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

Contents

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 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, 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 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 and display the latency advantage of the hardware implementation.

2 System Overview

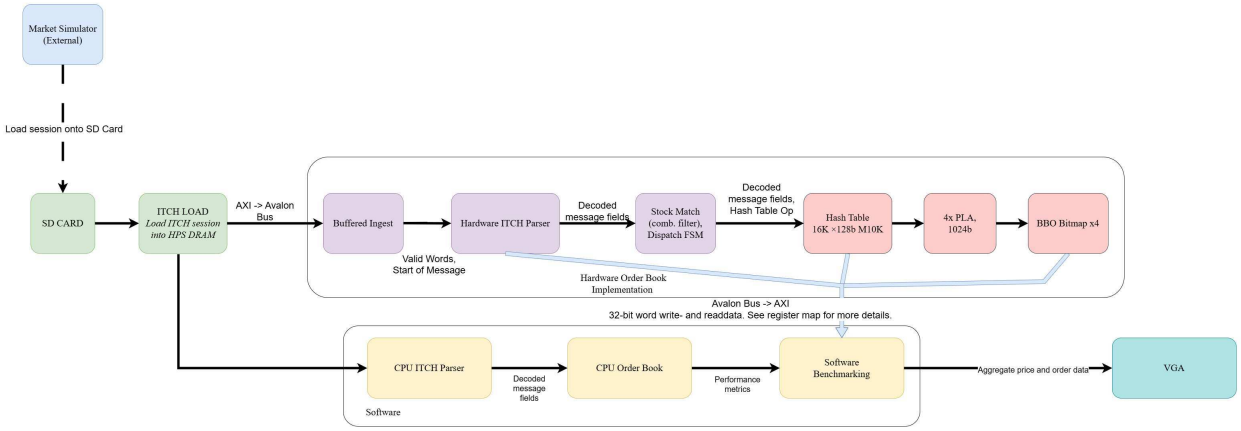


Figure 1: System block diagram

A binary file of emulated ITCH messages, each prefixed with a two-byte big-endian length header, is loaded onto the SD card of the DE1-SoC. `itch_load` (Section 3.2.3) maps this session file into HPS DRAM and programs all user-selectable parameters, including the symbol-filter registers and per-stock price baselines. The ITCH messages are over a lightweight AXI bridge into the FPGA's Avalon-MM slave in 8 KB chunks, alternating between the two ping-pong buffers. Once the HPS asserts the `start` bit in the control register, the buffers are consumed by `buffered_ingest` (Section 3.1.1), which uses the length fields to extract the body bytes of the messages which are fed to the ITCH parser.

The ITCH parser (Section 3.1.2) extracts the fixed-offset fields from each message (order reference number, stock locate, side, shares, price, etc.) and forwards these decoded fields to the hash table and price-level array (Section 3.1.3, 3.1.5), which together maintain the live order book. The hash table indexes all active orders by their 64-bit ITCH reference number, enabling constant-time lookup for Execute, Cancel, Delete, and Replace messages. The price-level array (PLA) updates aggregate resting quantity at one-cent resolution for each of the four tracked symbols, using the per-stock `PRICE_BASE` registers to convert raw ITCH prices into 10-bit bucket indices. Each PLA update also drives the corresponding bid/ask bitmap, which is consumed by the top-of-book (BBO) tracker. The BBO logic performs a two-stage priority encode over the bitmap to maintain `BEST_BID`, `BEST_ASK`, `BID_QTY`, and `ASK_QTY`.

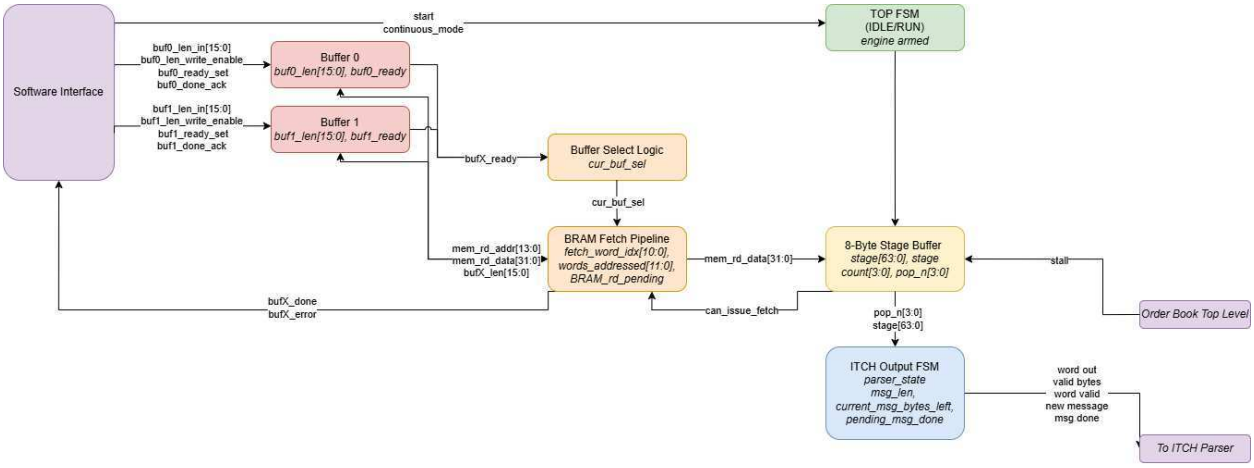


Figure 2: Buffered Ingest Block Diagram

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

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

Finally, a 640×480 VGA controller renders the live order-book state and various debugging metrics in real time.

3 Algorithms

3.1 Hardware

3.1.1 Ping Pong Buffer: `buffered_ingest.sv`

As shown in Figure 2, the `buffered_ingest` module uses a dual-buffer ping-pong scheme to sustain continuous ingestion of ITCH data from the HPS. Each buffer is prepared independently by software. After the software finishes loading data into a given buffer, it pulses `bufX_len_we`, causing the hardware to latch `bufX_len` from `bufX_len_in`. Software then asserts `bufX_ready_set`, marking the buffer ready for consumption by the ingest engine.

To enable latency benchmarking and to control when ingestion begins, the engine is armed when `start` is pulsed high. This latches `engine_armed` to 1, allowing the top-level FSM to leave its initial `IDLE` state once at least one buffer is ready and transition into `RUN`, activating the selected buffer to begin streaming and parsing its contents.

Once the engine transitions into the `RUN` state, it activates whichever buffer was selected, using round-robin arbitration logic if both buffers are ready. The active buffer's length is

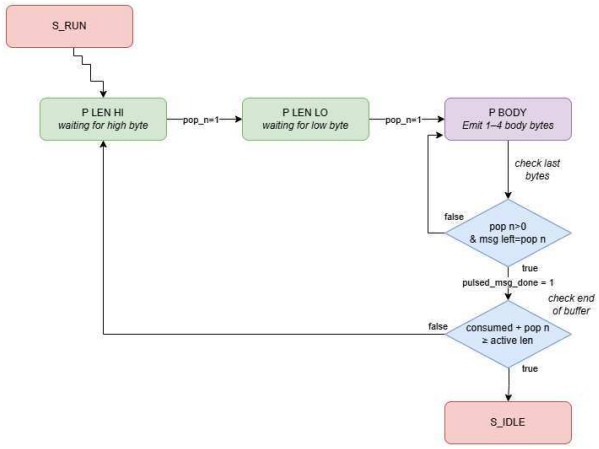


Figure 3: Buffered Ingest Stage-Buffer and Parser FSM

latched into the internal `active_len` register, and the BRAM fetch pipeline is reset to begin reading from the start of that buffer. As 32-bit words are fetched from memory, the hardware tracks the number of bytes consumed using `consumed_reg` and increments the BRAM address only when the stage buffer has sufficient space, as indicated by `can_issue_fetch`. This incoming BRAM data is merged into an 8-byte circular stage buffer, which allows 1–4 bytes to be popped per cycle and passed to the ITCH parser depending on message boundaries, stall conditions, and the number of bytes remaining in the current message.

Figure 3 illustrates this FSM. In `P_LEN_HI` and `P_LEN_LO`, the FSM pops exactly one byte each cycle to assemble the two-byte ITCH length field, checking for zero-length messages and ensuring the declared length does not exceed `active_len`. Once in `P_BODY`, the FSM emits 1–4 bytes per cycle depending on `pop_n`, decrements `msg_left`, which tracks remaining bytes in the message body based on the stated length, and asserts `new_message` on the first body emission. When `msg_left == pop_n`, the final bytes are popped and the FSM pulses `msg_done`. After each message completes, the FSM also checks the end-of-buffer condition `consumed + pop_n` $\geq$ `active_len` to ensure parsing does not overrun the programmed buffer boundary.

This parser operates directly on the stage buffer output, first extracting the two-byte ITCH message length and then streaming the message body to the downstream ITCH decoder. The hardware asserts `new_message` when the first body word is emitted and pulses `msg_done` one cycle after the final byte. Normal completion of a buffer occurs when the number of consumed bytes matches the programmed buffer length and the stage buffer is empty, at which point the engine transitions the buffer into the `DONE` state. If the parser encounters a zero-length message or a message body that would extend beyond the end of the buffer, the hardware instead transitions the buffer into the `ERROR` state and records the appropriate error code. In both cases, the engine returns to `IDLE` and waits for software to acknowledge completion via `bufX_done_ack`, after which the buffer is cleared and may be reused. This ping-pong mechanism allows one buffer to be processed while the other is being refilled.

3.1.2 ITCH Parser: `itch_parser.sv`

The ITCH Parser receives ITCH message bodies in 1–4 byte chunks from *buffered_ingest* and stores them sequentially in a 64-byte scratch buffer. A *msg_done* pulse from the upstream *buffered_ingest* indicates the full message is present in the scratch buffer and decoding begins. The parser extracts the desired fields from supported ITCH messages using fixed byte offsets in the scratch buffer, as defined by the NASDAQ ITCH 5.0 protocol. Unsupported types are safely ignored. Upon completion, the ITCH Parser asserts *msg_valid* for one cycle, indicating to the downstream hashtable that the decoded fields are ready for consumption. *msg_count* tracks the total number of successfully decoded messages.

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

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

The following fields are extracted by the parser:

Field	Offset	Decoded for	Meaning
type	0	all	ITCH message type
stock_locate	1..2	all	16-bit NASDAQ stock identifier
order_ref	11..18	A/F/E/X/D/U/C	64-bit ITCH order reference number
side	19	A/F ('B'/'S')	Buy/Sell Indicator
shares	20..23	A/F	32-bit share quantity at submission
shares	19..22	E/X/C	32-bit executed/cancelled share quantity
shares	27..30	U	32-bit new share quantity (replacement)
new_order_ref	19..26	U	64-bit replacement order reference
price	32..35	A/F, C	32-bit ITCH price in $0.0001 units (low 24 bits stored downstream)
price	31..34	U	32-bit new ITCH price for replacement

3.1.3 Hash Table: `hash_table.sv`

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; full 64-bit reference for collision check
`price`	24	0–$2^{24}-1$	Raw 24-bit integer from ITCH
`qty`	24	0–$2^{24}-1$	Shares remaining on this order
`side`	1	B/S	Buy = 0, Sell = 1
`symbol`	14	0–$2^{14}-1$	Decoded symbol ID (no padding)
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 de-correlate bursts of consecutive references that would otherwise cluster into adjacent slots and inflate probe chains. The hash is purely combinational and generates the initial probe index. On a collision, single increment linear probing with wrap-around is applied.

Load Factor and Probe Budget. For open addressing with linear probing, the expected number of slot reads is approximately $\frac{1}{2}(1 + 1/(1-\alpha))$ for a successful lookup and $\frac{1}{2}(1 + 1/(1-\alpha)^2)$ 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.

The M10K RAM blocks impose a single-cycle read latency, so each probe step traverses three FSM states (`READ_ISSUE` → `READ_WAIT` → `COMPARE`) and therefore takes three cycles. The first probe shortcuts directly from `S_IDLE` into `READ_WAIT`, so a one-probe hit completes in four cycles end-to-end (1 + `WAIT` + `COMPARE` + `DONE`). A typical lookup at $\alpha \leq 0.5$ resolves in 4–10 cycles, and the worst-case probe-phase latency is bounded by `MAX_CHAIN` × 3 cycles. These assumptions hold only so long as the table does not exceed half-full; the `HT_LOAD` and `MAX_PROBE` registers are exposed as MMIO registers so that the software benchmarker can audit this at the end of each run.

Insertion, Deletion, and Look Up. The hash table implements three primitive operations: Insertion, Deletion, and Look Up. *Insert* hashes the incoming reference and probes forward from $h(ref)$ until either the matching reference is found — in which case the slot is *updated in place* and `HT_LOAD` is left unchanged (used to service ITCH 'E' partial-fill rewrites that retain the original `order_ref`) — or an empty slot is encountered, in which case the full record is written and `HT_LOAD` is incremented. *Look Up* probes forward until it finds the matching reference, an empty slot, or exceeds the maximum probe chain length `MAX_CHAIN`. *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_CHAIN` and runs in the same FSM as the deletion itself; a new message cannot begin processing until the shift completes.

Finite-State Machine. Figure 4 shows the full FSM that sequences all hash-table operations. On reset, the FSM enters `S_INIT_CLEAR`, which walks all 16,384 slots and writes zero to each, replacing a synthesis-hostile for-loop reset. Normal operation begins in `S_IDLE`: when `op_valid` is asserted, the FSM issues a BRAM read (`S_READ_ISSUE` → `S_READ_WAIT`) and compares the slot contents in `S_COMPARE`. A match triggers either an in-place update (INSERT-on-match) or the backward-shift delete subroutine. A miss loops back to `S_READ_ISSUE` with the probe index incremented. All paths converge in `S_DONE`, which returns to `S_IDLE` on the next cycle.

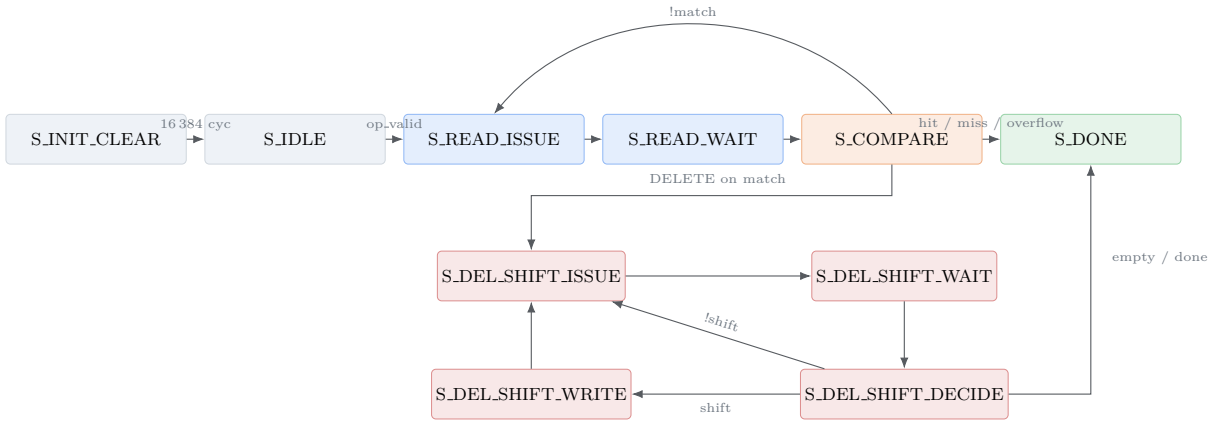


Figure 4: Hash-table FSM. The probe pipeline runs across the top row (`S_READ_ISSUE` → `S_READ_WAIT` → `S_COMPARE`, looping back on `!match`). On a DELETE hit, control transfers to the backward-shift subroutine. `S_DEL_SHIFT_DECIDE` branches three ways: `shift` routes through `S_DEL_SHIFT_WRITE` (clear the moved slot, advance scan); `!shift` returns directly to `S_DEL_SHIFT_ISSUE` (advance scan, no writeback); an empty slot or exceeded `MAX_CHAIN` exits to `S_DONE`.

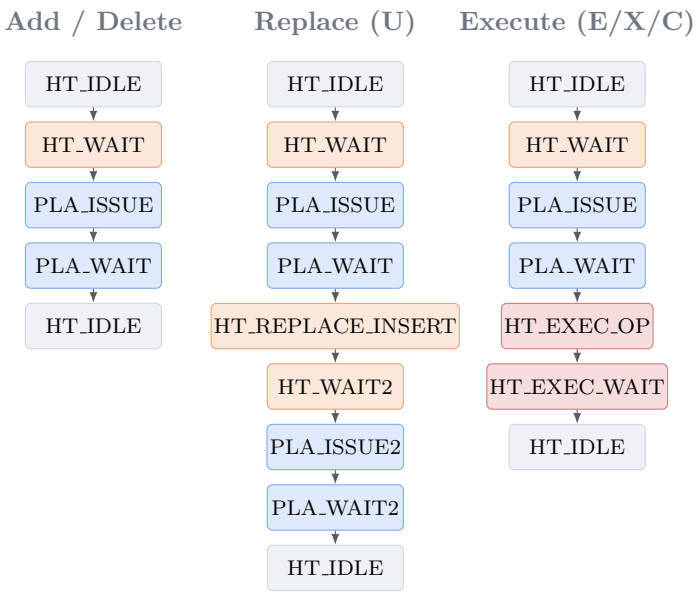


Figure 5: Dispatch FSM

3.1.4 Dispatch FSM: implemented in `orderbook_core.sv`

The Dispatch FSM coordinates all hash table and PLA updates by acting as the bridge between the ITCH parser and these structures. When a message is decoded, the FSM determines the required sequence of primitive hash-table operations (INSERT, DELETE, LOOKUP) and issues them in the correct order. Several ITCH message types are semantically composite and therefore expand into multi-stage update sequences.

