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

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: November 2025

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 [14, 20]. 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 [19, 7]. Deep RL models, including recurrent architectures, have demonstrated enhanced performance in inventory management and order placement, particularly in stressed environments [22, 5]. Moreover, multi-objective RL frameworks have applied Pareto optimization to manage trade-offs between latency, volatility resilience, and slippage reduction [10]. 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 [4**?**]. Studies highlight their advantages in power efficiency, deterministic execution, and real-time data handling, which are essential for HFT infrastructures [11, 6]. Applications range from pre-built IP libraries for networking and protocol parsing [11] to FPGA-based system-on-chip frameworks for algorithmic execution [1]. Furthermore, integration of FPGAs with technologies like RDMA has enabled near-zero-copy communication pipelines, further reducing latency across trading networks [6]. 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 [**?** 21]. Dynamic FPGA reconfiguration has been proposed as a means to accommodate evolving RL or deep learning models in real time [2]. Early prototypes of AI-augmented FPGA trading systems suggest significant reductions in end-to-end latencies for market-making strategies [9, 8]. 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 [**?** 17], 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.

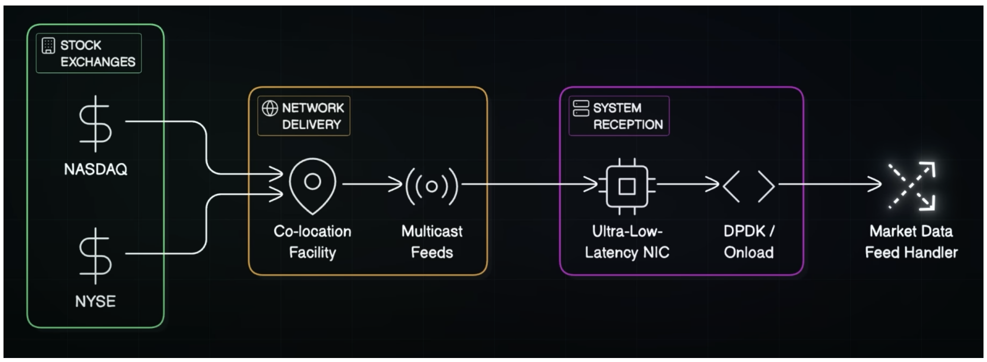


Figure 1.1: High Level Design - Part 1

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.

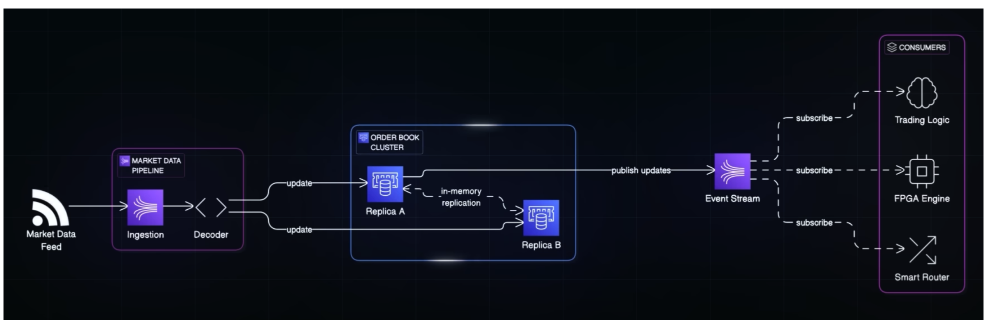


Figure 1.2: Order Book Management and Event-Driven Propagation

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 design, 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 details the architecture’s event-driven core, which is responsible for propagating market state changes with deterministic, nanosecond-level precision.

