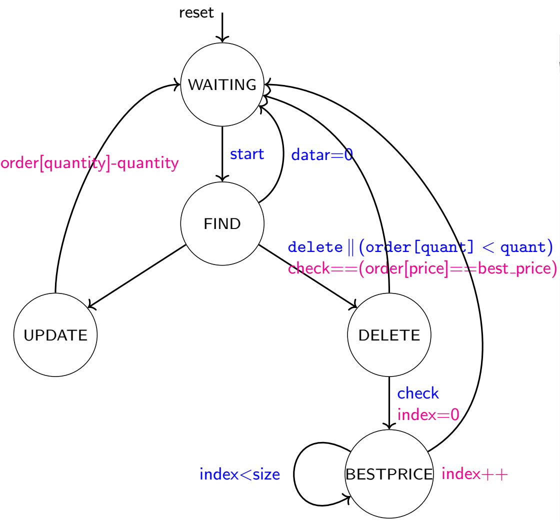
Order Book



Add Order FSM

- Triggers when new 32-bit data needs to be added to BRAM
- START and PROGRESS states manage Memory Manager signal control
- Provides necessary write signals to BRAM
- Monitors all Memory Manager states and I/O signals
- Latency : 8 clock cycles (O(1))

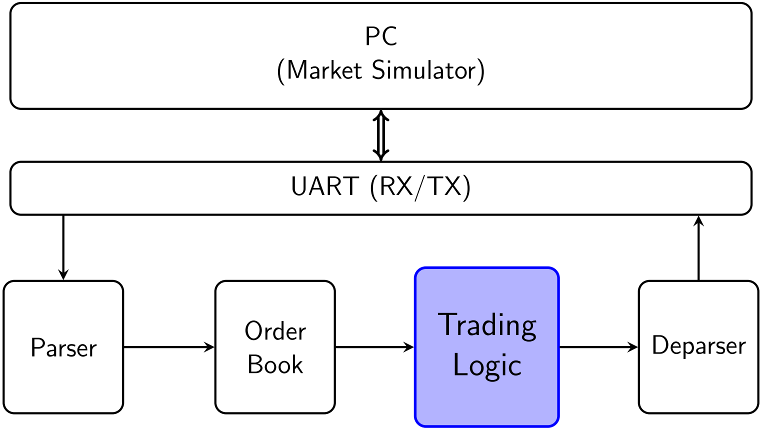
Order Book



Decrease Order FSM

- Activates when trade execution requires order deletion from BRAM
- Searches BRAM for specified order ID, then writes 0 to perform deletion
- Updates best_price when deleted order had the best price value
- Latency : 260 clock cycles (O(n))

Trading Logic

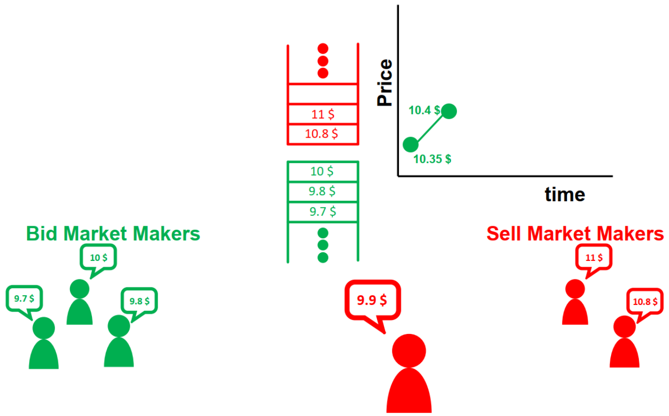


Trading Mechanics

10 $
9.8 $
9.7 $
Bid Market Makers
10 $
9.7 $
9.8 $

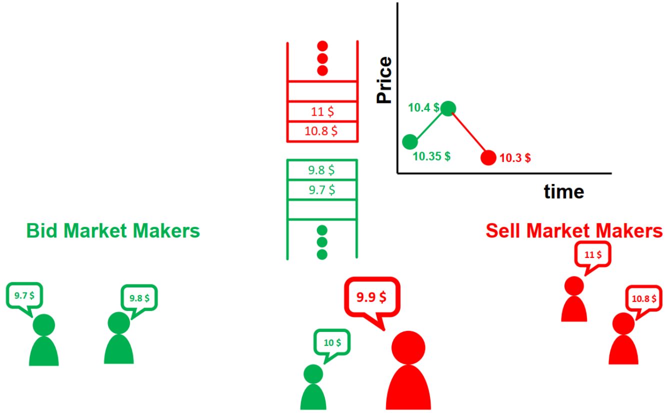
Order books organize buy and sell orders with bids (green) and asks (red) showing available prices and quantities.

