INF8503

Design Project

Final Report

Hardware ITCH Protocol Parser on FPGA

Implementation of a Speculative Multi-Decoder Architecture
for Ultra-Low-Latency Market Data Processing

Jean-Claude Junior Raymond

Department of Computer and Software Engineering
Polytechnique Montréal

Contents

1 Introduction

1.1 Project Description

This project aims to design and implement a hardware parser for the ITCH protocol on FPGA, targeting high-frequency trading (HFT) applications. The ITCH protocol is a market data dissemination protocol used by NASDAQ to transmit real-time information about orders and stock transactions.

Our implementation is based on the speculative, macro-driven architecture proposed by Zhang [**?**], which we have adapted and extended for deployment on the PYNQ-Z2 platform. This report documents our complete implementation, validation methodology, and results.

1.1.1 High-Frequency Trading Context

High-Frequency Trading represents an ultra-optimized variant of algorithmic trading characterized by several fundamental aspects. First, automation is complete: orders are placed automatically by algorithms without human intervention in the decision loop. Second, opportunity windows are extremely short, typically lasting only milliseconds or even microseconds for arbitrage opportunities. Third, there exists a genuine race for latency where, to be profitable, an HFT system must process market data very quickly, make instantaneous decisions, and send orders faster than competitors.

1.1.2 The Importance of Information Asymmetry

Trading fundamentally relies on exploiting information asymmetries. The principle is simple: whoever receives information first can act before others and capture profit opportunities. Although these asymmetries are shorter today than in the past due to technological advances, they remain impossible to eliminate completely due to fundamental physical constraints.

The first constraint is physical distance: the speed of light imposes a fundamental limit on signal propagation time. A signal traveling 300 km via fiber optic requires approximately 1 ms. The second constraint is transmission speed: transmission technologies (fiber optic, microwave, laser links) have different latency characteristics. The third constraint is computation speed: the time required to process received information and generate a response.

Among these three factors, only processing and reaction time represents a lever truly exploitable through engineering. This is precisely the objective of this project: minimizing this processing time through a dedicated hardware implementation.

1.1.3 Project Objectives

The main objective is to build a hardware ITCH parser capable of reading an ITCH data stream continuously byte by byte, automatically detecting the message type among the 9 supported types, extracting essential fields from each message (order reference, price, quantity, etc.), generating a uniform canonical output enabling simplified downstream processing, and achieving a processing latency of a single clock cycle after receiving the last byte of a message.

1.2 Literature Review

This project is primarily based on the article by R. Zhang entitled *"Speculative, Macro-Driven FPGA Architecture for Ultra-Low-Latency ITCH Parsing in High-Frequency Trading Systems"* published on TechRxiv in 2023 [**?**]. This article proposes an innovative FPGA architecture using speculative parallel decoding to minimize ITCH message parsing latency.

The approach presented in the reference article is distinguished by several innovations. Speculative decoding constitutes the first innovation: rather than waiting for message type identification before starting parsing, all decoders start simultaneously from the first byte, thus eliminating type determination latency. The macro-driven architecture represents the second innovation: using Verilog macros allows automatic generation of decoders for each message type from a high-level protocol description. The suppression mechanism constitutes the third innovation: decoders that do not match the received message type automatically deactivate after the first byte, thus avoiding erroneous outputs.

The article reports parsing latencies on the order of a few nanoseconds, representing a significant advantage compared to traditional software implementations that typically require several microseconds.

1.3 Adopted Approach

1.3.1 Limitations of Traditional Processors

Traditional processors (CPUs) present several fundamental limitations for HFT applications. Regarding high execution latency, modern CPUs sequentially execute ISA (Instruction Set Architecture) instructions. Even with sophisticated pipelines, this approach generates significant overhead including instruction fetch from memory, instruction decoding, register and memory access, execution in arithmetic units, and result write-back.

More problematic still, CPU latency is non-deterministic. The same code can take different amounts of time from one execution to another due to hardware and software interrupts, operating system preemption, cache effects (hits vs. misses), branch prediction (correct vs. incorrect), and memory bus contention. In the HFT context, a single delay can be very costly.

1.3.2 FPGA Advantages

FPGAs (Field-Programmable Gate Arrays) offer an elegant solution to these problems. Deterministic latency allows hardware logic to produce its results in a fixed and known number of clock cycles. Very low latency is made possible by parallel hardware processing that allows performing many operations simultaneously. The absence of system overhead means there is no operating system, no unsolicited interrupts. Finally, data path customization allows optimizing the architecture specifically for the ITCH protocol.

Our approach therefore consists of implementing nine parallel decoders on FPGA, each specialized for a specific ITCH message type, operating simultaneously according to a speculative scheme.

2 Software Design

2.1 Tools and Development Environment

The development of this project employed a four-phase validation methodology, combining original work with infrastructure provided by the article author. This progressive approach allows detecting errors as early as possible in the development cycle.

Figure 1 illustrates this four-phase methodology. The first phase consists of a pure Python software simulation (golden model) developed entirely as original work to serve as a reference implementation. The second phase uses Cocotb with Icarus Verilog—this framework was provided by the article author, and was extended to support the additional decoders implemented in this project. The third phase employs Vivado Simulator for pre-synthesis validation. The fourth phase validates the system on the PYNQ target with real hardware.

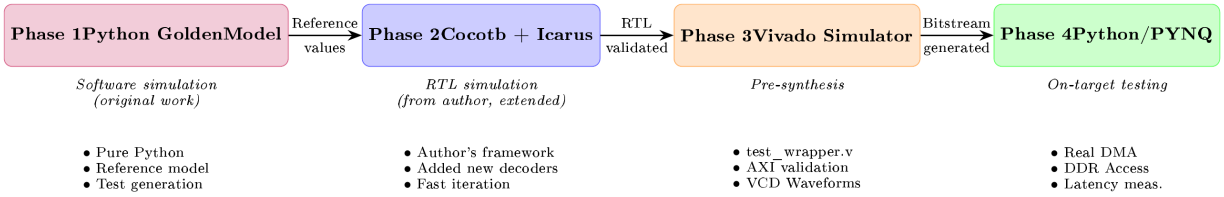

Figure 1: Four-phase validation methodology.

2.1.1 Phase 1: Python Golden Model (Original Work)

Before any hardware implementation, a software reference model was developed entirely in Python. This golden model implements exactly the same parsing logic as the hardware: it reads a byte stream, identifies the message type based on the first byte, extracts fields according to the ITCH 5.0 format, and produces a canonical data structure.

The golden model serves two essential purposes: first, to validate the correctness of the parsing algorithm in a pure software environment before RTL implementation; second, to generate expected reference values that are used to verify hardware parser outputs in subsequent phases. This approach enables exhaustive validation—thousands of random messages can be generated and verified automatically.