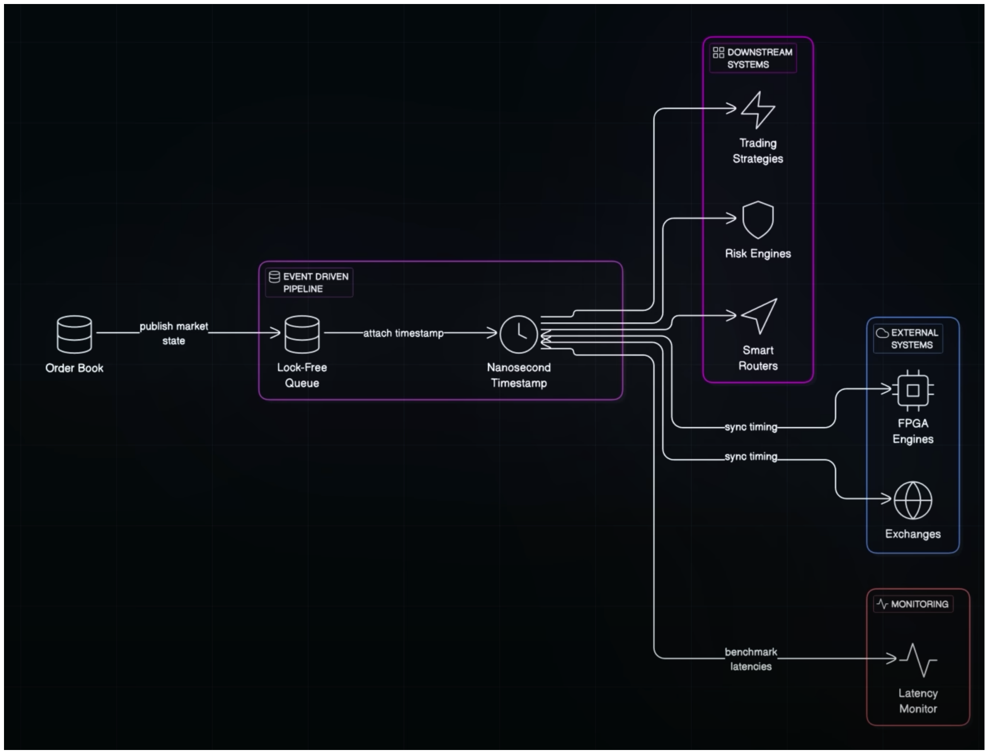


Figure 1.3: Event-Driven Pipeline and Nanosecond-Precision Timing

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.

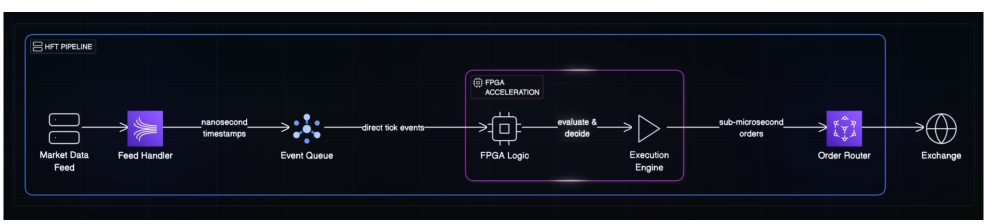


Figure 1.4: FPGA-Accelerated Tick-to-Trade Pipeline

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

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.

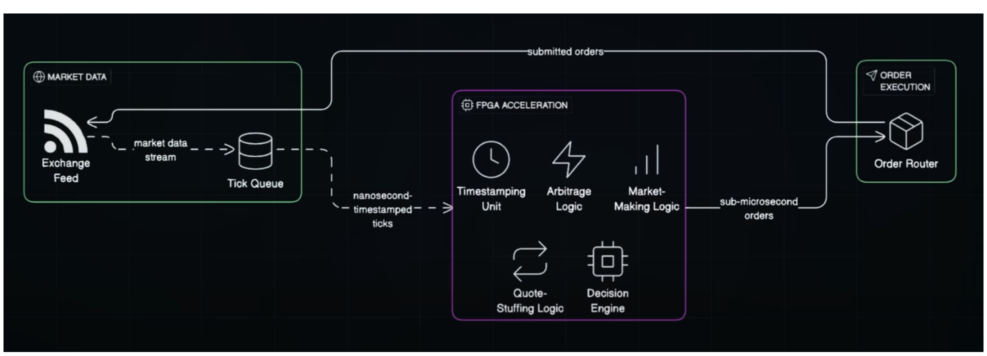


Figure 1.5: Internal Architecture of the FPGA Acceleration Module

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.

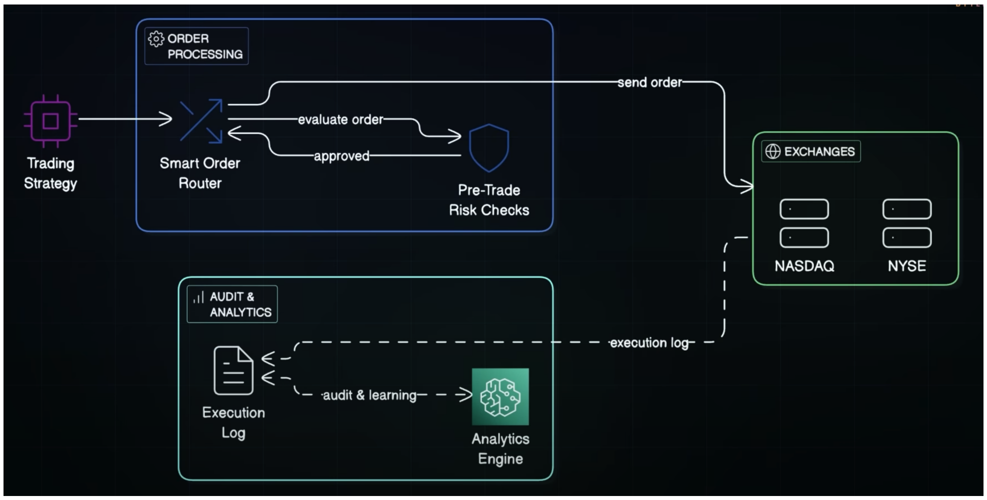


Figure 1.6: Order Processing and Post-Trade Analytics

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 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**