Price Consensus



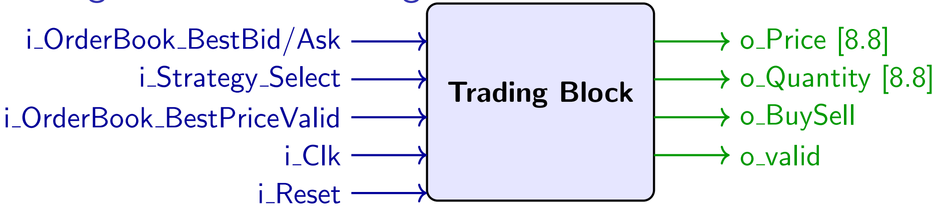
Price consensus forms through the continuous interaction between buyers and sellers in the market.

Market Makers



Market makers provide essential liquidity by placing orders on both sides of the book, profiting from the bid-ask spread.

Trading Module Block Diagram



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

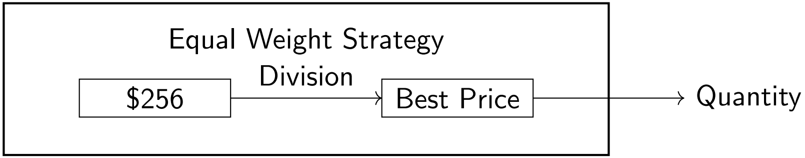
Signal Control and Visualization

- **VIO Interface:** User selects the trading strategy in real-time; all key IO ports are exposed for monitoring market activity.
- **ILA Debugging:** Captures handshaking signals (‘valid‘, ‘ready‘, ‘ack‘, ‘done‘) across design levels for debugging and analysis.

Equal Weight Strategy

Strategy Overview

- Allocate funds equally among N stocks
- Total capital: \$1024
- 4 stocks = \$256 per stock
- Quantity = \$256/Best Price
- Execute only sell orders in this example



Example:

- Best Price = \$10.4
- Quantity = \$256/\$10.4 = 24.615 units

Division Implementation in Hardware

Challenge: Division in Hardware

- Division is a complex operation in hardware
- High resource utilization
- Multiple clock cycles to complete
- Requires special handling for timing

Solution: AXI-based Division IP Core

- Use Xilinx Division Generator IP core
- AXI-Stream interface for data flow
- Pipeline structure with fixed latency
- Simplified integration with trading logic

AXI Division IP Core Signals

AXI-Stream Interface Signals

- `aclk` — Clock input
- `s_axis_divisor_tvalid` — Divisor valid signal
- `s_axis_divisor_tdata` — Divisor data (Best Price)
- `s_axis_dividend_tvalid` — Dividend valid signal
- `s_axis_dividend_tdata` — Dividend data (e.g., $256)
- `m_axis_dout_tvalid` — Output valid signal
- `m_axis_dout_tdata` — Output data (Quantity)

Timing Notes

- Input operands are validated via `tvalid` handshaking
- Output becomes valid when `m_axis_dout_tvalid` is asserted
- Typical latency: 20–30 clock cycles
- Pipelined: New input accepted every clock cycle

Strategy 2: Momentum-Based (Moving Average)

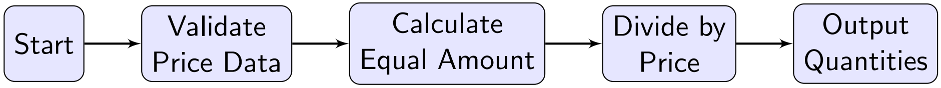
Core Idea

- Track a **moving average of depth 8** for each stock (hardware-friendly power-of-2).
- Maintain a rolling sum history: `Sum / 8 = Moving Average`.
- **Buy if:** Current Price $<$ Moving Average (Undervalued).
- **Sell if:** Current Price $>$ Moving Average (Overvalued).

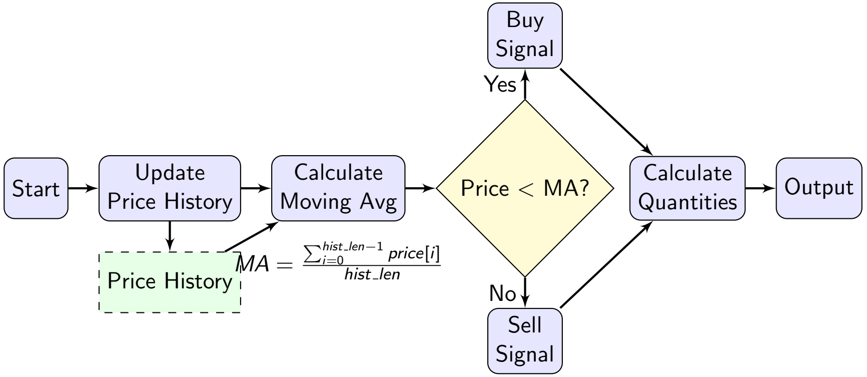
Implementation Highlights

- Use a history buffer of size $8 \times N$ (N = number of stocks).
- Right shift ($>>$ 3) replaces division for efficient averaging.
- Logic:
  - `if (BestPrice < MovingAvg) → Buy`
  - `else → Sell`
- Simple decision logic per stock based on trend.