Add / Delete (A/F/D). Add and Delete messages require only a single hash-table operation followed by a single PLA update: INSERT for A/F and DELETE for D.

Order Replace (U). An Order Replace message is logically an *delete-then-insert* on the old and new order references, respectively. Because ITCH Replace carries both references and the new price/quantity in a single message, the hardware performs a fused two-operation sequence: (1) DELETE the old order, issuing a PLA decrement at its original bucket, then (2) INSERT the new order, issuing a PLA increment at the new bucket.

Order Execute / Cancel (E/X/C). Execution and cancel messages reduce an existing order's quantity. The FSM first performs a LOOKUP to retrieve the order's current price, side, and quantity. The executed or canceled shares are subtracted; the FSM then either DELETEs the order (full fill) or INSERTs it back with the reduced quantity (partial fill). In both cases, the PLA is decremented at the original bucket.

Figure 5 shows the three execution paths implemented in hardware. All paths begin and end in `HT_IDLE`, but differ in the number and ordering of hash-table and PLA stages.

3.1.5 Price-Level: `pla.sv`

The Price-Level Array (PLA) is implemented as four independent dual-port BRAM banks — one per (stock, side) pair, i.e. 2 stocks × {bid, ask}. Each bank is 1024 × 24 b in M10K and stores only the aggregate share quantity at each penny bucket for that side of that stock. The bank is indexed by:

$$\text{bucket} = \texttt{((price - PRICE\_BASE) / 100)[9:0]}$$

The ITCH price is in \$0.0001, thus /100 gives 1-cent buckets. Quartus emits this as multiply-by-reciprocal. The HPS sets the price base as the first_price − \$5.12 so the live quote sits mid-window. Out-of-window updates are dropped and counted in `OOW_COUNT[stock]`.

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 3.2.4) performs the identical floor operation, so that the `BEST_BID` / `BEST_ASK` register comparison against the software book is exact rather than approximate.

The 1024-bucket window centered on `PRICE_BASE` + \$5.12 covers a ±\$5.12 range around the live quote (the bank spans \$10.24 end to end). Prices that fall outside this window are silently dropped from the hardware path and counted in the per-pair `OOW_COUNT[stock]` MMIO register; they do *not* fire `parse_err_sticky`, which is reserved for parser/framing errors. The software reference still processes these messages, so a non-zero `OOW_COUNT` is the benchmarker's audit signal that the replay window has drifted beyond the MVP's price range and a new `PRICE_BASE` should be programmed.

PLA Update FSM. (Port A). Figure 6 shows the three-state control FSM that sequences each read–modify–write update to the PLA. In `S_IDLE`, the PLA waits for an incoming update request signaled by `update_valid`; when asserted, the FSM issues a BRAM read on Port A for the target bucket and advances to `S_READ_PEND`. One cycle later, the old quantity (`dout_a`) is available and the combinational arithmetic block computes the new quantity and checks for underflow/overflow. If the update is valid, the writeback occurs in this same cycle and the FSM emits a one-cycle bitmap pulse by asserting `set_valid` together with `set_active` to indicate whether the bucket is now non-zero. The FSM then enters `S_DONE`, which produces a single-cycle `done` pulse before returning to `S_IDLE` to accept the next update.

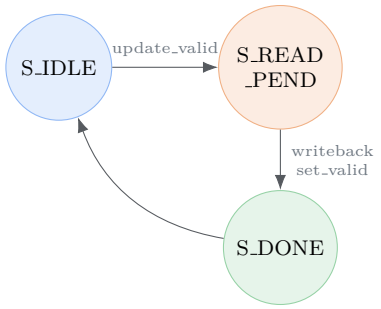


Figure 6: PLA read–modify–write control FSM.

PLA Querying. (Port B). Port B of the dual-port BRAM is dedicated to the top-of-book logic (Section 3.1.6), allowing bucket quantities to be queried with one-cycle latency while Port A handles updates. This lets queries and updates run in parallel so the BBO view remains current. The bitmap is updated on Port A writeback via `set_valid` and `set_active`.

3.1.6 Top-of-Book Tracking: `bbo_bitmap.sv`

`BEST_BID` and `BEST_ASK` are maintained by a 2 stage priority encoder (shown in Figure 7) operating over a bitmap in fabric which tracks which price levels are currently active for a given side of each stock. Each PLA writeback asserts `set_valid` together with `set_active` and `set_price`, toggling a single bit in the array. The parameter `IS_MAX` selects whether each encoder is tracking highest (bid) or lowest (ask) active price. Stage 1 of the encoder partitions the price levels into 32 groups of 32 bits. Each 32-bit group is OR-reduced combinationally, producing a `group_any` flag indicating the presence of any active levels, and a local priority scan identifies the best bit within that group. Stage 2 priority-encodes the 32 `group_any` flags, using a rightmost-set for bid (`IS_MAX=1`), and a leftmost-set for ask (`IS_MAX=0`), and combines the selected group index with its best-bit offset to form a final `best_price`. End-to-end this is two clock cycles from a PLA `set_valid` pulse to an updated `best_price`: one cycle to commit the bitmap write into the `active` register, and one cycle through the `group_any_r` pipeline register before the stage-2 combinational priority encoder resolves the new best.

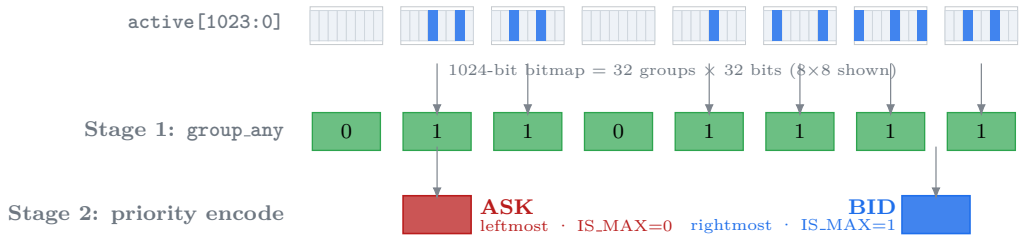


Figure 7: Two-stage priority encoder used for top-of-book tracking.

3.1.7 Diagnostics

The `HT_LOAD` and `MAX_PROBE` registers in Table 4 expose the current occupancy and worst-case probe chain observed since the last counter reset. The per-pair `OOW_COUNT[stock]`

registers expose how many updates were dropped because their price fell outside the PLA window, and `BBO_BEST[stock]` exposes the live `best_price` and validity bit for each (stock, side) pair. Together with the parser's `MSG_COUNT` and `ERR_COUNT`, 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.

3.1.8 Phase-Locked Loop: 50 MHz → 100 MHz, embedded in `orderbook_core`

A phase-locked loop (PLL) is used to multiply the DE1-SoC's 50 MHz reference clock up to the 100 MHz fabric clock used by the order-book pipeline for faster processing. An `altpll` primitive is instantiated *inside* `orderbook_core.sv` because a separate PLL component in Qsys conflicts with the HPS-SDRAM handoff (the SDRAM controller has its own PLL tree; Qsys reset bridging trips on a second hard-IP clock source it didn't create). Embedding inside the peripheral keeps the clock tree self-contained. Quartus reports an maximum actual frequency of 75.67 MHz for the PLL output from post-fit timing analysis.

3.2 Software

3.2.1 Historical Data Emulator: `historical_itch_simulator.py`

In order to generate realistic ITCH test inputs for the FPGA pipeline, the Historical Data Emulator extracts a targeted subset of messages from NASDAQ's historical TotalView-ITCH datasets. A select sample of free datasets are available at `https://emi.nasdaq.com/ITCH/NasdaqITCH`. We used data from December 30th 2019. As historical datasets are on the order of gigabytes, these datasets are preprocessed in Python to concentrate the test set on the message types and stock symbols relevant to our experiments. These messages are prepended with the necessary length fields.

Filtering logic. The extractor streams the raw ITCH feed and saves only a targeted subset of messages to a binary file according to the following rules:

1. **Stock Directory (R).** Write the message only if its stock symbol is in the user–supplied set of desired symbols.
2. **Add / Add–MPID (A, F).**
   - If the stock is in the desired set, write the message and insert its order reference into a `tracked_orders` set.
   - Otherwise, write the message with a small probability p_{noise} to preserve background traffic.
3. **Modify / Cancel / Execute / Replace (E, C, X, D, U).**
   - If the order reference number is in `tracked_orders`, write the message.
   - Otherwise, write it only with probability p_{noise}.

4. **All Other Message Types**

   - Write it only with probability p_{noise}.

5. **Termination.** Stop once a user–specified message limit is reached.

3.2.2 ITCH Stream Generator

The `generate_robust_itch.py` script constructs synthetic, length–prefixed ITCH message streams for stressing the FPGA ingest and order–book pipeline. Each message is emitted in the same framing format used by the userspace loader: a 2-byte big-endian length prefix followed by the ITCH payload. This allows the generated files to be consumed directly by both `itch_load` and the ARM reference implementation `itch_arm`.

The generated streams are designed to produce high turnover in the orderbook hash table, PLA, and BBO logic. The live book is continuously mutated by adds, executes, deletes, and optional replaces. In contrast, `generate_full_itch.py` implements a simpler add-heavy stream intended for basic parser, filter, and BBO bring-up.

Stock Directory. The generator emits an initial block of Stock Directory (`R`) messages for three symbols. In real NASDAQ ITCH feeds, Stock Directory messages appear near the beginning of the session and define the mapping from ticker strings to numeric `stock_locate` IDs. The FPGA filters by these numeric IDs, not by ticker strings.

RKLB	1	tracked symbol
FLNC	2	tracked symbol
NOISE	99	background traffic

All subsequent generated messages reference these numeric `stock_locate` IDs.

Reference allocation. Each tracked symbol receives a separate order-reference range using `REF_BASE` and `REFS_PER_STOCK`. In normal mode these ranges are chosen to avoid unnecessary hash collisions, so correctness tests are not dominated by collision artifacts.

The optional clustered collision mode deliberately constructs references that all hash to the same FPGA hash bucket. This mirrors the FPGA hash function,

$$\texttt{hash} = \texttt{ref[13:0]} \oplus \texttt{ref[27:14]} \oplus \texttt{ref[41:28]}.$$

The generator sets the upper hash slices equal so they cancel under XOR, leaving the requested collision bucket. This mode is used to stress linear probing, `MAX_CHAIN`, and backward-shift delete behavior.

Churn model. The generator is organized around churn cycles. A cycle is a generator-level concept, not an FPGA clock cycle. For each cycle and each tracked stock, the script:

1. Computes bid/ask prices from the selected price model,

2. Issues a bid-side Add (`A`),

3. Optionally issues a Replace (`U`),

4. Issues an Execute (`E`) on an eligible live order,

5. Repeats the same sequence for the ask side,

6. Trims each side to the configured target live depth by deleting oldest orders.

The default price mode is a monotonic trend: bids rise over time and asks fall toward a final tight spread. A demo-oriented `--price-mode wave` option instead makes prices move up and down so the VGA BBO changes more visibly.

The `--target-depth` parameter is the target number of live orders per stock side. Thus a target depth of 200 corresponds to roughly 800 live tracked orders across RKLB bid, RKLB ask, FLNC bid, and FLNC ask.

Executes and replaces. Execute messages may be partial or full-fill. The normal execute quantity is controlled by `--execute-qty`, while `--full-fill-exec-ratio` controls what percentage of executes consume the full remaining quantity. The generator clamps execute quantity to the live order quantity, so generated executes do not underflow the software model.

Execute target selection is controlled by `--execute-policy`, which can choose newest, oldest, round-robin, or aged live orders. The aged policy is used to test executes against orders that have survived multiple churn cycles rather than only same-cycle adds.

Replace messages delete an old reference and insert a fresh reference with a new price/quantity. This matches the FPGA implementation, which handles `U` as a delete leg followed by an insert leg while preserving the original order side.

Noise and tracked density. After generating tracked RKLB/FLNC events, the script spreads them through a larger stream containing `NOISE` messages. These messages have `stock_locate=99`; they exercise the parser and filter path but should not update the tracked RKLB/FLNC order book.

For visual demos, `--tracked-fraction` can make RKLB/FLNC messages a larger fraction of the file so that the VGA more frequently shows `LOC=1` and `LOC=2` instead of mostly `LOC=99`. The `--bbo-churn-demo` option can also periodically delete current best bid/ask levels so the displayed BBO moves more actively.

Intentional misses. The script can inject invalid Deletes and Executes against nonexistent references using `--bad-delete-count` and `--bad-execute-count`. These are used to validate that the FPGA reports misses through `HT_MISS_COUNT` without hanging or corrupting the live book.

Output and expected counters. The final stream consists of:

- Stock Directory (`R`) messages for RKLB, FLNC, and NOISE,

- Tracked event messages (`A`/`D`/`E`/`U`),

- Interspersed NOISE messages,
- Optional intentionally bad deletes/executes.

At generation time, the script prints the tracked message mix and expected FPGA counters. The expected hash-table operation count is computed from the fact that adds and deletes perform one hash operation, while executes and replaces normally perform two hash operations. The expected miss count is the number of intentionally bad delete and execute messages. These printed values are used to check the FPGA's `HT_OPS_TOTAL`, `HT_MISS_COUNT`, `HT_LOAD`, and final BBO behavior during board testing.

3.2.3 ITCH Load: `itch_load.c`

`itch_load` drives the FPGA ingest pipeline from the ARM processor. It is responsible for streaming the ITCH messages from the SD card memory into the ping-pong buffers of the FPGA, configuring user defined hardware register values using AXI–Lite and collecting detailed diagnostics including hardware progressing time. The `itch_load` HPS pipeline is represented in Figure 8.

Mapping the ITCH file into HPS DRAM. First, the ITCH session file is opened from the SD card and `mmap()`–mapped into the HPS DRAM address space.

Mapping the hardware. `itch_load` opens `/dev/mem` and `mmap()`s the AXI–Lite bridge region containing the `orderbook_core` registers and the two 8 KB ingest buffers. All subsequent configuration and data movement is performed via direct MMIO writes into this mapped region.

Pre–scan of the ITCH file. Before the HPS triggers ingestion to begin, the session file is scanned for Stock Directory (R) messages. These messages are identified using the 2-byte length prefix followed by the message-type byte at offset 0 of each payload similar to in the ITCH parser. The bytes corresponding to the stock symbols and locate numbers are then polled according to the NASDAQ protocol. This resolves user–supplied ticker strings into their session–local `stock_locate` IDs, and also extracts the first observed price for each tracked symbol. These values are later used to program the hardware filter registers and to center the PLA price window.

Hardware configuration. Using MMIO writes, `itch_load` sets

- The per–symbol filter registers (`FILTER_ID0..3`) which tell the hardware which stock_locate IDs to accept
- The filter control mask (Pass–All or selective mode), controlling which stock IDs are active
- The per–symbol PLA price baselines (`PRICE_BASE0..3`),

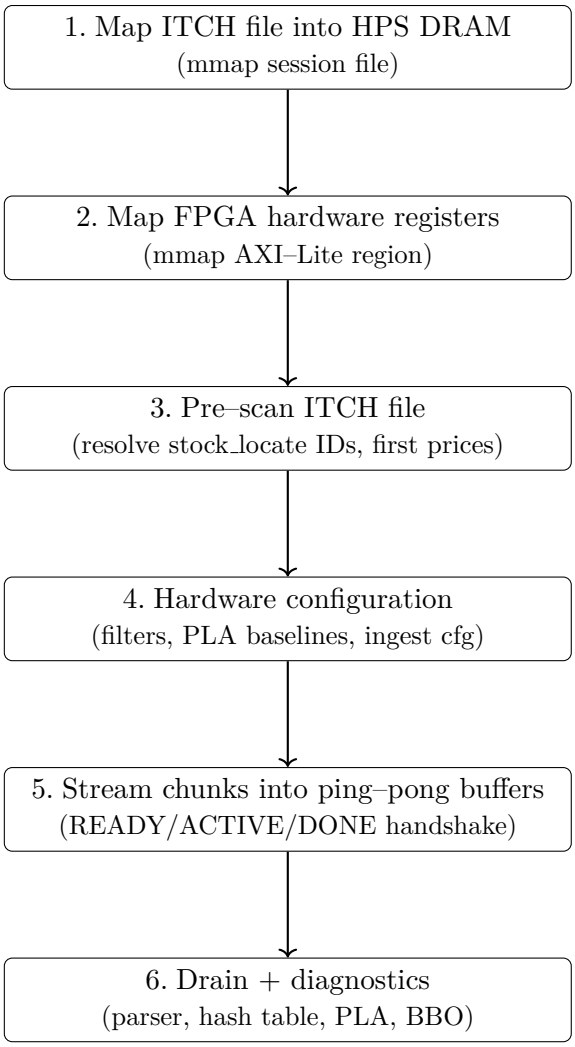


Figure 8: `itch_load` Software Pipeline

- Ingest configuration bits (enable, continuous mode),
- Sends counter reset pulse for all counters

Writing to the ping–pong buffers. The ITCH message file is streamed by `itch_load` from the HPS DRAM buffer into the FPGA in 8 KB chunks into the two ping-pong buffers. This executes in the loop below:

1. Wait for the target buffer to be free by monitoring the handshake flags
2. Copy a whole number of framed ITCH messages into the buffer,
3. Write the chunk length and assert the corresponding length write enable
4. Assert the `READY` bit for the buffer

5. For the first chunk from the file, pulse the `start` flag to trigger `buffered_ingest` to begin parsing from the buffers

Ping–pong handshake. The ping–pong HPS handshake consists of three `W_BUF`n`_FLAGS`:

- **READY** (bit 0): asserted by the software to indicate that the buffer has been filled with new data
- **ACTIVE** (bit 1): asserted by the hardware while it is consuming the buffer,
- **DONE** (bit 2): asserted by the FPGA once processing of the buffer is complete.