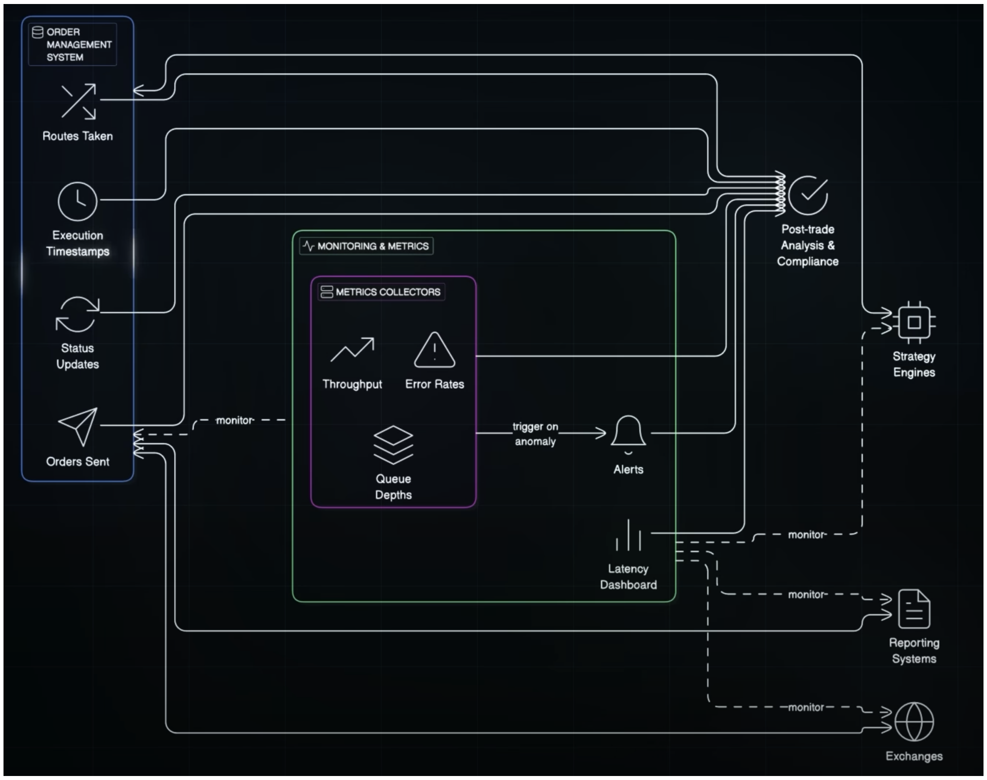


Figure 1.7: System-Wide Monitoring and Metrics Infrastructure

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 op-

erators.

- **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.

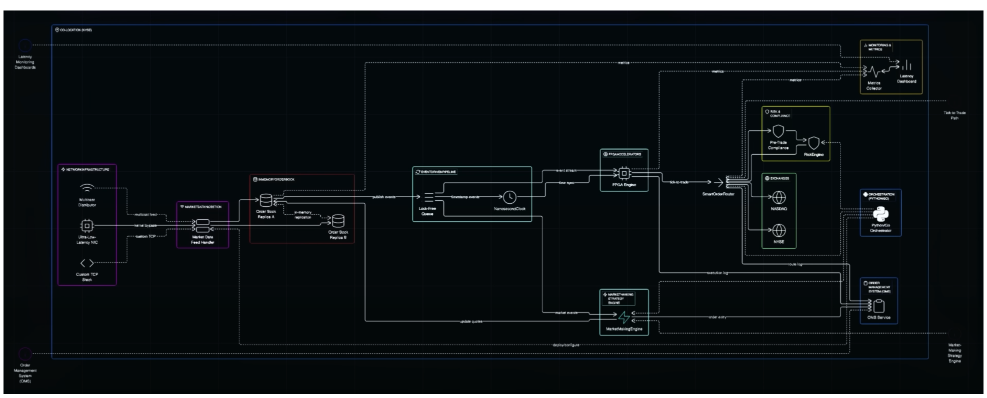


Figure 1.8: Unified High-Frequency Trading System Architecture

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. 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 [19, 14, 7, 22, 15, 20, 5].

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 [11**?** , 4, 1]. 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 [19, 14, 7, 22].

Recent advancements in multi-objective RL introduced Pareto front optimization, enabling agents to balance competing objectives such as profitability, inventory variance, and latency sensitivity [10]. 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 [11] 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 [4, 1].

Comparative research consistently shows that FPGAs outperform GPUs in latency-critical tasks, achieving predictable and energy-efficient performance even under peak market load [18]. Industrial reports underscore this transition, noting that leading trading firms are migrating core strategy components to FPGA fabrics for deterministic performance [16]. 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 [3].