2.1.2 Phase 2: Cocotb RTL Simulation (Extended from Author)

The article author provided a Cocotb (Coroutine-based Cosimulation TestBench) framework for RTL simulation using Icarus Verilog. This Python-based verification environment was originally designed to test the core decoder architecture.

To support the 9 message types implemented in this project, the original test infrastructure was extended by adding the new decoders to the existing files. The modifications included updating the payload generators and comparison logic for each new message type.

Figure 2 presents the organization of the Cocotb tests. The central Makefile orchestrates the execution of three Python test files: `test_integrated.py` for complete pipeline testing with all 9 message types, `test_parser_canonical.py` for canonical output validation at the parser level, and `test_valid_drop_abort.py` for mid-packet abort recovery testing. These tests use common helper functions (`payload_generator`, `compare`, `recorder`, `reset`) that encapsulate reusable functionality

such as valid ITCH message generation and result comparison. The Design Under Test (DUT) is the `integrated.v` module that combines the parser and latch stage.

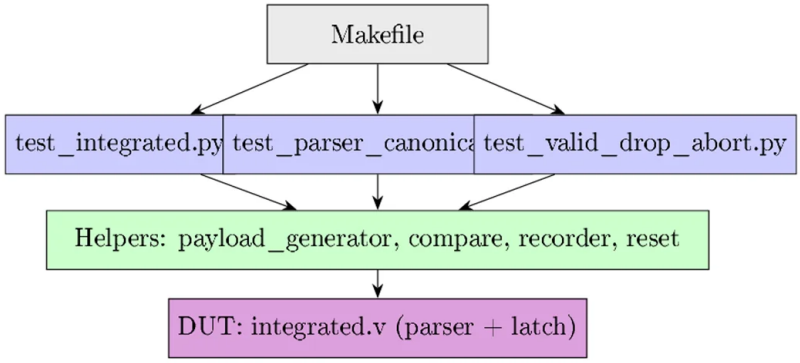

Figure 2: Cocotb test organization (extended from author's framework).

2.1.3 Phase 3: Pre-Synthesis Validation with Vivado Simulator

For FPGA implementation, validation was transferred to Vivado Simulator. This transition was essential to guarantee compatibility with Xilinx primitives, validate timing before synthesis, and verify AXI interface behavior which is not simulated by Cocotb.

2.1.4 Test Environment Comparison

The choice of test environment depends on the development phase and aspects to validate. Figure 3 compares the characteristics of the three RTL/hardware environments (phases 2-4). Cocotb uses Python as the test language with Icarus/Verilator as simulator, offering fast iteration speed but only approximate timing fidelity and no AXI interface support. Vivado Simulator uses Verilog with xsim, offering medium iteration speed but cycle-accurate fidelity with full AXI interface support and waveform generation. Finally, PYNQ uses Python on real hardware with slower iteration speed but perfect fidelity including real DMA, enabling debugging via registers.

Characteristic	**Cocotb**	**Vivado Sim**	**PYNQ**
Test Language	Python	Verilog	Python
Simulator	Icarus/Verilator	xsim	Real Hardware
Iteration Speed	Fast	Medium	Slow
Timing Fidelity	Approximate	Cycle-accurate	Real
AXI Interfaces	No	Yes	Yes + DMA
Debugging	VCD + Python	Waveforms	Registers

Figure 3: Test environment comparison (Phases 2-4).

2.2 Functional Model Description

2.2.1 Multi-Decoder Architecture

The system core is a multi-decoder architecture composed of 9 ITCH decoders operating in parallel from the first received byte. This speculative approach completely eliminates the latency that would be necessary in a traditional architecture to first identify the message type before routing to the appropriate decoder.

Figure 4 illustrates this architecture. The input signals `byte_in[7:0]` and `valid_in` are distributed simultaneously to all 9 decoders (ADD Type 0, CANCEL Type 1, DELETE Type 2, EXEC Type 3, through Broken Type 8). Each decoder produces two types of outputs: a `valid[n]` signal indicating that this decoder has finished parsing a message that matches it, and a set of `fields[n]` containing data extracted from the message.

The signals `valid[0]` through `valid[8]` form a one-hot vector `valid_vec[8:0]` where at most one bit is active at a time, indicating which decoder recognized the message. This vector serves as the selection signal for a 9:1 multiplexer that routes fields from the active decoder to the canonical output (`parsed_type`, `order_ref`, `shares`, `price`, etc.).

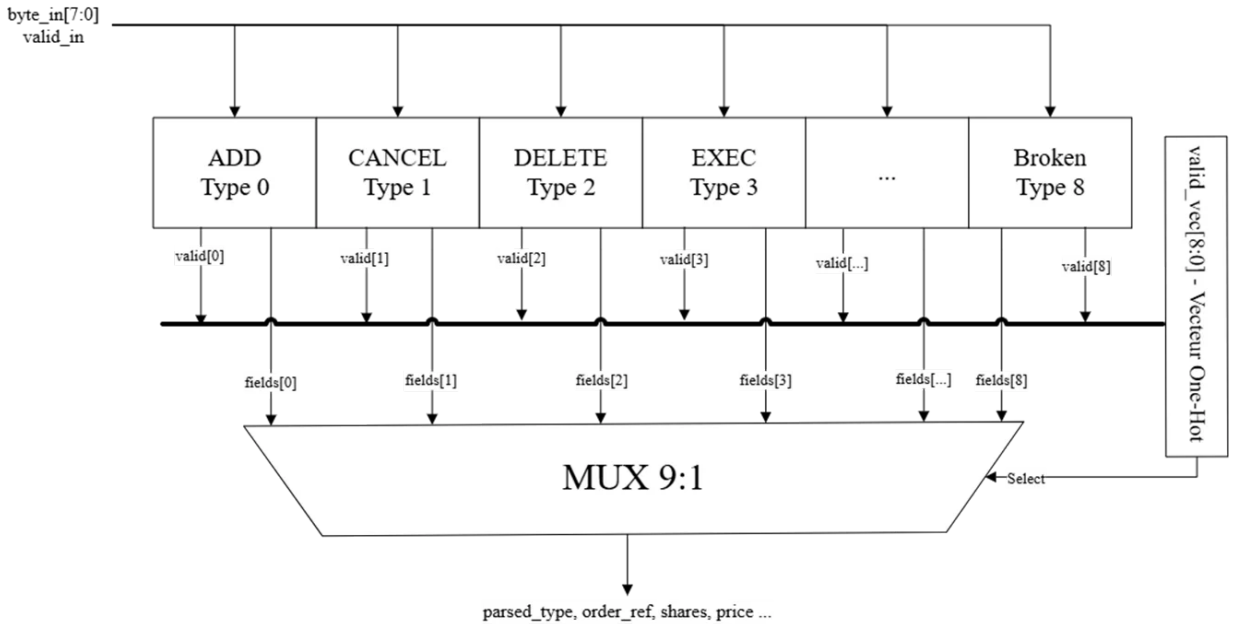

Figure 4: Multi-decoder parser architecture.

