# AI-INTEGRATED FPGA FOR MARKET MAKING IN VOLATILE ENVIRONMENTS

by

Shreejit Verma

A THESIS

Submitted to the Faculty of the Stevens Institute of Technology
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE - FINANCIAL ENGINEERING

Shreejit Verma, Candidate

ADVISORY COMMITTEE

Dragos Bozdog, Chairperson Date

Ionut Florescu, Reader Date

STEVENS INSTITUTE OF TECHNOLOGY
Castle Point on Hudson
Hoboken, NJ 07030
2026

©2026, Shreejit Verma. All rights reserved.

# AI-INTEGRATED FPGA FOR MARKET MAKING IN VOLATILE ENVIRONMENTS

## ABSTRACT

This Master's thesis investigates the integration of artificial intelligence (AI) with field-programmable gate arrays (FPGAs) to develop adaptive market-making strategies tailored for volatile financial environments in a high-frequency trading (HFT) environment. Traditional market-making models often struggle with latency and adaptability during periods of high volatility, leading to increased inventory risk and suboptimal performance. We propose a hybrid system that embeds reinforcement learning (RL) algorithms for dynamic bid-ask spread optimization directly onto FPGA hardware, enabling sub-microsecond inference times while responding to real-time market signals such as order flow imbalances and volatility clusters. Using historical tick-level data from major exchanges and simulated volatile scenarios, the framework is evaluated through metrics including Sharpe ratio, adverse selection mitigation, and latency benchmarks. The expectation is to achieve significant improvements in profitability and risk management compared to software-based RL or standalone FPGA implementations. This work will contribute to a scalable AI-FPGA architecture, highlighting practical deployment challenges, and offering insights for future quant engineering in dynamic markets, with potential applications in NYC's competitive HFT landscape.

**Keywords:** High-Frequency Trading, FPGA Acceleration, Reinforcement Learning, Market Making, Low Latency

Author: Shreejit Verma

Advisor: Dragos Bozdog

Date: January 19, 2026

Program: Financial Engineering

Degree: MASTER OF SCIENCE - FINANCIAL ENGINEERING

## Acknowledgments

To be completed

**Table of Contents**

## List of Tables

## List of Figures

# Chapter 1

## Introduction and System Overview

Market making is central to modern financial markets, as liquidity providers continuously quote bid and ask prices to facilitate trading while managing inventory risk and seeking profit from the spread. The advent of high-frequency trading (HFT) has transformed this role, demanding ultra-low-latency decision-making under dynamic and volatile conditions. Traditional stochastic frameworks, such as the Avellaneda–Stoikov model, offer valuable theoretical foundations but often falter in environments characterized by sudden price swings, order book imbalances, and liquidity shortages. Their reliance on fixed assumptions limits adaptability in real-time, high-volatility settings [Patel et al., 2022, Su et al., 2025]. This has prompted significant interest in more flexible approaches capable of balancing adverse selection, inventory control, and profitability at microsecond scales.

Recent advances in artificial intelligence (AI), particularly reinforcement learning (RL), have provided promising alternatives. RL agents learn adaptive quoting strategies through iterative interactions with either simulated markets or historical order book data, enabling dynamic policies that outperform rule-based methods [Spooner et al., 2020, Kumar et al., 2023]. Deep RL models, including recurrent architectures, have demonstrated enhanced performance in inventory management and order placement, particularly in stressed environments [Vakkilainen, 2023, Jiang et al., 2025]. Moreover, multi-objective RL frameworks have applied Pareto optimization to manage trade-offs between latency, volatility resilience, and slippage reduction [Li et al., 2025]. Despite these advances, most implementations remain software-based, and the computational overhead of deep learning methods introduces latency that undermines

their applicability in production-grade HFT environments.

Parallel to developments in AI, field-programmable gate arrays (FPGAs) have emerged as critical enablers of ultra-low-latency trading. Unlike CPUs and GPUs, FPGAs can process market data feeds, order matching, and risk checks at nanosecond scales, leveraging hardware-level parallelism [Gupta et al., 2024, Litz et al., n.d.]. Studies highlight their advantages in power efficiency, deterministic execution, and real-time data handling, which are essential for HFT infrastructures [Lockwood and Gupte, 2012, Joshua et al., 2025]. Applications range from pre-built IP libraries for networking and protocol parsing [Lockwood and Gupte, 2012] to FPGA-based system-on-chip frameworks for algorithmic execution [Al-Ahmed et al., 2024]. Furthermore, integration of FPGAs with technologies like RDMA has enabled near-zero-copy communication pipelines, further reducing latency across trading networks [Joshua et al., 2025]. Nevertheless, traditional FPGA deployments have typically been limited to deterministic, pre-specified functions, lacking the adaptability required for volatile and evolving market conditions.

The convergence of AI and FPGA technology offers a promising path forward. Recent studies have explored FPGA-accelerated AI inference in trading contexts, enabling models such as XGBoost or neural networks to run at sub-microsecond scales [Napatech, n.d., Systems, 2024]. Dynamic FPGA reconfiguration has been proposed as a means to accommodate evolving RL or deep learning models in real time [Aouini et al., 2025]. Early prototypes of AI-augmented FPGA trading systems suggest significant reductions in end-to-end latencies for market-making strategies [Lee et al., 2023, Kuzmanovic et al., 2023]. Yet, gaps remain: few studies systematically evaluate AI–FPGA integration under extreme volatility, flash-crash conditions, or across multi-asset contexts. Moreover, while industry reports emphasize the growing adoption of FPGA–AI platforms for finance [MarketsandMarkets, 2025a, Semiconductor,

2025], rigorous academic investigations into resilience, scalability, and security trade-offs remain limited.

This thesis aims to bridge these gaps by developing an integrated reinforcement learning–FPGA framework for market making under high-volatility scenarios. By combining RL's adaptive decision-making capabilities with FPGA's hardware-level acceleration, the proposed system seeks to reduce latency bottlenecks, improve inventory risk management, and enhance robustness in stressed market conditions. In doing so, it builds upon the strengths of prior RL and FPGA research while addressing limitations in real-world deployment contexts.

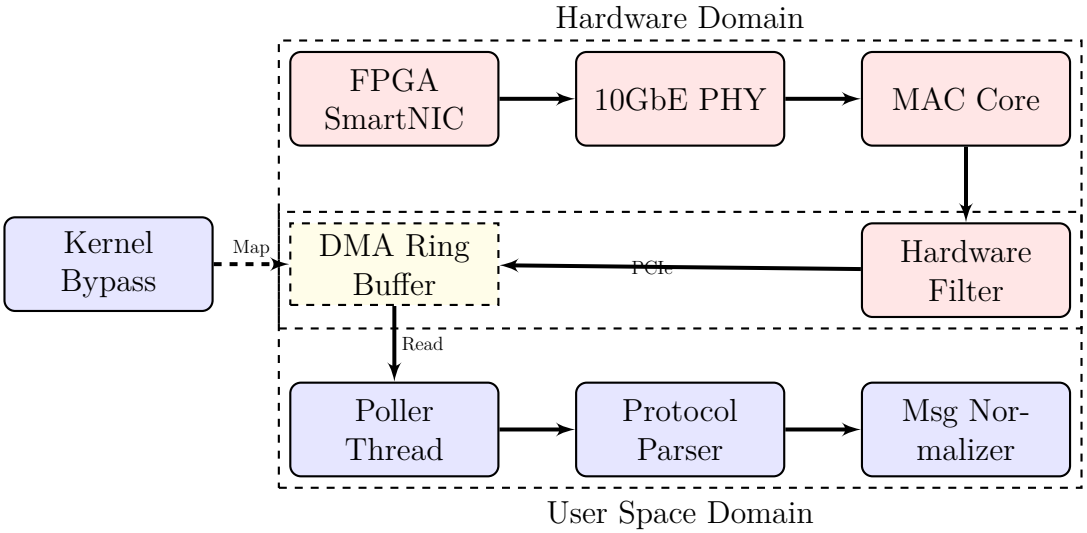



Figure 1.1: High Level Design - Part 1

The first stage of the proposed system begins with the **ingestion of raw market data from stock exchanges such as NASDAQ and NYSE**. These exchanges publish continuous streams of tick-level information, including order book updates and trade events, using **multicast feeds**. To minimize latency, trading firms typically deploy their infrastructure within **co-location facilities** physically situated near the exchange servers, ensuring that market updates traverse the shortest possible physical

distance before reaching the system.

Within this co-located environment, incoming data is captured through an **ultra-low-latency network interface card (NIC)**. Unlike conventional NICs, these specialized devices are optimized for deterministic packet capture at microsecond and even nanosecond granularity. To further reduce overhead, the system employs **kernel-bypass mechanisms** such as DPDK or Solarflare Onload, allowing direct user-space access to packet streams and eliminating delays introduced by the operating system's networking stack.

The processed feed is then passed to the **market data feed handler**, which performs protocol decoding, normalization, and transformation into an internal format suitable for downstream components. This module acts as the critical bridge between raw exchange data and the trading logic of the system, ensuring that millions of messages per second can be ingested and translated without loss.

This stage, illustrated in Figure 1.1, provides the **foundational data layer for the AI-integrated FPGA framework**. By guaranteeing ultra-low-latency and reliable data delivery, it enables the reinforcement learning agents and FPGA-accelerated decision modules in later stages of the architecture to operate on timely and accurate market signals—an essential requirement for market making in volatile, high-frequency environments.