Combining Strategies (1 & 2)

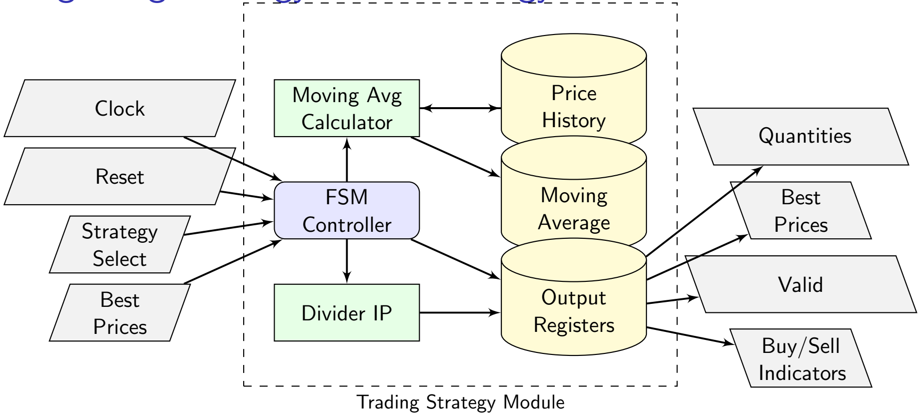


$$amount = \frac{TOTAL}{N} \quad quantity = \frac{amount}{price}$$



$$MA = \frac{\sum_{i=0}^{hist\_len-1} price[i]}{hist\_len}$$

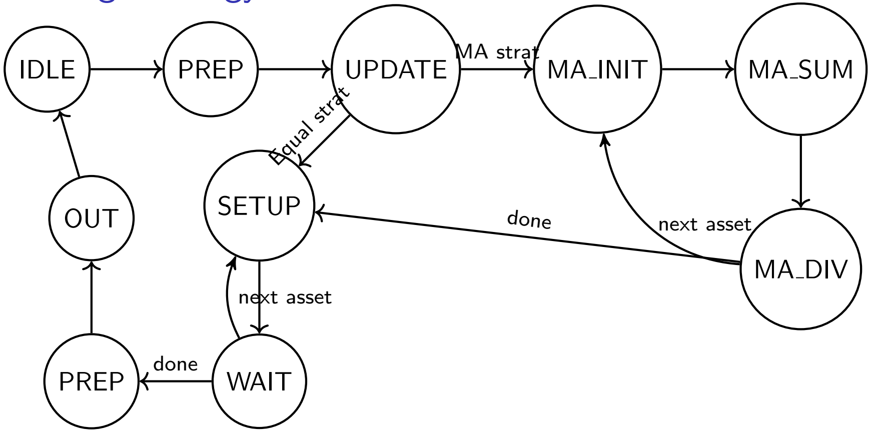
Integrating Strategy 1 and Strategy 2



Combining Strategies (1 & 2)

- **Strategy 1:** Equal weight allocation (fast, reactive).
- **Strategy 2:** Trend-aware, adaptive based on past prices.
- Use a shared Division IP Core for final quantity computation in both.

Trading Strategy FSM



Combining Strategies (1 & 2)

- The Division IP core has a latency of 35 cycles.
- Combined latency $\approx$ 160 cycles.
- Fixed-point arithmetic is consistently used for all operations.

Strategy 3: Minimum Variance Portfolio (MVP) – Key Concepts

Step-by-Step Mathematical Formulation

- **Setup:** For a universe of N stocks, let Σ 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:
  $$\min_{\mathbf{w}} \ \mathbf{w}^T \Sigma \mathbf{w}$$
- **Constraint:** Full investment (weights sum to 1):
  $$\mathbf{w}^T \mathbf{1} = 1$$
- **Closed-form Solution:**
  $$\mathbf{w}_{\text{opt}} = \frac{\Sigma^{-1}\mathbf{1}}{(\Sigma^{-1}\mathbf{1})^T \mathbf{1}}$$

Strategy 3: Dynamic Rebalancing and Hardware Implementation

Interpretation

- Allocations are driven by inverse risk (via Σ^{-1})
- No return assumptions are required — pure risk minimization