Diagnostics. `itch_load` monitors for when all messages from the message file have been passed to and ingested by `buffered_ingest`, as indicated by `busy=0` following the final write of the ping pong buffers. The hardware status registers, including parser state, hash-table load and probe depth, last operation results, PLA out-of-window counts, and BBO buckets are then read and used to generate a summary file.

3.2.4 Software Reference Implementation (HPS ARM): `itch_arm.c`

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 from the session file, performs message framing and parsing in C, maintains its own CPU-resident order book, and measures its own latency using ARM cycle counters. Its internal structure mirrors the hardware: a 16,384-slot hash 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 locate and update existing orders, and replace messages remove the old order and insert the new one with updated fields.

Separate from this reference path, a benchmark and diagnostics tool interacts with 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.

3.2.5 Live Hardware Monitor: `vga_writer.c`

`vga_writer` provides a real-time visual dashboard of the FPGA HFT pipeline by writing directly to the `vga_display_0` character buffer. It maps the lightweight AXI bridge to read the `orderbook_core` registers and renders key status fields—`buffered_ingest` engine state, ping–pong buffer flags, hash-table diagnostics, best bid/ask buckets, and the last parsed ITCH message—onto a 40×30 VGA text grid. This also reads the ARM and FPGA result files to display throughput, latency, and BBO agreement, enabling rapid, at-a-glance validation of correctness and system health.

The FPGA hardware implements the tiling rasterizer, 8×8 glyph bitmapping, and bridges the dual-port AXI–Lite BRAM to the VGA output region in real time, requiring no clock-domain crossings since both the HPS bridge and the VGA pixel clock are derived from the same fabric clock.

4 Resource Budgets

The DE1-SoC's Cyclone V 5CSEMA5F31C6 provides 397 RAM blocks (per Quartus's device model). The following budget estimated allocation these blocks to the large structures (hash table, price-level array, VGA character memory) and leaves small FIFOs, shift registers, and staging buffers to be inferred into MLABs, which Quartus prefers for wide-and-shallow memories under roughly 640 bits.

The final resource usage reported by Quartus is summarised in Table 2.

Top-level Entity	`soc_system_top`
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (ALMs)	25,573 / 32,070 (80%)
Total registers	47,686
Total pins	362 / 457 (79%)
Total virtual pins	0
Total block memory bits	2,097,152 / 4,065,280 (52%)
Total RAM blocks	256 / 397 (64%)
Total DSP blocks	0 / 87 (0%)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1 / 6 (17%)
Total DLLs	1 / 4 (25%)

Table 2: Quartus Analysis & Synthesis summary for `soc_system_top`.

4.1 Ping-Pong Ingest Buffers

`buffered_ingest` utilises two 8KB BRAM buffers for the ping-pong ingest path. Each buffer is implemented as a simple dual-port M10K memory with a 32-bit read port for the BRAM fetch pipeline and a 32-bit write port for HPS loading.

Each 8KB buffer requires:

8,192 bytes = 65,536 bits . Mapped onto a 32-bit-wide M10K configuration, this corresponds to two blocks in parallel for width and four blocks in series for depth, for a total of 8M10K blocks per buffer. The two-buffer ping-pong system therefore consumes 16M10K blocks.

4.2 Hash Table and Price-Level Array

The hash table and price-level array (PLA) account for the majority of on-chip memory usage. Sizing follows from three constraints: a target load factor of 0.5 for the hash table, a ± $ 10.24 per-symbol window for the PLA (at penny granularity), and a total RAM-block budget that must also accommodate VGA character memory, the ingest buffer, text buffers, and small FIFOs, with headroom for synthesis overhead.

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 ≤ 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 the remaining RAM-block headroom but leaves less margin for synthesis overhead.

M10K Block Estimates. M10K blocks store 10,240 bits each and are configurable in widths from × 1 up to × 40 . 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	∼230
Price-level array	8,192	64	524K	∼64
Ping-pong buffers (2 × 8KB)	8,192	32	131K	∼16
Character ROM (VGA)	256	128	33K	∼4
Text buffer (VGA, 80×60)	4,800	8	38K	∼4
Parser → $bookFIFO$	64	128	8K	∼1
Stats / counter memories	–	–	∼20K	∼2
MVP total			∼2,851K	∼321

Table 3: On-chip memory budget. The MVP estimate assumed 321 RAM blocks (72% of the originally budgeted 446).

The hash-table estimate assumes a 128-bit-wide M10K configuration (× 40 packed into four blocks wide, repeated over the depth dimension). Quartus typically lands in the range of 210–240 blocks depending on how aggressively it merges the valid-bit and symbol-ID fields into adjacent words. The PLA, at 8,192 × 64 bits and mapped onto a × 32 M10K configuration, requires two blocks in parallel for width and 32 deep for a total of roughly 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.

The resource usage report (Table 2) shows that this estimate was conservative: the final implementation uses 256 of 397 RAM blocks (%), leaving 141 blocks of headroom for synthesis overhead, error counters, and the `MSG_RATE` ring buffer.

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 M10Ks. 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 remains well under 100 of the 679 available blocks.

5 Hardware/Software Interface

The hardware/software interface is implemented as a *lightweight* Avalon-MM register block and a small set of on-chip memories, instantiated in Qysys as `orderbook_core_0`, and attached to the FPGA-to-HPS lightweight AXI bridge. The interconnection between the HPS and FPGA translates AXI transactions into Avalon-MM accesses to the control/status registers, the ping-pong ingest buffers, and the VGA character-buffer BRAM. The ARM core replays ITCH data from a session file in DRAM into the ping-pong buffers, while the FPGA fabric performs parsing, hashing, and order-book updates on those buffered chunks. The same interface is used for configuration and observability: software writes slow-control settings (ingest enable, filters, price baselines) and reads back message counters, error flags, hash-table diagnostics, and top-of-book summaries, as well as programming the VGA text grid for the live `vga_writer` display. Al offsets below are **byte offsets** from that mapped virtual (or physical) base; each register is one 32-bit word.

Memory-Mapped Register Map

Table 4 lists the memory-mapped registers exposed by `orderbook_core_0` on the lightweight FPGA–to–HPS bridge. Price fields follow the same encoding used internally by the hardware: the FPGA stores raw ITCH prices with four implied decimal places (e.g. 1500000 = \$150.0000), but the BEST_BID and BEST_ASK registers report the penny-bucket index $\lfloor price/100 \rfloor$ relative to the latched per-symbol baseline. This matches the layout of the 1024-entry price-level arrays. Software reconstructs a dollar price via $(base + offset \times 100)/10000$.

The top-of-book fields (BEST_BID, BEST_ASK, BID_QTY, ASK_QTY, BID_DEPTH, ASK_DEPTH) are exposed so he HPS can check the hardware order book against the software reference after replaying the same ITCH stream. Raw counters such as MSG_COUNT confirm that messages were ingested, but they do not guarantee correct parsing, hashing, or aggregation. Reading the FPGA's top-of-book and depth values at the end of a run provides a direct structural comparison with the C reference and catches logic bugs that latency numbers alone cannot reveal.

The CONTROL register forms the slow-control plane: infrequent settings written by the ARM that do not participate in the per-message datapath. These include enabling or disabling ping-pong buffer ingest, selecting which filtered symbols are active for display or statistics, and pulsing counter resets so each run starts from a clean state. Under normal operation these writes occur only at startup or between runs, not once per ITCH message, and therefore lie entirely off the latency-critical path.

Table 4: Memory-Mapped Registers

Byte Offset	Name	Purpose / Fields
Core Control and Status		
0x0000	CONTROL	[0] START ingest; [4] reset counters & sticky bits; [3:2] active_stock (unused in final design)
0x0004	STATUS	FIFO empty/full flags; parse_err sticky; parser FSM state
0x0008	MSG_COUNT	Total parsed messages (from parser)
0x000C	SCRATCH	General-purpose R/W scratch register
0x0010	ID	Magic constant `0xCAFE4840`
0x0014	ERR_COUNT	Ingest/parse error counter (cleared by CONTROL[4])
Filter Subsystem		
0x0018	FILTER_ID_0	16-bit stock_locate for slot 0
0x001C	FILTER_ID_1	16-bit stock_locate for slot 1
0x0020	FILTER_ID_2	16-bit stock_locate for slot 2
0x0024	FILTER_ID_3	16-bit stock_locate for slot 3
0x002C	FILTER_CTRL	[3:0] enable per slot; [4] Pass-All mode
Ingest Engine (Ping–Pong Buffers)		
0x0040	INGEST_CFG	[0] enable ingest; [1] continuous mode
0x0044	INGEST_STAT	Busy, armed, error flags (from ingest FSM)
0x0048	BUF0_LEN	Valid byte count for buffer 0
0x004C	BUF1_LEN	Valid byte count for buffer 1
0x0050	BUF0_FLAGS	[0] ready_set; [1] done_ack; active/-done/error reflected in STATUS
0x0054	BUF1_FLAGS	[0] ready_set; [1] done_ack
Parser Debug (Last Message Latched)		
0x0058	PARSER_STAT	[0] run_done sticky; parser FSM state
0x005C	PARSED_MSG_1	{side, type, stock_locate}
0x0060	PARSED_MSG_2	Parsed price (ITCH units)
0x0064	PARSED_MSG_3	Parsed shares
0x0068	PARSED_MSG_4_HI	Parsed order_ref[63:32]
0x006C	PARSED_MSG_4_LO	Parsed order_ref[31:0]
Hash Table Diagnostics		
0x0070	HT_LOAD	Occupied slots
0x0074	HT_MAX_PROBE	Worst probe chain
0x0078	HT_LAST_OP	{hit, op, dispatch_state, ready, done}
0x007C	HT_LAST_RESULT	{price_out[23:0], qty_out[7:0]}

Byte Offset	Name	Purpose / Fields
0x0080	HT_OPS_TOTAL	Total HT operations
0x0084	HT_MISS_COUNT	Total misses
0x0088	HT_DIAG	Full HT debug bitfield
0x008C	HT_STUCK_CYCLES	Cycles spent in current non-IDLE state
0x0090	HT_STUCK_MAX	Worst stall observed
0x0094	HT_CUR_REF_LO	Current ref[31:0]
0x0098	HT_CUR_REF_HI	Current ref[63:32]
0x009C	HT_PROBE	{chain_len, probe_idx}
0x00A0	HT_DELETE_PTRS	{hole_idx, scan_idx}
PLA (Price-Level Array) and BBO		
0x0100	PLA_CTRL	[0] clear PLA errors (broadcast)
0x0120–0x012C	BBO_BEST[0..3]	{best_valid, best_price[9:0]} per stock-/side pair
0x0140–0x014C	PRICE_BASE[0..3]	24-bit ITCH-unit baseline per stock
0x0150–0x015C	OOW_COUNT[0..3]	Out-of-window drops per stock
VGA Character Buffer Interface		
0x0200	CHAR_MEM_ADDR	Index into 40×30 text grid
0x0204	CHAR_MEM_DATA	Write ASCII + color to selected cell
BRAM Windows		
0x0800–0x47FF	itch_mem (16 KB)	Ping–pong ITCH staging buffer (byte-write, 32-bit read)

6 Group Reflections

6.1 Division of Work

- **Hardware/Software Interfaces** — Marcus
- **Hardware ITCH Parsing** — Sid, Sarah
- **Hash Table** — Shawn
- **Bitmap, Price-Level Array, Orderbook Core/Dispatch FSM** — Shawn
- **Software Pipeline** — Marcus, Sarah
- **Hardware Configuration** — Marcus
- **VGA** — Marcus
- **Test Generation and Robust Stress Testing** — Sid
- **Historical Testing** — Sarah

6.2 Lessons Learned

- Simple add-only streams were useful for bring-up, but mixed streams with deletes, executes, replaces, misses, and collisions were necessary to build confidence in the full pipeline.
- Hardware visibility was essential. Exposing internal state through debug registers made stalls and edge cases much easier to diagnose.
- Software reference checks were important for separating FPGA logic bugs from stale state, file-transfer issues, cache effects, or misleading display output.
- The HPS/software side matters as much as the FPGA logic: file transfer, kernel modules, memory mapping, and cache behavior all affected testing.
- Streaming hardware needs careful handshaking. Small timing details around valid/-done pulses, stalls, and buffer ownership can determine whether the whole pipeline keeps moving.
- Stateful hardware data structures require more edge-case testing than pure datapaths. Deletes, replacements, partial fills, and collisions exposed behavior that simple inserts never would.
- VGA output was valuable for demos, but terminal counters and logs were necessary for precise validation.

6.3 Future Advice

- Agree early on what performance metrics and demo cases will be most convincing. This makes it easier to design tests that directly support the final demonstration.
- Keep the instructor or project advisor in the loop throughout development, especially when deciding what results to collect.
- Build observability into the hardware from the beginning. Debug registers, counters, and status fields are much cheaper to add early than during final debugging.
- Treat the software/HPS path as part of the system, not just a way to launch tests. File movement, memory mapping, kernel modules, and cache behavior can all affect results.
- Test stateful structures with adversarial cases early: collisions, deletes, replacements, full fills, misses, and repeated reuse of the same price levels.
- Make test generation reproducible. A good generator with documented command lines is more useful than a few hand-created binary files.
- Preserve time for Quartus/Qsys integration and rebuilds. Hardware iteration is slow, so debugging hooks and stable build scripts matter.

7 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.

Appendices

A Testing Results

This section summarizes the main board tests performed on the DE1-SoC for the ITCH ingest and orderbook pipeline. It preserves the test narrative: what each test was designed to validate, what was observed, and what confidence each test provides.

A.1 How Results Were Judged

The most important correctness fields from `itch_load` are:

```
MSG_COUNT       total messages parsed from the file
ERR_COUNT       framing / ingest errors; should normally be 0
HT_LOAD         live orders left in the hash table
HT_MAX_PROBE    worst hash probe chain observed
HT_OPS_TOTAL    completed hash-table operations
HT_MISS_COUNT   lookup/delete operations that did not find a ref
HT_DIAG         hash/orderbook FSM state snapshot
HT_STUCK        live and maximum non-idle stall counter
```

For normal valid-ref generated files, `HT_MISS_COUNT = 0` is expected. For intentional-miss tests, a nonzero miss count is the expected result.

The VGA was used as a live dashboard for BBO state and ARM-vs-FPGA comparison. Before clean comparisons, stale scoreboard files were removed.

A.2 Baseline Robust Execute Test

Central point: Prove that the FPGA handles a large 1M-message file with adds, deletes, and partial executes, not just add-only traffic.

Generated with:

```
python3 market_sim/generate_robust_itch.py \
  --output sw/userspace/input/robust_exec_itch.bin
```

Tracked RKLB/FLNC mix:

```
A adds      = 4000
D deletes   = 3200
E executes  = 4000 partial fills
U replaces  = 0
```

Observed FPGA result:

```
MSG_COUNT      = 1000000
ERR_COUNT      = 0
HT_LOAD        = 800
HT_OPS_TOTAL   = 15200
HT_MISS_COUNT  = 0
Final BBO      RKLB $5.00/$5.02, FLNC $2.00/$2.02
```

What it revealed: Executes were genuinely present. `HT_OPS_TOTAL = 15200` matches the expected operation count. Final BBO values matched the generator endpoints.

A.3 Replace Test

Central point: Validate that the replace path correctly handles add, delete, execute, and replace messages.

Generated with:

```
python3 market_sim/generate_robust_itch.py \
  --include-replaces \
  --output sw/userspace/input/robust_replace_itch.bin
```

Expected tracked mix:

```
A adds      = 4000
D deletes   = 3200
E executes  = 4000
U replaces  = 796
```

Observed: clean completion, no errors, no misses, correct VGA BBO.

What it revealed: Replace logic works on a nontrivial stream. Replace messages delete the old ref and insert the new ref while preserving order side.

A.4 Higher-Churn 2k-Cycles Test

Central point: Increase tracked RKLB/FLNC churn cycles while keeping the file size at 1M messages.

Generated with:

```
python3 market_sim/generate_robust_itch.py \
  --include-replaces \
  --cycles 2000 \
  --target-depth 200 \
  --output sw/userspace/input/robust_replace_2kcycles.bin
```

Observed: no parser errors, no timeouts, correct VGA BBO.

What it revealed: The pipeline remains stable under denser activity.

A.5 Deep Book Test

Central point: Increase live-order depth so the hash table carries more state while churn continues.

Generated with:

```
python3 market_sim/generate_robust_itch.py \
  --include-replaces \
  --cycles 2000 \
  --target-depth 500 \
  --output sw/userspace/input/robust_replace_deep.bin
```

Observed: correct VGA behavior, no errors.

What it revealed: The design handles deeper live books while processing mixed operations.

A.6 2M-Message Sustained Ingest Test

Central point: Stress sustained file ingest and ping-pong buffering.