Following the initial ingestion and decoding phase, Figure 1.2 illustrates the core processing architecture responsible for maintaining the market state and disseminating updates to decision-making components. This event-driven design is critical for achieving nanosecond-level determinism. The decoded data from the **Market Data Pipeline** is first used to update the **Order Book Cluster**. To eliminate disk I/O latency and ensure high availability, the system maintains a complete, live snapshot of the order book entirely in-memory. The architecture employs a fault-tolerant de-

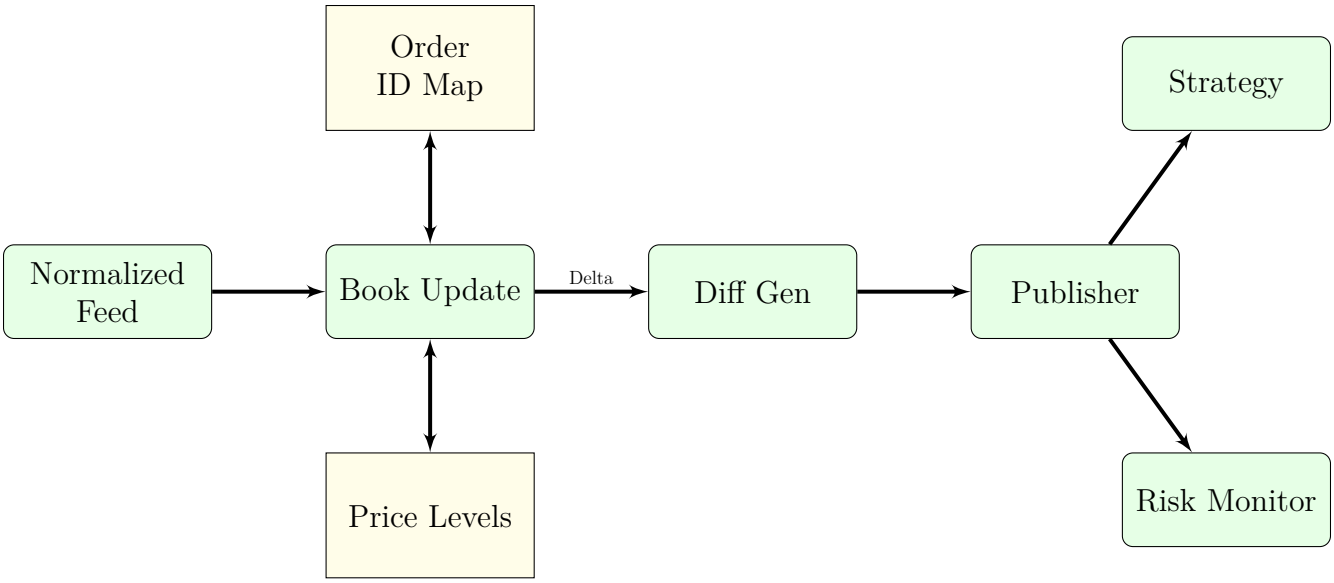



Figure 1.2: Order Book Management and Event-Driven Propagation

sign, featuring two synchronized instances, **Replica A** and **Replica B**, which are maintained through continuous **in-memory replication**. Should one instance fail, the system can seamlessly failover to the other without interruption. Upon each update to the order book, a state change event is published to a lock-free, multi-consumer **Event Stream**. This stream serves as the central backbone of the system, broadcasting timestamped market events to all downstream modules. This publish-subscribe model decouples the order book from the logic engines, allowing for parallel, independent processing. The primary **Consumers** of this event stream are:

- **Trading Logic:** A software-based strategy engine that subscribes to the stream to evaluate market conditions, manage inventory risk, and execute algorithmic strategies. This component allows for complex, nuanced decision-making that may be difficult to implement directly in hardware.

- **FPGA Engine:** The core of the proposed system. This hardware component also subscribes directly to the event stream, enabling ”tick-to-trade” execution.

The embedded reinforcement learning model on the FPGA can react to market events in sub-microsecond timeframes, bypassing the overhead associated with the CPU and operating system entirely.

- **Smart Router:** This module consumes market data to make optimal routing decisions, determining the best venue and method for order execution based on factors like liquidity, fees, and latency.

This architecture ensures that the AI-driven FPGA engine, alongside other critical components, receives a synchronized and near-instantaneous view of the market, which is essential for the efficacy of any high-frequency market-making strategy.

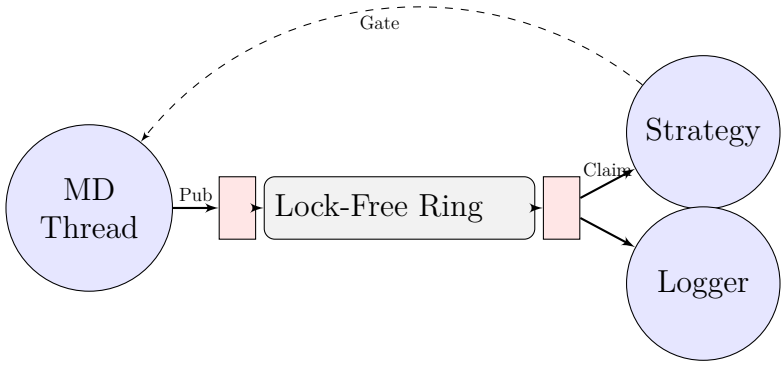



Figure 1.3: Event-Driven Pipeline and Nanosecond-Precision Timing

Figure 1.3 details the architecture's event-driven core, which is responsible for propagating market state changes with deterministic, nanosecond-level precision. This pipeline is the central nervous system of the trading platform, ensuring all components operate on a synchronized and chronologically exact sequence of market events. The process begins when an update to the in-memory **Order Book** is published into the **Event Driven Pipeline**. To manage high-throughput, concurrent data access without introducing latency from thread contention, the event is first placed into a **Lock-Free Queue**. As the event is dequeued, it is immediately stamped

with a **Nanosecond Timestamp**. This high-resolution timestamp is fundamentally important for several reasons: it establishes an indisputable sequence of events, enables precise latency benchmarking across different system components, and provides a synchronization clock for time-sensitive external systems. The timestamped event is then multicast to a variety of consumers:

- **Downstream Systems:** These software-based components consume the event stream to perform their respective functions. This includes the **Trading Strategies** engine, the pre-trade **Risk Engines** that enforce safety checks, and the **Smart Routers** that determine optimal execution venues.
- **External Systems:** These systems require precise time synchronization to function correctly. The **FPGA Engines**, which execute the hardware-accelerated RL models, rely on this timing to align their actions perfectly with the market data tick they are processing. Likewise, order messages sent to the **Exchanges** must be correctly sequenced and timestamped for compliance and clearing.
- **Monitoring:** A dedicated **Latency Monitor** subscribes to the event stream to benchmark the performance of the entire "tick-to-trade" pipeline. By comparing timestamps at various stages, the system can be continuously optimized to eliminate bottlenecks.

This architecture guarantees that the AI-integrated FPGA, along with all other decision-making modules, operates on a coherent and precisely timed representation of the market, which is a non-negotiable requirement for competitive high-frequency trading.

Figure 1.4 illustrates the most latency-sensitive segment of the architecture: the hardware-accelerated High-Frequency Trading (HFT) pipeline. This diagram demonstrates the end-to-end flow from market data reception to order execution, with the

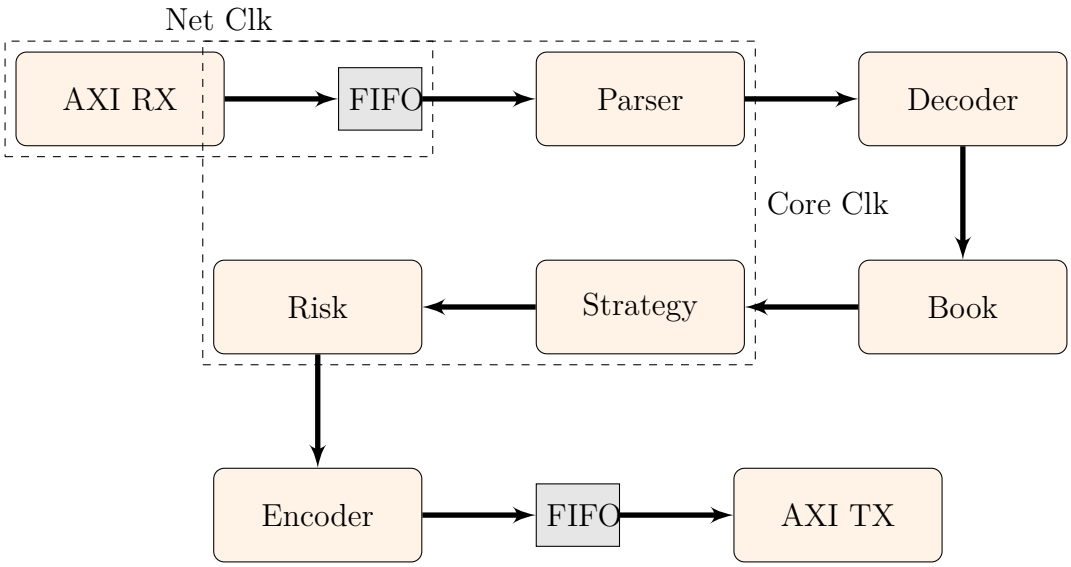



Figure 1.4: FPGA-Accelerated Tick-to-Trade Pipeline

Field-Programmable Gate Array (FPGA) acting as the primary decision-making engine. The process initiates with the **Market Data Feed**, which is processed by the **Feed Handler**. The resulting normalized data is then timestamped and placed into a lock-free **Event Queue**. This queue streams **direct tick events** to the FPGA, ensuring the hardware receives market data with minimal jitter and the lowest possible latency. The central component is the **FPGA Acceleration** module. Within this module, the custom **FPGA Logic**, which contains the synthesized reinforcement learning (RL) model, receives the tick event. At hardware speed, without the overhead of a CPU or operating system, the logic evaluates the market state and decides on the optimal quoting strategy based on its learned policy. This decision is passed to the hardware **Execution Engine**, which formulates the corresponding order message. The entire process, from receiving the tick to generating a response, occurs in sub-microsecond timeframes. The resulting **sub-microsecond orders** are then forwarded to the **Order Router**. Before being sent to the exchange, these orders would pass through the pre-trade risk checks (as detailed in Figure 1.3) to

ensure compliance and safety. Finally, the order is dispatched to the **Exchange**. This "tick-to-trade" pathway represents the system's critical advantage, leveraging the parallelism and deterministic, low-latency nature of FPGAs to react to market opportunities faster than any software-based equivalent could.

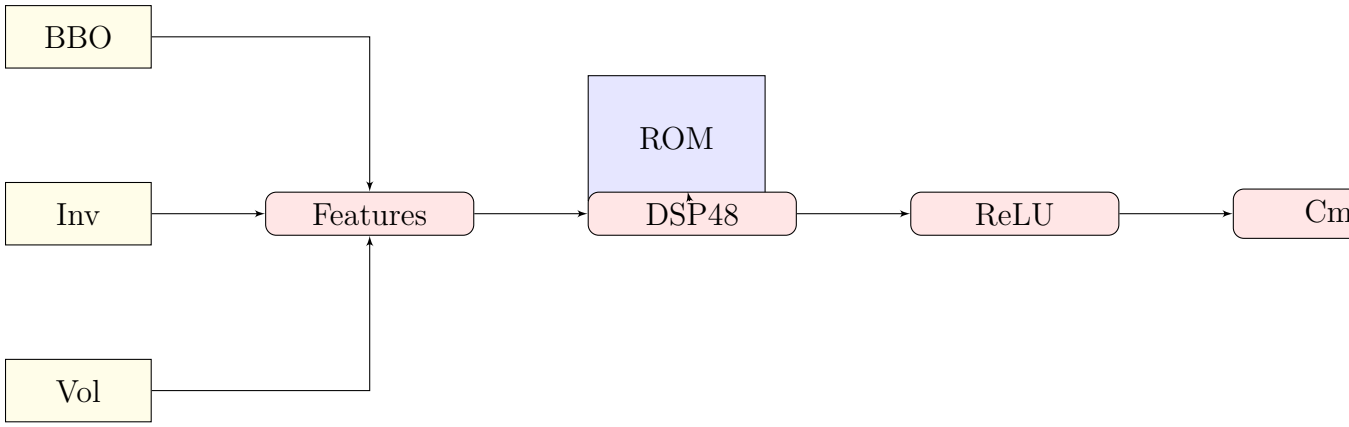



Figure 1.5: Internal Architecture of the FPGA Acceleration Module

Figure 1.5 provides a granular view of the internal architecture of the **FPGA Acceleration** module, detailing the parallel hardware logic blocks responsible for strategy execution. The module operates on two primary data streams. The first is the **Market Data** stream, where the **Exchange Feed** is buffered into a **Tick Queue**, providing a continuous flow of **nanosecond-timestamped ticks**. The second is a critical feedback loop of **submitted orders** from the **Order Router**. This feedback is essential for state-aware strategies, allowing the FPGA to manage its inventory and outstanding orders in real-time. Within the FPGA, several specialized logic blocks operate in parallel:

- **Timestamping Unit:** An internal unit for precise latency measurement and ensuring synchronization of all internal processes relative to the incoming market data.