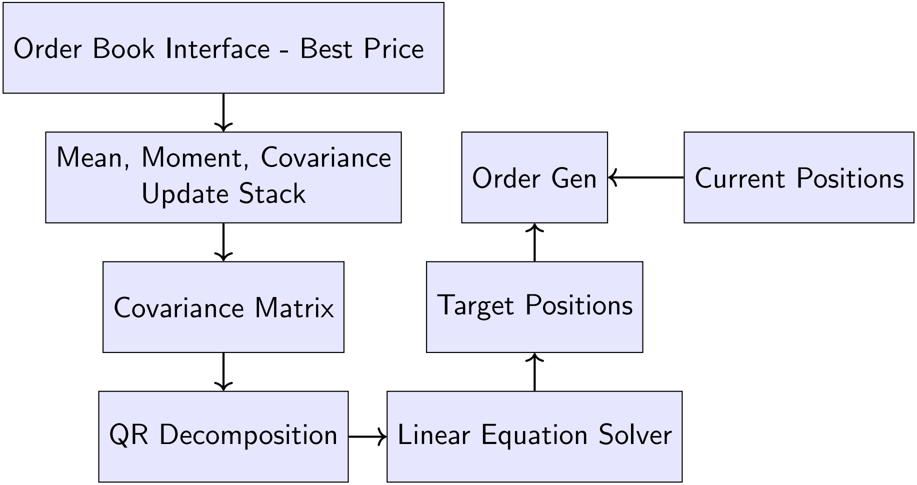
Why Rebalance?

- Σ (covariance) changes over time — market dynamics shift
- Portfolio must adapt: $\mathbf{w}_{\text{opt}}$ varies with updated Σ
- Rebalance periodically with new data to maintain minimum risk profile

Hardware View

- Matrix inversion logic for Σ^{-1} (expensive computation)
- RAM storage for live covariance matrix updates
- Division modules (reused from Strategy 1 & 2)

Strategy 3: Dynamic Rebalancing and Hardware Implementation



Covariance Matrix Computation (K)

System Flow

- **Input:** Prices retrieved from the Order Book Builder (latest p and Previous q)
- **Return Computation:** For each stock, return is computed as:

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

- **Covariance Estimation:**

$$\boldsymbol{K} = \mathbb{E}[XX^T] - \mathbb{E}[X]\mathbb{E}[X]^T$$

  where X is the random vector of stock returns
- **Portfolio Weights (Minimum Variance):**

$$\boldsymbol{w} = \frac{\boldsymbol{K}^{-1}\mathbf{1}}{(\boldsymbol{K}^{-1}\mathbf{1})^T\mathbf{1}}$$

Recursive Update of Covariance Matrix $\boldsymbol{K}$

Expectation Formulas at Step N

$$\mathbb{E}_N[XX^T] = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_i \boldsymbol{x}_i^T \quad (5)$$

$$\mathbb{E}_N[X]\mathbb{E}_N[X]^T = \left(\frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_i\right)\left(\frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_i^T\right) \quad (6)$$

Online Update at each Step $N+1$

$$\boldsymbol{K}_{N+1} = \mathbb{E}_{N+1}[XX^T] - \mathbb{E}_{N+1}[X]\,\mathbb{E}_{N+1}[X]^T \quad (7)$$

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

$$\mathbb{E}_{N+1}[X] = \frac{N\,\mathbb{E}_N[X] + \boldsymbol{x}_{N+1}}{N+1} \quad (9)$$

Matrix Inverse Computation

We need to compute the quantity:

$$w = \frac{K^{-1}\mathbf{1}}{(K^{-1}\mathbf{1})^T\mathbf{1}} \tag{1}$$

Where:

- K is our matrix
- $\mathbf{1}$ is a vector of all ones
- We need to find w efficiently

Key Insight: We don't actually need to invert K explicitly!
Instead of explicitly computing K^{-1}, we can solve:

$$v = K^{-1}\mathbf{1} \Rightarrow Kv = \mathbf{1} \tag{2}$$

Then normalize to get w:

$$w = \frac{v}{v^T\mathbf{1}} = \frac{v}{\sum_i v_i} \tag{3}$$

Challenge: Numerical stability, especially on FPGAs with fixed-point representation

QR Decomposition

To improve numerical stability, we use QR decomposition:

$$K = QR \tag{4}$$

Where:

- Q is an orthogonal matrix ($Q^{-1} = Q^T$)
- R is an upper triangular matrix

Our linear system becomes:

$$Kv = \mathbf{1} \tag{5}$$
$$QRv = \mathbf{1} \tag{6}$$
$$Rv = Q^{-1}\mathbf{1} = Q^T\mathbf{1} \tag{7}$$

Two-step process:

1. Compute $Q^T\mathbf{1}$ by applying QR Decomposition using Givens Rotation
2. Solve the upper triangular system $Rv = Q^T\mathbf{1}$

Advantages:

- More numerically stable than direct inversion
- Upper triangular system is easily solved by back-substitution

Givens Rotations for Upper Triangulation

Goal

Convert a 4×4 matrix to upper triangular form:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \rightarrow \begin{bmatrix} a'_{11} & a'_{12} & a'_{13} & a'_{14} \\ 0 & a'_{22} & a'_{23} & a'_{24} \\ 0 & 0 & a'_{33} & a'_{34} \\ 0 & 0 & 0 & a'_{44} \end{bmatrix} \tag{8}$$

2×2 Example: Convert to Upper Triangular

Given $\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, let $\theta = \arctan\left(\frac{a_{21}}{a_{11}}\right)$.