Generated with:

```
python3 market_sim/generate_robust_itch.py \
  --include-replaces \
  --messages 2000000 \
  --cycles 3000 \
  --target-depth 500 \
  --output sw/userspace/input/robust_replace_2m.bin
```

Observed: successful 2M-message run, correct VGA behavior.

What it revealed: The ingest path and orderbook can run for long streams, not only 1M-message tests.

A.7 Full-Fill Execute Test

Central point: Validate the execute branch where an order is fully consumed and removed.

Generated with:

```
python3 market_sim/generate_robust_itch.py \
  --execute-qty 100 \
  --output sw/userspace/input/robust_fullfill_exec.bin
```

Expected:

```
A adds      = 4000
E executes  = 4000 full fills
HT_LOAD     = 0
HT_OPS_TOTAL = 12000
HT_MISS_COUNT = 0
```

Observed: correct behavior, empty BBO after ingest.

What it revealed: Full-fill execute path works correctly.

A.8 Collision Saturation Test

Central point: Determine whether the hash table survives pathological collision pressure without hanging.

Generated with:

```
python3 market_sim/generate_robust_itch.py \
  --include-replaces \
  --collision-mode clustered \
  --output sw/userspace/input/robust_collision_churn.bin
```

Observed FPGA result:

```
MSG_COUNT      = 1000000
ERR_COUNT      = 0
HT_LOAD        = 64 / 16384
HT_MAX_PROBE   = 64
HT_OPS_TOTAL   = 12478
HT_MISS_COUNT  = 10882
HT_DIAG        hash_state=IDLE ob_state=IDLE
HT_STUCK       cycles=0 max=465
```

What it revealed: The design did not hang under extreme collision pressure. Misses are expected once the 64-entry probe limit is exceeded.

A.9 Aged Execute Test

Central point: Test executes against older orders that have lived through several churn cycles.

Generated with:

```
python3 market_sim/generate_robust_itch.py \
  --include-replaces \
  --execute-policy aged \
  --execute-min-age 5 \
  --full-fill-exec-ratio 25 \
  --messages 1000000 \
  --cycles 1000 \
  --target-depth 200 \
  --output sw/userspace/input/robust_aged_exec.bin
```

Observed FPGA result:

```
MSG_COUNT      = 1000000
ERR_COUNT      = 0
HT_LOAD        = 800 / 16384
HT_MAX_PROBE   = 1
HT_OPS_TOTAL   = 15589
HT_MISS_COUNT  = 8
```

What it revealed: The run completed cleanly, but 8 unexpected misses were observed. This indicates a small correctness gap in the aged-execute path.

A.10 Intentional Miss Test

Central point: Verify that the hash table reports misses correctly for invalid delete/execute refs.

Generated with:

```
python3 market_sim/generate_robust_itch.py \
  --bad-delete-count 20 \
  --bad-execute-count 20 \
  --output sw/userspace/input/robust_miss_test.bin
```

Observed FPGA result:

```
HT_OPS_TOTAL   = 15240
HT_MISS_COUNT  = 40
```

What it revealed: Expected behavior. The design handles intentional misses cleanly.

A.11 Diagnostics Added After Early Hangs

Additional MMIO debug registers were added:

```
0x22 HT_DIAG
0x23 HT_STUCK_CYCLES
0x24 HT_CUR_REF_LO
0x25 HT_CUR_REF_HI
0x26 HT_PROBE
0x27 HT_STUCK_MAX
0x28 HT_DELETE_PTRS
```

What it revealed: Later successful runs ended with both FSMs idle and no stalls, confirming stability.

A.12 Current Confidence

The tests above give confidence that the FPGA path handles:

- large-file ingest,
- stock-directory filtering,
- RKLB/FLNC BBO maintenance,
- partial and full-fill executes,
- deletes and replaces,
- deeper live books,
- sustained 2M-message ingest,
- collision saturation without hanging,
- intentional hash misses without hanging.

Remaining useful tests include clean collision workloads, investigation of the aged-execute misses, cold-boot stability, and timing analysis.

A.13 Combined Stress Tiers

These tests combine the hardest dimensions already tested separately.

A.13.1 Tier 1: Hard Mostly-Clean Mixed Workload

Central point: Combine replaces, deeper book state, more cycles, larger file size, and a mix of partial and full-fill executes without intentionally forcing pathological hash collisions.

Generated with:

```
python3 market_sim/generate_robust_itch.py \
  --include-replaces \
  --cycles 2500 \
  --target-depth 500 \
  --full-fill-exec-ratio 25 \
  --replace-interval 2 \
  --messages 2000000 \
  --output sw/userspace/input/robust_combo_clean_2m.bin
```

Pass criteria:

```
MSG_COUNT matches the file
ERR_COUNT = 0
HT_MISS_COUNT = 0
HT_DIAG reports hash_state=IDLE and ob_state=IDLE
HT_STUCK cycles=0 at the end
BBO matches ARM/VGA expectations
```

Observed FPGA result:

```
MSG_COUNT      = 2000000
ERR_COUNT      = 0
HT_LOAD        = 2000 / 16384
HT_MAX_PROBE   = 1
HT_OPS_TOTAL   = 45492
HT_MISS_COUNT  = 0
HT_DIAG        hash_state=IDLE ob_state=IDLE
HT_STUCK       cycles=0 max=195
```

What it revealed: This harder clean workload passed. The design handled a 2M-message file with replaces, deeper live state, frequent full-fill executes, and no intentional collision pressure. `HT_LOAD = 2000` shows a substantially larger live book than the baseline, while `HT_MISS_COUNT = 0` and idle FSMs show the mixed operation stream remained coherent.

A.13.2 Tier 2: Clean Collision Mixed Workload

Central point: Combine real collision pressure with replaces and full-fill executes, while keeping the live colliding set below `MAX_CHAIN = 64` so the correct result should still have no misses.

Generated with:

```
python3 market_sim/generate_robust_itch.py \
  --include-replaces \
  --collision-mode clustered \
  --target-depth 10 \
  --cycles 1000 \
  --full-fill-exec-ratio 25 \
  --replace-interval 2 \
  --output sw/userspace/input/robust_combo_collision_clean.bin
```

Pass criteria:

```
ERR_COUNT = 0
HT_MAX_PROBE > 1
HT_MAX_PROBE <= 64
HT_MISS_COUNT = 0
HT_DIAG reports hash_state=IDLE and ob_state=IDLE
HT_STUCK cycles=0 at the end
```

First attempt (with `--target-depth 15`) produced:

```
MSG_COUNT      = 1000000
ERR_COUNT      = 0
HT_LOAD        = 63 / 16384
HT_MAX_PROBE   = 64
HT_OPS_TOTAL   = 17416
HT_MISS_COUNT  = 3026
HT_DIAG        hash_state=IDLE ob_state=IDLE
HT_STUCK       cycles=0 max=463
```

What it revealed: This run was stable but not a clean collision test. The file still reached the `MAX_CHAIN = 64` boundary, producing many misses. This is another collision-saturation/no-hang result, not proof of zero-miss collision correctness. Reducing the target depth to 10 leaves more margin below the probe-chain limit.

A.13.3 Tier 3: Adversarial Collision Saturation

Central point: Intentionally exceed the collision capacity while also including replaces and mixed partial/full-fill executes. This tier is judged by no-hang behavior, not by zero misses.

Generated with:

```
python3 market_sim/generate_robust_itch.py \
  --include-replaces \
  --collision-mode clustered \
  --cycles 2000 \
  --target-depth 500 \
  --full-fill-exec-ratio 25 \
```

```
  --replace-interval 2 \
  --messages 2000000 \
  --output sw/userspace/input/robust_combo_collision_saturation_2m.bin
```

Pass criteria:

```
ERR_COUNT = 0
HT_MAX_PROBE = 64
HT_MISS_COUNT > 0 is expected
HT_DIAG reports hash_state=IDLE and ob_state=IDLE
HT_STUCK cycles=0 at the end
No timeout diagnostic block from itch_load
```

Observed FPGA result:

```
MSG_COUNT      = 2000000
ERR_COUNT      = 0
HT_LOAD        = 64 / 16384
HT_MAX_PROBE   = 64
HT_OPS_TOTAL   = 24982
HT_MISS_COUNT  = 22662
HT_DIAG        hash_state=IDLE ob_state=IDLE
HT_STUCK       cycles=0 max=465
```

What it revealed: Tier 3 passed as an adversarial saturation test. The file forced the hash table to the `MAX_CHAIN = 64` boundary, generated many expected misses, and still completed without parser errors or a hardware hang. The final idle FSM states are the key correctness signal for this tier.

B Presentation Slides

FPGA HFT Order Book Core

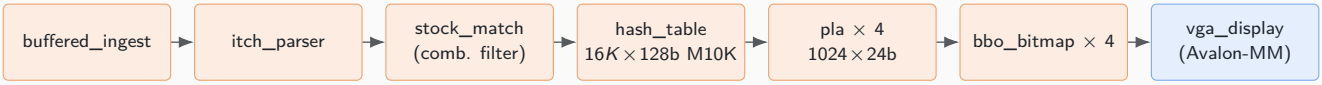
ITCH 5.0 + 16K-slot hash table + hardware BBO @ 100 MHz on DE1-SoC

Shawn Kathuria · Sarah Hagan · Shiyao Marcus Lam · Siddharth Raykar

Spring 2026

CSEE W4840: Embedded Systems (SP26)

Pipeline



Clock. `altpll` inside `orderbook_core` doubles 50 MHz → 100 MHz; looped back as the Avalon-MM slave clock, drives the entire fabric. **Backpressure.** `ht_stall` (= `ht_pending` ∨ `ht_state` ≠ `HT_IDLE`) freezes byte emission in `buffered_ingest`, which back-pressures the HPS via `buf_ready/buf_done`. No drops, no torn frames. **Worst-case latency.** ∼100 cycles (64-deep probe + backward-shift delete) ≈ 1 μs.

PLL — 50 MHz → 100 MHz, embedded in `orderbook_core`

An `altpll` primitive is instantiated *inside* `orderbook_core.sv`. A separate PLL component in Qsys conflicts with the HPS-SDRAM handoff (the SDRAM controller has its own PLL tree; Qsys reset bridging trips on a second hard-IP clock source it didn't create). Embedding inside our peripheral keeps the clock tree self-contained.

Platform Designer wiring:

- `ref_clk` = clock sink (50 MHz from `CLOCK_50`).
- `clk_out` = clock source (100 MHz).
- Loop `clk_out` → peripheral `clk` by hand in Qsys.
- `pll_locked` gates the Qsys reset controller.

Every intra-module hop is already a one-cycle stage, so doubling the clock closed timing on the first compile with no RTL changes. VGA derives its 25 MHz pixel clock as `hcount[0]` off the raw 50 MHz Qsys clock — no extra PLL.

```
altpll #(
  .bandwidth_type ("AUTO"),
  .compensate_clock ("CLK0"),
  .clk0_divide_by (1),
  .clk0_duty_cycle (50),
  .clk0_multiply_by (2),
  .clk0_phase_shift ("0"),
  .inclk0_input_frequency(20000),
  .intended_device_family("Cyclone V"),
  .lpm_type ("altpll"),
  .operation_mode ("NORMAL"),
  .pll_type ("AUTO"),
  .port_clk0 ("PORT_USED"),
  .port_inclk0 ("PORT_USED"),
  .port_locked ("PORT_USED"),
  .width_clock (5)
) u_pll_50_to_100 (
  .inclk ({1'b0, ref_clk}),
  .clk (pll_clk_w),
  .locked (pll_locked_w)
);
assign clk_out = pll_clk_w[0];
```

Software-to-Hardware Entrance: `itch_load.c`

Details in Appendix

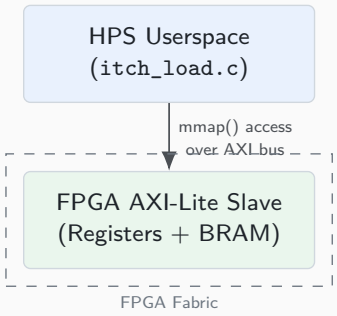
Major Flow:

1. `open("/dev/mem")` and `mmap()` the AXI-Lite bridge.
2. **Pre-scan**: Resolve ticker symbols to `stock_locate` IDs and find first prices.
3. **Config**: Program hardware filter registers and price baselines via MMIO.
4. **Ingestion**: Feed 8 KB chunks into ping-pong BRAM via direct pointer writes.

Ping-Pong Handshake (`W_BUFn_FLAGS`):

- `READY` (bit 0): CPU → FPGA (data is ready).
- `ACTIVE` (bit 1): FPGA internal (processing).
- `DONE` (bit 2): FPGA → CPU (finished).

Invariant: CPU waits for `flags & 3 == 0` before overwriting.




```
// Pointer-based MMIO
volatile uint32_t *regs = (uint32_t *)map;

// Tight ping-pong poll (READY=0 & ACTIVE=0)
while ((regs[FLAGS_OFFSET] & 3u) != 0) {
    if (++spins >= TIMEOUT) break;
}
// Direct copy to BRAM, set length, pulse READY
regs[LEN_OFFSET] = chunk_size;
regs[FLAGS_OFFSET] = 1u; // READY=1

// Wait for DONE (bit 2) from FPGA
while ((regs[FLAGS_OFFSET] & 4u) == 0) { ... }

// Switch to opposite buffer
```

`buffered_ingest` — ping-pong over altsyncram

16 KB on-chip BRAM, 4096 × 32-bit words, split into two 8 KB halves. HPS writes one half via Avalon-MM byte writes while the FPGA drains the other.

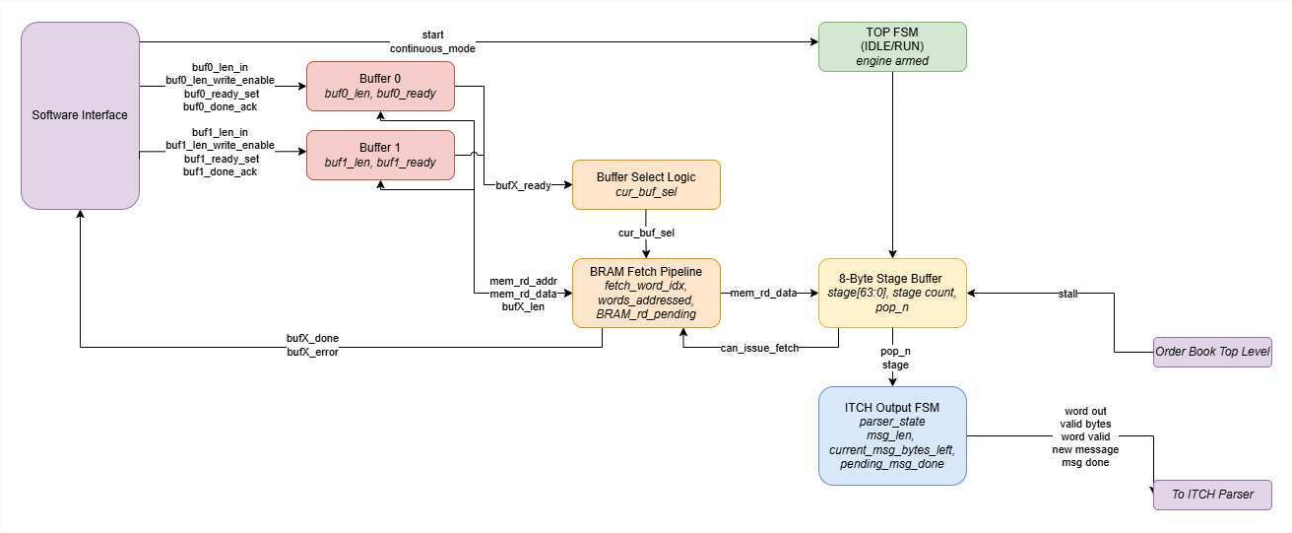
Word interface to the parser.

- `new_message`: 1-cycle pulse before any payload bytes.
- `word_valid`: 1-cycle pulse with 1–4 payload bytes in `word_out`, `valid_bytes` $\in [1, 4]$.
- `msg_done`: 1-cycle pulse *one cycle after* the last `word_valid`, so the parser's NBA writes to `msg_buf` have settled before decode.

Wire frame: 2-byte big-endian per-message length, then ITCH payload. Internal 8-byte FIFO staging ring decouples 32-bit BRAM word reads from the up-to-4-bytes-per-cycle parser output.

```
altsyncram #(
  .operation_mode("BIDIR_DUAL_PORT"),
  .width_a(32), .widthad_a(12),
  .numwords_a(4096),
  .width_byteena_a(4),
  .ram_block_type("M10K"),
  .outdata_reg_a("UNREGISTERED"),
  .outdata_reg_b("UNREGISTERED"),
  .read_during_write_mode_mixed_ports
               ("DONT_CARE")
) itch_mem_inst ( ... );
```

`buffered_ingest` — Buffered Ingest Block Diagram

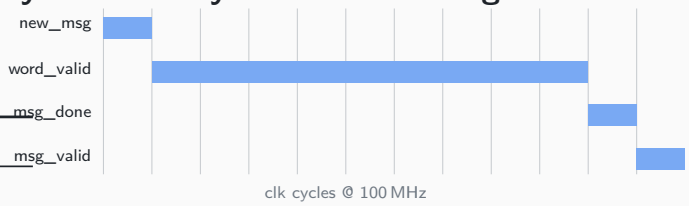


`itch_parser` — 32-bit word interface, 64-byte buffer

ITCH 5.0 messages top out at 50 bytes (P/Q/I). The parser uses a 64-byte scratch (`msg_buf`) for headroom. Up to four payload bytes are written per cycle at `byte_count`; on `msg_done` the type byte selects which fixed-offset fields to latch.