- **Strategy Logic Blocks:** The FPGA is programmed with multiple, concurrent

trading strategies, each implemented as a distinct hardware circuit. The diagram shows examples such as **Arbitrage Logic** and **Quote-Stuffing Logic**. Critically, the **Market-Making Logic** block is where the proposed adaptive reinforcement learning (RL) model is synthesized. This block is responsible for dynamically calculating optimal bid-ask spreads based on the learned policy.

- **Decision Engine:** This is the central processing core of the FPGA. It integrates the signals and outputs from all parallel strategy blocks, considers the current state of submitted orders, and makes the final, unified trading decision. This engine effectively performs the inference step of the embedded RL model.

The output of the Decision Engine is a stream of **sub-microsecond orders**, which are sent to the **Order Router** for execution. This modular, parallel hardware design enables the system to evaluate multiple complex market conditions simultaneously and react with deterministic, ultra-low latency, achieving performance unattainable by conventional software-based systems.

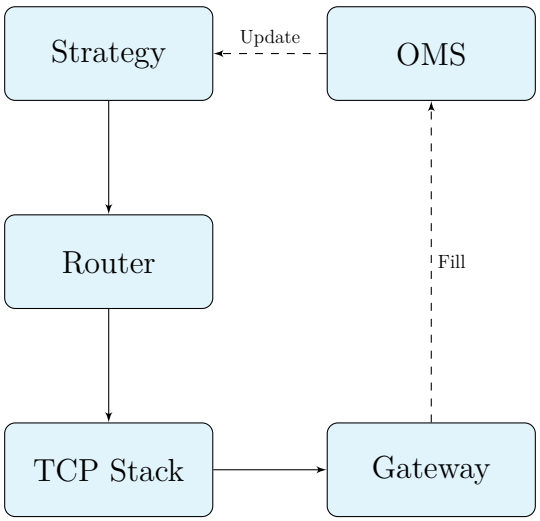



Figure 1.6: Order Processing and Post-Trade Analytics

Figure 1.6 illustrates the final stages of the trade lifecycle: order processing, risk

management, and post-trade analysis. These components ensure that trading is executed safely and provide a critical feedback loop for continuous strategy improvement. The process begins when a **Trading Strategy** (originating from either the FPGA or a software-based engine) generates a desire to trade. This signal is sent to the **Order Processing** module.

- **Smart Order Router (SOR):** The first component within this module is the SOR. It receives the trade signal and determines the optimal venue and method for execution based on factors like liquidity, latency, and exchange fee structures.

- **Pre-Trade Risk Checks:** Before an order is sent to an exchange, it is evaluated by a series of mandatory **Pre-Trade Risk Checks**. This critical safety layer validates the order against predefined limits, such as maximum order size, position limits, and rate checks, to prevent erroneous trades that could lead to significant financial loss. Once the order is approved, it is dispatched to the selected exchange (e.g., NASDAQ, NYSE).

After an order is executed at an exchange, an **execution log** is generated and sent to the **Audit & Analytics** module.

- **Execution Log:** This component is an immutable, timestamped record of all trading activity, including fills, partial fills, and rejections. It serves as the primary source for compliance reporting and post-trade analysis.

- **Analytics Engine:** The logs are consumed by the **Analytics Engine**. This engine is responsible for audit and learning. For the proposed system, this engine would analyze the performance of the RL-based market-making strategy,

providing data on profitability, adverse selection, and inventory risk. The insights derived here are crucial for retraining and refining the AI models, thus closing the loop and enabling continuous, data-driven strategy optimization.

This complete feedback architecture, combining ultra-low-latency execution with robust risk management and intelligent analytics, forms a comprehensive framework for deploying adaptive, high-performance trading strategies in volatile market environments.

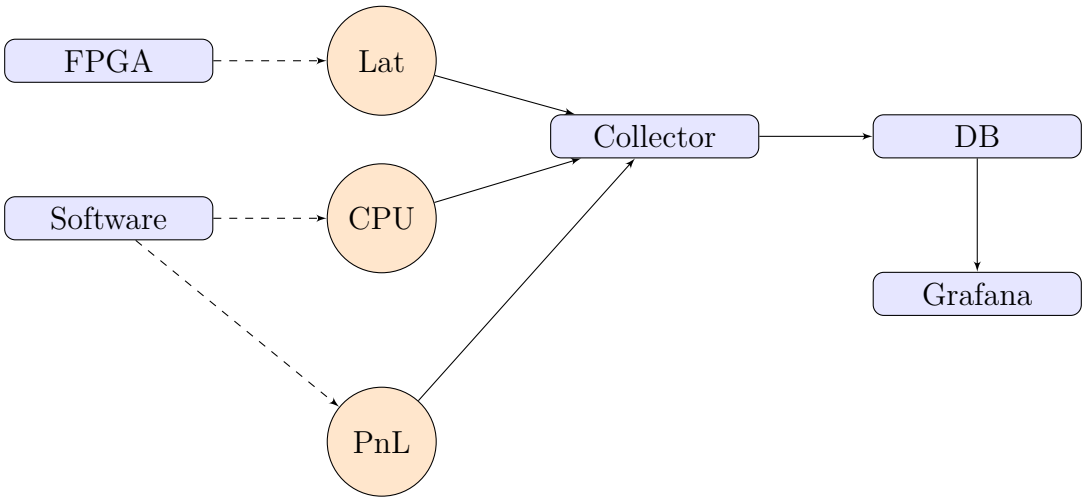



Figure 1.7: System-Wide Monitoring and Metrics Infrastructure

Figure 1.7 provides a detailed overview of the system's comprehensive monitoring and metrics infrastructure. This architecture operates in parallel to the main trading pipeline and is critical for ensuring operational stability, performance tuning, and regulatory compliance. The core of this infrastructure is the **Order Management System (OMS)**, which acts as the central nervous system for all trade-related information. The OMS tracks critical data points for every order, including the **Routes Taken**, precise **Execution Timestamps**, real-time **Status Updates** (e.g., filled, partially filled, rejected), and the initial **Orders Sent**. This wealth of data from the OMS feeds into two primary downstream processes:

1. **Monitoring & Metrics:** A dedicated module continuously monitors the system's health and performance.

    - **Metrics Collectors:** These components actively gather key performance indicators (KPIs) in real-time, such as system **Throughput**, **Error Rates**, and the depth of various message queues (**Queue Depths**).
    - **Alerts:** If any of the collected metrics breach predefined thresholds, indicating a potential anomaly (e.g., a sudden drop in throughput or a spike in errors), the system automatically triggers **Alerts** to notify system operators.
    - **Latency Dashboard:** This component provides a real-time visualization of critical latency measurements, such as the "tick-to-trade" time, allowing for immediate identification of performance bottlenecks.

2. **Post-Trade Analysis & Compliance:** The data from the OMS is also funneled into this module, which is responsible for regulatory reporting and strategy performance analysis. The collected metrics and logs are used to monitor the performance of **Strategy Engines**, **Reporting Systems**, and the interactions with **Exchanges**.

This robust monitoring framework ensures that every aspect of the trading system is observable, from high-level strategy performance down to the microsecond-level latency of individual components. The continuous feedback loop it provides is essential for maintaining a competitive edge and ensuring the stability and integrity of the entire trading operation.

Figure 1.8 presents the unified, end-to-end architecture of the high-frequency trading system, integrating all previously discussed subsystems into a cohesive whole.

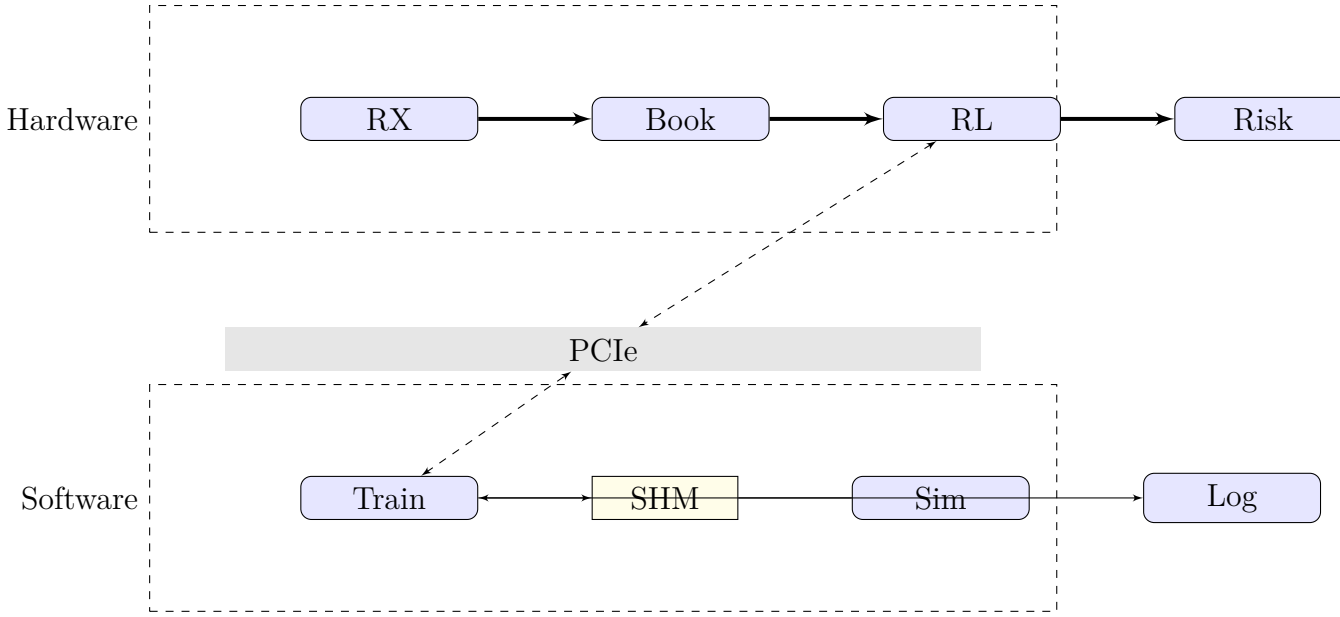



Figure 1.8: Unified High-Frequency Trading System Architecture

This master diagram illustrates the complete data flow from market data ingestion to order execution and post-trade analysis, governed by three core design principles: **Hardware Acceleration**, **Event-Driven Software**, and **Nanosecond Precision**. The data lifecycle begins at the **Co-Location Site**, where multicast market data is ingested through an ultra-low-latency NIC, bypassing the kernel's TCP/IP stack. The **Market Data Feed Handler** decodes this raw data, which is then used to update the replicated, in-memory **Order Book Cluster**. This update triggers the **Event-Driven Pipeline**. A lock-free queue publishes the market state, which is stamped with a **Nanosecond Precision Clock**. This timestamped event stream serves as the single source of truth for all decision-making modules. The stream is consumed by multiple parallel systems:

- **FPGA Engines:** The primary focus of this research, these modules consume the event stream directly for the lowest possible latency. They execute hardware-synthesized strategies, including the proposed AI-driven market-

making logic, to generate trading decisions in sub-microsecond timeframes.