2.2.2 Supported Message Types

Table 1 presents the 9 ITCH message types supported by our implementation. These types cover essential order book operations: adding, modifying, and deleting orders, as well as execution and transaction notifications.

Table 1: Supported ITCH message types.

Type	Name	Char	Size	Extracted Fields
0	Add Order	'A'	36 B	order_ref, side, shares, stock, price
1	Cancel Order	'X'	23 B	order_ref, canceled_shares
2	Delete Order	'D'	9 B	order_ref
3	Executed Order	'E'	30 B	order_ref, executed_shares, match_id
4	Replace Order	'U'	27 B	old_order_ref, new_order_ref, shares, price
5	Trade	'P'	40 B	order_ref, side, shares, stock, price, match_id
6	Add Order MPID	'F'	40 B	order_ref, side, shares, stock, price, attribution
7	Exec with Price	'C'	36 B	order_ref, exec_shares, match_id, price
8	Broken Trade	'B'	19 B	reason_code, match_id

The Add Order message (Type 'A') signals the arrival of a new order in the order book with its unique reference, side (buy 'B' or sell 'S'), quantity, 8-character stock symbol, and limit price. The Delete Order message (Type 'D') is the shortest with only 9 bytes, signaling complete removal of an order identified by its reference. The Trade message (Type 'P') is the longest with 40 bytes, representing an executed transaction with all details including the match_id for traceability.

2.2.3 Suppression Mechanism

The suppression mechanism is crucial for proper operation of the speculative architecture. When a decoder receives the first byte of a message and determines it is not its type, it must ignore subsequent bytes until the end of the message to correctly resynchronize with the next message.

Figure 5 shows the SystemVerilog `itch_length` function that implements this lookup table. This function takes the message type character (`msg_type`) as input and returns the corresponding length in bytes. For example, character 'A' (0x41) returns 36 for Add Order, 'D' (0x44) returns 9 for Delete Order, and 'P' (0x50) returns 40 for Trade. The `default` case returns 2 as a fallback for unrecognized types, allowing graceful recovery.

```
function automatic logic [5:0] itch_length(input logic [7:0] msg_type);
    case (msg_type)
        "A": return 36;  // Add Order
        "F": return 40;  // Add Order - MPID Attribution
        "X": return 23;  // Cancel Order
        "U": return 27;  // Replace Order
        "D": return 9;   // Delete Order
        "E": return 30;  // Executed Order
        "C": return 36;  // Executed Order With Price
        "P": return 40;  // Trade
        "B": return 19;  // Broken Trade
        default: return 2; // Fallback
    endcase
endfunction
```

Figure 5: Message length lookup table.

Concretely, when a decoder (for example the Add Order decoder) receives a first byte 'D' (Delete), it consults this table, obtains the value 9, and enters suppression mode for 9 cycles. During this time, it ignores all incoming bytes and produces no output. At cycle 10, it is ready to receive the first byte of the next message.

2.2.4 Decoder Temporal Operation

Figure 6 presents the timing diagram for parsing a DELETE message, the shortest of the 9 types with only 9 bytes. This diagram illustrates cycle-by-cycle decoder behavior.

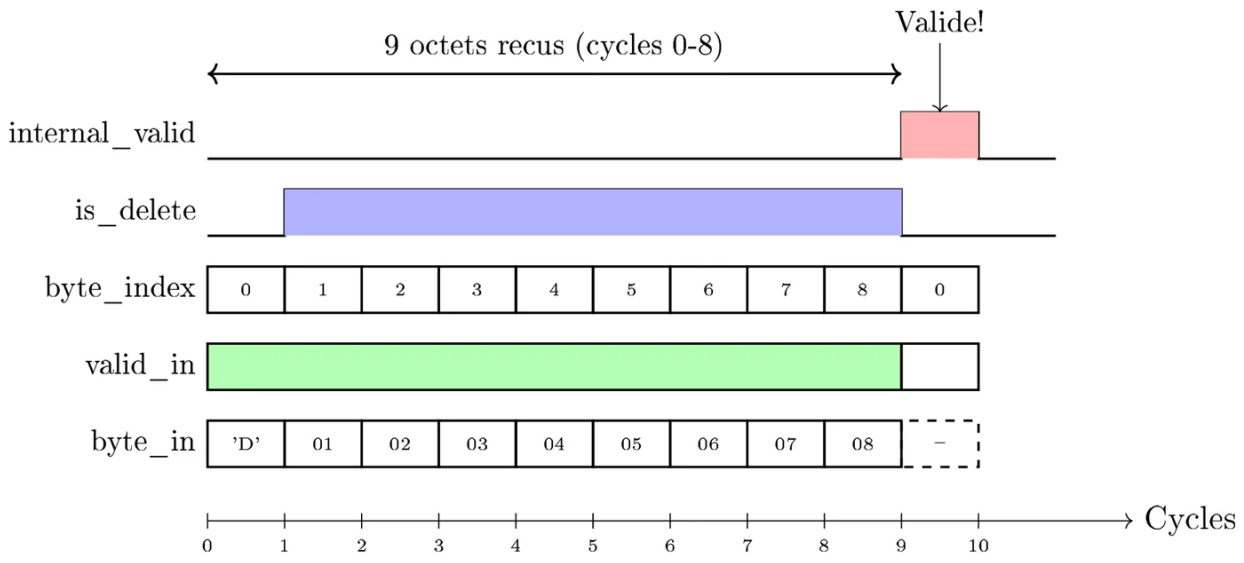

Figure 6: DELETE message parsing timing diagram.

At cycle 0, the `byte_in` signal contains character 'D' (0x44). The DELETE decoder recognizes its type and activates the `is_delete` signal. The `byte_index` counter is at 0. From cycles 1 to 8, the 8 bytes of the order reference (01, 02, 03, 04, 05, 06, 07, 08 in this example) are received sequentially. The `valid_in` signal remains active throughout the message duration, and `byte_index` increments each cycle. The `is_delete` signal remains active throughout parsing.

At cycle 9, the decoder has received all necessary bytes. The `internal_valid` signal activates for one cycle (represented by the red "Valid!" rectangle), signaling that parsed fields are valid and ready to be latched. From cycle 10 onward, the decoder is ready for the next message, `byte_index` returns to 0, and `valid_in` can become inactive if no new message arrives immediately.

2.3 Results

2.3.1 Validation with Vivado Simulator

Vivado simulation enables detailed analysis of parser behavior with cycle-accurate fidelity. Figure 7 shows waveforms during parsing of an Add Order MPID message (Type 'F', 40 bytes).