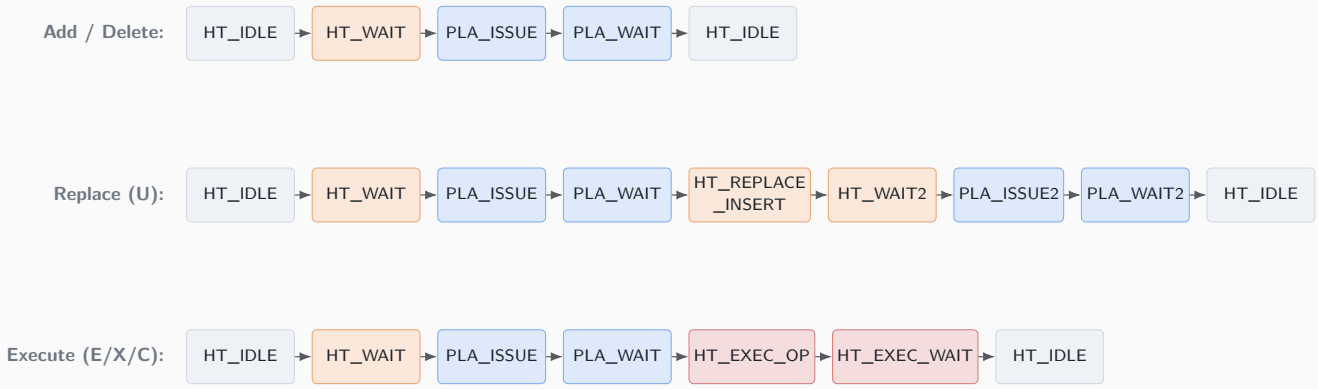
Field	Offset	Decoded for
type	0	all
stock_locate	1..2	all
order_ref	11..18	A/F/E/X/D/U/C
shares	19..22	A/F/E/X/U/C
side	19	A/F ('B'/'S')
price	23 or 32	A/F (23); U new price (32)
new_order_ref	19..26	U

Cycle anatomy of one A-message



9 `word_valid` cycles for a 36-byte A-payload, 1 cycle of `msg_done`, then 1 cycle of `msg_valid` when the dispatch FSM latches.

Dispatch FSM — 10 states, one row per op type



A/D: one HT op + one PLA delta. **U**: first HT op is a DELETE that returns the *original* side; PLA decrement at old bucket, INSERT new ref with `ht_side_orig`, PLA increment. **E/X/C**: LOOKUP returns old price/sym; PLA decrement by clamped fill; then DELETE on full fill or in-place INSERT (reduced qty) on partial.

Hash table — 2^{14} slots, 128 bits each

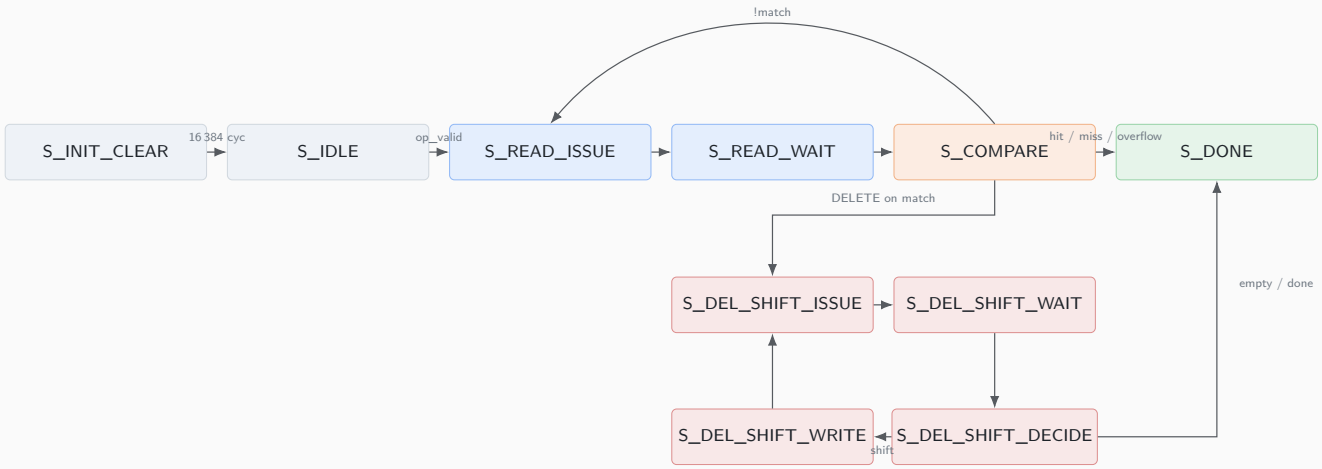
- 16 384 slots, 128 bits each, M10K.
- Hash = 14-bit XOR fold of `order_ref`: `ref[13:0] ^ ref[27:14] ^ ref[41:28]`.
- Linear probing, `MAX_CHAIN`=64; overflow pulses on exceed.
- 1-cycle BRAM read; the FSM bridges with `S_READ_ISSUE` → `S_READ_WAIT` → `S_COMPARE`.
- At reset, `S_INIT_CLEAR` walks all 16 384 slots and writes zero (replaces a synthesis-hostile for-loop reset).
- INSERT on `ref`-match updates in place without bumping `ht_load` (E partial-fill rewrite).

128-bit slot layout

```
[127:114] sym (14b) // ITCH stock_locate
[113] side // 0=B, 1=S
[112:89] qty (24b)
[ 88:65] price (24b) // ITCH units ($0.0001)
[ 64:1] ref (64b) // ITCH order ref
[ 0] valid
```

Exactly 128 bits, no padding. `sym` was widened from a 2-bit filter index to the full 14-bit ITCH locate so the PLA/BBO can fan out without a sideband.

Hash-table FSM



Shift criterion (robin-hood-style):

`shift_needed` $=$ `rd_valid` $\wedge$ (`hole_idx` $-$ `rd_hash`) $<$ (`hole_idx` $-$ `scan_idx`). Bounded by `MAX_CHAIN`=64 — no tombstones, no asymptotic degradation under churn. `S_DONE` returns to `S_IDLE` on the next cycle.

Backward-shift delete

hash(A)=10, hash(B)=10, hash(C)=11, hash(D)=11.

Naive tombstone breaks future lookups of C and D.

Backward-shift pulls each into its earliest legal slot, preserving the probe invariant.

4-state subroutine inside the HT FSM. Cost bounded by `MAX_CHAIN`.

PLA — 1024 buckets, true dual-port, sign+magnitude delta

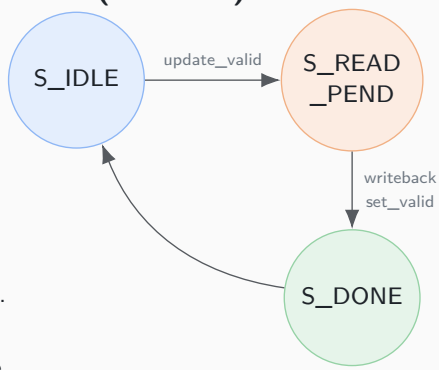
4 banks = 2 stocks × {bid, ask}. Each bank is 1024 × 24b in M10K (`(* ramstyle="M10K" *)`). Bucket = `((price − PRICE_BASE) / 100)[9:0]` — ITCH price is in \$0.0001, /100 gives 1-cent buckets; Quartus emits multiply-by-reciprocal. HPS sets `PRICE_BASE = first_price − $5.12` so the live quote sits mid-window. Out-of-window updates are dropped and counted in `OOW_COUNT[stock]`.

Two ports:

- **A** (update): FSM reads in `S_READ_PEND` and writes back the *same* cycle (`addr_a` held to `saved_price`; combinational `we_a`).
- **B** (query): combinational `addr_b`, registered `query_qty_valid` one cycle later. The BBO/VGA can poll without disturbing the update FSM.

`update_delta` is `signed [QTY_BITS:0]`; the arith block sign-extends to `QTY_BITS+2` and reads the top two bits for underflow (negative) and overflow ($\geq 2^{QTY\_BITS}$). Bitmap is poked with `set_active = (new_qty != 0)` on writeback.

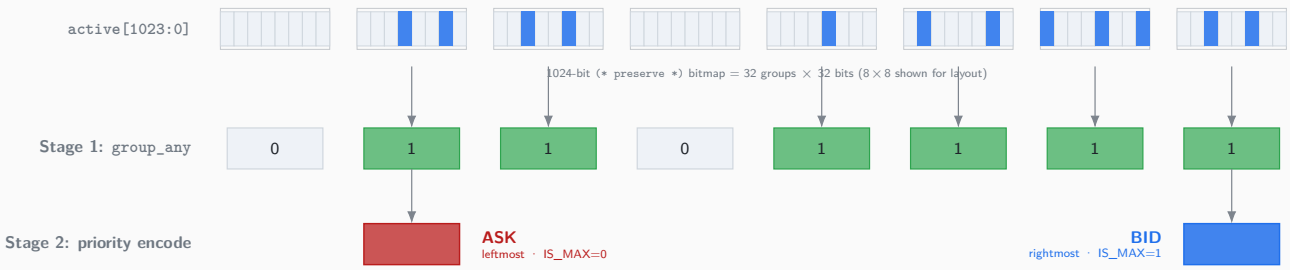
FSM (3 states)



To BBO:

```
set_valid // 1-cycle pulse
set_active // qty != 0 after update
set_price // bucket index
```

Two-stage priority encoder for BBO



Bitmap kept in fabric (`(* preserve *)`); pushing to M10K would serialize the encoder. Stage 1 OR-reduces each 32-bit group combinationally and *registers* `group_any` between stages. Stage 2 priority-encodes 32 group flags — rightmost-set for bid (`IS_MAX=1`), leftmost-set for ask (`IS_MAX=0`). One cycle from `set_valid` to `best_price`.

MMIO register map (`orderbook_core`, word offsets)

Offset	Name	Purpose
0x00	CONTROL	[0] START ingest, [4] reset counters & sticky bits
0x01	STATUS	[8] parse_err sticky
0x02	MSG_COUNT	messages parsed
0x04	ID	magic `0xCAFE4840`
0x06–0x09	FILTER_ID[0..3]	16-bit `stock_locate` per slot
0x0A	ERR_COUNT	framing/ingest errors
0x0B	FILTER_CTRL	[3:0] enable per slot, [4] Pass-All
0x10	INGEST_CFG	[0] enable, [1] continuous
0x12–0x15	BUF*n*_LEN/FLAGS	per-half length, ready, done, active, error
0x1C–0x21	HT_LOAD ... HT_MISS	load, worst probe, ops, misses
0x22–0x28	HT_DIAG, HT_STUCK, HT_CUR_REF, HT_PROBE, ...	deep HT debug
0x40	PLA_CTRL	[0] clear PLA errors (broadcast)
0x48–0x4B	BBO_BEST[0..3]	{best_valid, best_price[9:0]} per pair
0x50–0x53	PRICE_BASE[0..3]	24-bit ITCH-unit base per stock
0x54–0x57	OOW_COUNT[0..3]	out-of-window drops per pair
0x200..	itch_mem	16 KB byte-write / 32-bit read window

VGA Dashboard: 40×30 Text-Mode Tile Grid

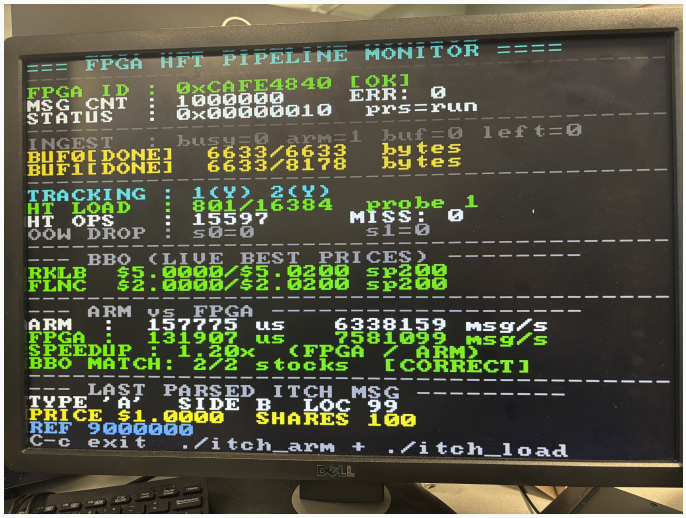
Details in Appendix

HPS: Writing to the dashboard

- Separate process (`vga_writer`) polls MMIO status.
- Writes 32-bit "tuples" to BRAM at offset 0x8000.
- **Cell format**: `{R[8], G[8], B[8], ASCII[8]}`.

FPGA: Tiling & Rasterization

- **Tiling**: `char_col = x[9:4], char_row = y[8:4]` (16×16 pixels/cell).
- **Addressing**: `addr = row*40 + col`.
- **Glyph ROM**: 8x8 bitmap lookup; foreground color applied if bit is set.



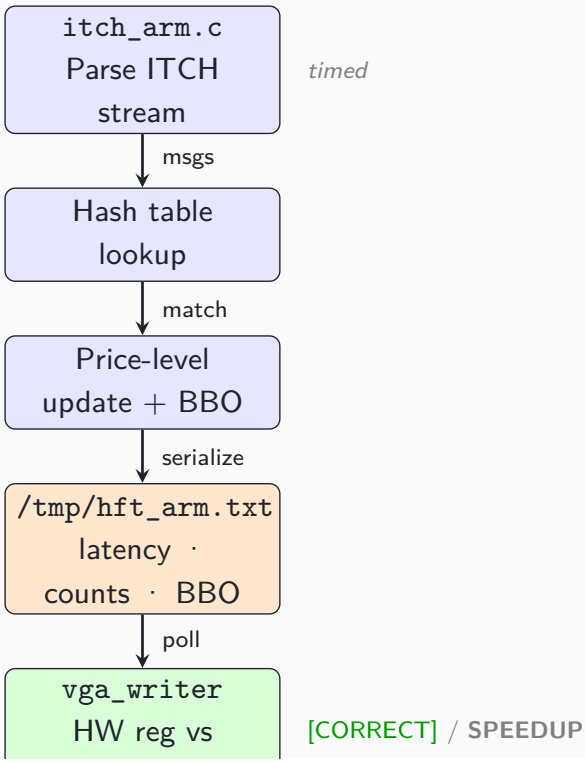
The rasterizer bridges the dual-port AXI-Lite BRAM to the VGA output region in real-time, requiring no clock-domain crossings as both the HPS bridge and VGA pixel clock (derived) share the fabric clock.

ARM Software Reference (Golden Model)

Purpose: Golden-model baseline for correctness validation and speedup measurement.

Key outputs:

- `total_msgs` / `msgs_filtered`
- Latency (via `CLOCK_MONOTONIC`)
- BBO buckets → `/tmp/hft_arm.txt`



`generate_robust_itch.py` — controllable test streams

Purpose. Generate framed ITCH binaries that drive both the ARM twin and the FPGA with the same repeatable workload.

How it builds a file

- Emits Stock Directory (`R`) messages first: `RKLB=1`, `FLNC=2`, `NOISE=99`.
- Runs a configurable number of **cycles**; each cycle creates RKLB/FLNC bid+ask churn.
- Tracks a software live-order book so generated deletes, executes, and replaces target valid refs unless a miss test asks otherwise.
- Spreads tracked messages through a larger stream of `NOISE` messages to exercise parser+filter behavior.

Useful terms

- **Cycle**: one generator iteration over RKLB/FLNC bid and ask activity; not an FPGA clock cycle.
- **Churn**: adds/deletes/executes/replaces changing the live book.
- **Tracked**: RKLB/FLNC messages that pass hardware filters.
- **NOISE**: synthetic locate 99 traffic that should be parsed but filtered out.
- **Full-fill**: execute consumes the full remaining order qty.

Key knobs.

Flag	Effect
`-include-replaces`	add `U` replace traffic
`-target-depth N`	control live book depth
`-collision-mode clustered`	force hash collisions
`-bad-delete-count / -bad-execute-count`	intentional misses
`-tracked-fraction, -price-mode wave, -bbo-churn-demo`	livelier VGA demos

Hardware Test Matrix

Test	Generator	MSG_CNT	HT_OPS	HT_MISS	Verdict
Baseline robust	A/D/E (4k/3.2k/4k)	1 M	15 200	0	BBO match
Replace	+ `-include-replaces`	1 M	–	0	BBO match
Higher-churn 2k	+ `-cycles 2000`	1 M	–	0	VGA OK
Deep book	+ `-target-depth 500`	1 M	–	0	VGA OK
2M sustained	2M-message ingest	2 M	–	0	No hang
Full-fill execute	`-execute-qty 100`	–	12 000	0	HT_LOAD=0
Collision saturation	`-collision-mode clustered`	1 M	12 478	10 882	*Expected, no hang*
Aged execute	`-execute-policy aged`	1 M	15 597	0	Clean Re-run
Intentional miss	20 bad-D + 20 bad-E	1 M	15 240	40	Exact Match
Tier 1 mixed-clean	2M, depth 500, full-fills 25%	2 M	45 492	0	BBO match
Tier 2 clean coll.	clustered, depth 10	1 M	–	3026	Saturation
Tier 3 adversarial	clustered, 2M, depth 500	2 M	24 982	22 662	No hang

Every passing run ends with `hash_state=IDLE`, `ob_state=IDLE`, `HT_STUCK cycles=0`. Worst `HT_STUCK_MAX`: 465 cycles (collision-cluster file).