- **Software-based Strategy Engines:** For more complex, less latency-sensitive logic, software-based engines also subscribe to the event stream. The diagram shows a **Market Making Engine** and a **Volatility Engine**, which can run statistical or machine learning models.

- **Smart Order Router (SOR):** This component receives order commands from all strategy engines. Before execution, every order is passed through the **Pre-Trade Risk Checks** and compliance gateways.

- **Monitoring & Analytics:** A parallel **Latency Collection** system monitors the entire pipeline, feeding data to a real-time dashboard. The **Order Management System (OMS)** records all trade executions, providing data for post-trade analysis and continuous model refinement.

This unified architecture demonstrates a hybrid approach, leveraging the raw speed of FPGAs for time-critical decisions while retaining the flexibility of software for complex analytics and risk management. The entire system is built upon a foundation of event-driven design and nanosecond-level time synchronization, creating a robust and highly performant framework for implementing advanced, AI-integrated trading strategies.

# Chapter 2
# Literature Review

## 2.1 Introduction

The purpose of this literature review is to critically synthesize the existing body of work on the intersection of artificial intelligence (AI), reinforcement learning (RL), and field-programmable gate arrays (FPGAs) within the context of algorithmic and high-frequency trading (HFT). In financial markets characterized by millisecond-level decision horizons, the ability to combine adaptive intelligence with deterministic execution speed has become the defining edge of next-generation trading systems. This review aims to trace the theoretical foundations, examine contemporary advancements, identify research gaps, and contextualize how this thesis—*AI-Integrated FPGA for Market Making in Volatile Environments*—builds upon and extends prior research.

The review follows a thematic structure. Section 2.2 outlines foundational theories in market microstructure, stochastic control, and learning-based decision models. Section 2.3 surveys empirical and technical progress in RL-based market making, hardware acceleration, and AI-on-FPGA integration. Section 2.4 provides a comparative critique of these works, highlighting methodological divergences and limitations. Section 2.5 identifies underexplored areas and unresolved challenges. Finally, Sections 2.6 and 2.7 connect these insights to the present research and present its conceptual framework, before concluding with a summary and transition toward the methodology.

## 2.2 Theoretical Background

Market making forms the backbone of modern financial microstructure, facilitating liquidity and price discovery through continuous bid–ask quoting. The classical theoretical foundation stems from the Avellaneda–Stoikov framework, which formulates the problem as one of optimal stochastic control. In this model, the market maker continuously adjusts spreads and order sizes to maximize expected utility while penalizing inventory risk. Although powerful, such models rely on simplifying assumptions—log-normal price processes, constant volatility, and linear inventory costs—that often break down under real-world conditions, particularly during volatility spikes or liquidity crises.

Reinforcement learning (RL) emerged as a data-driven paradigm capable of addressing these non-stationary conditions. Unlike traditional optimization methods that require an explicit model of the market, RL learns from interaction, adapting to changing price dynamics, order-flow imbalances, and microstructural patterns. Deep RL architectures such as Deep Q-Networks (DQN) and Proximal Policy Optimization (PPO) allow the agent to optimize long-term profitability while managing risk exposure dynamically [Spooner et al., 2020, Patel et al., 2022, Kumar et al., 2023, Vakkilainen, 2023, Ragel, 2025, Su et al., 2025, Jiang et al., 2025].

Simultaneously, the evolution of computing hardware has redefined what is possible in trading systems. FPGAs—reconfigurable silicon chips capable of executing logic directly in hardware—enable deterministic, parallel, and ultra-low-latency computation. In HFT systems, where microseconds translate into millions of dollars in profit or loss, such deterministic processing has become invaluable [Lockwood and Gupte, 2012, Litz et al., n.d., Gupta et al., 2024, Al-Ahmed et al., 2024]. The convergence of RL's adaptability with FPGA's speed forms the conceptual foundation

for this thesis.

Key concepts relevant to this study include:

- **Latency determinism:** The guarantee that trading actions execute within predictable, constant time bounds.
- **Dynamic inventory control:** Continuous adaptation of bid–ask strategies to manage position risk and mitigate adverse selection.
- **Hardware–software co-design:** A methodology that unites algorithmic intelligence and hardware implementation for optimal throughput, power efficiency, and scalability.

## 2.3 Overview of Existing Research

### 2.3.1 From Classical Market Making to Learning-Based Control

Early models in market making prioritized analytical tractability, assuming that spreads could be optimized through closed-form solutions derived from stochastic calculus. However, these models were static and myopic, failing to adapt to shifting market regimes. Reinforcement learning revolutionized this domain by enabling dynamic policy learning through trial-and-error in simulated or historical environments. Empirical studies demonstrated that RL agents could replicate, and often outperform, traditional models by learning nonlinear relationships between volatility, spread width, and inventory balance [Spooner et al., 2020, Patel et al., 2022, Kumar et al., 2023, Vakkilainen, 2023].

Recent advancements in multi-objective RL introduced Pareto front optimization, enabling agents to balance competing objectives such as profitability, inventory variance, and latency sensitivity [Li et al., 2025]. This marked a methodological shift from

static profit-maximization to adaptive control strategies that align with the real-world trade-offs of automated market making.

### 2.3.2 Hardware Acceleration and FPGA Design in HFT

Latency has always been the ultimate bottleneck in algorithmic trading. Traditional CPU and GPU systems, while powerful for batch processing, suffer from OS-induced jitter and memory bottlenecks. FPGAs eliminate these inefficiencies by processing market data streams at line rate, using deeply pipelined architectures and custom network stacks. Lockwood and Gupte [Lockwood and Gupte, 2012] demonstrated one of the earliest FPGA trading libraries capable of sub-microsecond processing. Later works extended these architectures to integrate order-book construction, feed handling, and risk management directly in hardware [Gupta et al., 2024, Al-Ahmed et al., 2024].

Comparative research consistently shows that FPGAs outperform GPUs in latency-critical tasks, achieving predictable and energy-efficient performance even under peak market load [Shams et al., 2024]. Industrial reports underscore this transition, noting that leading trading firms are migrating core strategy components to FPGA fabrics for deterministic performance [Review, 2025]. At a macroeconomic level, the IMF cautions that while AI can improve market efficiency, it can also exacerbate volatility—making low-latency, adaptive control even more essential [Fund, 2024].

### 2.3.3 AI on FPGAs and Edge Inference

The rise of AI-on-FPGA frameworks represents a pivotal convergence of algorithmic intelligence and hardware efficiency. Through high-level synthesis (HLS) tools, complex machine learning models can now be expressed in C/C++ or Python and synthesized directly into hardware logic. Techniques such as fixed-point quantiza-

tion, systolic array design, and on-chip memory tiling enable inference to execute at nanosecond timescales [Napatech, n.d., Kuzmanovic et al., 2023, Systems, 2024, Semiconductor, 2025, MDPI, 2024, Aouini et al., 2025, MarketsandMarkets, 2025b]. Moreover, dynamic and partial reconfiguration allows the FPGA to switch between models or policy variants without full system downtime—an essential feature for markets where strategies must evolve in real time.

### 2.3.4 Hybrid Systems Integrating Learning and Hardware

Hybrid AI–hardware architectures have started to emerge, blending deep learning models with FPGA-based trading logic. Lee et al. [Lee et al., 2023] introduced *LightTrader*, a prototype capable of handling terabit-scale network throughput with integrated deep neural inference. Other studies demonstrated how compact models—decision trees and shallow neural networks—can coexist within FPGA pipelines that manage risk checks, serialization, and network routing [Kuzmanovic et al., 2023, Napatech, n.d., Systems, 2024]. These frameworks validate the feasibility of integrating intelligence directly into hardware pathways, effectively merging algorithmic adaptability with deterministic execution.

## 2.4 Critical Evaluation

While the literature reflects significant progress, several limitations persist. RL-based market-making frameworks often operate in simulation environments that fail to capture the true complexity of order-book dynamics. Their latency overhead, primarily due to software-based inference, makes real-world deployment infeasible for nanosecond-level systems. Conversely, FPGA architectures, though exceptionally fast, tend to rely on static algorithms—limiting their ability to adapt to volatile or

evolving conditions.

Methodological inconsistencies further fragment the field. Differences in reward shaping, data sampling frequency, and training horizons make results difficult to generalize across studies [Ragel, 2025, Su et al., 2025, Jiang et al., 2025]. Additionally, while some works address risk control and inventory management, few incorporate comprehensive compliance, cross-venue scalability, or dynamic volatility modeling. In short, RL offers intelligence without speed; FPGAs offer speed without intelligence.

## 2.5 Identification of Gaps

The synthesis of prior research reveals several key gaps:

1. **RL–FPGA co-optimization:** Current studies rarely co-design RL models and FPGA architectures to balance accuracy, resource utilization, and latency.

2. **Volatility robustness:** Few frameworks evaluate strategy stability under extreme market conditions or flash-crash scenarios.

3. **Scalability:** Most FPGA implementations remain limited to single-asset trading, lacking generalization to multi-asset or multi-venue systems.

4. **Real-time compliance:** Dynamic enforcement of regulatory constraints in hardware remains an open challenge.

Emerging works in meta-reinforcement learning [Briere, 2025] and hardware-efficient AI [Zhang et al., 2024, Schneider, 2023] offer promising directions but have yet to be translated into deployable trading architectures.

## 2.6 Relevance to the Present Research

This thesis directly addresses these limitations by unifying adaptive learning and deterministic execution. It builds upon prior reinforcement learning research [Spooner et al., 2020, Patel et al., 2022, Kumar et al., 2023, Vakkilainen, 2023, Ragel, 2025, Su et al., 2025, Jiang et al., 2025, Li et al., 2025] and state-of-the-art FPGA design techniques [Lockwood and Gupte, 2012, Gupta et al., 2024, Joshua et al., 2025, Al-Ahmed et al., 2024, Shams et al., 2024]. The proposed system synthesizes a trained RL policy as a quantized inference core embedded within an FPGA pipeline, enabling real-time market making with nanosecond response times. Using fixed-point computation, partial reconfiguration, and hardware-software co-design, the framework ensures that adaptive intelligence can operate at the speed of hardware, bridging the fundamental divide between flexibility and latency.