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 quantization, systolic array design, and on-chip memory tiling enable inference to execute at nanosecond timescales [**?** 8, 21, 17, MDPI, 2, 12]. 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. [9] 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 [8**?** , 21]. 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 [15, 20, 5]. Additionally, while some works address risk control and inventory management, few incorporate comprehensive compliance, crossvenue 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 [**?**] and hardware-efficient AI [**?** **?**] 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 [19, 14, 7, 22, 15, 20, 5, 10] and state-of-the-art FPGA design techniques [11, 4, 6, 1, 18]. 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

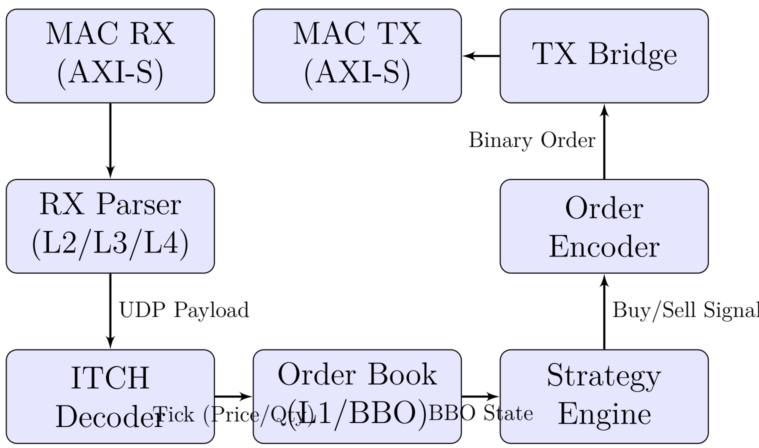
This literature review has charted the evolution of market-making methodologies from classical stochastic control to reinforcement learning and, finally, to hardware-accelerated AI systems. While RL frameworks introduce unprecedented adaptability, their computational overhead constrains real-world viability. 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 an FPGA Matching Engine

An FPGA-based matching engine is not a monolithic entity but rather a system of interconnected, highly-specialized modules working in harmony. Figure 3.1 illustrates the high-level architecture, showing the data flow from network ingress to egress.

Figure 3.1: High-Level Data Flow in the Implemented FPGA Matching Engine



3.1 Description of Key Modules

The system is composed of several core modules, each designed to perform a specific task with minimal latency.

RX Parser (`rx_parser`) *Responsibility:* Inspects incoming Ethernet packets to extract the UDP payload containing market data.

Implementation: A Finite State Machine (FSM) that tracks byte offsets to strip Ethernet (14 bytes), IP (20 bytes), and UDP (8 bytes) headers, streaming only the payload to the downstream decoder.

ITCH Decoder (`itch_decoder`) *Responsibility:* Parses the raw binary payload to identify specific market messages (e.g., ”Add Order”).

Logic: It accumulates streaming data into a buffer and extracts fixed-width fields (Timestamp, Price, Quantity) once a complete message is received, normalizing them into a unified ”Tick” format.

Order Book (`book2`) *Responsibility:* Maintains the ”Best Bid and Offer” (BBO) state in hardware registers.

Implementation: Optimized for single-cycle updates. It compares incoming ticks against the current BBO registers (‘bid_px0‘, ‘ask_px0‘) and updates them immediately if the new price is more competitive.

Strategy Engine (`strat_decide`) *Responsibility:* The decision-making core. It compares the current market BBO against pre-calculated ”fair value” thresholds.

Logic: Combinatorial logic checks conditions like ‘(Ask ¡ FairPrice - Threshold)‘ to trigger a BUY signal instantly when the market moves favorably.

Order Encoder (`order_encode`) *Responsibility:* Formats the internal BUY/SELL signal into a valid exchange-compliant binary message (e.g., OUCH).

Logic: A simple state machine that constructs a multi-word packet with the required headers, Price, and Quantity fields, ready for transmission.

TX Bridge (`tx_bridge`) *Responsibility:* Adapts the encoder’s output to the MAC layer’s timing requirements, ensuring valid AXI-Stream handshaking before the packet is placed on the wire.

Chapter 4

FPGA System Architecture and Methodology

To validate the hypothesis that an AI-integrated FPGA can outperform traditional market-making strategies, we propose a hybrid system architecture. This architecture is founded upon a deterministic, pure FPGA tick-to-trade (T2T) pipeline by replacing its static decision logic with a hardware-accelerated reinforcement learning (RL) inference core.

This design is composed of two primary components, both specified in the underlying technical architecture:

- **The Ultra-Low-Latency (ULL) Data Path:** A ”pure-in-gates” FPGA pipeline responsible for all operations on the critical path, from network ingress to order egress.
- **The Hybrid Control Plane:** A software-based system (running on a host CPU) responsible for training the RL model and updating its parameters on the FPGA via a non-critical control path.

