Low-Latency FPGA Market Data Parser and Hardware Order Book

CSEE 4840 Design Document

Shawn Kathuria (shk2199), Shiyao Lam (sml2286), Sarah Hagan (sah2267), Siddharth Raykar (sr4102)

Spring 2026

1 Introduction

High-frequency trading firms use FPGAs to process market data and execute trades at nanosecond-scale latencies. The critical performance advantage comes from eliminating the CPU entirely from the data path: raw Ethernet frames carrying exchange data are received, parsed, and acted upon in FPGA fabric, bypassing the operating system, kernel network stack, and all associated software overhead.

This project implements the core of such a system on the DE1-SoC using a hardware market data processing pipeline that receives simulated NASDAQ ITCH 5.0 protocol messages from an external device, parses them entirely in SystemVerilog, and maintains a sorted hardware order book in on-chip memory. The system will support Add Order, Order Executed, Order Cancel, Order Delete, and Order Replace message types. A VGA display will show the live state of the order book: best bid/ask, depth at each price level, and real-time statistics including messages processed and parse latency. Simultaneously, we will implement the same parsing and book-building logic in C on the HPS ARM core and rigorously benchmark the two paths to quantify the latency advantage of the hardware implementation.

A prior 4840 project (HFT Book Builder, Spring 2024) attempted a similar system but routed all data through the ARM processor via TCP sockets and the Linux kernel before forwarding it to the FPGA over the Avalon bus. This defeated the purpose of hardware acceleration. Our project eliminates the ARM from the critical data path entirely: a market data simulator transmits ITCH messages over a direct serial link to the FPGA fabric via GPIO, where they are received, framed, deserialized, parsed, and used to update the order book—all without CPU involvement.

2 System Overview

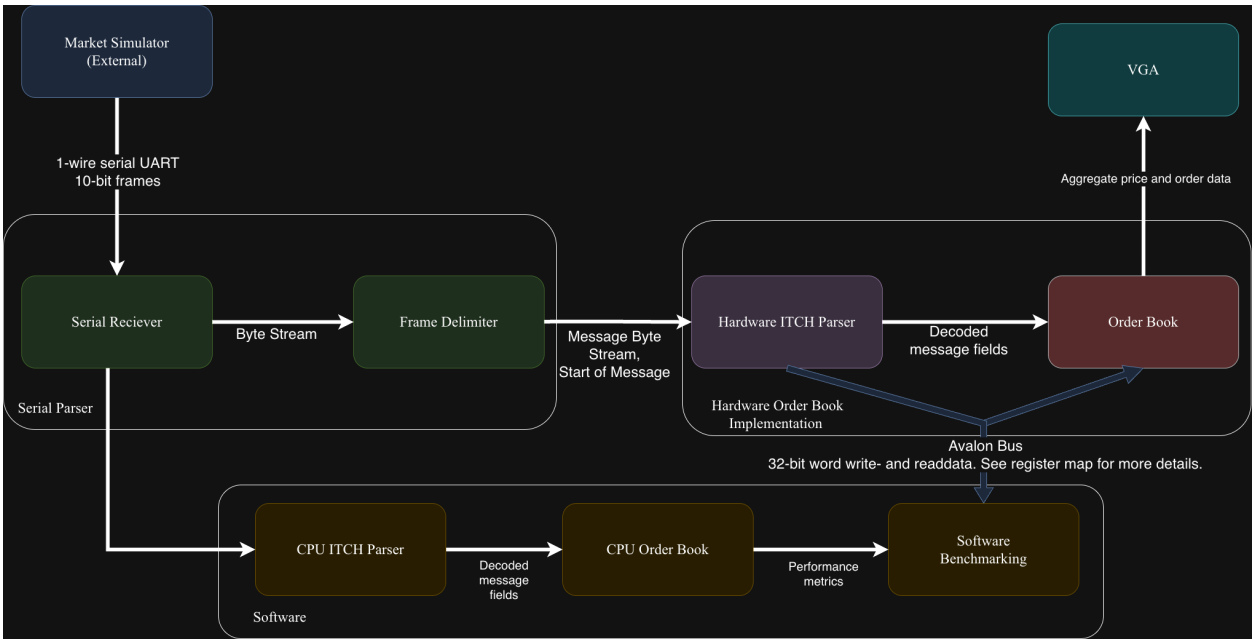

Figure 1: System block diagram

The market simulator (**3.3.1**) sends historical NASDAQ itch data as a serial datastream into a GPIO of the DE1-SOC. This pin is oversampled by the serial receiver (**3.1.1**) such that each bit lasts for CLK/BAUD_SEL cycles, where CLK is the on-board 50 MHz clock, and BAUD_SEL is the baud rate we set using the register described below. Serial communication occurs via the UART protocol, meaning that a byte of information is passed to the ITCH parser at a time, along with a valid and frame_error signal indicating whether that particular UART frame is trustworthy.