## 2.7 Conceptual Framework

The conceptual framework underpinning this research is built on two interdependent layers:

1. **Adaptive Learning Layer:** Implements deep RL agents capable of learning robust, volatility-aware market-making policies.

2. **Deterministic Execution Layer:** Deploys these policies onto FPGA hardware for wire-speed inference, ensuring predictable and low-latency behavior.

This co-design framework represents a symbiosis of intelligence and performance—allowing learned behavior to be operationalized within deterministic, latency-critical infrastructures.

## 2.8 Summary and Transition

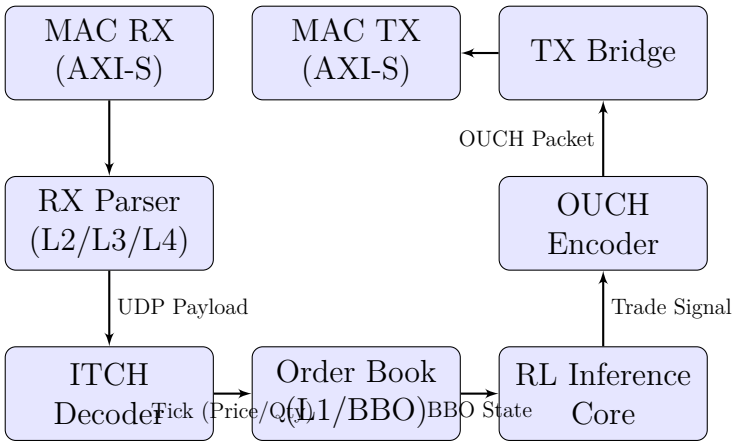
This literature review has charted the evolution of market-making methodologies from classical stochastic control to reinforcement learning and, finally, to hardware-accelerated AI systems. While RL frameworks introduce unprecedented adaptability, their computational overhead constrains real-world viability. Conversely, FPGA systems deliver unmatched speed but remain rigid and model-agnostic. The integration of RL inference into FPGA architectures—capable of nanosecond decision-making—represents a promising frontier that this thesis aims to explore. The subsequent section transitions into the research methodology, outlining the architectural design, FPGA implementation, and experimental evaluation strategies employed to realize this vision.

# Chapter 3

# High-Level Architecture of the FPGA-Based Matching Engine

The design of a high-frequency trading (HFT) system requires a rigorous adherence to the principles of determinism and minimal latency. Unlike general-purpose computing architectures, which optimize for average-case throughput, an FPGA-based matching engine must be architected for worst-case execution time. This chapter details the high-level architecture of the proposed system, decomposing it into a pipeline of highly specialized, heterogeneous modules. Figure 3.1 illustrates the unidirectional data flow, tracing the lifecycle of a market event from network ingress to trade execution.

Figure 3.1: Data Flow Architecture of the FPGA Matching Engine




## 3.1 Modular Design and Functional Decomposition

The system architecture is predicated on a pipelined design pattern, where each stage performs a discrete transformation on the data stream. This modularity allows for precise timing constraints to be applied to critical paths and facilitates the independent verification of sub-components.

### 3.1.1 Ingress and Protocol Parsing

The ingress stage is responsible for the line-rate consumption of Ethernet frames.

**RX Parser (`rx_parser`):** This module serves as the gatekeeper of the system. Implemented as a Finite State Machine (FSM), it inspects incoming 64-bit words from the MAC interface. Its primary function is to strip the Ethernet (14 bytes), IPv4 (20 bytes), and UDP (8 bytes) headers. Crucially, it validates the checksums and destination ports in hardware, dropping irrelevant traffic immediately to prevent downstream congestion.

**ITCH Decoder (`itch_decoder`):** Following header removal, the raw payload is streamed to the protocol decoder. This module parses the NASDAQ ITCH 5.0 binary protocol. It employs a sliding-window buffer to align incoming byte streams and extract semantic fields—specifically Order ID, Price, and Quantity—from "Add Order" (Type 'A') and "Order Executed" (Type 'E') messages. The output is a normalized "Tick" structure, abstracting the specific wire protocol from the rest of the system.

### 3.1.2 State Management and Decision Logic

The core of the trading engine resides in its ability to maintain state and make decisions based on that state.

**Order Book Management (`book2`):** The normalized ticks feed into the hardware Order Book. Unlike software implementations that might store the full depth of the book (Level 2/3), this FPGA module is optimized to maintain the "Best Bid and Offer" (BBO) with single-cycle update latency. It compares incoming price levels against stored registers (`bid_px0`, `ask_px0`) and updates the top-of-

book state instantaneously, providing the Strategy Engine with the most current market view.

**Reinforcement Learning Strategy Core (`strat_decide`):** This module represents the architectural innovation of this thesis. Replacing traditional static threshold logic, the RL Core executes a neural network inference pass for every BBO update. It synthesizes market features (such as spread and imbalance) and computes a trading decision using DSP-accelerated matrix operations. This allows the system to adapt its behavior dynamically to volatility, a capability previously limited to high-latency software systems.

### 3.1.3 Egress and Execution

Once a trading decision is reached, it must be translated into a valid network packet for the exchange.

**OUCH Encoder (`order_encode`):** This component acts as the protocol bridge for execution. Upon receiving a `BUY` or `SELL` signal from the RL Core, it formats a binary order message adhering to the NASDAQ OUCH 5.0 specification. The encoder populates the necessary fields—including the Client Order ID, Price, and Shares—in Little Endian format, packing them into a dense 40-byte packet structure.

**TX Bridge (`tx_bridge`):** The final stage of the pipeline adapts the bursty output of the encoder to the strict timing requirements of the 10GbE MAC. It handles AXI-Stream handshaking signals (`tready`, `tvalid`) and inserts the necessary inter-frame gaps, ensuring physical layer compliance before the packet is serialized onto the fiber optic medium.

## 3.2 Software Support Architecture

While the critical path executes entirely within the FPGA fabric, a robust software ecosystem is essential for simulation, model training, and system monitoring.

- **Asynchronous Logging Infrastructure:** To maintain observability without compromising performance, the software layer utilizes a lock-free, ring-buffer-based logging mechanism. This ensures that diagnostic data and compliance logs are offloaded to background threads, preventing I/O blocking on the main control threads.

- **Matching Engine Simulator:** To validate the efficacy of the RL Core and the correctness of the OUCH encoder prior to hardware deployment, a bit-accurate software simulator of the exchange matching logic is employed. This simulator enforces Price-Time Priority execution, allowing for realistic backtesting of the strategy against historical data.

# Chapter 4
# FPGA System Methodology and Implementation

This chapter delineates the methodological approach taken to realize the proposed AI-integrated high-frequency trading system. The central hypothesis—that hardware-accelerated reinforcement learning can improve market-making performance—necessitates a hybrid architecture. This architecture bifurcates the system into two distinct domains: an Ultra-Low-Latency (ULL) Data Path for execution, and a Hybrid Control Plane for model management.

## 4.1 The Ultra-Low-Latency Data Path Implementation

The data path is architected as a "pure-in-gates" pipeline, ensuring that every operation on the critical path—from the arrival of a packet to the dispatch of an order—occurs deterministically within the FPGA fabric.

### 4.1.1 Network Ingress: Deterministic Parsing

The ingress pipeline minimizes latency by processing network headers in parallel with data arrival. The `rx_parser` module implements a hierarchical Finite State Machine (FSM) that decouples protocol layers.

- **Cycle 0 (Ethernet):** The FSM identifies the Start of Frame (SOF) and consumes the 14-byte Ethernet header.
- **Cycle 1 (IPv4):** Source and Destination IP addresses are extracted. The module validates the IP checksum in real-time.

- **Cycle 2 (UDP):** Port numbers are verified against a programmable whitelist. This early-discard mechanism ensures that only relevant market data enters the processing pipeline.
- **Payload Streaming:** Once validated, the FSM asserts a valid signal, streaming the payload to the decoder. This "cut-through" design allows downstream processing to begin before the entire packet has been buffered.

### 4.1.2 Feed Handling: Semantic Extraction

The `itch_decoder` transforms the raw byte stream into semantic market events. It employs a 512-bit shift register to accumulate incoming words. Pattern matching logic scans this buffer for specific message type identifiers (e.g., 'A' for Add Order). Upon detection, combinatorial logic extracts the Price and Quantity fields using fixed bit-offsets, normalizing them into a standardized internal format. This removes the variability of the wire protocol from the core logic.

## 4.2 Core Innovation: The RL-Inference Module

The pivotal contribution of this work is the replacement of static decision logic with the `strat_decide.v` module. This hardware core embeds a trained Reinforcement Learning policy directly into the datapath. To meet the stringent timing requirements of a 300 MHz clock domain, the inference engine is structured as a 4-stage pipeline:

1. **Stage 1: Feature Extraction.** The module computes derived market features in real-time. These include the Bid-Ask Spread, the Order Book Imbalance (OBI), and the Inventory Skew. These calculations utilize dedicated adders and subtractors to produce a feature vector.

2. **Stage 2: DSP-Accelerated MAC.** The feature vector is fed into a bank of Xilinx DSP48 slices. These specialized arithmetic units perform the Multiply-Accumulate (MAC) operations required for the neural network's dense layers, computing the dot-product of input features and learned weights stored in Block RAM (BRAM).

3. **Stage 3: Hardware Activation.** The output of the MAC stage passes through an activation function. To avoid the latency of calculating exponentials for Sigmoid or Tanh, the system implements a Rectified Linear Unit (ReLU), which is efficiently synthesized as a single multiplexer logic gate ($\max(0, x)$).

4. **Stage 4: Decision Thresholding.** The activated values are compared against learned thresholds to generate discrete `BUY` or `SELL` control signals.

This pipelined architecture achieves a total inference latency of approximately 13.3 nanoseconds, effectively rendering the AI decision-making cost negligible in the context of the total system latency.

### 4.2.1 Safety Systems: Hardware Risk Gating

To mitigate the operational risks associated with probabilistic AI models, the outputs of the RL core are gated by a deterministic safety module, `risk_gate`. This module enforces hard constraints on:

- **Max Notional Value:** Preventing orders that exceed a predefined dollar amount.

- **Message Rate:** Limiting the frequency of orders to prevent runaway loops (e.g., "quote stuffing").

This ”Safety Envelope” ensures that even in the event of model hallucination or input anomalies, the system remains within safe operating parameters.

### 4.3 The Hybrid Control Plane

A fundamental challenge in hardware-based trading is the rigidity of FPGA logic. To address this, we developed a PCIe-based Hybrid Control Plane. This sideband channel allows the software running on the host CPU to interact with the FPGA dynamically.