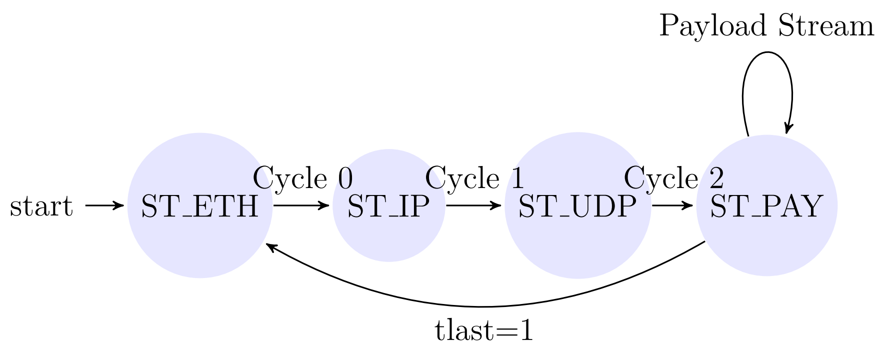
4.1 The Ultra-Low-Latency T2T Data Path (FPGA)

Our data path foundation is a deterministic, end-to-end T2T pipeline architecture. This design ensures our system operates with deterministic, sub-microsecond latency, eliminating OS jitter and software overheads.

4.1.1 Network Ingress & Parsing (rx_parser)

The system ingests 10/25GbE market data feeds directly from the PHY. The `rx_parser` module implements a Finite State Machine (FSM) to strip headers and extract the payload.

Figure 4.1: State Machine for RX Parser



The FSM cycles through:

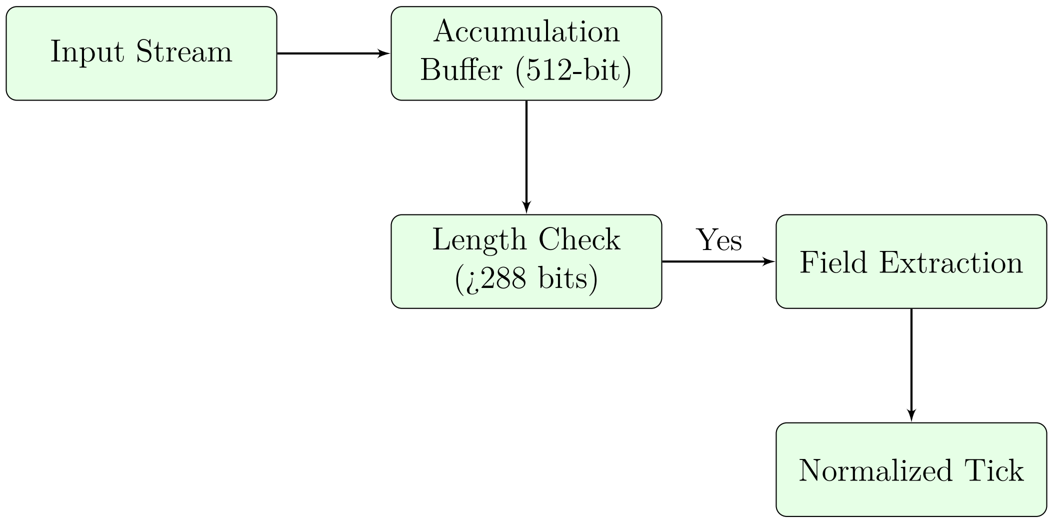
1. **ST_ETH:** Consumes the 14-byte Ethernet header.
2. **ST_IP:** Extracts Source/Dest IP from the 20-byte IPv4 header.
3. **ST_UDP:** Extracts Source/Dest Ports from the 8-byte UDP header and asserts `header_valid`.
4. **ST_PAY:** Streams the remaining payload data to the decoder until `tlast` is asserted.

4.1.2 Feed Handling & Book Building (itch_decoder)

The `itch_decoder` module is responsible for identifying and parsing specific market data messages.

Logic flow:

Figure 4.2: ITCH Decoder Logic Flow



- **Accumulation:** Incoming 64-bit words are shifted into a large internal buffer (‘buffer‘).
- **Detection:** The logic checks if the buffer contains enough bits for a full ”Add Order” message (Type ’A’, 36 bytes).
- **Extraction:** Once valid, it extracts the Order ID, Price, and Quantity fields using fixed bit-offsets and generates a single-cycle `tick_valid` pulse.

4.1.3 Strategy & Risk (The Core Integration)

4.2 Core Integration: The RL-Inference Module

The key innovation of this thesis is the replacement of the static `strat_decide` module (Listing 5). The baseline implementation uses a simple, threshold-based logic (e.g., `(ask_px0 + thresh_buy < fair_px)`). This is precisely the static model our literature review identifies as suboptimal in volatile markets.

We will replace this module with a custom-designed **RL-Inference Core**.