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

QR Decomposition with Givens Rotations

To perform QR decomposition:

1. Apply $N(N-1)/2$ Givens rotations to make all lower-triangular elements zero
2. Each rotation zeroes one element below the diagonal
3. The composition of all rotations gives Q^{-1}
4. The resulting upper triangular matrix is R

To compute $Q^T\mathbf{1}$:

- Augment matrix K with a column of ones $\mathbf{1}$
- Apply the same rotations to this column as well

Givens Rotations

- Operates on two rows at a time
- Zeroes out a single element
- Ideal for FPGA implementation with CORDIC

FPGA Implementation of Givens Rotations

Implementation using CORDIC

- One CORDIC block for computing $\arctan(A[j,k], A[i,k])$
- Two CORDIC blocks for performing the rotations
- **Pipelined implementation:** One rotator for all N elements in a row

Parallelization

- Two Givens rotations performed in parallel if they don't share rows
- Algorithm is divided into stages with multiple parallel rotations

CORDIC Implementation Details

CORDIC allows efficient computation of:

- Trigonometric functions (sine, cosine, arctan)
- Vector rotation

Algorithm Summary

Algorithm 1 QR Decomposition with Givens Rotations

```
 1: Augment $K$ with a column of ones: $[K|\mathbf{1}]$
 2: for $k = 1$ to $N - 1$ do                                  ▷ Process each column
 3:     for $j = k + 1$ to $N$ do                   ▷ Process each row below diagonal
 4:         Compute $\theta = \arctan(A[j,k], A[k,k])$
 5:         for $i = k$ to $N + 1$ do                      ▷ Include augmented column
 6:             Apply rotation to $(A[k,i], A[j,i])$ using $\theta$
 7:         end for
 8:     end for
 9: end for
10: $R \leftarrow$ upper triangular part of transformed matrix
11: $Q^T\mathbf{1} \leftarrow$ transformed augmented column
12: Solve $Rv = Q^T\mathbf{1}$ by back-substitution
13: $w \leftarrow v / \sum_i v_i$
```

Order Generation: Mathematical Model

$$\text{For each asset } i: \tag{9}$$

$$\Delta_i = \text{target}_i - \text{position}_i \tag{10}$$

$$\text{shares}_i = \frac{\Delta_i}{\text{price}_i} \tag{11}$$

$$\text{direction}_i = \begin{cases} \text{BUY}, & \text{if shares}_i > 0 \\ \text{SELL}, & \text{otherwise} \end{cases} \tag{12}$$

$$\text{quantity}_i = |\text{shares}_i| \times \text{account\_value} \tag{13}$$

$$\tag{14}$$

Hardware Implementation:

- Fixed-point arithmetic for precision control
- Parallel processing across all assets
- $O(1)$ execution time regardless of portfolio size

Trading System Summary

Strategy Overview

Strategy	Post-Synth Sim	Post-Impl Timing	FPGA Impl
Strategy 1	✓	✓	✓
Strategy 2	✓	✓	✓
Strategy 3	✓	✗	✗

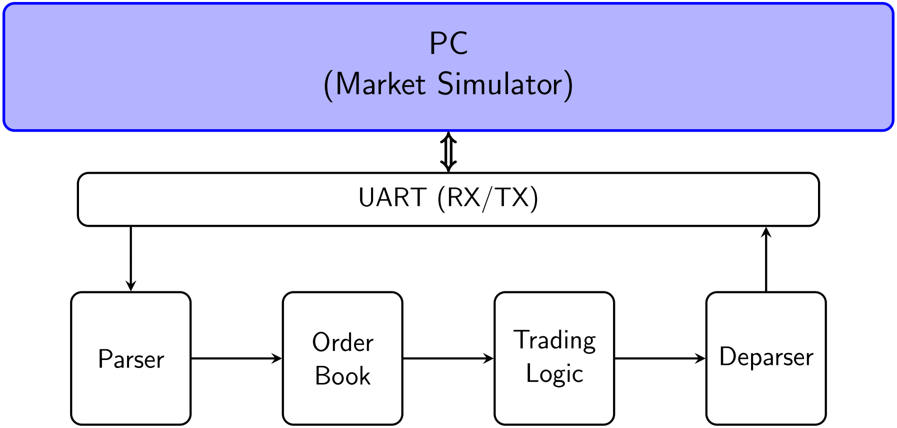
Notes

- Post-implementation simulation failed for Strategy 3 on Artix FPGA despite STA pass.