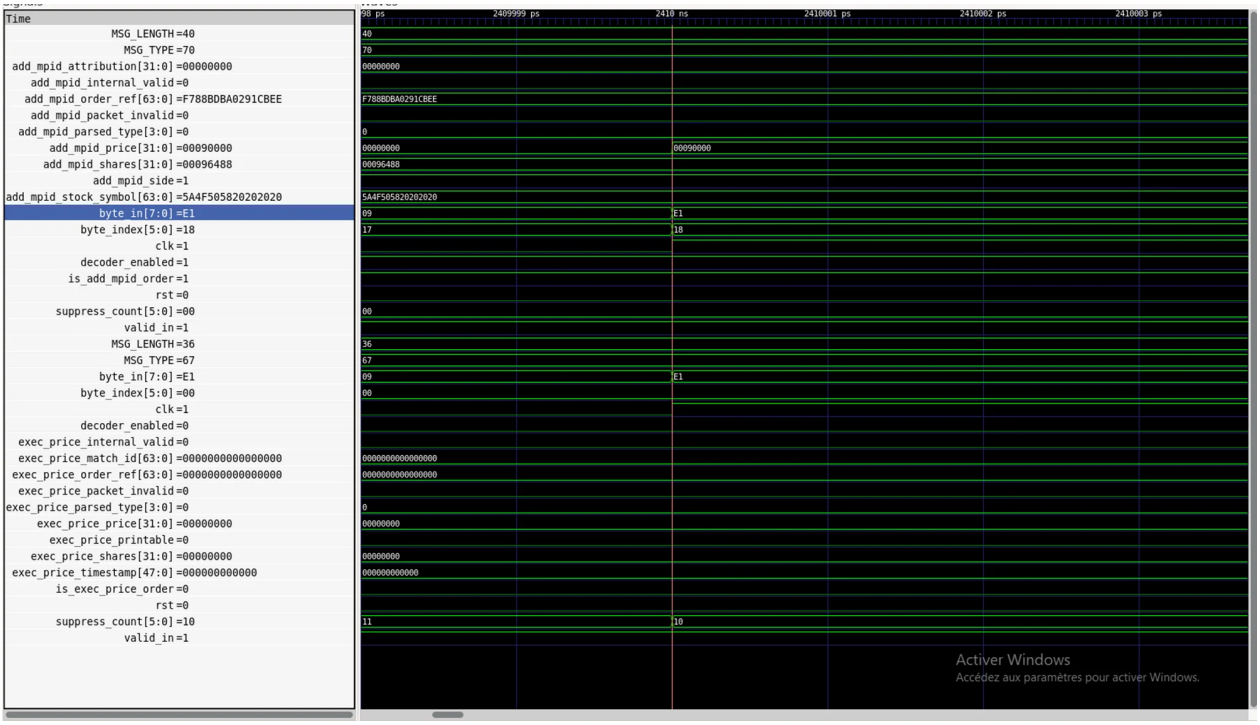

Figure 7: Vivado simulation – Add Order MPID message parsing.

In this capture, several important signals are observed. The signals `MSG_LENGTH=40` and `MSG_TYPE=70` (character 'F' in hexadecimal) confirm this is an Add Order MPID message. The signal `add_mpid_order_ref[63:0]` progressively accumulates the order reference over 8 bytes. The signals `add_mpid_price`, `add_mpid_shares`, and `add_mpid_side` show extracted fields. The signal `is_add_mpid_order=1` confirms this decoder is active while `is_exec_price_order=0` shows another decoder (Execute with Price) is in suppression mode with `suppress_count` decrementing.

Figure 8 presents the module-level testbench validating latched outputs after a DELETE message.

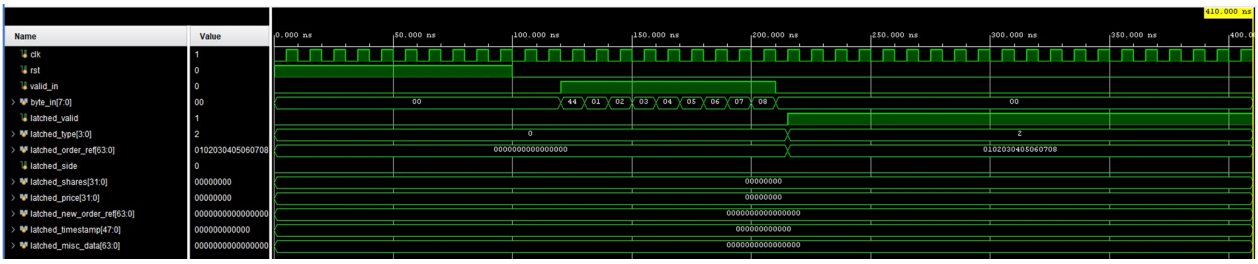

Figure 8: Vivado simulation – module-level testbench.

Here we observe the complete sequence: `byte_in` goes from 00 to 44 (character 'D') then to bytes 01-08 of the order reference. The `latched_valid` signal goes to 1 once parsing is complete, `latched_type` indicates 2 (code for DELETE), and `latched_order_ref` contains the value 0102030405060708 correctly assembled. Other fields (`latched_shares`, `latched_price`, etc.) remain at 0 as they are unused by this message type.

Figure 9 shows AXI interface validation at the system level.

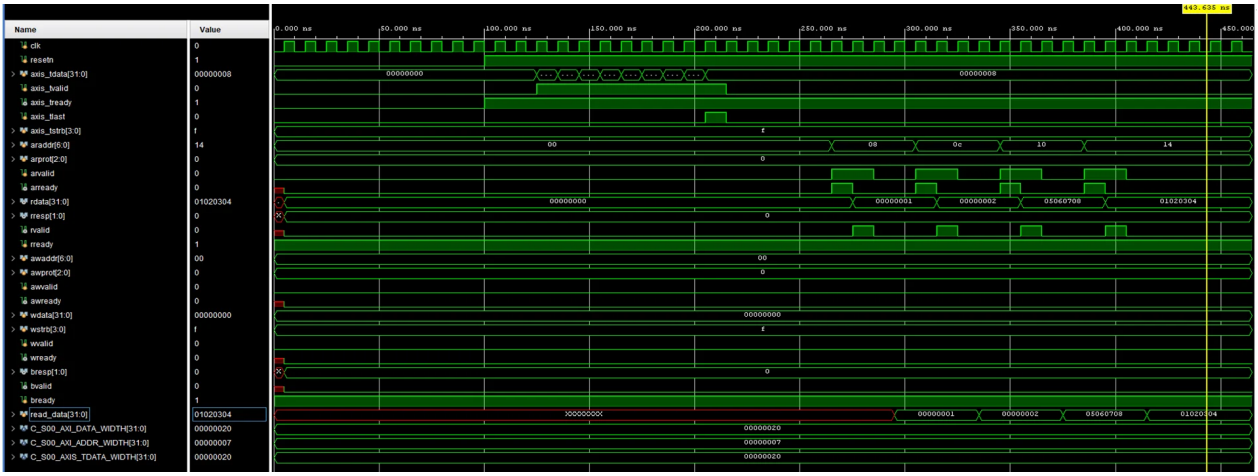

Figure 9: AXI interface simulation.