- **Inputs:** This new module will receive the same high-speed signals from the book builder (e.g., `bid_px0`, `ask_px0`) but will also be fed additional real-time state features, such as order flow imbalances, volatility metrics (calculated in hardware), and the current inventory state (held in registers).

- **Logic:** The core itself will be a pipelined neural network (or other RL-based model) implemented directly in Verilog/VHDL. It will be architected to meet the aggressive latency budget of the module it replaces (approx. 30–100 ns).

- **Outputs:** The module will output a `buy` or `sell` decision and the dynamically calculated `in_px` and `in_qty`. These outputs feed directly into the *existing* `risk_gate` module (Listing 6).

This design retains the safety of the original pipeline. The AI's decisions are still subject to deterministic, hardware-based pre-trade risk checks (e.g., `notional_limit`, `msg_rate_limit`). The AI can *propose* a trade, but the hard-wired risk module provides the final ”pass” or ”kill” signal, mitigating adverse selection from a misbehaving model.

4.3 The Hybrid Control Plane for Model Management

A key challenge in AI-FPGA integration is model training and updating. The FPGA is for *inference*, not *training*. We will leverage the optional PCIe/DMA control plane for this purpose. This ”slower” sideband channel is critical and will be used for:

1. **Model Deployment:** The CPU-based software stack will be responsible for training/retraining the RL model. After training, the optimized model weights (parameters) will be written into the FPGA's registers (e.g., BRAMs, LUTs)

via the AXI-Lite register file (Listing 9). This allows for dynamic model updates without recompiling the FPGA.

2. **Telemetry and Monitoring:** We will use this same PCIe path to export high-resolution timestamps, counters, and latency histograms, which is essential for our evaluation.

This hybrid approach ensures the critical trading path remains purely in hardware and is *never* back-pressured by the software/control plane. The implementation of this high-performance software plane is detailed in Section 5.

Chapter 5

Ultra-Low-Latency Software Baseline Implementation

As outlined in the evaluation methodology, a critical component of this research is to benchmark the AI-integrated FPGA system against a state-of-the-art, ”pure software” implementation. This section details the architecture and implementation of this ultra-low-latency (ULL) software baseline, which is designed to achieve tick-to-trade latencies under one microsecond. This system also serves as the foundation for the ”Hybrid Control Plane” used for model training and management.

5.1 Target Performance Metrics

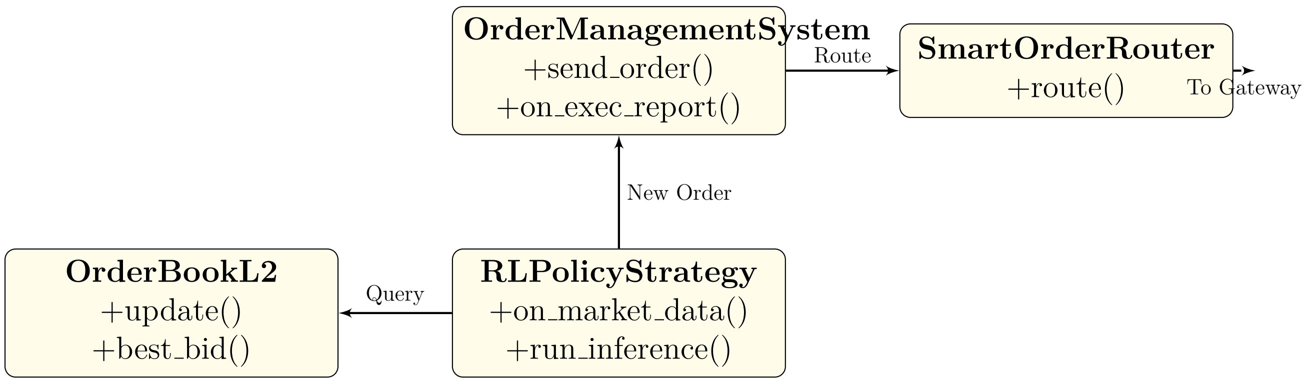
The software system is engineered to meet aggressive latency targets for each stage of the tick-to-trade pipeline:

- **Network Processing (Parsing):** ¡ 100ns
- **Book Update:** ¡ 50ns
- **RL Inference (Software):** ¡ 100ns
- **Risk Check:** ¡ 50ns
- **Order Generation:** ¡ 100ns
- **Total Target Tick-to-Trade:** ¡ 1 microsecond (1,000 nanoseconds)

5.2 Software System Architecture

The software baseline follows a layered, six-stage pipeline. The interactions between the core C++ components are illustrated in Figure 5.1.

Figure 5.1: C++ Software Component Diagram



The pipeline consists of:

1. **Hardware & Kernel Bypass:** Utilizes 10/25GbE NICs (e.g., Mellanox/Intel) with kernel-bypass technologies like DPDK or Solarflare Onload to ingest raw packets directly into user space, eliminating OS overhead.