The ITCH parser (**3.1.2**) uses an FSM to extract relevant message fields (i.e. ITCH reference number, type of command). This is passed to the Hash Table and price-level array (**3.1.3**), and these together maintain the live order book. So, the hash table indexes orders by 64-bit ITCH reference number for fast lookup on Execute, Cancel, Delete, and Replace messages, while the price array aggregates resting quantity at penny granularity for each of the 4 tickers. A small top-of-book racker observes each update and maintains BEST_BID, BEST_ASK, BID_QTY, and ASK_QTY in dedicated registers, exposed to the HPS over the Avalon-MM bridge alongside message counts, parse latency, and error counters.

A 640x480 VGA controller displays the live order book state. The top half shows a depth-of-book visualization for the selected stock: bid levels on the left in green, ask levels on the right in red, with horizontal bars proportional to quantity at each price level. The bottom half shows statistics: total messages parsed, messages per second, hardware parse latency (in clock cycles), current best bid/ask and spread, and error counters.

In parallel, the serial input is forwarded to the ARM processor. A C program on the ARM core implements identical ITCH parsing and order book logic in software. Cycle-accurate timestamps (from the ARM performance counter) are recorded for each message to produce a latency distribution that is directly compared against the hardware path

A Linux kernel module exposes the FPGA's status registers (message counts, latency counters, best bid/ask, error flags) to userspace via `/dev/itch_accel`. A userspace program reads these registers to produce benchmark reports.

3 Algorithms

3.1 Hardware

3.1.1 Serial Receiver

The Serial Receiver is an FSM, with four states: IDLE, START, DATA, STOP. The output depends on both rx and the current state, making this a Mealy Machine. The baud rate of the serial receiver has its default setting at approximately 100 Kbps, however it can go up to approximately 3Mbps using UART and even more for other serial regimes.

IDLE

The UART protocol has a default value of 1 (when no signal is being sent over the line). During the IDLE state, at each positive clock edge the receiver compares the current input bit (henceforth called INPUT) with the previous one. If it is 0, and the previous input was 1, then there is a falling edge — meaning that we've possibly just encountered the start bit. We transition to **START**.

START

At this point, we do not know if we encountered a genuine start signal or a random fluctation. To determine this, we operate under the assumption that it was a genuine start bit, and go to the middle of the bit (in $417/2 = 217$ clock cycles). If INPUT is still 0, this strongly implies that we are actually in a start bit. We transition to **DATA**. If INPUT = 1 again, it was a false alarm. Return to **IDLE**.

DATA

If we're here, we know that we just received a start bit. Now, we must actually load the incoming input data into an output register to be passed downstream later. The safest bet is to always sample in the midpoint of each bit cycle. Since the length of each bit cycle is constant (dictated by the CLK and baudrate) we simply wait for 434 cycles before sampling and loading a particular data bit into the shift register. Once we have done this for 8 bits, we move to **STOP**.

STOP

We now have to check if the stop bit (which shoud come immediately next) is valid or

not. We check the midpoint of the next bit pulse, and if it is 1 (as it should be), we assert that the frame was valid (valid = 1), and that there was no frame error. If it is 0, then we assert a frame error. In either case, we return to **IDLE**.

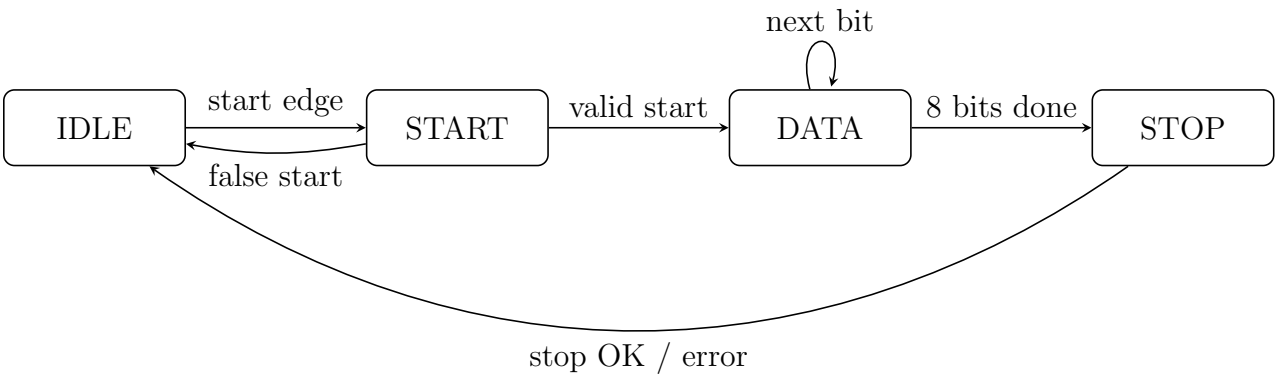

Figure 2: UART receiver FSM.