Demo Test 1 — Full-Fill Executes

Purpose. Show that the execute path handles the strongest execute case: an order is completely filled and must disappear from both the hash table and the displayed BBO.

File	`robust_fullfill_exec.bin`
Generator knob	`-execute-qty 100`
Tracked mix	4000 adds, 4000 full-fill executes
Expected final state	no live tracked orders

Observed result.

- Run completed with `ERR_COUNT=0`.
- `HT_LOAD=0`, confirming every tracked order was removed.
- `HT_MISS_COUNT=0`, so the executes found their refs.
- BBO section ends empty, which is the correct final state.

Demo note: run this first after a clean hardware reset/reboot, because `itch_load` clears counters but not old book state.

Demo Test 2 — Clean Mixed Workload

Purpose. Show the strongest clean capability test: sustained ingest, deeper live book state, replaces, partial executes, and full-fill executes without intentional bad refs or pathological collisions.

File	`robust_combo_clean_2m.bin`
Generator knobs	`-include-replaces`, `-messages 2000000`
Stress added	depth 500, replace interval 2, 25% full-fill executes
Expected correctness	zero misses, BBO match, idle FSMs

Observed FPGA result.

MSG_CNT	ERR	HT_LOAD	HT_OPS	HT_MISS
2 000 000	0	2000	45 492	0

- Hash table and orderbook FSMs returned to `IDLE`.
- `HT_LOAD=2000` shows a substantially live book.
- `HT_MISS=0` shows the mixed stream stayed coherent.

Demo test 3 — Collision Saturation

Purpose. Demonstrate the known hash-table limit honestly. This file forces many refs into one collision cluster, so the expected result is graceful saturation, not zero misses.

File	`robust_combo_collision_saturation_2m.bin`
Generator knobs	`-collision-mode clustered`, 2M messages
Stress added	replaces, 25% full-fills, depth 500
Pass condition	no hang; FSMs idle; misses are expected

Observed FPGA result.

MSG_CNT	ERR	HT_LOAD	HT_MAX_PROBE	HT_MISS
2 000 000	0	64	64	22 662

- `HT_MAX_PROBE=64` means the test hit the configured chain limit.
- Nonzero misses are expected once that artificial cluster saturates.
- The key result is stability: no parser error, no hang, final FSMs idle.

Hash-table Debug Registers

Problem. Early board runs could appear stuck with only the outer ingest/orderbook view visible. We needed to know whether the parser stopped, the dispatch FSM stopped, or the hash table itself was mid-operation.

Added MMIO visibility.

Register	What it exposes
`0x22 HT_DIAG`	hash FSM state, orderbook FSM state, op, msg type
`0x23 HT_STUCK_CYCLES`	live cycles spent in current non-idle HT state
`0x24–0x25 HT_CUR_REF`	order ref currently dispatched to the hash table
`0x26 HT_PROBE`	probe index and chain length
`0x27 HT_STUCK_MAX`	worst non-idle stall observed since reset/clear
`0x28 HT_DELETE_PTRS`	delete-shift hole and scan pointers

How we use them.

- If `HT_STUCK_CYCLES` climbs, the hash table is truly busy/stuck.
- If `HT_PROBE` advances, it is slow progress; if frozen, inspect the FSM state.
- Successful stress runs end with `hash_state=IDLE`, `ob_state=IDLE`, and `HT_STUCK cycles=0`.

Aged-Execute Test

Setup. –execute-policy aged –execute-min-age 5 –full-fill-exec-ratio 25 forces executes against orders that survived ≥ 5 replace/churn cycles, instead of the same-cycle adds the simpler tests hit.

Metric	Generator expected	Re-run observed
`HT_OPS`	15 597	15 597
`HT_MISS`	0	0
`HT_LOAD`	live older-order book	801

Takeaway. Executes against older live orders are now covered by a passing board test: no parser errors, no hash misses, expected op count, and the hash/orderbook FSMs return idle.

Appendix: `itch_load.c` Core Logic (1/2)

Opening the AXI-Lite window:

```
int fd = open("/dev/mem", O_RDWR | O_SYNC);
void *map = mmap(NULL, MAP_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd, FPGA_BASE);
volatile uint32_t *regs = (volatile uint32_t *)map;
```

Ping-pong buffer selection and write:

```
// Compute pointer to ping or pong region (BRAM starts at offset 512)
volatile uint32_t *buf_words = regs + 512;
if (use_buf1) buf_words += 8192 / 4;

// Wait for buffer free (READY=0 and ACTIVE=0)
while ((regs[flags_off] & 3u) != 0) { ... }

// Copy data into FPGA-side BRAM
for (size_t w = 0; w < chunk_size / 4; w++) {
    buf_words[w] = temp;
}

// Publish length, raise READY, pulse START (on first buffer)
regs[len_off] = (uint32_t)chunk_size;
regs[flags_off] = 1u; // bit 0 = ready
if (!started) { regs[W_CONTROL] = 0x1u; started = 1; }

// Wait for DONE (bit 2) then toggle buffer
while ((regs[flags_off] & 4u) == 0) { ... }
use_buf1 = !use_buf1;
```

Appendix: Performance Accounting (2/2)

Tracking pure FPGA throughput vs. Wall-clock time:

```
struct timespec t_total_start, t_total_end, c0, c1;
clock_gettime(CLOCK_MONOTONIC, &t_total_start);

// Inside the ingest loop:
clock_gettime(CLOCK_MONOTONIC, &c0); // Before READY=1
...
while ((regs[flags_off] & 4u) == 0) { ... } // Wait for DONE
clock_gettime(CLOCK_MONOTONIC, &c1); // After DONE observed

// Accumulate delta
accumulated_fpga_us += elapsed_us_between(&c0, &c1);
...
clock_gettime(CLOCK_MONOTONIC, &t_total_end);

// Calculation:
uint64_t elapsed_us = accumulated_fpga_us; // FPGA processing only
uint64_t wall_us = elapsed_us_between(&t_total_start, &t_total_end);
double msg_s = (double)msg_count * 1e6 / elapsed_us;
```

Note: `elapsed_us` measures the sum of hardware processing intervals, excluding host-side file I/O and pre-scanning overhead.

Appendix: Buffer Registers Deep Dive

W_CONTROL (0x00)

- [0] **start_pulse**: One-shot pulse to arm the ingest FSM. Asserted only for the first buffer.
- [4] **reset_counters**: Pulse to clear sticky diagnostics and msg counters.

W_INGEST_CFG (0x40)

- [0] **enable**: Allows FSM to process.
- [1] **continuous**: FPGA automatically toggles buffers. Enables "push-data-and-forget" flow.

W_INGEST_STAT (0x44)

- [1] **armed**: Ready for first buffer.
- [2] **busy**: Actively consuming data.
- [3] **active_buf**: 0 = Ping, 1 = Pong.
- [23:8] **left**: Words remaining in current chunk (useful for debugging hangs).

W_BUFn_FLAGS (0x50, 0x54)

Bit	Name	Meaning
0	READY	CPU sets when BRAM write is done.
1	ACTIVE	FPGA sets while reading.
2	DONE	FPGA sets when finished.
3	ERROR	Malformed ITCH detected.
20:5	CONS	Bytes consumed (diagnostics).

Note: The handshaking logic requires `READY` and `ACTIVE` to be 0 before the HPS can safely overwrite a buffer region.

Appendix: VGA Dashboard Core Logic

HPS: Mapping the character buffer

```
// Character buffer starts at word offset 0x2000 (0x8000 bytes)
volatile uint32_t *cbuf = regs + 0x2000;

static void vc_put(int row, int col, char c, uint32_t color) {
    // Pack {R, G, B, ASCII} into one 32-bit MMIO write
    // row*40 + col converts 2D tile coord to linear BRAM offset
    cbuf[row * 40 + col] = (color & 0xFFFFFF00u) | (uint8_t)c;
}
```

FPGA: Tiling and glyph lookup

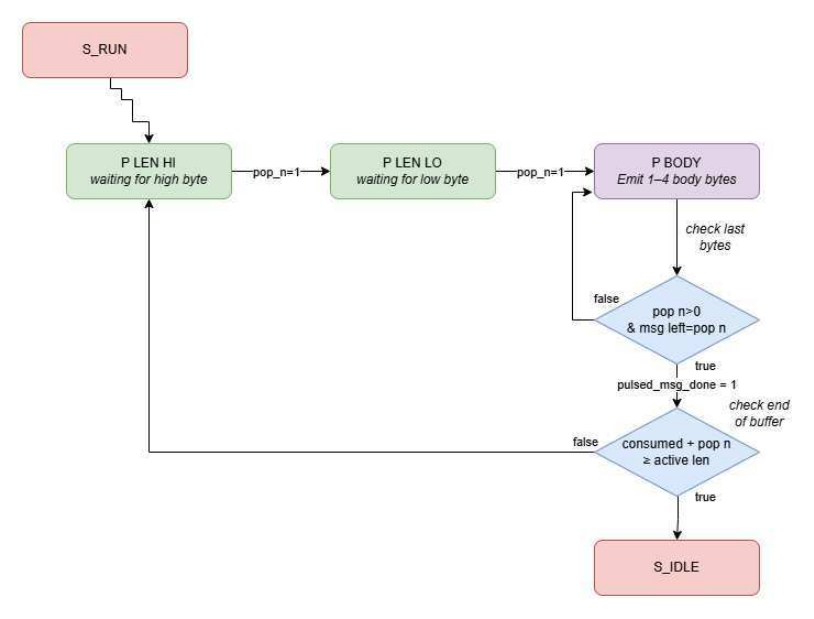
```
// 1. Tiling: convert pixel (x,y) to tile (col,row) by shifting right 4 (/16)
logic [5:0] char_col = pixel_x[9:4];
logic [4:0] char_row = pixel_y[8:4];

// 2. Linearize address for BRAM: addr = row*40 + col
assign char_addr = (char_row << 5) + (char_row << 3) + char_col;

// 3. Lookup ASCII byte in Glyph ROM (8x8 bitmap)
logic [7:0] font_bits = font_rom[{char_cell[7:0], sub_y}];
logic pixel_on = font_bits[sub_x];

// 4. Drive VGA signals
{VGA_R, VGA_G, VGA_B} = pixel_on ? char_cell[31:8] : 24'h000000;
```

Appendix: `buffered_ingest` — Parser FSM



Appendix: `buffered_ingest` — BRAM Fetch

```
/* ── BRAM fetch issue ──────────────────────────────── */
 * On fetch issue, use current fetch_word_idx, advance for next cycle,
 * and mark data as arriving next cycle.
 */
if (can_issue_fetch) begin
    fetch_word_idx  <= fetch_word_idx + 1'b1;
    words_addressed <= words_addressed + 1'b1;
    bram_rd_pending <= 1'b1;
end else begin
    bram_rd_pending <= 1'b0;
end

/* Update Counters */
consumed_reg <= consumed_reg + LENGTH_WIDTH'(pop_n);
if (parse_state == P_BODY && pop_n > 0) begin
    msg_left_reg <= msg_left_reg - LENGTH_WIDTH'(pop_n);
end
```

Appendix: `buffered_ingest` — Stage Update

```
/* — Stage update:
 * Pop pop_n bytes from stage,
 * then append 4 bytes if a BRAM word arrives. */
 * Stage layout: LSB = oldest. Pop = right-shift by pop_n*8.
 */
begin : stage_update
    /* Temporary variables for stage/count after pop and after optional BRAM fill. */
    automatic logic [63:0] s_after_pop;
    automatic logic [3:0]  c_after_pop;
    automatic logic [63:0] s_after_fill;
    automatic logic [3:0]  c_after_fill;

    /* Right-shift stage by pop_n bytes */
    s_after_pop = stage >> {pop_n, 3'b000};
    c_after_pop = stage_count - pop_n;

    if (bram_rd_pending) begin
        /* Insert 4 incoming bytes at byte position c_after_pop */
        s_after_fill = s_after_pop |
                       ({32'h0, mem_rd_data} << {c_after_pop, 3'b000});
        c_after_fill = c_after_pop + 4'd4;
    end else begin
        s_after_fill = s_after_pop;
        c_after_fill = c_after_pop;
    end

    stage       <= s_after_fill;
    stage_count <= c_after_fill;
end
```

C Code

C.1 Hardware

Listing 1: bbo_bitmap.sv: Bitmap-based best bid/offer selector.

```
module bbo_bitmap #(
    parameter int PRICE_BITS  = 13,
    parameter int GROUP_BITS  = 6,    // GROUP_BITS + WITHIN_BITS = PRICE_BITS
    parameter int WITHIN_BITS = 7,
    parameter int N_LEVELS    = 1 << PRICE_BITS,
    parameter bit IS_MAX      = 1'b1  // 1 = bid side, 0 = ask side
) (
    input  logic                    clk,
    input  logic                    rst,

    input  logic                    set_valid,
    input  logic                    set_active,
    input  logic [PRICE_BITS-1:0]   set_price,

    output logic                    best_valid,
    output logic [PRICE_BITS-1:0]   best_price
);

    localparam int N_GROUPS    = 1 << GROUP_BITS;
    localparam int GROUP_SIZE  = 1 << WITHIN_BITS;

    // Preserve the active-level bitmap in fabric so best-price selection
    // can be done as a wide parallel scan instead of a BRAM read sequence.
    (* preserve *) logic [N_LEVELS-1:0] active;

    always_ff @(posedge clk) begin
        if (rst) begin
            active <= '0;
        end else if (set_valid) begin
            active[set_price] <= set_active;
        end
    end

    logic [N_GROUPS-1:0]       group_any;
    logic [WITHIN_BITS-1:0]    group_idx [0:N_GROUPS-1];

    genvar g;
    generate
        for (g = 0; g < N_GROUPS; g++) begin : gen_stage1
            logic [GROUP_SIZE-1:0] grp;
            assign grp = active[g*GROUP_SIZE +: GROUP_SIZE];

            assign group_any[g] = |grp;

            always_comb begin
                group_idx[g] = '0;
                if (IS_MAX) begin
                    for (int i = GROUP_SIZE-1; i >= 0; i--) begin
                        if (grp[i]) begin
                            group_idx[g] = i[WITHIN_BITS-1:0];
                            break;
                        end
                    end
                end else begin
                    for (int i = 0; i < GROUP_SIZE; i++) begin
                        if (grp[i]) begin
                            group_idx[g] = i[WITHIN_BITS-1:0];
                            break;
                        end
                    end
                end
            end
        end
    endgenerate

    // Pipeline the group summaries to shorten the priority-encode path.
    logic [N_GROUPS-1:0]    group_any_r;
    logic [WITHIN_BITS-1:0] group_idx_r [0:N_GROUPS-1];

    always_ff @(posedge clk) begin
        if (rst) begin
            group_any_r <= '0;
            for (int g = 0; g < N_GROUPS; g++) group_idx_r[g] <= '0;
        end else begin
            group_any_r <= group_any;
            for (int g = 0; g < N_GROUPS; g++) group_idx_r[g] <= group_idx[g];
        end
    end

    logic [GROUP_BITS-1:0]  best_group;
```

```
    logic [WITHIN_BITS-1:0] best_within;

    always_comb begin
        best_group  = '0;
        best_within = '0;
        if (IS_MAX) begin
            for (int gg = N_GROUPS-1; gg >= 0; gg--) begin
                if (group_any_r[gg]) begin
                    best_group  = gg[GROUP_BITS-1:0];
                    best_within = group_idx_r[gg];
                    break;
                end
            end
        end else begin
            for (int gg = 0; gg < N_GROUPS; gg++) begin
                if (group_any_r[gg]) begin
                    best_group  = gg[GROUP_BITS-1:0];
                    best_within = group_idx_r[gg];
                    break;
                end
            end
        end
    end

    assign best_valid = (group_any_r != '0);
    assign best_price = {best_group, best_within};

endmodule
```