2. **Network & Parsing:** A zero-copy parser decodes Ethernet, IP, and UDP headers, followed by a protocol-specific (e.g., ITCH) decoder.

3. **Market Data & Book Management:** Normalized events update an in-memory L2/L3 order book. The book state is then used for feature extraction.

4. **Strategy & Decision:** Extracted features are fed into the software-based RL inference engine (see Listing 16) to produce a quote decision.

5. **Risk & Execution:** The generated order is passed through a branchless, pre-trade risk check (Listing 18) before being sent to an exchange gateway.

6. **Telemetry & Monitoring:** All stages are instrumented with high-precision timestamps (Listing 11) to track latency and performance metrics.

5.3 Core Infrastructure Components

To achieve nanosecond-level performance in software, several core infrastructure components are required, as detailed in Appendix B.

- **RDTSC Clock:** A high-precision timer (Listing 11) using the `RDTSC` (Read Time-Stamp Counter) CPU instruction, calibrated to system time, for accurate latency measurements.
- **Lock-Free SPSC Queue:** A single-producer, single-consumer queue is used for passing events between pipeline stages (e.g., network thread to book-builder thread) without mutex/lock contention, targeting ¡20ns per operation.
- **Huge Page Allocator:** (Listing 12) Allocates memory in large 2MB or 1GB ”huge pages” to reduce Translation Lookaside Buffer (TLB) misses and ensure critical data structures (like the order book) are locked in physical memory.

5.4 Network and Market Data Processing

The ingress pipeline is optimized for zero-copy, branchless processing.

- **Ethernet/IP/UDP Parser:** (Listing 13) This parser reads packet headers directly from the NIC's DMA buffer. It uses fast, branch-predictable checks and bitwise operations to extract the UDP payload, targeting ¡50ns.
- **ITCH 5.0 Decoder:** (Listing 14) This decoder uses optimization techniques such as jump tables for message type dispatch, SIMD instructions (AVX2) for field extraction, and pre-computed hash tables for symbol lookups.
- **Order Book L2 Implementation:** (Listing 15) A cache-friendly, array-based L2 order book is used instead of tree-based structures. This provides faster,

more deterministic updates for limited-depth books, targeting ¡30ns per update.

5.5 Strategy and Risk Management

This is the ”brain” of the software system, designed to mirror the RL logic on the FPGA.

- **Heuristic Strategy Logic:** (Listing 16) The baseline strategy implements an approximation of the Avellaneda-Stoikov market-making model. This heuristic approach calculates reservation prices and optimal spreads based on inventory risk and volatility, providing a deterministic performance baseline against which the AI-driven FPGA model can be compared.
- **Feature Extraction:** (Listing 17) This module calculates features (e.g., imbalance, spread, volatility) from the order book state to be fed into the strategy logic.
- **Pre-Trade Risk Checks:** (Listing 18) A critical safety component. This logic is implemented to be branchless using bitwise operations to check all limits (notional, position, rate) simultaneously in ¡30ns.

5.6 Performance Tuning and Benchmarking

The software implementation relies on extensive system-level tuning (see Listing 20) to ensure deterministic performance. This includes:

- **CPU Isolation:** Using `isolcpus` and `nohz_full` to dedicate specific CPU cores exclusively to the trading application, shielding them from OS jitter.
- **Network Tuning:** Disabling interrupt coalescing, increasing ring buffers, and pinning NIC interrupts to specific, isolated cores.

- **Memory Tuning:** Enabling huge pages and disabling NUMA balancing.

This highly optimized software system provides a formidable baseline for evaluating the performance and latency advantages of the proposed AI-integrated FPGA architecture. The full pipeline is benchmarked using the high-precision clock (Listing 19).

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 ??):** 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 ??): Detects message boundaries and aggregates fixed-width data records. **Order Book Update (Listing ??):** 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.2): 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.3):** 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 ??): Converts validated trade signals into exchange-compatible order messages (FIX/OUCH). **TX Bridge (Listing ??):** 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 ??): Provides a software-accessible control plane for updating parameters and reading telemetry metrics via PCIe/DMA. **Top-Level Integration (Listing ??):** 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)

```verilog
// (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
```

A.3.2 2. Strategy Logic: Decision Block (Verilog)

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

Listing A.2: 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.3.3 3. Hardware Pre-Trade Risk Gate (Verilog)

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

Listing A.3: 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.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.