3.1.2 ITCH Parser

The Parser (figure 3) extracts the following fields (as applicable per message type) from the ITCH messages, using a finite state machine that parses each message byte-wise as received from the serial parser, and skips other fields as applicable.

- Stock - Security locate code (2 bytes)
- Order Ref - order reference number (8 bytes)
- Price (4 bytes)
- Shares - number of shares (4 bytes)
- Side - 0='B' (buy), 1='S' (sell) (1 byte) Replacement reference number (8 bytes)

Our implementation the following message types which are detailed in the Nasdaq TotalView-ITCH 5.0 Protocol

- 'A' — Add Order (No MPID Attribution)
- 'D' — Order Delete
- 'U' — Order Replace
- 'E' — Order Executed (partial fill; reduce shares)
- 'C' — Order Executed with Price
- 'X' — Order Cancel

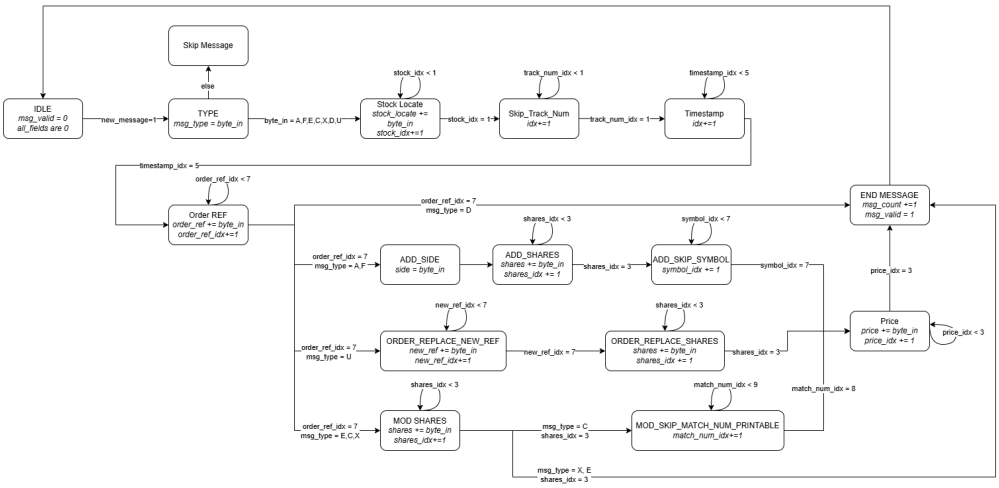

Figure 3: ITCH Protocol Parser

3.1.3 Hash Table and Order Book

The order book is maintained by two independent on-chip memory structures: a hash table keyed on the 64-bit ITCH order reference number, and a price-indexed aggregate array at penny granularity. The hash table answers the question *"where does order X live?"* for Execute, Cancel, Delete, and Replace messages, which identify the target order by reference number rather than by price. The price array answers the question *"what is the aggregate state at price P?"* and is the structure from which the VGA controller and the top-of-book registers (`BEST_BID`, `BEST_ASK`, `BID_QTY`, `ASK_QTY`) are read.

Both structures are mapped to M10K blocks. Sizing and block-count estimates are given in Resources.

Order Hash Table. The hash table is a 16,384-slot open-addressed table with linear probing. Each slot is 128 bits wide, packed as shown in Table 1. Linear probing is chosen over quadratic or double hashing for three reasons: the index update is a single incrementer with wrap, sequential probes are friendly to the M10K row-buffer behavior, and the `MAX_PROBE` diagnostic register gives direct runtime evidence that the load-factor assumption holds.

Field	Bits	Range	Notes
`valid`	1	–	Slot occupied
`order_ref`	64	ITCH 5.0	Key; stored for collision verification
`price`	24	\$0–\$1677.72	Fixed-point, four implied decimals
`qty`	24	0–$2^{24}-1$	Shares remaining on this order
`side`	1	B/S	
`symbol_id`	2	0–3	Index into `STOCK_FILTER[0..3]`
`reserved`	12	–	Padding to 128 b for M10K alignment
Total	128		

Table 1: Hash-table slot layout. The full 64-bit `order_ref` is stored so that collisions are resolved by direct key comparison, not by hash equality alone.

The hash function folds the order reference number down to the 14-bit table index via three-way XOR:

$$h(ref) = ref_{[13:0]} \oplus ref_{[27:14]} \oplus ref_{[41:28]}.$$

NASDAQ assigns order reference numbers sequentially across the session, so the low-order bits are already well distributed across the keyspace; the XOR-fold exists mainly to decorrelate bursts of consecutive references that would otherwise cluster into adjacent slots and inflate probe chains. The fold is implemented in combinational logic in a single cycle.