Summary:

- **Strategy 1 & 2:** End-to-end HFT system implemented and verified.
- **Strategy 3:** Verified till simulation; requires further timing analysis.

Python Market simulator



Python Exchange Simulator

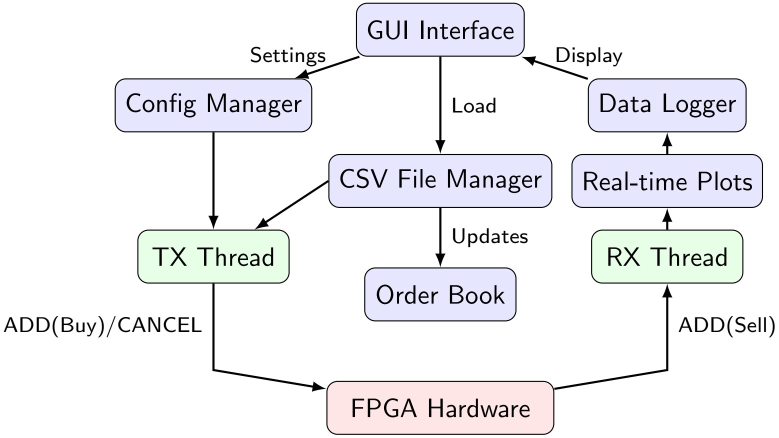
What It Does

- Generates market data (ADD:- Buy and Sell, CANCEL:- Execute the order) for testing the trading system
- Transmits data packets to FPGA via UART and receives trading responses
- Visualizes market data and system performance in real-time
- Supports configuration through GUI and config files

Key Features

- Multi-threaded with concurrent TX and RX processing
- Real-time price/quantity visualization for each stock
- Order book tracking with buy/sell order management (mirror of FPGA's Order Book)
- CSV import/export for test scenarios
- Configurable order generation parameters

Software Architecture



Exchange Simulator maintains local order book based on CSV data

Implementation Results I: Resource & Performance

Basys 3 FPGA Resource Utilization

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

Performance Metrics

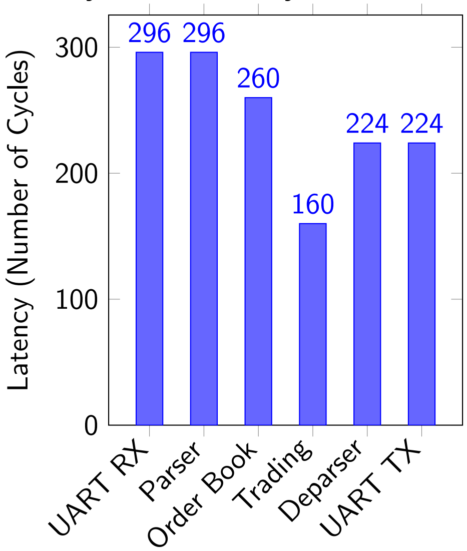
Metric	Value
F_{max}	104 MHz
Throughput	400 orders/sec
Latency (avg)	3.2 μs
Power Consumption	0.135 W
UART Baud Rate	115200 bps

Implementation Results II: Latency Analysis

System Optimizations

- Parallel order book updates
- Pipelined market data processing
- Optimized fixed-point calculations
- Memory addressing tuned for trading patterns

Latency Breakdown by Module:



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 16 K depth with Trigger Control

Key Lessons

- **Interface Definition**: Critical between modules – document interface control, timing diagrams, and dataflow correctly
- **Handshaking Protocols**: Need careful testing and verification to ensure reliable data transfer
- **Individual Testbenches**: Creating separate testbenches for each module saves significant debugging time
- **Data Format Conversion**: Fixed-point precision requires careful handling at each processing step
- **Endianness**: Big Endian/Little Endian conversions must be managed at each layer
- **Resource Optimization**: FPGA resource vs. performance tradeoffs require iterative optimization

Applied Concepts and System-Level Features

ILA	Synchronizers	Reset Sync
BRAM	VIO	DSP
Ready-Busy Handshake	Exception Handling	Message Rejection
SV & V Mixed	Parametric Design	Modularity
Pipelining	Parallelism	GUI Simulator
Fixed-point Precision	Clock Buffer	VIO Debug
Div AXI IP Core	AXI Interface	Fixed Latency