This simulation validates the complete AXI protocol. For the AXI-Stream interface (data input), we observe `axis_tdata` carrying message bytes, `axis_tvalid` indicating data validity, and `axis_tready` always at 1 because the parser accepts data without backpressure. For the AXI-Lite interface (register reading), we observe `araddr` containing the register address to read (00, 08, 0C, 10, 14...), `arvalid`/`arready` for address handshake, and `rdata` returning register values with `rvalid`/`rready` for data handshake.

2.3.2 Validation on PYNQ Target

Figure 10 presents the Python code used for final validation on the PYNQ-Z2 board.

```python
def run_all_tests():
    """Execute les 9 tests de types de messages"""
    tests = [
        ('ADD',         gen_add_order(...),         0),
        ('CANCEL',      gen_cancel_order(...),      1),
        ('DELETE',      gen_delete_order(...),      2),
        ('EXEC',        gen_exec_order(...),        3),
        ('REPLACE',     gen_replace_order(...),     4),
        ('TRADE',       gen_trade(...),             5),
        ('ADD_MPID',    gen_add_order_mpid(...),    6),
        ('EXEC_PRICE',  gen_exec_price(...),        7),
        ('BROKEN',      gen_broken_trade(...),      8),
    ]

    for name, msg_bytes, expected_type in tests:
        send_message(msg_bytes)       # DMA transfer
        time.sleep(0.001)
        msg = read_message()          # AXI-Lite read

        if msg and msg['type_code'] == expected_type:
            print(f"  V {name:12} - PASS")
        else:
            print(f"  X {name:12} - FAIL")

run_all_tests()  # Execute tous les tests
```

Figure 10: Python test code on PYNQ.

The `run_all_tests()` function defines a list of 9 tests, one for each message type. Each list entry is a tuple containing the test name (e.g., 'ADD', 'CANCEL'), a generator function that creates the binary payload conforming to the ITCH 5.0 protocol, and the expected type code (0 to 8).

The main loop iterates over these tests. For each test, `send_message(msg_bytes)` performs the DMA transfer of bytes to the parser. After a short 1 ms wait to allow parsing to complete, `read_message()` reads result registers via AXI-Lite and returns a dictionary containing parsed fields. Verification compares the read `type_code` with the expected value, displaying "PASS" or "FAIL" for each test.

Result: All 9 message types were successfully validated on the hardware target.

3 Hardware Architecture

3.1 Target Platform

The system is implemented on the PYNQ-Z2 board equipped with the Zynq-7000 SoC (XC7Z020-1CLG400C) from Xilinx. This SoC integrates on a single chip a Processing System (PS) based on a dual-core ARM Cortex-A9 processor and Programmable Logic (PL) based on an Artix-7 family FPGA fabric.

Figure 11 presents the high-level architecture of the complete system.

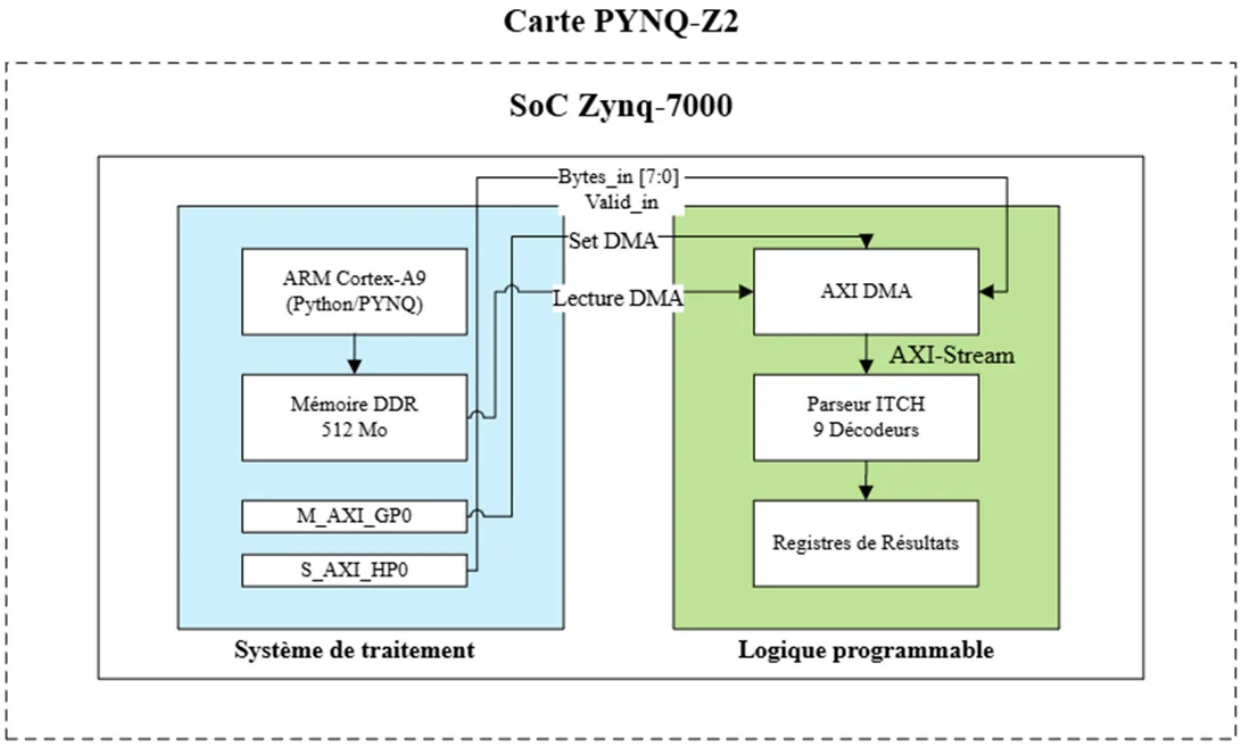

Figure 11: Overall architecture on Zynq-7000 SoC.

The left side (in blue) represents the Processing System containing the ARM Cortex-A9 processor running Python/PYNQ, 512 MB DDR memory for data storage, and master (`M_AXI_GP0`) and slave (`S_AXI_HP0`) AXI interfaces for communication with the programmable logic.

The right side (in green) represents the Programmable Logic containing the AXI DMA that reads data from DDR and converts it to AXI-Stream, the ITCH parser with its 9 parallel decoders that processes the incoming byte stream, and result registers that store parsed fields for CPU reading.