Load Factor and Probe Budget. For open addressing with linear probing, the expected number of slot reads is approximately $12(1 + 1/(1 - \alpha))$ for a successful lookup and $12(1 + 1/(1 - \alpha)^2)$ for an unsuccessful lookup (Knuth, TAOCP Vol. 3). At a target load factor of $\alpha \leq 0.5$ this gives a mean of roughly 1.5 probes for a hit and 2.5 for a miss. M10K blocks impose a two-cycle read latency (one address register, one output register), so a full probe-and-compare completes in approximately 2–3 clock cycles, yielding a mean end-to-end hash-table lookup of 3–6 cycles and a worst-case lookup bounded by `MAX_PROBE` times two cycles. These assumptions hold only so long as the table does not exceed half-full; the `HT_LOAD` register is exposed over the Avalon bridge precisely so that the software benchmarker can audit this at the end of each run.

Insertion, Deletion, and Replace. Add Order hashes the incoming reference, probes forward from $h(ref)$ until it finds a slot whose `valid` bit is 0, and writes the full record. Deletion (Execute that zeroes the order, Cancel, Delete) is handled by *backward-shift deletion* rather than by tombstones: the slot is cleared and subsequent slots in the probe chain are examined and shifted back into position if their natural hash index still sits at or before the freed slot. This keeps the table tombstone-free, which is important because tombstones would otherwise inflate `MAX_PROBE` monotonically over the course of a replay and break the load-factor analysis above. The shift loop is bounded by `MAX_PROBE` and runs in the same FSM state as the deletion itself; a new message cannot begin processing until the shift completes.

Order Replace is a semantically atomic *delete-then-insert* on the old and new reference numbers respectively, but because ITCH Replace carries both reference numbers and the new price/quantity in a single message, it is implemented as a fused operation: one hash-table read-modify-write cycle for the deletion and one for the new insertion, with the corresponding price-array decrement and increment issued in parallel on the dual-port price array.

Price-Level Array. Aggregate state is held in a flat array of 8,192 entries shared across all four symbols, with each entry holding the aggregate quantity and active-order count at one penny of price for one symbol. Entry layout is given in Table 2. The array is indexed by

$$index = (symbol\_id \ll 11) \mid price\_offset_{[10:0]},$$

giving each symbol a 2,048-entry window.

Field	Bits	Notes
`agg_qty`	32	Total shares at this penny level
`order_count`	16	Number of live orders at this level
`valid`	1	Level has at least one live order
`reserved`	15	Padding to 64 b
Total	64	

Table 2: Price-array entry layout. One entry per $(symbol\_id, penny\_offset)$ pair.

Penny Granularity and the Per-Symbol Window. The price array aggregates at penny granularity rather than at the native ITCH \$0.0001 tick. An incoming price is mapped by

$$price\_offset = \left\lfloor \frac{price - base[symbol\_id]}{100} \right\rfloor,$$

dropping the two low decimal digits of the four-decimal fixed-point representation. With 2,048 penny entries per symbol the window spans \$20.48; the base is latched from the median of the first 256 accepted messages for the symbol so that the window sits at approximately ±\$10.24 around the opening level. This comfortably covers a full-day price range for the intended liquid NASDAQ symbols (typical daily ranges for the four candidate symbols are under \$10).

Sub-penny ITCH prices, which appear on midpoint and hidden-liquidity orders for some message types, are rounded down to the nearest penny bucket for aggregate purposes; the individual order record in the hash table retains the full 24-bit price so that Execute, Cancel, and Replace messages decrement the correct quantity from the correct penny bucket regardless of sub-penny rounding. The software reference implementation on the HPS (Section **??**) performs the identical floor operation, so that the `BEST_BID` / `BEST_ASK` register comparison against the software book is exact rather than approximate.

Prices that fall outside the ±\$10.24 window trigger the `parse_error` sticky bit in `STATUS` and increment `ERR_COUNT`; the affected message is dropped from the hardware path but processed by the software reference, which gives the benchmarker a direct audit signal that

the replay window has drifted beyond the MVP's price range. Supporting unbounded price drift would require either a sparse price structure (a second hash table keyed on price) or runtime rebasing of the window, and is deferred as a stretch goal.

Aggregate Updates. Add Order at (*symbol_id*, *penny_offset*) increments `agg_qty` by the order quantity and `order_count` by one, setting `valid`. Execute (partial or full), Cancel, and Delete decrement `agg_qty` by the executed or cancelled share count and, for full-order removals, decrement `order_count` by one and clear `valid` if `order_count` reaches zero. Because the price array is dual-port M10K, the hash-table FSM can issue its decrement on one port while the top-of-book tracker (below) reads on the other.