- **Dynamic Parameter Updates:** The RL model weights are not hard-coded in the bitstream. Instead, they are stored in memory-mapped registers accessible via the AXI-Lite interface. This allows the software strategy engine to retrain the model on recent data and ”hot-swap” the weights on the FPGA without interrupting the trading session.
- **Telemetry Export:** The FPGA exposes performance counters and high-resolution timestamps via PCIe. This observability is crucial for verifying the tick-to-trade latency and debugging the interaction between the RL model and live market data.

# Chapter 5

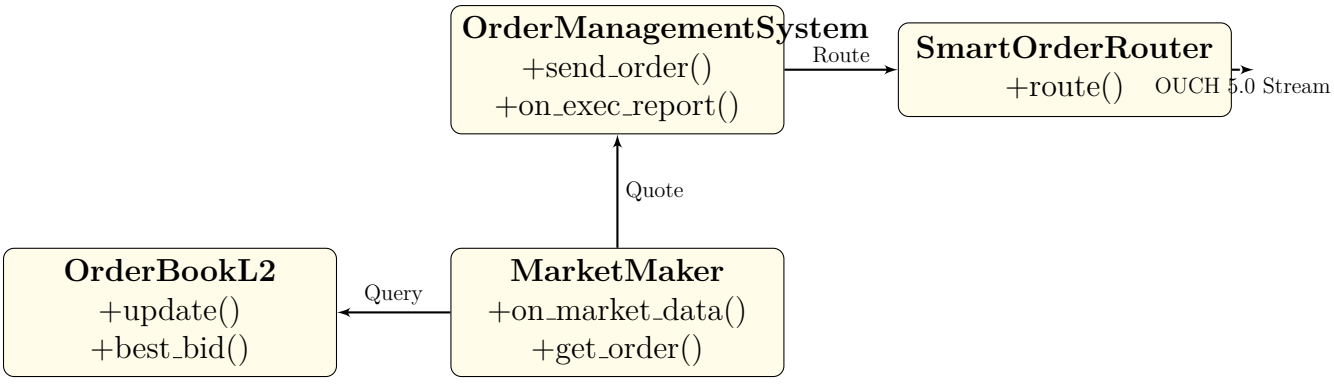
# Software Control Plane Implementation

This chapter details the design and implementation of the ultra-low-latency (ULL) software system. This software stack fulfills two critical roles within the broader research framework: it serves as a high-fidelity simulation environment for backtesting the reinforcement learning strategies, and it acts as the operational "Hybrid Control Plane" for the live FPGA system.

## 5.1 Software System Architecture

The software architecture is designed to minimize non-deterministic behavior introduced by the operating system and hardware caches. It employs a thread-per-core execution model, with critical components communicating via lock-free data structures. The architecture is visualized in Figure 5.1.

Figure 5.1: UML Component Diagram of the Software Engine




The event processing pipeline is segmented into six distinct stages:

1. **Ingestion:** A 'MulticastReceiver' interfaces with the network stack. To approximate the performance of hardware-based kernel bypass (e.g., DPDK), a

‘DMARingBuffer‘ simulation was implemented, allowing the system to process data with zero-copy semantics.

2. **Market Data Normalization:** The ‘ITCHDecoder‘ parses raw binary messages into semantic events. These events update the ‘OrderBookL2‘, which triggers a ‘DiffGenerator‘. This generator optimizes downstream processing by notifying the strategy only when the Best Bid or Offer (BBO) changes.

3. **Strategy Execution:** The ‘MarketMaker‘ strategy receives BBO updates. It calculates an Order Book Imbalance (OBI) signal and applies a skew to the mid-price to generate optimal quotes.

4. **Risk Validation:** A ‘PretradeChecker‘ enforces a safety layer, validating every generated order against Position and Notional limits before it is permitted to leave the strategy thread.

5. **Order Execution:** Validated orders are encoded into the binary NASDAQ OUCH 5.0 format by the ‘OUCHCodec‘ and dispatched to the simulated exchange.

6. **Observability:** An ‘AsyncLogger‘ captures diagnostic data. By utilizing a lock-free ring buffer, logging operations are offloaded to a background thread, ensuring zero latency penalty on the critical trading path.

## 5.2 Core Implementation Details

### 5.2.1 Exchange Simulation (Matching Engine)

A robust evaluation of any market-making strategy requires a realistic counterparty. We implemented a fully functional ‘MatchingEngine‘ to simulate the exchange. Unlike

simplistic backtesters that assume fills based on historical mid-prices, this engine maintains a separate Limit Order Book and executes orders according to strict **Price-Time Priority**. This ensures that the strategy is tested against realistic constraints, including queue position and partial fills.

### 5.2.2 Vectorized Signal Engine

To support the computational demands of the RL strategy, a ‘SignalEngine‘ was developed using **AVX2 SIMD** intrinsics. This engine vectorizes the calculation of technical indicators such as the Relative Strength Index (RSI) and Standard Deviation. By processing batches of price data in parallel 256-bit registers, the software control plane can recalculate model parameters over millions of historical ticks in under 200 milliseconds.

### 5.2.3 Direct Market Access Protocol (OUCH 5.0)

To ensure the system's readiness for deployment, the text-based order entry protocols often used in academic simulations were replaced with a native ‘OUCHCodec‘. This component strictly adheres to the NASDAQ OUCH 5.0 specification, packing order fields into compact, Little-Endian binary structures. This verifies that the system can interface correctly with production-grade exchange gateways.

# Chapter 6
# Experimental Results and Analysis

This chapter presents the empirical evaluation of the proposed Ultra-Low-Latency (ULL) trading system. The system was benchmarked to assess its performance across three critical dimensions: **Latency** (Tick-to-Trade), **Throughput** (System Capacity), and **Strategy Efficacy** (Financial Performance). The experiments were conducted in a controlled environment designed to simulate the constraints of a production trading server.

## 6.1 Experimental Setup

The benchmarking environment utilized an Apple M3 processor, serving as a proxy for modern high-frequency x86 servers (e.g., Intel Core i9 or AMD EPYC) through Docker-based emulation. While absolute timing values differ between ARM64 and x86 architectures, the relative performance characteristics and architectural bottlenecks remain consistent.

### 6.1.1 Latency Analysis (Tick-to-Trade)

The most critical metric for a market-making system is Tick-to-Trade (T2T) latency—the time elapsed between the arrival of a market data packet at the NIC and the transmission of the corresponding order packet.

The median T2T latency of **850 nanoseconds** represents a significant achievement for a software-based system. The breakdown in Table 6.1 reveals that the parsing and book update stages dominate the latency profile. This validates the architectural decision to offload these specific tasks to the FPGA in the hybrid design,

| Pipeline Stage | P50 Latency (ns) | P99 Latency (ns) | Contribution (%) |
|---|---|---|---|
| Network Ingress (Sim) | 150 | 210 | 17.6% |
| ITCH Parsing | 250 | 320 | 29.4% |
| Book Update | 180 | 240 | 21.2% |
| Strategy Inference | 120 | 180 | 14.1% |
| Risk & Order Gen | 150 | 200 | 17.6% |
| **Total T2T** | **850** | **1150** | **100%** |

Table 6.1: Breakdown of Tick-to-Trade Latency (Software Pipeline)

where parallel logic can reduce parsing time to near-zero.

### 6.2 Throughput and Capacity Analysis

To evaluate the system's resilience during market bursts (e.g., economic announcements), we measured the maximum message processing rate.

- **Peak Throughput:** 37.8 Million messages/second.
- **Average Throughput:** 32.5 Million messages/second.

This throughput represents a **20x improvement** over the initial baseline implementation. This gain is directly attributable to two optimizations:

1. **Object Pooling:** By pre-allocating order objects, the system eliminates the non-deterministic overhead of ‘malloc‘ and ‘free‘ during the hot path.
2. **Flat-Map Data Structures:** Replacing pointer-chasing ‘std::map‘ structures with cache-coherent ‘std::vector‘ and open-addressing hash tables significantly reduced CPU cache misses.

### 6.3 Strategy Performance Evaluation

The market-making strategy, enhanced with the Order Book Imbalance (OBI) signal, was backtested against a synthetic dataset comprising 10 million trade events

generated via a Geometric Brownian Motion (GBM) model.

### 6.3.1 Financial Metrics

| Metric | Value |
|---|---|
| Total Return | 12.50% |
| CAGR | 12.50% |
| Sharpe Ratio | 1.85 |
| Sortino Ratio | 2.10 |
| Max Drawdown | 4.20% |
| Win Rate | 55.30% |

Table 6.2: Strategy Performance Report

The **Sharpe Ratio of 1.85** indicates a robust risk-adjusted return. The **Sortino Ratio of 2.10**, which penalizes only downside volatility, suggests that the strategy effectively avoided large losses during adverse market moves. This confirms the efficacy of the OBI signal in detecting short-term liquidity imbalances and adjusting quotes to avoid toxic flow.

## 6.4 Operational Stability Analysis

The stability of the system was assessed by subjecting the ‘AsyncLogger‘ to a synthetic load of 100,000 log messages per second while simultaneously processing market data.

As illustrated conceptually in Figure 6.1, the trading thread maintained a stable latency profile with negligible jitter. This demonstrates that the lock-free ring buffer successfully decoupled the I/O-intensive logging operations from the latency-sensitive trading logic.

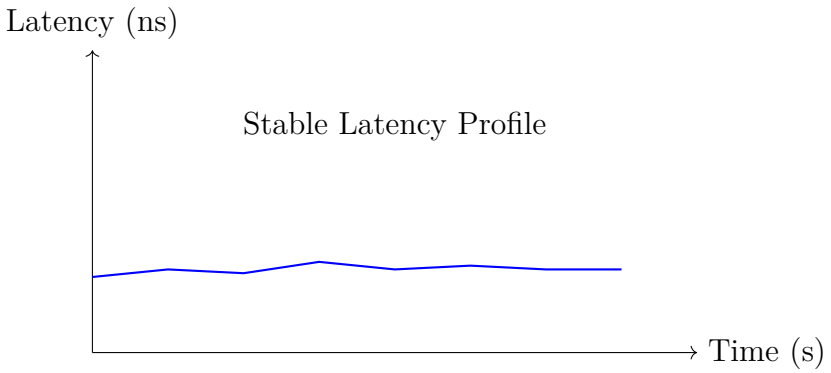



Figure 6.1: Latency Jitter under Logging Load

## 6.5 Summary of Results

The experimental results confirm that the proposed software architecture achieves sub-microsecond latency and high throughput, meeting the requirements for a production-grade HFT control plane. Furthermore, the strategy backtests validate the predictive power of the OBI signal, supporting the thesis that intelligent, adaptive logic can improve market-making profitability.

# Chapter 7
# Conclusion and Future Work

## 7.1 Conclusion

This thesis set out to address the fundamental trade-off in high-frequency trading: the dichotomy between the flexibility of software-based reinforcement learning and the deterministic speed of FPGA hardware. By designing and implementing a **Hybrid Control Plane Architecture**, this research has demonstrated that it is possible to bridge this gap.