The arrows show data flow: `Bytes_in[7:0]` and `Valid_in` enter the parser from the DMA, `Set DMA` and `Read DMA` represent CPU control commands to the DMA, and the AXI-Stream connection transports bytes from the DMA to the parser.

3.2 High-Level Architecture

3.2.1 Data Flow

Figure 12 illustrates data flow through the system in 7 sequential steps.

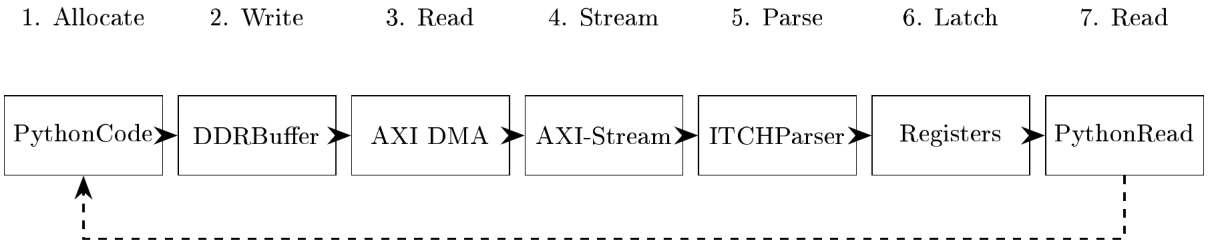

Figure 12: Data flow in 7 steps.

Step 1 (Allocation) sees Python code allocate a contiguous DMA buffer in DDR memory using PYNQ's `allocate` class. This buffer must be physically contiguous because DMA does not support paged virtual memory. Step 2 (Write) consists of writing ITCH message bytes to the buffer according to the ITCH 5.0 protocol format. Step 3 (Read) configures the DMA with source address (buffer) and transfer length via DMA control registers.

Step 4 (Stream) sees the DMA read from DDR via the high-performance `S_AXI_HP0` interface and send bytes via AXI-Stream to the parser, at one byte per clock cycle. Step 5 (Parse) has the ITCH parser process bytes in real-time with all 9 decoders operating in parallel. Step 6 (Latch) locks parsed fields in output registers at message end. Finally, Step 7 (Read) allows the CPU to read parsed results via the AXI-Lite interface.

The dashed arrow returning from "Python Read" to "Python Code" indicates that the process can repeat for the next message, with the buffer being reusable.

3.2.2 Vivado Block Design

Figure 13 shows the complete Vivado block design with all interconnected components.

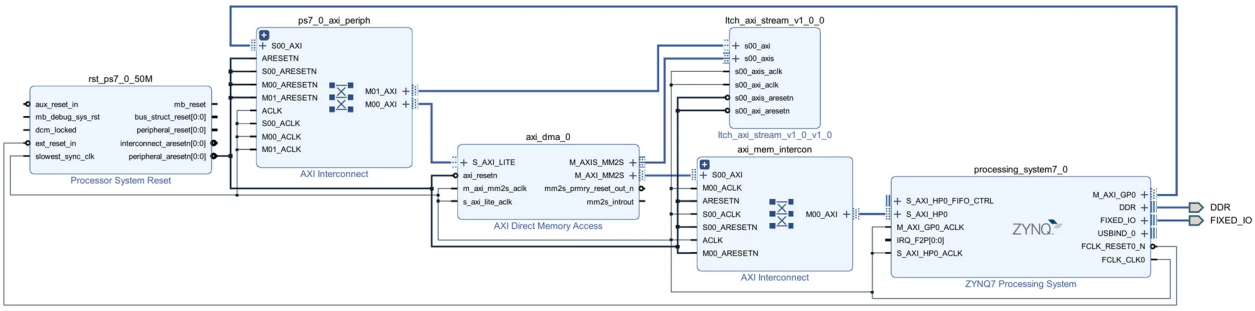

Figure 13: Vivado Block Design.

The `processing_system7_0` block (bottom right) represents the Zynq PS with its `M_AXI_GP0` (general purpose master) interface for control and `S_AXI_HP0` (high-performance slave) for DMA accesses. The `ps7_0_axi_periph` block is an AXI Interconnect that routes transactions from PS to peripherals: port `M00_AXI` to the DMA and port `M01_AXI` to our IP.

The `axi_dma_0` block is configured in MM2S (Memory-Mapped to Stream) mode only, reading from memory and producing an AXI-Stream. Its `M_AXIS_MM2S` output is connected to our

IP's `s00_axis` input. The `Itch_axi_stream_v1_0_0` block is our custom IP with two interfaces: `s00_axis` to receive the data stream and `s00_axi` to expose result registers.

The `axi_mem_intercon` block routes DMA memory requests to the PS HP0 port for high-performance DDR accesses. Finally, `rst_ps7_0_50M` manages synchronized reset signals for all components.

3.2.3 Parser IP Structure

Figure 14 details the internal structure of the `ITCH_axi_stream` IP.

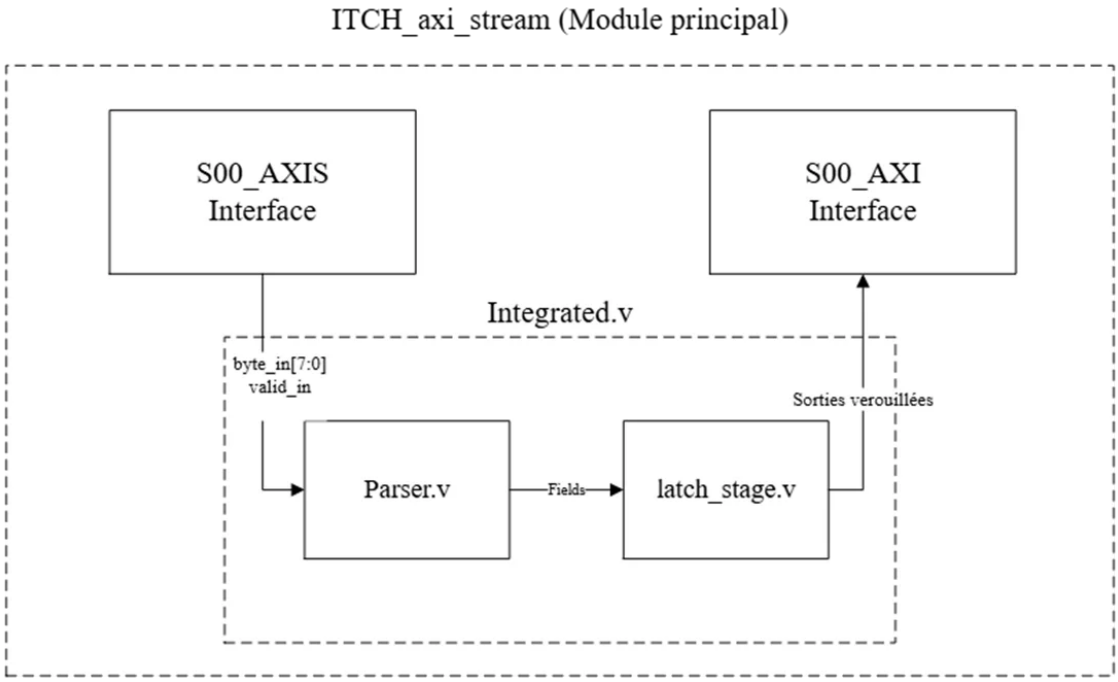

Figure 14: ITCH_axi_stream IP internal structure.