Top-of-Book Tracking. `BEST_BID` and `BEST_ASK` are maintained incrementally. Each price-array update is observed by a small tracker FSM which compares the updated penny bucket against the current best on that side:

- On a bid-side update at a price strictly greater than the current `BEST_BID`, the best bid is replaced and `BID_QTY` is updated to the new bucket's `agg_qty`.
- On an update at the current `BEST_BID` that leaves `valid`= 1, only `BID_QTY` is refreshed.
- On an update at the current `BEST_BID` that clears `valid`, the tracker scans downward from the current best within that symbol's window until it finds the next valid bucket, which becomes the new best bid.

The ask side is symmetric. The downward scan is the only non-constant-time operation in the book update path; it is bounded by the per-symbol window (2,048 entries) and in practice terminates within the first few iterations on any active book, since liquid names maintain continuous price levels near the inside. The scan is permitted to overlap with subsequent message processing by pipelining on the dual-port price array, so it does not block hash-table insertions unless the scan itself takes longer than the inter-message interval.

Diagnostics. The `HT_LOAD` and `MAX_PROBE` registers in Table 3 expose the current occupancy and worst-case probe chain observed since the last counter reset. The `BID_DEPTH` and `ASK_DEPTH` registers expose the count of valid penny buckets on each side for the currently selected `ACTIVE_STOCK`, and are computed incrementally as `valid` bits transition. Together these allow the software benchmarker to verify not only that the hardware path processes messages faster than the software path, but that it processes them *correctly* by comparing top-of-book state at known points in the replay against the software reference.

3.2 FPGA–HPS Bridge: `orderbook_core`

Again, market data bytes never traverse the ARM. The HPS participates only through a lightweight Avalon-MM slave, instantiated in Qsys as `orderbook_core_0` and exposed to Linux on the FPGA-to-HPS *lightweight* AXI bridge. All offsets below are **byte offsets** from that mapped virtual (or physical) base; each register is one 32-bit word.

The generated device tree names the peripheral `orderbook_core_0` with `compatible = "csee4840,orderbook_core-1.0"` and a `reg` window of `0x400` bytes on the lightweight bridge (physical base `0xFF200000` in the current `soc_system.dts`). Software will treat the register file as the single official interface for control, diagnostics, VGA text programming, and hardware-side benchmarking (message counts, latency counters, top-of-book snapshots). The planned Linux driver will expose these registers to userspace through `ioctl`; the software reference path for latency comparison uses ARM-local timers and does not rely on these CSRs for its own timing.

Memory-Mapped Register Map

Table 3 lists the target register map for the full system. Price fields exposed through this register map report penny-granularity offsets from the per-symbol base price: the hardware stores raw ITCH prices (four implied decimal places, e.g. 1500000 = \$150.0000) internally, but `BEST_BID` and `BEST_ASK` report the penny-bucket index $\lfloor price/100 \rfloor$ relative to the symbol's latched base, consistent with the price-level array layout. To recover a dollar price, software computes $(base + offset \times 100) \div 10000$.

We expose BEST_BID, BEST_ASK, BID_QTY, ASK_QTY, BID_DEPTH, and ASK_DEPTH over the lightweight bridge so the HPS can check the hardware order book against the software reference after replaying the same ITCH stream. Raw counters such as MSG_COUNT show that messages were accepted, but they do not prove that prices and aggregates were updated correctly. Reading the FPGA's top-of-book and level counts at known points in a trace (or at the end of a run) gives a direct state comparison with the C implementation and catches bugs in parsing, hashing, or price-level updates that latency numbers alone would miss.

The CONTROL register is the slow control plane: infrequent, coarse-grained settings written from the ARM that do not participate in the per-message datapath. We aim to implement enabling or disabling processing (RUN), choosing which of the four filtered symbols is active for display or statistics (ACTIVE_STOCK), selecting serial baud rate for bring-up (BAUD_SEL), and pulsing counter resets so each benchmark run starts from a clean slate. Under normal operation these writes happen at startup, between runs, or on rare UI changes, not once per ITCH message, so they are not on the latency-critical path from serial byte to book update.

Table 3: `orderbook_core` Avalon register map (byte offsets from bridge base).