Listing 2: buffered_ingest.sv: Ping-pong buffered ITCH stream ingest.

```
/*
 * Ping-pong ITCH ingest from external BRAM.
 *
 * The module reads 32-bit BRAM words, stages bytes in an 8-byte buffer,
 * and emits 1-4 body bytes per cycle to the ITCH parser.
 *
 * msg_done is delayed one cycle after the final word_valid so parser buffer
 * writes settle before decode.
 */
module buffered_ingest #(
    parameter int HALF_BUFFER_BYTES = 8192,
    parameter int LENGTH_WIDTH      = 16
) (
    input  logic        clk,
    input  logic        reset,

    input  logic        start,
    input  logic        continuous_mode,
    input  logic        clear_status,
    input  logic        stall,

    input  logic [LENGTH_WIDTH-1:0] buf0_len_in,
    input  logic [LENGTH_WIDTH-1:0] buf1_len_in,

    input  logic        buf0_len_we,
    input  logic        buf1_len_we,

    input  logic        buf0_ready_set,
    input  logic        buf1_ready_set,
    input  logic        buf0_done_ack,
    input  logic        buf1_done_ack,

    output logic [$clog2(2 * HALF_BUFFER_BYTES)-1:0] mem_rd_addr,
    input  logic [31:0]                              mem_rd_data,

    output logic [7:0]  byte_in,
    output logic        byte_valid,

    output logic [31:0] word_out,
    output logic [2:0]  valid_bytes,
    output logic        word_valid,
    output logic        new_message,
    output logic        msg_done,

    output logic        busy,
    output logic        armed,
    output logic        active_buf,
    output logic [LENGTH_WIDTH-1:0] bytes_consumed,
    output logic [LENGTH_WIDTH-1:0] current_msg_len,
    output logic [LENGTH_WIDTH-1:0] current_msg_bytes_left,
    output logic [LENGTH_WIDTH-1:0] buf0_len,
    output logic [LENGTH_WIDTH-1:0] buf1_len,
    output logic        buf0_ready,
    output logic        buf1_ready,
    output logic        buf0_active,
    output logic        buf1_active,
    output logic        buf0_done,
```

```
    output logic        buf1_done,
    output logic        buf0_error,
    output logic        buf1_error,
    output logic [3:0]  error_code
);

    localparam int MEM_ADDR_W = $clog2(2 * HALF_BUFFER_BYTES);
    localparam int WORD_IDX_W = $clog2(HALF_BUFFER_BYTES / 4);

    localparam logic [3:0] ERR_NONE      = 4'd0;
    localparam logic [3:0] ERR_ZERO_LEN  = 4'd1;
    localparam logic [3:0] ERR_TRUNCATED = 4'd2;

    typedef enum logic [1:0] {
        S_IDLE,
        S_RUN
    } state_t;

    typedef enum logic [1:0] {
        P_LEN_HI,
        P_LEN_LO,
        P_BODY
    } parse_t;

    state_t state;
    parse_t parse_state;

    logic engine_armed;
    logic cur_buf_sel;

    logic [LENGTH_WIDTH-1:0] active_len;
    assign active_len = cur_buf_sel ? buf1_len : buf0_len;

    logic [WORD_IDX_W:0] total_words;
    assign total_words = (WORD_IDX_W+1)'((active_len + 16'd3) >> 2);

    // BRAM data is captured only for issued reads; repeated stale reads are ignored.
    logic [WORD_IDX_W-1:0] fetch_word_idx;
    logic                  bram_rd_pending;
    logic [WORD_IDX_W:0]   words_addressed;

    assign mem_rd_addr = cur_buf_sel
        ? (MEM_ADDR_W'(HALF_BUFFER_BYTES) + (MEM_ADDR_W'(fetch_word_idx) << 2))
        : (MEM_ADDR_W'(fetch_word_idx) << 2);

    logic [63:0] stage;
    logic [3:0]  stage_count;

    logic [LENGTH_WIDTH-1:0] msg_len_reg;
    logic [LENGTH_WIDTH-1:0] msg_left_reg;
    logic [LENGTH_WIDTH-1:0] consumed_reg;
    logic [3:0]              error_code_reg;
    logic                    pending_msg_done;

    logic [3:0] pop_n;
    logic       want_emit_word;
    logic [3:0] body_pop_max;

    always_comb begin
        pop_n          = 4'd0;
        want_emit_word = 1'b0;
        body_pop_max   = 4'd0;

        if (state == S_RUN) begin
            unique case (parse_state)
                P_LEN_HI: if (stage_count >= 4'd1) pop_n = 4'd1;
                P_LEN_LO: if (stage_count >= 4'd1) pop_n = 4'd1;
                P_BODY: begin
                    if (!stall && stage_count > 0 && msg_left_reg > 0) begin
                        body_pop_max = (msg_left_reg >= 16'd4)
                                       ? 4'd4
                                       : 4'(msg_left_reg);
                        if (4'(stage_count) < body_pop_max)
                            body_pop_max = stage_count;
                        pop_n          = body_pop_max;
                        want_emit_word = (body_pop_max != 4'd0);
                    end
                end
                default: ;
            endcase
        end
    end

    // Project stage occupancy before issuing the next BRAM read. This prevents
    // overflow even though the read data arrives one cycle after the request.
    logic [4:0] stage_count_proj;
    logic       can_issue_fetch;

    always_comb begin
        stage_count_proj = {1'b0, stage_count} +
                           (bram_rd_pending ? 5'd4 : 5'd0) -
                           {1'b0, pop_n};
```

```
    can_issue_fetch = (state == S_RUN) &&
                      (words_addressed < total_words) &&
                      (stage_count_proj <= 5'd4);
end

always_ff @(posedge clk) begin : main_fsm
    word_valid  <= 1'b0;
    new_message <= 1'b0;
    msg_done    <= 1'b0;
    byte_valid  <= 1'b0;
    byte_in     <= 8'h00;

    if (reset) begin
        state              <= S_IDLE;
        parse_state        <= P_LEN_HI;
        engine_armed       <= 1'b0;
        cur_buf_sel        <= 1'b0;
        fetch_word_idx     <= '0;
        words_addressed    <= '0;
        bram_rd_pending    <= 1'b0;
        stage              <= 64'h0;
        stage_count        <= 4'd0;
        buf0_len           <= '0;
        buf1_len           <= '0;
        buf0_ready         <= 1'b0;
        buf1_ready         <= 1'b0;
        buf0_active        <= 1'b0;
        buf1_active        <= 1'b0;
        buf0_done          <= 1'b0;
        buf1_done          <= 1'b0;
        buf0_error         <= 1'b0;
        buf1_error         <= 1'b0;
        error_code_reg     <= ERR_NONE;
        consumed_reg       <= '0;
        msg_len_reg        <= '0;
        msg_left_reg       <= '0;
        pending_msg_done   <= 1'b0;
        word_out           <= 32'h0;
        valid_bytes        <= 3'h0;
    end else if (clear_status) begin
        state              <= S_IDLE;
        parse_state        <= P_LEN_HI;
        engine_armed       <= 1'b0;
        fetch_word_idx     <= '0;
        words_addressed    <= '0;
        bram_rd_pending    <= 1'b0;
        stage              <= 64'h0;
        stage_count        <= 4'd0;
        buf0_ready         <= 1'b0;
        buf1_ready         <= 1'b0;
        buf0_active        <= 1'b0;
        buf1_active        <= 1'b0;
        buf0_done          <= 1'b0;
        buf1_done          <= 1'b0;
        buf0_error         <= 1'b0;
        buf1_error         <= 1'b0;
        error_code_reg     <= ERR_NONE;
        consumed_reg       <= '0;
        msg_len_reg        <= '0;
        msg_left_reg       <= '0;
        pending_msg_done   <= 1'b0;
        word_out           <= 32'h0;
        valid_bytes        <= 3'h0;
    end else begin
        if (buf0_len_we) buf0_len <= buf0_len_in;
        if (buf1_len_we) buf1_len <= buf1_len_in;

        if (buf0_ready_set) begin
            buf0_ready <= 1'b1;
            buf0_done  <= 1'b0;
            buf0_error <= 1'b0;
        end
        if (buf1_ready_set) begin
            buf1_ready <= 1'b1;
            buf1_done  <= 1'b0;
            buf1_error <= 1'b0;
        end

        if (buf0_done_ack && !buf0_active) begin
            buf0_done  <= 1'b0;
            buf0_error <= 1'b0;
        end
        if (buf1_done_ack && !buf1_active) begin
            buf1_done  <= 1'b0;
            buf1_error <= 1'b0;
        end

        if (start) engine_armed <= 1'b1;

        // Pop consumed bytes first, then append fresh BRAM data at the tail.
        begin : stage_update
            automatic logic [63:0] s_after_pop;
```

```
    automatic logic [3:0]  c_after_pop;
    automatic logic [63:0] s_after_fill;
    automatic logic [3:0]  c_after_fill;

    s_after_pop = stage >> {pop_n, 3'b000};
    c_after_pop = stage_count - pop_n;

    if (bram_rd_pending) begin
        s_after_fill = s_after_pop |
                       ({32'h0, mem_rd_data} << {c_after_pop, 3'b000});
        c_after_fill = c_after_pop + 4'd4;
    end else begin
        s_after_fill = s_after_pop;
        c_after_fill = c_after_pop;
    end

    stage       <= s_after_fill;
    stage_count <= c_after_fill;
end

if (can_issue_fetch) begin
    fetch_word_idx  <= fetch_word_idx + 1'b1;
    words_addressed <= words_addressed + 1'b1;
    bram_rd_pending <= 1'b1;
end else begin
    bram_rd_pending <= 1'b0;
end

consumed_reg <= consumed_reg + LENGTH_WIDTH'(pop_n);
if (parse_state == P_BODY && pop_n > 0) begin
    msg_left_reg <= msg_left_reg - LENGTH_WIDTH'(pop_n);
end

// Register body bytes with word_valid so the parser sees pre-shift data.
if (parse_state == P_BODY && want_emit_word) begin
    word_valid <= 1'b1;
    word_out   <= 32'h0;
    if (pop_n >= 4'd1) word_out[7:0]   <= stage[7:0];
    if (pop_n >= 4'd2) word_out[15:8]  <= stage[15:8];
    if (pop_n >= 4'd3) word_out[23:16] <= stage[23:16];
    if (pop_n >= 4'd4) word_out[31:24] <= stage[31:24];
    valid_bytes <= pop_n[2:0];
end

// Delay msg_done so the parser decodes after its byte writes settle.
if (pending_msg_done) begin
    msg_done         <= 1'b1;
    pending_msg_done <= 1'b0;
end

if (state == S_RUN) begin
    unique case (parse_state)
        P_LEN_HI: begin
            if (consumed_reg >= active_len && stage_count == 0) begin
                if (cur_buf_sel) begin
                    buf1_active <= 1'b0;
                    buf1_done   <= 1'b1;
                    buf1_ready  <= 1'b0;
                end else begin
                    buf0_active <= 1'b0;
                    buf0_done   <= 1'b1;
                    buf0_ready  <= 1'b0;
                end
                state       <= S_IDLE;
                parse_state <= P_LEN_HI;
            end else if (pop_n == 4'd1) begin
                msg_len_reg[15:8] <= stage[7:0];
                parse_state <= P_LEN_LO;
            end
        end

        P_LEN_LO: begin
            if (pop_n == 4'd1) begin
                automatic logic [LENGTH_WIDTH-1:0] full_len;
                full_len = {msg_len_reg[15:8], stage[7:0]};
                msg_len_reg[7:0] <= stage[7:0];

                if (full_len == 16'h0) begin
                    error_code_reg <= ERR_ZERO_LEN;
                    if (cur_buf_sel) begin
                        buf1_active <= 1'b0;
                        buf1_done   <= 1'b1;
                        buf1_ready  <= 1'b0;
                        buf1_error  <= 1'b1;
                    end else begin
                        buf0_active <= 1'b0;
                        buf0_done   <= 1'b1;
                        buf0_ready  <= 1'b0;
                        buf0_error  <= 1'b1;
                    end
                    state       <= S_IDLE;
                    parse_state <= P_LEN_HI;
```

```
                        end else if (consumed_reg + LENGTH_WIDTH'(pop_n) +
                                     full_len > active_len) begin
                            error_code_reg <= ERR_TRUNCATED;
                            if (cur_buf_sel) begin
                                buf1_active <= 1'b0;
                                buf1_done   <= 1'b1;
                                buf1_ready  <= 1'b0;
                                buf1_error  <= 1'b1;
                            end else begin
                                buf0_active <= 1'b0;
                                buf0_done   <= 1'b1;
                                buf0_ready  <= 1'b0;
                                buf0_error  <= 1'b1;
                            end
                            state       <= S_IDLE;
                            parse_state <= P_LEN_HI;
                        end else begin
                            msg_left_reg <= full_len;
                            new_message  <= 1'b1;
                            parse_state  <= P_BODY;
                        end
                    end
                end

                P_BODY: begin
                    if (pop_n > 0 && msg_left_reg == LENGTH_WIDTH'(pop_n)) begin
                        pending_msg_done <= 1'b1;

                        if (consumed_reg + LENGTH_WIDTH'(pop_n) >= active_len) begin
                            if (cur_buf_sel) begin
                                buf1_active <= 1'b0;
                                buf1_done   <= 1'b1;
                                buf1_ready  <= 1'b0;
                            end else begin
                                buf0_active <= 1'b0;
                                buf0_done   <= 1'b1;
                                buf0_ready  <= 1'b0;
                            end
                            state       <= S_IDLE;
                            parse_state <= P_LEN_HI;
                        end else begin
                            parse_state <= P_LEN_HI;
                        end
                    end
                end

                default: parse_state <= P_LEN_HI;
            endcase
        end

        if (state == S_IDLE) begin
            if (engine_armed && (buf0_ready || buf1_ready)) begin
                automatic logic next_buf_sel;

                if (buf0_ready && buf1_ready)      next_buf_sel = ~cur_buf_sel;
                else if (buf0_ready)               next_buf_sel =  1'b0;
                else                               next_buf_sel =  1'b1;

                cur_buf_sel <= next_buf_sel;

                if (next_buf_sel) begin
                    buf1_active <= 1'b1;
                    buf0_active <= 1'b0;
                end else begin
                    buf0_active <= 1'b1;
                    buf1_active <= 1'b0;
                end

                fetch_word_idx   <= '0;
                words_addressed  <= '0;
                bram_rd_pending  <= 1'b0;
                stage            <= 64'h0;
                stage_count      <= 4'd0;
                consumed_reg     <= '0;
                msg_len_reg      <= '0;
                msg_left_reg     <= '0;
                error_code_reg   <= ERR_NONE;
                pending_msg_done <= 1'b0;
                parse_state      <= P_LEN_HI;
                state            <= S_RUN;
            end else if (!continuous_mode && engine_armed) begin
                engine_armed <= 1'b0;
            end
        end
    end
end

assign busy                  = (state != S_IDLE);
assign armed                 = engine_armed;
assign active_buf            = cur_buf_sel;
assign bytes_consumed        = consumed_reg;
assign current_msg_len       = msg_len_reg;
```

```
    assign current_msg_bytes_left = msg_left_reg;
    assign error_code             = error_code_reg;

endmodule
```