The main module (outer dashed rectangle) contains two top-level interfaces. The `S00_AXIS` interface (top left) receives the AXI-Stream from the DMA with signals `TDATA`, `TVALID`, `TREADY`, and `TLAST`. The `S00_AXI` interface (top right) exposes result registers via AXI-Lite with read (AR, R) and write (AW, W, B) channels.

The `Integrated.v` module (inner dashed rectangle) encapsulates the parsing logic. It receives `byte_in[7:0]` and `valid_in` from the AXI-Stream interface. The `Parser.v` submodule contains the 9 parallel decoders and multiplexing logic. Its `Fields` outputs (parsed fields) are sent to the `latch_stage.v` submodule that maintains stable values in registers until the next valid message. Latched outputs are then exposed via the AXI-Lite interface for CPU reading.

3.3 Integration

3.3.1 AXI Interface Overview

Figure 15 presents all AXI interfaces used in the system as a summary table.

Interface	Type	Width	Function
M_AXI_GP0	AXI4 Master	32 bits	CPU control of DMA and ITCH IP
S_AXI_HP0	AXI4 Slave	64 bits	High-perf memory access for DMA
S_AXI_LITE (DMA)	AXI4-Lite	32 bits	DMA configuration registers
M_AXI_MM2S	AXI4 Master	32 bits	DMA memory read channel
M_AXIS_MM2S	AXI4-Stream	8 bits	DMA stream output to parser
S00_AXIS	AXI4-Stream	8 bits	Parser byte input
S00_AXI	AXI4-Lite	32 bits	Result registers (read-only)

Figure 15: AXI interface summary.

The `M_AXI_GP0` interface is AXI4 Master type with 32-bit width, used by the CPU to control the DMA and access ITCH IP registers. The `S_AXI_HP0` interface is AXI4 Slave type with 64-bit width, providing high-performance memory access for DMA transfers to/from DDR. The DMA's `S_AXI_LITE` interface is AXI4-Lite type with 32 bits for DMA controller configuration registers. The `M_AXI_MM2S` interface is the DMA's memory read channel as AXI4 Master 32 bits. The `M_AXIS_MM2S` interface is the DMA's stream output as AXI4-Stream 8 bits, carrying bytes to the parser. The `S00_AXIS` interface is the parser input as AXI4-Stream 8 bits receiving bytes from the DMA. Finally, the `S00_AXI` interface exposes parser result registers as AXI4-Lite 32 bits read-only.

3.3.2 AXI-Lite Register Map

Figure 16 details the register map exposed by the parser via the AXI-Lite interface.

Table: AXI-Lite Register Map

Offset	Name	Bits	Description
0x00	Reserved	[31:0]	Reserved for future use
0x04	Status	[31:0]	Status register
0x08	LATCHED_VALID	[0]	Valid message flag (1 = latched)
0x0C	LATCHED_TYPE	[3:0]	Message type code (0-8)
0x10	ORDER_REF_LO	[31:0]	Order reference number [31:0]
0x14	ORDER_REF_HI	[31:0]	Order reference number [63:32]
0x18	SIDE	[0]	Buy/Sell indicator (0=Buy, 1=Sell)
0x1C	SHARES	[31:0]	Number of shares
0x20	PRICE	[31:0]	Price (4 implicit decimals)
0x24	NEW_ORDER_LO	[31:0]	New order ref [31:0] (Replace only)
0x28	NEW_ORDER_HI	[31:0]	New order ref [63:32]
0x2C	TIMESTAMP_LO	[31:0]	Timestamp [31:0]
0x30	TIMESTAMP_HI	[15:0]	Timestamp [47:32]
0x34	MISC_DATA_LO	[31:0]	Misc data [31:0] (symbol/match_id)
0x38	MISC_DATA_HI	[31:0]	Misc data [63:32]

Figure 16: AXI-Lite register map.

The register at offset 0x00 is reserved for future use. The status register at 0x04 contains general

state information. The `LATCHED_VALID` register at 0x08 contains a single bit [0] indicating whether a valid message has been parsed and latched (1 = message latched). The `LATCHED_TYPE` register at 0x0C contains the message type code on bits [3:0], value from 0 to 8 corresponding to the 9 supported types.

The `ORDER_REF_LO` (0x10) and `ORDER_REF_HI` (0x14) registers contain the 64-bit order reference, split into low [31:0] and high [63:32] parts. The `SIDE` register at 0x18 contains the buy/sell indicator on bit [0] where 0 = Buy ('B') and 1 = Sell ('S'). The `SHARES` (0x1C) and `PRICE` (0x20) registers contain the number of shares and price with 4 implicit decimals respectively.

The `NEW_ORDER_LO/HI` (0x24, 0x28) registers are used only for Replace Order messages. The `TIMESTAMP_LO/HI` (0x2C, 0x30) registers contain the 48-bit timestamp. The `MISC_DATA_LO/HI` (0x34, 0x38) registers contain additional data varying by message type, typically the symbol or match_id.

3.4 Results

3.4.1 Resource Utilization

Table 2 presents FPGA resource utilization after synthesis and implementation with Vivado 2022.1.

Table 2: FPGA resource utilization.

Resource	Used	Available	Utilization
LUTs	2,578	53,200	4.85%
Flip-Flops	3,705	106,400	3.48%
BRAM	0	140	0%
DSP	0	220	0%

Resource utilization is remarkably low: less than 5% of LUTs and less than 3.5% of Flip-Flops. No BRAM blocks are used because lookup tables are implemented in combinational logic, and no DSPs are needed because there are no complex arithmetic calculations. This low footprint leaves considerable margin for future extensions such as additional decoders, a hardware order book, or a decision engine.

3.4.2 Latency Analysis

Figure 17 presents detailed latency analysis for each message type.

Message Type	Length (bytes)	Parse Cycles	Latency @ 100MHz
Delete Order ('D')	9	10	100 ns
Broken Trade ('B')	19	20	200 ns
Cancel Order ('X')	23	24	240 ns
Replace Order ('U')	27	28	280 ns
Executed Order ('E')	30	31	310 ns
Add Order ('A')	36	37	370 ns
Exec with Price ('C')	36	37	370 ns
Trade ('P')	40	41	410 ns
Add Order MPID ('F')	40	41	410 ns

Figure 17: Latency analysis by message type.