Offset	Name	R/W	Description
0x00	CONTROL	R/W	[0] run enable; [3:2] active stock select (4 symbols); [4] reset statistics counters (pulse or level, as implemented); [7:5] baud rate select for serial bring-up.
0x04	STATUS	R	[2:0] parser FSM state (diagnostic); [3] FIFO full; [4] FIFO empty; [8] parse error sticky/latched.
0x08	MSG_COUNT	R	Total messages parsed (or accepted into the book, per implementation).
0x0C	LATENCY	R	Cycle count from first byte of message (or defined start-of-frame) to completion of book update for that message.
0x10	BEST_BID	R	Best bid penny-bucket index: $\lfloor price_{ITCH}/100 \rfloor - base[sym]$. Recover dollars via $(base + val \times 100) \div 10000$.
0x14	BEST_ASK	R	Best ask penny-bucket index (same encoding as BEST_BID).
0x18	BID_QTY	R	Aggregate shares at best bid.
0x1C	ASK_QTY	R	Aggregate shares at best ask.
0x20	BID_DEPTH	R	Count of active bid price levels.
0x24	ASK_DEPTH	R	Count of active ask price levels.
0x28	ERR_COUNT	R	Framing, parse, and FIFO overflow errors (combined or banked per implementation).
0x2C	MSG_RATE	R	Messages observed in the last one-second window.
0x30–0x5F	STOCK_FILTER[0..3]	R/W	Four 8-byte ASCII NASDAQ symbols (NUL-padded), two 32-bit words per symbol, for filtering displayed or processed traffic.
0x60	HT_LOAD	R	Number of occupied slots in the order-reference hash table.
0x64	MAX_PROBE	R	Maximum probe chain length observed (hash quality diagnostic).
0x68	CHAR_MEM_ADDR	R/W	Index into a small on-chip *text* buffer the VGA block reads when drawing captions (not the order book).
0x6C	CHAR_MEM_DATA	W	Write one character (ASCII or font table index) into the cell selected by CHAR_MEM_ADDR. Used only for labels/titles.

Software Header (Byte Offsets and Control Bit Masks)

Listing 1 gives the canonical C definitions shared by the kernel driver and userspace tools. They must stay in lockstep with Table 3.

Listing 1: `orderbook_regs.h` (shared kernel/userspace offsets).

```
/* orderbook_regs.h -- byte offsets from MMIO base (orderbook_core)
   */

#ifndef ORDERBOOK_REGS_H
#define ORDERBOOK_REGS_H

#include <stdint.h>

/* DT: reg span 0x400 @ lightweight bridge (e.g. phys 0xFF200000) */

#define OB_REG_CONTROL           0x00u
#define OB_REG_STATUS            0x04u
#define OB_REG_MSG_COUNT         0x08u
#define OB_REG_LATENCY           0x0Cu
#define OB_REG_BEST_BID          0x10u
#define OB_REG_BEST_ASK          0x14u
#define OB_REG_BID_QTY           0x18u
#define OB_REG_ASK_QTY           0x1Cu
#define OB_REG_BID_DEPTH         0x20u
#define OB_REG_ASK_DEPTH         0x24u
#define OB_REG_ERR_COUNT         0x28u
#define OB_REG_MSG_RATE          0x2Cu

#define OB_REG_STOCK_FILTER_BASE   0x30u
#define OB_REG_STOCK_FILTER_STRIDE 0x08u   /* 8 bytes = 2 x 32-bit
   words */
#define OB_REG_STOCK_FILTER_COUNT  4u

#define OB_REG_HT_LOAD           0x60u
#define OB_REG_MAX_PROBE         0x64u
#define OB_REG_CHAR_MEM_ADDR     0x68u
#define OB_REG_CHAR_MEM_DATA     0x6Cu

/* CONTROL (OB_REG_CONTROL) */
#define OB_CTRL_RUN                (1u << 0)
#define OB_CTRL_ACTIVE_STOCK_SHIFT 2u
#define OB_CTRL_ACTIVE_STOCK_MASK  (3u << OB_CTRL_ACTIVE_STOCK_SHIFT
   )
#define OB_CTRL_RESET_COUNTERS     (1u << 4)
#define OB_CTRL_BAUD_SEL_SHIFT     5u
#define OB_CTRL_BAUD_SEL_MASK      (7u << OB_CTRL_BAUD_SEL_SHIFT)
```

```
static inline uint32_t ob_ctrl_active_stock(uint32_t ctrl)
{
    return (ctrl & OB_CTRL_ACTIVE_STOCK_MASK) >>
       OB_CTRL_ACTIVE_STOCK_SHIFT;
}

static inline uint32_t ob_ctrl_set_active_stock(uint32_t ctrl,
   uint32_t idx)
{
    return (ctrl & ~OB_CTRL_ACTIVE_STOCK_MASK)
         | ((idx << OB_CTRL_ACTIVE_STOCK_SHIFT) &
            OB_CTRL_ACTIVE_STOCK_MASK);
}

#endif /* ORDERBOOK_REGS_H */
```

3.3 Software

3.3.1 Market Data Simulator (external host)

NASDAQ has free Nasdaq TotalView-ITCH data for a select sample of dates available at `itchdata.nasdaq.com`. Data from December 30th 2019 was processed using Python, to isolate the desired message types for 4 stocks (Figure 4). These are saved in a binary file with message length headers. To run a simulation, this binary file is processed and messages are directly passed to the serial port of an external host.