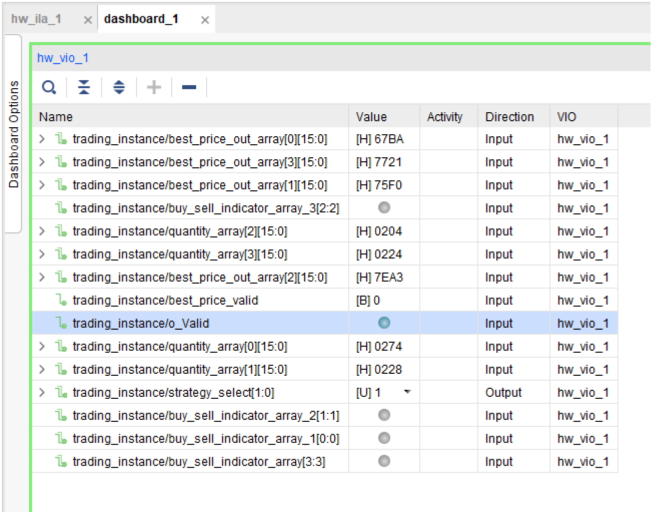
Key Design Features:

- Ready-busy handshake for reliable data transfer
- Mixed SystemVerilog & Verilog implementation
- Parametric design allows scaling to more stocks
- Exception handling for invalid messages
- Pipelined architecture for throughput
- Modular design structure
- Synchronizers for clock domain crossing
- ILA/VIO for real-time debugging
- AXI IP cores for complex operations
- Fixed-point with different precision across modules
- Parallel processing of multiple stocks
- GUI-based exchange simulator for testing
- BRAM for efficient on-chip memory
- DSP blocks for accelerated computation

Future Work

1. **Ethernet Interface:** Full Stack Devolopmemnt including Network Layer
2. **Buy/Sell Support:** Add capability to handle both buy and sell orders
3. **NSE/BSE Formats:** Support additional exchange data formats
4. **More Message Types:** Handle cancellations, modifications, and other actions
5. **More Stock IDs:** Scale system to track additional securities simultaneously
6. **Trading Logics:** Implement various algorithmic strategies including Strategy 3
7. **Reduced Latency:** Further optimize critical paths in hardware design
8. **Software Feedback:** Add AI-based adaptive strategy adjustments

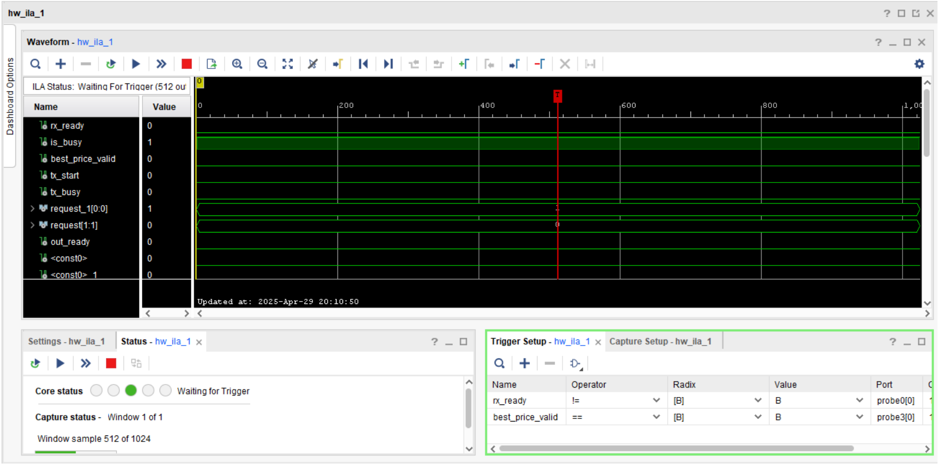
Demonstration - ILA, VIO and Testing GUI



Name	Value	Activity	Direction	VIO
trading_instance/best_price_out_array[0][15:0]	[H] 67BA		Input	hw_vio_1
trading_instance/best_price_out_array[3][15:0]	[H] 7721		Input	hw_vio_1
trading_instance/best_price_out_array[1][15:0]	[H] 75F0		Input	hw_vio_1
trading_instance/buy_sell_indicator_array_3[2:2]	●		Input	hw_vio_1
trading_instance/quantity_array[2][15:0]	[H] 0204		Input	hw_vio_1
trading_instance/quantity_array[3][15:0]	[H] 0224		Input	hw_vio_1
trading_instance/best_price_out_array[2][15:0]	[H] 7EA3		Input	hw_vio_1
trading_instance/best_price_valid	[B] 0		Input	hw_vio_1
trading_instance/o_Valid	●		Input	hw_vio_1
trading_instance/quantity_array[0][15:0]	[H] 0274		Input	hw_vio_1
trading_instance/quantity_array[1][15:0]	[H] 0228		Input	hw_vio_1
trading_instance/strategy_select[1:0]	[U] 1		Output	hw_vio_1
trading_instance/buy_sell_indicator_array_2[1:1]	●		Input	hw_vio_1
trading_instance/buy_sell_indicator_array_1[0:0]	●		Input	hw_vio_1
trading_instance/buy_sell_indicator_array[3:3]	●		Input	hw_vio_1