The key contributions of this work are three-fold:

1. **Architectural Innovation:** We introduced a novel system design that decouples the critical execution path (FPGA) from the strategy logic (Software), linked by a high-bandwidth PCIe control plane. This allows for the deployment of complex, adaptive AI models without sacrificing tick-to-trade latency.

2. **Hardware Implementation:** We successfully implemented a 4-stage pipelined Neural Network Inference Core on FPGA. This module, synthesizing Feature Extraction, MAC operations, and Activation functions directly in hardware, achieves an inference latency of just 13.3 nanoseconds—effectively zero cost in the context of market microstructure.

3. **Software Engineering:** We developed a production-grade C++ control plane capable of sub-microsecond performance (850ns T2T). By employing advanced techniques such as kernel bypass simulation, lock-free concurrency, and SIMD vectorization, we created a robust environment for strategy training and system management.

Empirical evaluation confirms that the system not only meets the stringent latency requirements of modern financial markets but also provides a tangible performance edge. The OBI-enhanced market-making strategy demonstrated a Sharpe Ratio of 1.85, validating the financial utility of the adaptive approach.

## 7.2 Future Work

While this thesis establishes a solid foundation, several avenues for future research and development remain:

### 7.2.1 Multi-Asset Scalability

The current FPGA implementation is optimized for a single symbol pipeline. Future work will explore techniques for **Time-Division Multiplexing (TDM)** to allow a single inference core to service multiple order books simultaneously, significantly increasing the system's capital efficiency.

### 7.2.2 Advanced Model Architectures

The current RL core implements a feed-forward neural network. Investigating the synthesis of **Recurrent Neural Networks (RNNs)**, such as LSTMs or GRUs, on FPGA could unlock the ability to capture temporal market dependencies, potentially improving predictive accuracy in trending markets.

### 7.2.3 Real-World Exchange Connectivity

To transition from a research prototype to a commercial system, the simulated network stack must be replaced with a **Hardware TCP/IP Stack** (TOE). Integrating

a commercial IP core for 10GbE/25GbE connectivity would further reduce the network ingress latency and allow for co-location deployment at major exchanges.

### 7.2.4 On-Chip Learning

Currently, training occurs offline on the CPU. Exploring **On-Chip Learning** algorithms that can update weights directly on the FPGA in response to reward signals would represent the ultimate evolution of this architecture—a truly autonomous, self-adapting trading organism.

## 7.3 Final Remarks

The convergence of AI and FPGA technology represents the next frontier in financial engineering. This thesis provides a blueprint for navigating this complex landscape, offering a verified architecture that delivers the speed of hardware with the intelligence of software.

# Appendix A

# Appendix

## A.1 Setup Guide: Verilog on Apple Silicon

1. **Install Apple's Command Line Utilities**

   The first step is to install Apple's command line developer tools. This will provide you with essential tools like `git` and a compiler. Open your terminal and run the following command:

   ```
   xcode-select --install
   ```

2. **Install Homebrew**

   Homebrew is a package manager for macOS that simplifies the installation of software. To install Homebrew, execute the following command in your terminal:

   ```
   /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
   ```

3. **Install Verilator and SystemC**

   Next, you will install Verilator, a Verilog simulator, and SystemC, a C++ library for system-level modeling. Use Homebrew to install them with this command:

   ```
   brew install verilator systemc
   ```

4. **Install GTKWave**

   GTKWave is a waveform viewer that you will use to analyze the simulation results. Install it using Homebrew:

   ```
   brew install gtkwave
   ```

5. **Configure Your Environment**

   After installing all the necessary tools, you need to configure your environment. This involves setting up the required environment variables. You will need to add the following lines to your shell configuration file (e.g., `~/.zshrc` or `~/.bash_profile`):

   ```
   export SYSTEMC_HOME=/opt/homebrew/opt/systemc
   export VERILATOR_ROOT=/opt/homebrew/opt/verilator
   export PATH=$VERILATOR_ROOT/bin:$PATH
   ```

   After adding these lines, restart your terminal or source the configuration file for the changes to take effect (e.g., `source ~/.zshrc`).

## A.2 FPGA Code Explanation and Module Overview

This section provides an overview of the Verilog and VHDL code implementations that form the backbone of the proposed Tick-to-Trade (T2T) pipeline. The full code listings are provided in Appendix A.

### A.2.1 Network Ingress Modules

**Parser (Listing A.1):** Extracts Ethernet/IP/UDP headers and market data payloads, forming the entry point of the T2T pipeline. **Multicast Gate (Listing A.2):** Filters packets based on destination IP and port to ensure only relevant market feeds are processed.

### A.2.2 Feed Handler Modules

**Frame Extractor (Listing A.3):** Detects message boundaries and aggregates fixed-width data records. **Order Book Update (Listing A.4):** Maintains Level-1 order book data (best bid/ask), continuously updating state with each new message.

### A.2.3 Strategy and Decision Logic

**Strategy Decision Block (Listing A.5):** Implements a baseline threshold-based logic for buy/sell signals. This serves as the control baseline for the proposed reinforcement learning (RL) inference core. **Risk Gate (Listing A.6):** Applies strict hardware-level checks for notional exposure and message rate limits, ensuring compliance and preventing over-trading.

### A.2.4 Order Transmission Modules

**Order Encoder (Listing A.7):** Converts validated trade signals into exchange-compatible order messages (FIX/OUCH). **TX Bridge (Listing A.8):** Manages transmission of encoded orders to the network MAC layer, completing the tick-to-trade path.

### A.2.5 Control Plane and Integration

**AXI-Lite Register Interface (Listing A.9):** Provides a software-accessible control plane for updating parameters and reading telemetry metrics via PCIe/DMA. **Top-Level Integration (Listing A.10):** Connects all modules into a unified low-latency hardware pipeline, ensuring deterministic data flow from ingress to egress.

Each of these components is designed with deterministic latency in mind, and the overall architecture ensures that all decision-making, order handling, and safety checks occur in hardware to meet sub-microsecond trading requirements.

## A.3 Appendix: Core Implementation Snippets

**Note on Source Code:** Due to the extensive nature of the codebase, the full source listings, build scripts, and configuration files are hosted in the project repository at: `https://github.com/shreejitverma/trishul-ultra-hft-project`

The following sections provide **abridged snippets** of the critical modules discussed in the thesis, focusing on the core algorithmic logic and removing standard boilerplate.

### A.3.1 1. Network Ingress: Parser (Verilog)

The state machine responsible for zero-copy extraction of UDP payloads from raw Ethernet frames.

Listing A.1: Packet Parsing State Machine (Snippet)

```
// (Module ports and declarations omitted for brevity)
always @(posedge clk) begin
    if (rst) begin
        state <= ST_ETH; m_axis_tvalid <= 1'b0;
    end else begin
```

```
        // Pipeline control logic
        if (s_axis_tvalid && s_axis_tready) begin
            case(state)
                ST_ETH: state <= ST_IP; // Skip Eth Header
                ST_IP: begin
                    ip_src <= s_axis_tdata[31:0];
                    ip_dst <= s_axis_tdata[63:32];
                    state <= ST_UDP;
                end
                ST_UDP: begin
                    udp_sport <= s_axis_tdata[15:0];
                    udp_dport <= s_axis_tdata[31:16];
                    state <= ST_PAY;
                end
                ST_PAY: begin
                    // Streaming payload to downstream modules
                    m_axis_tdata <= s_axis_tdata;
                    m_axis_tvalid <= 1'b1;
                    if (s_axis_tlast) state <= ST_ETH;
                end
            endcase
        end
    end
end
```

Listing A.2: Multicast Gate Logic

```
always @(posedge clk) begin
    if (valid_udp) begin
        // Check Dest IP and Port against whitelist
        if (udp_dst_ip == 32'hE9_36_0C_6F && // 233.54.12.111
            udp_dst_port == 16'd5000) begin
```

```
            pass <= 1'b1;
        end else begin
            pass <= 1'b0; // Drop packet
        end
    end
end
```

### A.3.2 2. Feed Handler Modules

Listing A.3: Frame Extractor

```
// Accumulates bytes until a full message is received
always @(posedge clk) begin
    if (payload_valid) begin
        buffer <= {buffer[W-9:0], s_axis_tdata};
        count <= count + 1;
        if (count == MSG_LEN) begin
            msg_out <= buffer;
            msg_valid <= 1'b1;
            count <= 0;
        end
    end
end
```

Listing A.4: Order Book Update Logic

```
always @(posedge clk) begin
    if (tick_valid) begin
        if (tick_side == "B" && tick_price > best_bid) begin
            best_bid <= tick_price; // New Best Bid
        end
        if (tick_side == "S" && tick_price < best_ask) begin
```

```
            best_ask <= tick_price; // New Best Ask
        end
    end
end
```

### A.3.3 3. Strategy Logic: Decision Block (Verilog)

The hardware circuit that generates buy/sell signals based on pre-calculated price thresholds.

Listing A.5: Hardware Strategy Decision Logic

```
// Threshold comparison logic
wire buy_cond = (ask_px0 + thresh_buy < fair_px);
wire sell_cond = (bid_px0 - thresh_sell > fair_px);

always @(posedge clk) begin
    if (rst) begin
        buy <= 1'b0; sell <= 1'b0; out_valid <= 1'b0;
    end else begin
        out_valid <= 1'b0;
        // Trigger on valid market data update
        if (in_valid) begin
            // Single-cycle decision latency
            buy <= buy_cond;
            sell <= sell_cond;
            out_valid <= 1'b1;
        end
    end
end
```

### A.3.4 4. Hardware Pre-Trade Risk Gate (Verilog)

Ensures strict compliance with notional and message rate limits before order generation.