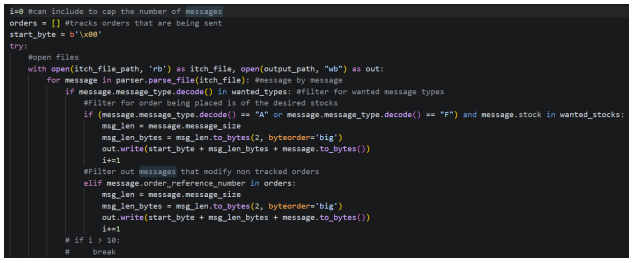

Figure 4: Historical Data Filtering

3.3.2 Software Reference Implementation (HPS ARM)

The software reference implementation on the HPS ARM core serves as an independent baseline for the FPGA design. It consumes the raw ITCH byte stream directly, performs message framing and parsing in C, maintains its own CPU-resident order book, and measures its own latency using ARM performance counters. A natural software structure is an order lookup table keyed by the 64-bit ITCH order reference number together with per-symbol price-level state for aggregate quantity and top-of-book tracking; add messages insert new orders, execute/cancel/delete messages look up and modify existing orders, and replace messages remove the old order and insert the new one. Separate from this reference path, a

benchmark and diagnostics tool accesses the FPGA through the FPGA–HPS bridge to read hardware-side registers such as message counts, latency counters, error flags, and top-of-book summaries. Keeping these two software roles separate ensures that the CPU reference remains a fair standalone baseline while still allowing the FPGA implementation to be monitored and validated.

4 Resource Budgets

The Serial Receiver runs at the FPGA's 50 MHz, with a variable baud rate which can be set using the BAUD_SEL register specified above. For our MVP, we will be using a UART system for the Serial Receiver, due to its relative simplicity and ease of implementation. UART is a 10-bit frame — a start bit, a byte of data, and a stop bit. With a clock frequency CLK, and a baud rate BAUD, each bit will last for CLK/BAUD cycles (let this be T). To ensure proper sampling, we must sample in the middle of T, as the edges are fuzzier. As a result, we must round T to the nearest so we can count up to T/2, however this will introduce some rounding error, R cycles/bit. Over a full frame, this error will accumulate to be 10R, and could potentially lead to sampling at the wrong time if the value of T is small enough.

This imposes a fundamental ceiling on the baudrate, as if it is too high, then R will be too large a fraction of T. This problem is especially acute when T is in the single digits, since other non-idealities like transmitter baud mismatch, start-edge quantization, and normal noise/jitter must also be taken into account. We estimate that the upper limit for a safe baud rate is approximately 3.5 Mbps.

The DE1-SoC's Cyclone V 5CSEMA5F31C6 provides 446 M10K blocks (4,884 Kb on-chip RAM (Cyclone V Handbook). The following budget targets M10K for the large structures (hash table, prie array, character memory) and leaves the small FIFOs and shift regisrers to be inferred into MLABs by Quartus, which is where the synthesis tool plrefers to place wide-and-shallow memories under approx. 640bits.

4.1 Serial Receiver

The Serial Receiver runs at the FPGA's 50 MHz system clock, with a variable baud rate set through the `BAUD_SEL` field of the `CONTROL` register. For our MVP, we will be using a UART system for the Serial Receiver, due to its relative simplicity and ease of implementation. UART is a 10 bit frame, a start bit a byte of data, and a stop bit. With a clock frequency of `CLK`, and a baud rate `BAUD`, each bit will last for `CLK/BAUD` cycles (let this be T). To ensure proper sampling, we must sample in the middle of T, as the edges are fuzzier. As a result, we must round T to the nearest integer so we can count up to $T/2$; this introduces a rounding error or R cycles per bit. OVer a full frame, this error accumulates to $10R$, and could potentially lead to sampling at the wrong time if the value of T is small enough.

This imposes a fundamental ceiling on the baud rate; if it is too high, R becomes too large a fraction of T. This problem is especially acute when T is in the single digits, since other non-idealities like transmitter baud mismatch, start-edge quantization, and normal noise/jitter must also be taken into account. We estimate that the upper limit for a safe baud rate is

approximately 3.5 Mbps. The Serial Receiver itself consumes negligible embedded memory: the FSM state, bit counter, and 8-bit shift register fit entirely in fabric registers, and a small 256-entry receieve FIFO ($\sim$2.5 Kb) is expected to be inferred into a single MLAB.