VIO

Demonstration - ILA, VIO and Testing GUI



ILA Status: Waiting For Trigger (512 ou

Name	Value
rx_ready	0
is_busy	1
best_price_valid	0
tx_start	0
tx_busy	0
request_1[0:0]	1
request[1:1]	0
out_ready	0
<const0>	0
<const0>_1	0

```
Updated at: 2025-Apr-29 20:10:50
```

Core status ● Waiting for Trigger

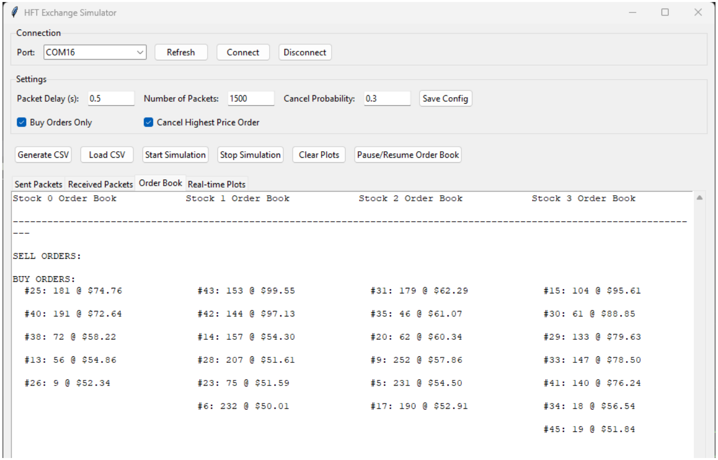
Capture status - Window 1 of 1

Window sample 512 of 1024

Name	Operator	Radix	Value	Port
rx_ready	!=	[B]	B	probe0[0]
best_price_valid	==	[B]	B	probe3[0]

ILA

Demonstration - ILA, VIO and Testing GUI



Port: COM16

Packet Delay (s): 0.5 Number of Packets: 1500 Cancel Probability: 0.3

[✓] Buy Orders Only [✓] Cancel Highest Price Order

```
Stock 0 Order Book            Stock 1 Order Book            Stock 2 Order Book            Stock 3 Order Book

------------------------------------------------------------------------------------------------------------------------
---

SELL ORDERS:

BUY ORDERS:
  #25: 181 @ $74.76             #43: 153 @ $99.55             #31: 179 @ $62.29             #15: 104 @ $95.61

  #40: 191 @ $72.64             #42: 144 @ $97.13             #35: 46 @ $61.07              #30: 61 @ $88.85

  #38: 72 @ $58.22              #14: 157 @ $54.30             #20: 62 @ $60.34              #29: 133 @ $79.63

  #13: 56 @ $54.86              #28: 207 @ $51.61             #9: 252 @ $57.86              #33: 147 @ $78.50

  #26: 9 @ $52.34               #23: 75 @ $51.59              #5: 231 @ $54.50              #41: 140 @ $76.24

                                #6: 232 @ $50.01              #17: 190 @ $52.91             #34: 18 @ $56.54

                                                                                            #45: 19 @ $51.84
```

Testing GUI

Conclusion

Key Outcomes

- Implemented high-performance FPGA-based trading system with microsecond response time
- Developed efficient order book management handling multiple assets simultaneously
- Successfully integrated communication protocols with algorithmic trading strategies
- Applied advanced hardware design techniques including pipelining and fixed-point arithmetic
- Validated system performance using real-world market data scenarios

Individual Contributions

Team Member	Contributions
Jaideep	UART RX and TX, Python-based Market Exchange Simulator
Shubham	Parser and Deparser modules
Rakesh	Order Book implementation
Pavan	Trading Algorithms Implementation

References

[1] NASDAQ, “NASDAQ TotalView-ITCH 5.0 Specification,” Technical Document, 2020.

[2] Kahssay, N., Kahssay, E., & Wang, Z. (2019). “An HFT (High Frequency Trading) Accelerator,” *IEEE Transactions on FPGA Implementation*.

[3] Leber, C., Geib, B., & Litz, H. (2011). “High frequency trading acceleration using FPGAs,” In *2011 21st International Conference on Field Programmable Logic and Applications* (pp. 317-322). IEEE.

[4] Xilinx Inc., “Artix-7 FPGA Family Data Sheet,” DS181 (v1.10), 2019.

[5] Digilent Inc., “Basys 3 FPGA Board Reference Manual,” 2019.

[6] Jain, S. (2025). “High Frequency Trading: A Cutting-Edge Application of FPGA,” *IEEE Talk on HFT*.

Thank You

Questions?