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

    // Sample system clock vs RDTSC multiple times
    for (int i = 0; i < SAMPLES; ++i) {
```

```
        auto sys_start = std::chrono::high_resolution_clock::
    now();
        uint64_t tsc_start = __rdtscp(&aux);

        std::this_thread::sleep_for(std::chrono::milliseconds
    (100));

        auto sys_end = std::chrono::high_resolution_clock::now
    ();
        uint64_t tsc_end = __rdtscp(&aux);

        auto elapsed_ns = std::chrono::duration_cast<std::
    chrono::nanoseconds>
                              (sys_end - sys_start).count();
        factors[i] = static_cast<double>(elapsed_ns) / (
    tsc_end - tsc_start);
    }

    // Use median factor to reject OS jitter/outliers
    std::sort(factors, factors + SAMPLES);
    tsc_to_ns_factor_.store(factors[SAMPLES / 2], std::
    memory_order_release);
}
```

Listing A.4: 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;
    }

```

```
    Price optimal_bid = static_cast<Price>(reservation_price -
    half_spread);
    Price optimal_ask = static_cast<Price>(reservation_price +
    half_spread);

    // 4. Order Generation (Quantized to tick size)
    // (Order construction code omitted...)
    if (optimal_bid > 0) order_queue_.push(buy_order);
    if (optimal_ask > optimal_bid) order_queue_.push(
   sell_order);
}
```

Listing A.5: Heuristic Strategy Inference Logic

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

Optimized Ethernet header parsing using pointer arithmetic.

```
ParsedPacket parse(const uint8_t* packet, size_t len,
   Timestamp ts) noexcept {
    ParsedPacket result{};
    result.timestamp_ns = ts;

    // Fast checks for packet validity
    if (ULTRA_UNLIKELY(len < 42)) return result;

    const auto* eth = reinterpret_cast<const EthernetHeader*>(
   packet);
    if (ULTRA_UNLIKELY(ntohs_fast(eth->ethertype) != 0x0800))
```

```
    return result; // IPv4

    const auto* ip = reinterpret_cast<const IPv4Header*>(
  packet + 14);
    if (ULTRA_UNLIKELY(ip->protocol != 17)) return result; //
  UDP

    // Extract pointers directly from buffer (Zero-Copy)
    const uint8_t ip_hdr_len = (ip->version_ihl & 0x0F) * 4;
    const size_t header_len = 14 + ip_hdr_len + 8; // Eth + IP
    + UDP

    result.payload = packet + header_len;
    result.payload_len = len - header_len;
    result.valid = true;
    return result;
}
```

Listing A.6: Ethernet/IP/UDP Header Parsing

Bibliography

[1] Al-Ahmed, S. A. et al. (2024). A framework for high-frequency trading algorithms on fpga using soc. *Alexandria Engineering Journal*, 97:112–125.

[2] Aouini, S. et al. (2025). Dynamic fpga reconfiguration for scalable embedded artificial intelligence.

[3] Fund, I. M. (2024). Artificial intelligence can make markets more efficient—and more volatile.

[4] Gupta, D. et al. (2024). Fpga for high-frequency trading: Reducing latency in financial systems. *IEEE Xplore*.

[5] Jiang, B. et al. (2025). Resolving latency and inventory risk in market making with reinforcement learning.

[6] Joshua, C., Safaei, L. K., Bargh, H. G., Beshel, J. A., and Motahar, S. (2025). Architecting low-latency interconnects for high-frequency trading using fpgas and rdma. *ResearchGate*.

[7] Kumar, S. et al. (2023). Deep reinforcement learning for high-frequency market making. *Proceedings of Machine Learning Research*, 189.

[8] Kuzmanovic, S. et al. (2023). Acceleration of trading system back end with fpgas using high-level synthesis. *Electronics*, 12(3):520.

[9] Lee, S. et al. (2023). 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)*.

[10] Li, H. et al. (2025). A multi-objective reinforcement learning approach with pareto fronts. *Expert Systems with Applications*, 262:125484.

[11] Lockwood, J. W. and Gupte, N. (2012). A low-latency library in fpga hardware for high-frequency trading (hft). In *20th Annual IEEE Symposium on High-Performance Interconnects*.

[12] MarketsandMarkets (2025). Ai impact analysis on fpga industry: Key revenue insights.

[MDPI] MDPI. Special issue: Advanced ai hardware designs based on fpgas.

[14] Patel, J. et al. (2022). A reinforcement learning approach to improve the performance of the avellaneda-stoikov market-making algorithm. *PLOS ONE*, 17(12):e0277042.

[15] Ragel, N. (2025). Reinforcement learning for systematic market making strategies. Doctoral dissertation.

[16] Review, G. B. . F. (2025). Why more trading firms are moving to fpga for low-latency gains.

[17] Semiconductor, L. (2025). Contextual ai: Enhancing edge intelligence with fpga technology.

[18] Shams, A. et al. (2024). Comparative study of fpga and gpu for high-performance computing and ai. *ResearchGate*.

[19] Spooner, T. et al. (2020). Deep reinforcement learning for market making. pages 1892–1900.

[20] Su, Z. et al. (2025). Market making strategies with reinforcement learning.

[21] Systems, F. (2024). The role of fpgas in ai acceleration.

[22] Vakkilainen, A. (2023). Further optimizing market making with deep reinforcement learning. Master's thesis, Aalto University.