4.2 Hash table and Price-Level Array

The hash table and price-level array as descrobed in the previous section(s) account for the bulk of on-chip memory usage. Sizing follows from three constraints: a target load factor of $\alpha \leq 0.5$ for the hash table, a $\pm$\$10.24 per-symbol window for the price array (at penny granulrity), and a total M10K budget that must also accomodate VGA character memory and small FIFOs with headroom for synthesis-tool inefficiency.

Capacity Planning. A liquid NASDAQ symbol maintains on the order of a few thousand live orders on its lit book at any instant. For a 15-minute replay window over four filtered symbols, the combined resident order count is expected to peak below 8,000; a 16,384-slot hash table gives $\alpha \leq 0.5$ under this load, with the `HT_LOAD` register available for runtime verification. A longer replay (up to a full trading day) is treated as a stretch goal and would require growing the hash table to 32,768 slots, which fits within remaining M10K headroom but leaves less margin for synthesis overhead.

M10K Block Estimates. M10K blocks store 10,240 bits each and are configurable in widths from $\times$1 up to $\times$40 (Cyclone V device handbook). Wider ports require more blocks in parallel; deeper memories require more blocks in series. Quartus handles this packing automatically, but block-count estimates can be computed from the widest supported configuration.

Structure	Depth	Width (b)	Bits	Est. M10K
Hash table (MVP)	16,384	128	2,097K	$\sim$230
Hash table (stretch, 32K slots)	32,768	128	4,194K	$\sim$410
Price-level array	8,192	64	524K	$\sim$64
Character ROM (VGA)	256	128	33K	$\sim$4
Text buffer (VGA, 80$\times$60)	4,800	8	38K	$\sim$4
Parser $\rightarrow$ book FIFO	64	128	8K	$\sim$1
Stats / counter memories	–	–	$\sim$20K	$\sim$2
MVP total			$\sim$2,720K	$\sim$305

Table 4: On-chip memory budget. MVP total consumes approximately 61% of available M10K (305 of 446 blocks), leaving $\sim$141 blocks of headroom for synthesis overhead, banked error counters, and the MSG_RATE ring buffer. Moving to the stretch-goal 32K hash table pushes the total to $\sim$485 blocks and would require compensating reductions elsewhere.

The hash-table estimate assumes 128-bit-wide M10K configuration ($\times$40 packed into 4 blocks wide, repeated over the depth dimension). Quartus will in practice land somewhere in the range of 210–240 blocks depending on how aggressively it merges the valid bit and symbol-id fields into adjacent words. The price array, at 8,192 $\times$ 64 bits and mapped onto

$\times$32 M10K configuration, requires two blocks in parallel for width and 32 deep for a total of $\sim$64 blocks. Character ROM and text buffer are small enough that their M10K cost is dominated by the minimum allocation granularity of one block per inferred memory.

MLAB Usage. The UART receive FIFO, parser intermediate registers, and any shift registers or delay lines inside the VGA timing generator are expected to be inferred into MLABs rather than M10K. Each MLAB provides 640 bits of simple dual-port SRAM, and Quartus automatically selects MLAB over M10K for memories below approximately 32×20 bits. Total MLAB usage is expected to remain well under 100 of the 679 available blocks.

4.3 Logic (ALMs)

Logic utilization has not been estimated in detail at the design stage, but the major consumers are: the ITCH parser FSM (dominated by message-type dispatch and field-extraction logic, estimated at a few hundred ALMs), the hash-table insertion and backward-shift deletion FSM (a few hundred ALMs plus one 14-bit comparator and the XOR-fold hash function), the top-of-book tracker and the downward-scan FSM for best-price updates (a few hundred ALMs), the VGA timing and rendering logic, and the Avalon-MM slave register file. The 5CSEMA5 provides 85,000 ALMs; none of these consumers individually is expected to approach even 1% of that, and logic is not expected to be the binding resource. A full post-synthesis utilization report will be included in the final project report.

4.4 DSP Blocks

No DSP blocks are required. The only multiplication in the datapath is the penny-to-tick reconstruction for the `BEST_BID` and `BEST_ASK` registers ($\mathit{offset} \times 100$), which is implemented as a shift-and-add tree rather than a multiplier instantiation.

5 References

1. NASDAQ. NASDAQ TotalView-ITCH 5.0 Specification. 2020.

2. Litz, H. et al. High Frequency Trading Acceleration using FPGAs. IEEE Intl. Conf. on Field Programmable Logic and Applications, 2012.

3. Thenappan, C. et al. HFT Book Builder Implemented on DE1-SoC FPGA Board. CSEE 4840 Final Report, Columbia University, Spring 2024.

4. Chhabra, A. et al. Hardware Acceleration of Market Order Decoding. CSEE 4840 Final Report, Columbia University, Spring 2012.