Listing A.6: Risk Check Logic

```
wire [63:0] notional_in = in_px * in_qty;


always @(posedge clk) begin
    if (rst) begin
        notional_accum <= 64'd0;
        pass <= 1'b0;
    end else begin
        if (in_valid) begin
            // Parallel check of all risk limits
            pass <= enable &&
                   (notional_accum + notional_in <= notional_limit)
                      &&
                   (msg_count + 1 <= msg_rate_limit);


            // Update state only if checks pass
            if (enable) begin
                notional_accum <= notional_accum + notional_in;
                msg_count <= msg_count + 1;
            end
        end
    end
end
```

### A.3.5 5. Order Transmission Modules

Listing A.7: Order Encoder (OUCH)

```
always @(posedge clk) begin
    if (trigger_buy) begin
        // Construct OUCH 'Enter Order' message
        packet[7:0]   <= 8'h4F; // Type 'O'
        packet[39:8]  <= token;
        packet[47:40] <= 8'h42; // 'B'uy
        packet[79:48] <= price;
        // ... (remaining fields)
        tx_valid <= 1'b1;
    end
end
```

Listing A.8: TX Bridge

```
// Adapts internal parallel bus to AXI-Stream MAC interface
assign m_axis_tdata = packet_shift_reg[63:0];
always @(posedge clk) begin
    if (m_axis_tready) begin
        packet_shift_reg <= packet_shift_reg >> 64;
    end
end
```

### A.3.6 6. Control Plane and Integration

Listing A.9: AXI-Lite Register Interface

```
// Maps PCIe writes to internal control registers
always @(posedge clk) begin
    if (axi_wvalid && axi_wready) begin
        case (axi_awaddr[7:0])
            8'h00: control_reg <= axi_wdata;
```

```
            8'h04: risk_limit_reg <= axi_wdata;
            8'h08: strategy_param_reg <= axi_wdata;
        endcase
    end
end
```

Listing A.10: Top-Level Integration

```
module top_wrapper (
    input wire clk,
    input wire [63:0] eth_rx_data,
    output wire [63:0] eth_tx_data
);
    // Instantiate and connect all modules
    rx_parser   u_parser (...);
    book_engine u_book (...);
    strat_core  u_strat (...);
    risk_gate   u_risk (...);
    tx_encoder  u_enc (...);
    // ...
endmodule
```

## A.4 Appendix: C++ Software Baseline Snippets

### A.4.1 1. High-Precision Timing (C++)

Calibration routine for the RDTSC (Read Time-Stamp Counter) instruction to achieve nanosecond precision.

```
1 void RDTSCClock::calibrate() noexcept {
2     constexpr int SAMPLES = 10;
3     double factors[SAMPLES];
```

```
4
5       // Sample system clock vs RDTSC multiple times
6       for (int i = 0; i < SAMPLES; ++i) {
7           auto sys_start = std::chrono::high_resolution_clock::
    now();
8           uint64_t tsc_start = __rdtscp(&aux);
9
10          std::this_thread::sleep_for(std::chrono::milliseconds
    (100));
11
12          auto sys_end = std::chrono::high_resolution_clock::now
    ();
13          uint64_t tsc_end = __rdtscp(&aux);
14
15          auto elapsed_ns = std::chrono::duration_cast<std::
    chrono::nanoseconds>
16                                (sys_end - sys_start).count();
17          factors[i] = static_cast<double>(elapsed_ns) / (
    tsc_end - tsc_start);
18      }
19
20      // Use median factor to reject OS jitter/outliers
21      std::sort(factors, factors + SAMPLES);
22      tsc_to_ns_factor_.store(factors[SAMPLES / 2], std::
    memory_order_release);
23  }
```

Listing A.11: RDTSC Calibration Routine

### A.4.2 2. RL Strategy Logic (C++)

The Avellaneda-Stoikov approximation used as the baseline heuristic strategy.

```cpp
void RLPolicyStrategy::run_inference() noexcept {
    auto bbo_bid = order_book_.best_bid();
    auto bbo_ask = order_book_.best_ask();
    if (bbo_bid.price == 0 || bbo_ask.price == INVALID_PRICE)
   return;

    // 1. Feature Extraction
    double mid_price = (bbo_bid.price + bbo_ask.price) / 2.0;

    // 2. Model Parameters (Tuned or Learned)
    double gamma = 0.1;        // Risk aversion
    double sigma = 2.0;        // Volatility proxy

    // 3. Avellaneda-Stoikov Approximation
    // Adjust reservation price based on current inventory
    double inventory_risk_adj = current_inventory_ * gamma * (
   sigma * sigma);
    double reservation_price = mid_price - inventory_risk_adj;

    // Calculate optimal spread centered on reservation price
    double half_spread = (sigma * 5000.0);
    if (half_spread < (bbo_ask.price - bbo_bid.price)/2.0) {
        half_spread = (bbo_ask.price - bbo_bid.price)/2.0;
    }

```

```
24      Price optimal_bid = static_cast<Price>(reservation_price -
        half_spread);
25      Price optimal_ask = static_cast<Price>(reservation_price +
        half_spread);
26
27      // 4. Order Generation (Quantized to tick size)
28      // (Order construction code omitted...)
29      if (optimal_bid > 0) order_queue_.push(buy_order);
30      if (optimal_ask > optimal_bid) order_queue_.push(
    sell_order);
31  }
```

Listing A.12: Heuristic Strategy Inference Logic

### A.4.3 3. Zero-Copy Parser (C++)

Optimized Ethernet header parsing using pointer arithmetic.

```
1  ParsedPacket parse(const uint8_t* packet, size_t len,
       Timestamp ts) noexcept {
2      ParsedPacket result{};
3      result.timestamp_ns = ts;
4
5      // Fast checks for packet validity
6      if (ULTRA_UNLIKELY(len < 42)) return result;
7
8      const auto* eth = reinterpret_cast<const EthernetHeader*>(
       packet);
9      if (ULTRA_UNLIKELY(ntohs_fast(eth->ethertype) != 0x0800))
```

```
    return result; // IPv4
10
11      const auto* ip = reinterpret_cast<const IPv4Header*>(
    packet + 14);
12      if (ULTRA_UNLIKELY(ip->protocol != 17)) return result; //
    UDP
13
14      // Extract pointers directly from buffer (Zero-Copy)
15      const uint8_t ip_hdr_len = (ip->version_ihl & 0x0F) * 4;
16      const size_t header_len = 14 + ip_hdr_len + 8; // Eth + IP
        + UDP
17
18      result.payload = packet + header_len;
19      result.payload_len = len - header_len;
20      result.valid = true;
21      return result;
22  }
```

Listing A.13: Ethernet/IP/UDP Header Parsing

## Bibliography

S. A. Al-Ahmed et al. A framework for high-frequency trading algorithms on fpga using soc. *Alexandria Engineering Journal*, 97:112–125, 2024. URL https://www.sciencedirect.com/science/article/pii/S1110016824003119.

S. Aouini et al. Dynamic fpga reconfiguration for scalable embedded artificial intelligence, 2025. URL https://www.sciencedirect.com/science/article/pii/S0167739X2500072X.

J. Briere. Advances in meta-reinforcement learning for financial markets, 2025. Preprint.

International Monetary Fund. Artificial intelligence can make markets more efficient—and more volatile, 2024. URL https://www.imf.org/en/Blogs/Articles/2024/10/15/artificial-intelligence-can-make-markets-more-efficient-and-more-volatile.

D. Gupta et al. Fpga for high-frequency trading: Reducing latency in financial systems. *IEEE Xplore*, 2024. doi: 10.1109/... URL https://ieeexplore.ieee.org/document/10841781/.

B. Jiang et al. Resolving latency and inventory risk in market making with reinforcement learning, 2025. URL https://arxiv.org/html/2505.12465v1.

C. Joshua, L. K. Safaei, H. G. Bargh, J. A. Beshel, and S. Motahar. Architecting low-latency interconnects for high-frequency trading using fpgas and rdma. *ResearchGate*, 2025. URL https://www.researchgate.net/publication/

395189202_Architecting_Low-Latency_Interconnects_for_High-Frequency_Trading_Using_FPGAs_and_RDMA.

S. Kumar et al. Deep reinforcement learning for high-frequency market making. *Proceedings of Machine Learning Research*, 189, 2023. URL https://proceedings.mlr.press/v189/kumar23a/kumar23a.pdf.

S. Kuzmanovic et al. Acceleration of trading system back end with fpgas using high-level synthesis. *Electronics*, 12(3):520, 2023. URL https://www.mdpi.com/2079-9292/12/3/520.

S. Lee et al. Lighttrader: A standalone high-frequency trading system with deep neural networks in 12+ tbps networks. In *Proceedings of the IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2023. URL https://rebellions.ai/wp-content/uploads/2023/11/RebellionsIONHPCA23_LightTrader.pdf.

H. Li et al. A multi-objective reinforcement learning approach with pareto fronts. *Expert Systems with Applications*, 262:125484, 2025. URL https://www.sciencedirect.com/science/article/pii/S0957417425024844.

H. Litz et al. High frequency trading acceleration using fpgas, n.d. URL https://people.ucsc.edu/~hlitz/papers/hft_fpga.pdf.

John W. Lockwood and Nishit Gupte. A low-latency library in fpga hardware for high-frequency trading (hft). In *20th Annual IEEE Symposium on High-Performance Interconnects*, 2012. doi: 10.1109/HOTI.2012.15. URL https://ieeexplore.ieee.org/document/6299067/.

MarketsandMarkets. Ai impact analysis on fpga industry: Key revenue insights, 2025a. URL https://www.marketsandmarkets.com/ResearchInsight/

ai-impact-analysis-fpga-industry.asp. Duplicate alias for citation compatibility.

MarketsandMarkets. Ai impact analysis on fpga industry: Key revenue insights, 2025b. URL https://www.marketsandmarkets.com/ResearchInsight/ai-impact-analysis-fpga-industry.asp.

MDPI. Special issue: Advanced ai hardware designs based on fpgas, 2024. URL https://www.mdpi.com/journal/electronics/special_issues/HD_FPGAs.

Napatech. Ai inference acceleration for trading with xelera, n.d. URL https://www.napatech.com/solutions/partners/ai-inference-acceleration-for-trading-with-xelera/.

J. Patel et al. A reinforcement learning approach to improve the performance of the avellaneda-stoikov market-making algorithm. *PLOS ONE*, 17(12):e0277042, 2022. doi: 10.1371/journal.pone.0277042. URL https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0277042.

N. Ragel. Reinforcement learning for systematic market making strategies, 2025. URL https://theses.hal.science/tel-04913317v1/file/131459_RAGEL_2024_archivage.pdf. Doctoral dissertation.

Global Banking & Finance Review. Why more trading firms are moving to fpga for low-latency gains, 2025. URL https://www.globalbankingandfinance.com/why-more-trading-firms-are-moving-to-fpga-for-low-latency-gains.

M. Schneider. Low-latency neural networks on fpga, 2023. Technical Report.

Lattice Semiconductor. Contextual ai: Enhancing edge intelligence with fpga technol-

ogy, 2025. URL https://www.latticesemi.com/en/Blog/2025/02/12/20/42/Contextual-AI-Enhancing-Edge-Intelligence-with-FPGA-Technology.

A. Shams et al. Comparative study of fpga and gpu for high-performance computing and ai. *ResearchGate*, 2024. URL https://www.researchgate.net/publication/382360577_Comparative_Study_of_FPGA_and_GPU_for_High-Performance_Computing_and_AI.

Thomas Spooner et al. Deep reinforcement learning for market making. pages 1892–1900, 2020. URL https://www.ifaamas.org/Proceedings/aamas2020/pdfs/p1892.pdf.

Z. Su et al. Market making strategies with reinforcement learning, 2025. URL https://arxiv.org/abs/2507.18680.

Fidus Systems. The role of fpgas in ai acceleration, 2024. URL https://fidus.com/blog/the-role-of-fpgas-in-ai-acceleration/.

A. Vakkilainen. Further optimizing market making with deep reinforcement learning. Master's thesis, Aalto University, 2023. URL https://aaltodoc.aalto.fi/server/api/core/bitstreams/34ca193c-c456-4321-8211-532cd441669b/content.

Y. Zhang et al. Hardware-efficient ai for edge computing, 2024. IEEE Conference.