Total parsing latency follows the formula: Latency $= (\text{Length} + 1) \times T_{clk}$ where the "+1" corresponds to the latch stage cycle and $T_{clk} = 10$ ns at 100 MHz.

The Delete Order ('D') message is fastest with 9 bytes requiring 10 cycles or 100 ns. The Broken Trade ('B') message with 19 bytes requires 20 cycles or 200 ns. Medium-sized messages like Cancel Order ('X', 23 bytes), Replace Order ('U', 27 bytes), and Executed Order ('E', 30 bytes) have latencies of 240 ns, 280 ns, and 310 ns respectively.

The longest messages are Add Order ('A') and Exec with Price ('C') with 36 bytes each requiring 37 cycles or 370 ns, and Trade ('P') and Add Order MPID ('F') with 40 bytes each requiring 41 cycles or 410 ns.

These latencies represent only parsing time within the FPGA. Total system latency would also include DMA transfer time and AXI-Lite register access time.

4 Discussion

4.1 Design Choices Justification

4.1.1 Platform Choice

The choice of the PYNQ-Z2 platform is justified by several factors. The Zynq-7000 SoC architecture integrates the ARM processor and FPGA fabric on the same chip, enabling efficient communication via standardized AXI interfaces without crossing slow external buses. The PYNQ ecosystem with its Python/Jupyter interface greatly facilitates development, testing, and demonstration, reducing debug time compared to a purely embedded approach. Zynq-7020 resources are sufficient for our application (less than 5% used), leaving significant margin for extensions. Finally, the PYNQ-Z2 board is affordable and widely available in academic contexts.

4.1.2 Speculative Architecture Choice

The speculative architecture with nine parallel decoders was chosen to eliminate message type determination latency. In a traditional sequential architecture, one would first need to identify the type (1 cycle), then route to the correct decoder (1 cycle), before starting parsing. The speculative approach eliminates these 2 cycles. Moreover, control simplicity is ensured because all decoders operate autonomously, without a complex central controller. The suppression mechanism is local to each decoder. Finally, extensibility is facilitated because adding a new message type simply requires adding an additional decoder without modifying others.

The resource tradeoff is acceptable: 9 decoders consume approximately 9 times more logic than a single decoder, but utilization remains below 5% of available resources.

4.1.3 Interface Choices

The AXI-Stream interface for data input is ideal for continuous streams. It requires no addressing, just a byte stream with simple control signals (valid, ready, last), and interfaces naturally with the DMA. The AXI-Lite interface for results is a lightweight protocol perfect for register access, supporting reads/writes to specific addresses without the complexity of full AXI4 burst mode.

4.2 Lessons Learned

This project successfully validated several aspects. The technical feasibility of implementing a high-performance ITCH parser on FPGA is demonstrated with modest resources. The speculative architecture works as expected and achieves the targeted minimum latency of one cycle after each message ends. PYNQ integration proved very effective for rapid development and validation. The three-phase test methodology allowed detecting problems at each abstraction level before moving to the next.

Regarding technical lessons, the suppression mechanism proved critical for system robustness. Without it, non-matching decoders would produce erroneous outputs or desynchronize from the message stream. AXI interface integration required particular attention to protocol details, especially correct ready/valid signal management and timing compliance. Python prototyping with Cocotb considerably accelerated development, with the ability to write tests in Python rather than Verilog enabling rapid iteration on decoder logic. Finally, the Python golden model proved indispensable for automated validation and debugging.

Difficulties encountered include initial DMA configuration with correct parameters (data width, burst mode) which required several iterations. The ITCH protocol uses big-endian format while

the ARM processor is little-endian, requiring conversion either in the parser or in Python code. Although the design is relatively simple, several iterations were needed to achieve timing closure at 100 MHz.

4.3 Limitations and Future Improvements

4.3.1 Current Limitations

Current limitations include the number of message types limited to 9 of the many ITCH types defined in the 5.0 protocol. There is no error handling for malformed messages or transmission errors. Register-only output requires CPU intervention for each message, limiting overall throughput. Finally, the absence of a network interface requires loading data into DDR before processing.

4.3.2 Future Improvements

In the short term, the following improvements are envisioned: adding an AXI-Stream output path for pipelined processing without CPU intervention, invalid message detection with error signaling, and performance counters.

In the medium term, support for additional ITCH message types, an Ethernet MAC interface for direct packet processing, and upstream UDP/IP decoding would be beneficial.

In the long term, a real-time hardware order book, a simple decision engine for automatic order generation, and porting to a high-performance platform (Alveo, Stratix) represent possible evolutions.

5 Conclusion

This project demonstrated the feasibility and effectiveness of a hardware ITCH parser implemented on FPGA for high-frequency trading applications. The speculative architecture with nine parallel decoders achieves minimum latency of one cycle after each message ends, meeting the strict requirements of the HFT domain.

Key achievements include complete support for the 9 major ITCH message types, speculative parallel decoding eliminating type determination latency, clean integration with the PYNQ ecosystem, a complete simulation and test infrastructure in three phases with a Python golden model, and efficient FPGA resource utilization (less than 5% of LUTs and FFs).

Achieved latencies range from 100 ns for the shortest message (Delete Order) to 410 ns for the longest (Trade, Add Order MPID) at 100 MHz. Modest resource utilization (2,578 LUTs, 3,705 FFs, 0 BRAM, 0 DSP) leaves significant margin for identified future extensions.

This implementation validates the core concepts presented in the reference article [**?**] and demonstrates their practical applicability on an accessible academic platform. The complete source code and documentation are available on GitHub [**?**].

6 References

[1] R. Zhang, "Speculative, Macro-Driven FPGA Architecture for Ultra-Low-Latency ITCH Parsing in High-Frequency Trading Systems," *TechRxiv*, 2023. [Online]. Available: https://www.techrxiv.org/users/924957/articles/1296717

[2] J.-C. J. Raymond, "FPGA Speculative ITCH Parser," *GitHub repository*, 2025. [Online]. Available: https://github.com/JJrRay/FPGA-speculative-itch-parser

[3] NASDAQ, "ITCH 5.0 Specification," *NASDAQ Trader*, 2020. [Online]. Available: https://www.nasdaqtrader.com/content/technicalsupport/specifications/dataproducts/NQTVITCHSpecification.pdf

[4] Xilinx, "PYNQ: Python Productivity for Zynq," *PYNQ Documentation*, 2023. [Online]. Available: https://pynq.readthedocs.io/

[5] ARM, "AMBA AXI and ACE Protocol Specification," *ARM Developer*, 2021. [Online]. Available: https://developer.arm.com/documentation/ihi0022/latest