Listing 3: hash_table.sv: Linear-probing order reference table.

```
/* verilator lint_off WIDTH */
/* verilator lint_off UNUSED */
module hash_table #(
    parameter int SLOT_BITS   = 14,
    parameter int NUM_SLOTS   = 1 << SLOT_BITS,
    parameter int MAX_CHAIN   = 64
) (
    input  logic        clk,
    input  logic        rst,

    input  logic        op_valid,
    input  logic [1:0]  op,
    input  logic [63:0] ref_in,
    input  logic [23:0] price_in,
    input  logic [23:0] qty_in,
    input  logic        side_in,
    input  logic [13:0] sym_in,
    output logic        ready,

    output logic        done,
    output logic        hit,
    output logic [23:0] price_out,
    output logic [23:0] qty_out,
    output logic        side_out,
    output logic [13:0] sym_out,

    output logic [SLOT_BITS:0] ht_load,
    output logic [7:0]         max_probe,
    output logic               overflow,

    // These debug outputs make a hardware stall diagnosable through MMIO.
    output logic [3:0]         dbg_state,
    output logic [1:0]         dbg_cur_op,
    output logic [63:0]        dbg_cur_ref,
    output logic [SLOT_BITS-1:0] dbg_probe_idx,
    output logic [SLOT_BITS-1:0] dbg_hole_idx,
    output logic [SLOT_BITS-1:0] dbg_scan_idx,
    output logic [7:0]         dbg_chain_len,
    output logic               dbg_rd_valid,
    output logic               dbg_ref_match,
    output logic               dbg_shift_needed
);

    localparam logic [1:0] OP_INSERT = 2'b00;
    localparam logic [1:0] OP_LOOKUP = 2'b01;
    localparam logic [1:0] OP_DELETE = 2'b10;

    localparam logic [SLOT_BITS-1:0] IDX_ONE  = 1;
    localparam logic [SLOT_BITS:0]   LOAD_ONE = 1;

    function automatic logic [127:0] pack_slot(
        input logic        v,
        input logic [63:0] r,
        input logic [23:0] p,
        input logic [23:0] q,
        input logic        s,
        input logic [13:0] sym
    );
        pack_slot = {sym, s, q, p, r, v};
    endfunction

    function automatic logic        unpack_valid(input logic [127:0] w); unpack_valid = w[0];        endfunction
    function automatic logic [63:0] unpack_ref  (input logic [127:0] w); unpack_ref   = w[64:1];     endfunction
    function automatic logic [23:0] unpack_price(input logic [127:0] w); unpack_price = w[88:65];    endfunction
    function automatic logic [23:0] unpack_qty  (input logic [127:0] w); unpack_qty   = w[112:89];   endfunction
    function automatic logic        unpack_side (input logic [127:0] w); unpack_side  = w[113];      endfunction
    function automatic logic [13:0] unpack_sym  (input logic [127:0] w); unpack_sym   = w[127:114];  endfunction

    function automatic logic [SLOT_BITS-1:0] hash(input logic [63:0] r);
        hash = r[13:0] ^ r[27:14] ^ r[41:28];
    endfunction

    logic [127:0]           mem        [NUM_SLOTS];
    logic [SLOT_BITS-1:0]   mem_addr;
    logic [127:0]           mem_wdata;
    logic                   mem_we;
    logic [127:0]           mem_rdata;

    always_ff @(posedge clk) begin
        if (mem_we) mem[mem_addr] <= mem_wdata;
        mem_rdata <= mem[mem_addr];
```

```
end

typedef enum logic [3:0] {
    S_INIT_CLEAR,
    S_IDLE,
    S_READ_ISSUE,
    S_READ_WAIT,
    S_COMPARE,
    S_DELETE_SHIFT_ISSUE,
    S_DELETE_SHIFT_WAIT,
    S_DELETE_SHIFT_DECIDE,
    S_DELETE_SHIFT_WRITE,
    S_DONE
} state_t;

state_t state;

logic [1:0]             cur_op;
logic [63:0]            cur_ref;
logic [23:0]            cur_price, cur_qty;
logic                   cur_side;
logic [13:0]            cur_sym;

logic [SLOT_BITS-1:0]   probe_idx;
logic [SLOT_BITS-1:0]   hole_idx;
logic [SLOT_BITS-1:0]   scan_idx;
logic [7:0]             chain_len;

logic [SLOT_BITS:0]     load_r;
logic [7:0]             max_probe_r;
logic [SLOT_BITS:0]     init_cnt;

logic        rd_valid;
logic [63:0] rd_ref;
logic        ref_match;

assign rd_valid  = unpack_valid(mem_rdata);
assign rd_ref    = unpack_ref(mem_rdata);
assign ref_match = rd_valid && (rd_ref == cur_ref);

logic [SLOT_BITS-1:0] rd_hash;
logic [SLOT_BITS-1:0] dist_h, dist_j;
logic                 shift_needed;

assign rd_hash      = hash(rd_ref);
assign dist_h       = hole_idx - rd_hash;
assign dist_j       = hole_idx - scan_idx;
assign shift_needed = rd_valid && (dist_h < dist_j);

assign ready     = (state == S_IDLE);
assign ht_load   = load_r;
assign max_probe = max_probe_r;

assign dbg_state        = state;
assign dbg_cur_op       = cur_op;
assign dbg_cur_ref      = cur_ref;
assign dbg_probe_idx    = probe_idx;
assign dbg_hole_idx     = hole_idx;
assign dbg_scan_idx     = scan_idx;
assign dbg_chain_len    = chain_len;
assign dbg_rd_valid     = rd_valid;
assign dbg_ref_match    = ref_match;
assign dbg_shift_needed = shift_needed;

always_ff @(posedge clk) begin
    if (rst) begin
        state        <= S_INIT_CLEAR;
        load_r       <= '0;
        max_probe_r  <= '0;
        done         <= 1'b0;
        hit          <= 1'b0;
        overflow     <= 1'b0;
        mem_we       <= 1'b0;
        init_cnt     <= '0;
    end else begin
        done     <= 1'b0;
        overflow <= 1'b0;
        mem_we   <= 1'b0;

        case (state)

        // Clear table contents in hardware instead of relying on reset loops.
        S_INIT_CLEAR: begin
            mem_addr  <= init_cnt[SLOT_BITS-1:0];
            mem_wdata <= '0;
            mem_we    <= 1'b1;
            init_cnt  <= init_cnt + LOAD_ONE;
            if (init_cnt == NUM_SLOTS - 1) state <= S_IDLE;
        end

        S_IDLE: begin
            if (op_valid) begin
```

```
            cur_op    <= op;
            cur_ref   <= ref_in;
            cur_price <= price_in;
            cur_qty   <= qty_in;
            cur_side  <= side_in;
            cur_sym   <= sym_in;
            probe_idx <= hash(ref_in);
            chain_len <= 8'd1;
            mem_addr  <= hash(ref_in);
            state     <= S_READ_WAIT;
        end
    end

    S_READ_ISSUE: begin
        mem_addr <= probe_idx;
        state    <= S_READ_WAIT;
    end

    S_READ_WAIT: state <= S_COMPARE;

    S_COMPARE: begin
        if (chain_len > max_probe_r) max_probe_r <= chain_len;

        case (cur_op)
        OP_INSERT: begin
            if (ref_match) begin
                mem_addr  <= probe_idx;
                mem_wdata <= pack_slot(1'b1, cur_ref, cur_price,
                                       cur_qty, cur_side, cur_sym);
                mem_we    <= 1'b1;
                hit       <= 1'b1;
                state     <= S_DONE;
            end else if (!rd_valid) begin
                mem_addr  <= probe_idx;
                mem_wdata <= pack_slot(1'b1, cur_ref, cur_price,
                                       cur_qty, cur_side, cur_sym);
                mem_we    <= 1'b1;
                load_r    <= load_r + LOAD_ONE;
                hit       <= 1'b1;
                state     <= S_DONE;
            end else if (chain_len >= MAX_CHAIN) begin
                overflow <= 1'b1;
                hit      <= 1'b0;
                state    <= S_DONE;
            end else begin
                probe_idx <= probe_idx + IDX_ONE;
                chain_len <= chain_len + 8'd1;
                state     <= S_READ_ISSUE;
            end
        end

        OP_LOOKUP: begin
            if (ref_match) begin
                price_out <= unpack_price(mem_rdata);
                qty_out   <= unpack_qty(mem_rdata);
                side_out  <= unpack_side(mem_rdata);
                sym_out   <= unpack_sym(mem_rdata);
                hit       <= 1'b1;
                state     <= S_DONE;
            end else if (!rd_valid || chain_len >= MAX_CHAIN) begin
                hit   <= 1'b0;
                state <= S_DONE;
            end else begin
                probe_idx <= probe_idx + IDX_ONE;
                chain_len <= chain_len + 8'd1;
                state     <= S_READ_ISSUE;
            end
        end

        OP_DELETE: begin
            if (ref_match) begin
                price_out <= unpack_price(mem_rdata);
                qty_out   <= unpack_qty(mem_rdata);
                side_out  <= unpack_side(mem_rdata);
                sym_out   <= unpack_sym(mem_rdata);
                mem_addr  <= probe_idx;
                mem_wdata <= '0;
                mem_we    <= 1'b1;
                load_r    <= load_r - LOAD_ONE;
                hit       <= 1'b1;
                hole_idx  <= probe_idx;
                scan_idx  <= probe_idx + IDX_ONE;
                chain_len <= 8'd1;
                state     <= S_DELETE_SHIFT_ISSUE;
            end else if (!rd_valid || chain_len >= MAX_CHAIN) begin
                hit   <= 1'b0;
                state <= S_DONE;
            end else begin
                probe_idx <= probe_idx + IDX_ONE;
                chain_len <= chain_len + 8'd1;
                state     <= S_READ_ISSUE;
            end
```

```
                end

                default: state <= S_DONE;
                endcase
            end

            S_DELETE_SHIFT_ISSUE: begin
                mem_addr <= scan_idx;
                state    <= S_DELETE_SHIFT_WAIT;
            end

            S_DELETE_SHIFT_WAIT: state <= S_DELETE_SHIFT_DECIDE;

            S_DELETE_SHIFT_DECIDE: begin
                if (!rd_valid) begin
                    state <= S_DONE;
                end else if (shift_needed) begin
                    mem_addr  <= hole_idx;
                    mem_wdata <= mem_rdata;
                    mem_we    <= 1'b1;
                    state     <= S_DELETE_SHIFT_WRITE;
                end else begin
                    scan_idx  <= scan_idx + IDX_ONE;
                    chain_len <= chain_len + 8'd1;
                    if (chain_len >= MAX_CHAIN) state <= S_DONE;
                    else                        state <= S_DELETE_SHIFT_ISSUE;
                end
            end

            S_DELETE_SHIFT_WRITE: begin
                mem_addr  <= scan_idx;
                mem_wdata <= '0;
                mem_we    <= 1'b1;
                hole_idx  <= scan_idx;
                scan_idx  <= scan_idx + IDX_ONE;
                chain_len <= chain_len + 8'd1;
                if (chain_len >= MAX_CHAIN) state <= S_DONE;
                else                        state <= S_DELETE_SHIFT_ISSUE;
            end

            S_DONE: begin
                done  <= 1'b1;
                state <= S_IDLE;
            end

            default: state <= S_IDLE;
            endcase
        end
    end

endmodule
/* verilator lint_on UNUSED */
/* verilator lint_on WIDTH */
```

Listing 4: itch_parser.sv: ITCH message decoder for the FPGA pipeline.

```
/*
 * NASDAQ ITCH 5.0 message parser.
 *
 * msg_done is asserted one cycle after the final word_valid, so the parser
 * decodes only after all nonblocking writes into msg_buf have settled.
 */
module itch_parser (
    input  logic        clk,
    input  logic        reset,

    input  logic [7:0]  byte_in,
    input  logic        byte_valid,

    input  logic [31:0] word_in,
    input  logic [2:0]  valid_bytes,
    input  logic        word_valid,
    input  logic        new_message,
    input  logic        msg_done,
    input  logic        msg_error,

    output logic        msg_valid,
    output logic [7:0]  msg_type,
    output logic [15:0] stock_locate,
    output logic [63:0] order_ref,
    output logic [31:0] price,
    output logic [31:0] shares,
    output logic        side,
    output logic [63:0] new_order_ref,
    output logic [31:0] msg_count
);

    // ITCH 5.0 messages fit within 64 bytes, which leaves safe write headroom.
```

```
localparam int MSG_BUF_BYTES = 64;
localparam int BYTE_CNT_W    = $clog2(MSG_BUF_BYTES);

logic [7:0]            msg_buf [0:MSG_BUF_BYTES-1];
logic [BYTE_CNT_W-1:0] byte_count;

always_ff @(posedge clk) begin
    msg_valid <= 1'b0;

    if (reset) begin
        byte_count    <= '0;
        msg_count     <= 32'h0;
        msg_type      <= 8'h0;
        stock_locate  <= 16'h0;
        order_ref     <= 64'h0;
        price         <= 32'h0;
        shares        <= 32'h0;
        side          <= 1'b0;
        new_order_ref <= 64'h0;
        for (int i = 0; i < MSG_BUF_BYTES; i++) msg_buf[i] <= 8'h0;
    end else begin
        automatic logic [BYTE_CNT_W-1:0] write_pos;
        write_pos = new_message ? '0 : byte_count;

        if (new_message) begin
            byte_count <= '0;
        end

        if (word_valid) begin
            if (valid_bytes >= 3'd1 && {2'b0, write_pos} + 6'd0 < MSG_BUF_BYTES)
                msg_buf[write_pos + 0] <= word_in[7:0];
            if (valid_bytes >= 3'd2 && {2'b0, write_pos} + 6'd1 < MSG_BUF_BYTES)
                msg_buf[write_pos + 1] <= word_in[15:8];
            if (valid_bytes >= 3'd3 && {2'b0, write_pos} + 6'd2 < MSG_BUF_BYTES)
                msg_buf[write_pos + 2] <= word_in[23:16];
            if (valid_bytes >= 3'd4 && {2'b0, write_pos} + 6'd3 < MSG_BUF_BYTES)
                msg_buf[write_pos + 3] <= word_in[31:24];

            byte_count <= write_pos + BYTE_CNT_W'(valid_bytes);
        end

        if (msg_done) begin
            msg_type     <= msg_buf[0];
            stock_locate <= {msg_buf[1], msg_buf[2]};

            unique case (msg_buf[0])
                8'h41, 8'h46: begin
                    order_ref <= {msg_buf[11], msg_buf[12], msg_buf[13], msg_buf[14],
                                  msg_buf[15], msg_buf[16], msg_buf[17], msg_buf[18]};
                    side      <= (msg_buf[19] == 8'h53);
                    shares    <= {msg_buf[20], msg_buf[21], msg_buf[22], msg_buf[23]};
                    price     <= {msg_buf[32], msg_buf[33], msg_buf[34], msg_buf[35]};
                    msg_valid <= 1'b1;
                    msg_count <= msg_count + 1'b1;
                end

                8'h44: begin
                    order_ref <= {msg_buf[11], msg_buf[12], msg_buf[13], msg_buf[14],
                                  msg_buf[15], msg_buf[16], msg_buf[17], msg_buf[18]};
                    msg_valid <= 1'b1;
                    msg_count <= msg_count + 1'b1;
                end

                8'h55: begin
                    order_ref     <= {msg_buf[11], msg_buf[12], msg_buf[13], msg_buf[14],
                                      msg_buf[15], msg_buf[16], msg_buf[17], msg_buf[18]};
                    new_order_ref <= {msg_buf[19], msg_buf[20], msg_buf[21], msg_buf[22],
                                      msg_buf[23], msg_buf[24], msg_buf[25], msg_buf[26]};
                    shares        <= {msg_buf[27], msg_buf[28], msg_buf[29], msg_buf[30]};
                    price         <= {msg_buf[31], msg_buf[32], msg_buf[33], msg_buf[34]};
                    msg_valid     <= 1'b1;
                    msg_count     <= msg_count + 1'b1;
                end

                8'h45, 8'h58: begin
                    order_ref <= {msg_buf[11], msg_buf[12], msg_buf[13], msg_buf[14],
                                  msg_buf[15], msg_buf[16], msg_buf[17], msg_buf[18]};
                    shares    <= {msg_buf[19], msg_buf[20], msg_buf[21], msg_buf[22]};
                    msg_valid <= 1'b1;
                    msg_count <= msg_count + 1'b1;
                end

                8'h43: begin
                    order_ref <= {msg_buf[11], msg_buf[12], msg_buf[13], msg_buf[14],
                                  msg_buf[15], msg_buf[16], msg_buf[17], msg_buf[18]};
                    shares    <= {msg_buf[19], msg_buf[20], msg_buf[21], msg_buf[22]};
                    price     <= {msg_buf[32], msg_buf[33], msg_buf[34], msg_buf[35]};
                    msg_valid <= 1'b1;
                    msg_count <= msg_count + 1'b1;
                end

                default: begin end
```

```
                endcase
            end
        end
    end

endmodule
```

Listing 5: pla.sv: Per-price-level aggregate quantity memory.

```
module pla #(
    parameter int PRICE_BITS   = 13,
    parameter int QTY_BITS     = 24,
    parameter int PRICE_LEVELS = 1 << PRICE_BITS
) (
    input  logic                        clk,
    input  logic                        rst,

    input  logic                        update_valid,
    output logic                        update_ready,
    input  logic [PRICE_BITS-1:0]       update_price,
    input  logic signed [QTY_BITS:0]    update_delta,
    output logic                        done,

    output logic                        set_valid,
    output logic                        set_active,
    output logic [PRICE_BITS-1:0]       set_price,

    input  logic                        query_valid,
    input  logic [PRICE_BITS-1:0]       query_price,
    output logic                        query_qty_valid,
    output logic [QTY_BITS-1:0]         query_qty,

    output logic                        error_underflow,
    output logic                        error_overflow,
    input  logic                        clear_errors
);

    // Request block RAM inference for the price-level quantity table.
    (* ramstyle = "M10K" *)
    logic [QTY_BITS-1:0] mem [0:PRICE_LEVELS-1];

    logic [PRICE_BITS-1:0] addr_a;
    logic [QTY_BITS-1:0]   din_a;
    logic                  we_a;
    logic [QTY_BITS-1:0]   dout_a;

    logic [PRICE_BITS-1:0] addr_b;
    logic [QTY_BITS-1:0]   dout_b;

    always_ff @(posedge clk) begin
        if (we_a) mem[addr_a] <= din_a;
        dout_a <= mem[addr_a];
    end

    always_ff @(posedge clk) begin
        dout_b <= mem[addr_b];
    end

    typedef enum logic [1:0] {
        S_IDLE,
        S_READ_PEND,
        S_DONE
    } state_t;

    state_t state;

    logic [PRICE_BITS-1:0]    saved_price;
    logic signed [QTY_BITS:0] saved_delta;

    assign update_ready = (state == S_IDLE);

    // Extend by one extra bit so signed arithmetic can detect underflow
    // and overflow before the value is written back into the quantity RAM.
    logic signed [QTY_BITS+1:0] sum_ext;
    logic                       udf, ovf;
    logic [QTY_BITS-1:0]        new_qty;

    always_comb begin
        sum_ext = $signed({2'b00, dout_a}) +
                  $signed({saved_delta[QTY_BITS], saved_delta});
        udf     = sum_ext[QTY_BITS+1];
        ovf     = ~sum_ext[QTY_BITS+1] & sum_ext[QTY_BITS];
        new_qty = sum_ext[QTY_BITS-1:0];
    end

    always_comb begin
        addr_a = saved_price;
        din_a  = '0;
```

```
        we_a   = 1'b0;

        case (state)
        S_IDLE: begin
            if (update_valid) addr_a = update_price;
        end

        S_READ_PEND: begin
            if (!udf && !ovf) begin
                din_a = new_qty;
                we_a  = 1'b1;
            end
        end

        default: ;
        endcase
    end

    assign addr_b    = query_price;
    assign query_qty = dout_b;

    always_ff @(posedge clk) begin
        if (rst) query_qty_valid <= 1'b0;
        else     query_qty_valid <= query_valid;
    end

    always_ff @(posedge clk) begin
        if (rst) begin
            state           <= S_IDLE;
            saved_price     <= '0;
            saved_delta     <= '0;
            done            <= 1'b0;
            set_valid       <= 1'b0;
            set_active      <= 1'b0;
            set_price       <= '0;
            error_underflow <= 1'b0;
            error_overflow  <= 1'b0;
        end else begin
            done      <= 1'b0;
            set_valid <= 1'b0;

            if (clear_errors) begin
                error_underflow <= 1'b0;
                error_overflow  <= 1'b0;
            end

            case (state)
            S_IDLE: begin
                if (update_valid) begin
                    saved_price <= update_price;
                    saved_delta <= update_delta;
                    state       <= S_READ_PEND;
                end
            end

            S_READ_PEND: begin
                // The writeback and bitmap update occur only for valid arithmetic.
                if (udf) begin
                    error_underflow <= 1'b1;
                end else if (ovf) begin
                    error_overflow <= 1'b1;
                end else begin
                    set_valid  <= 1'b1;
                    set_price  <= saved_price;
                    set_active <= (new_qty != '0);
                end
                state <= S_DONE;
            end

            S_DONE: begin
                done  <= 1'b1;
                state <= S_IDLE;
            end

            default: state <= S_IDLE;
            endcase
        end
    end

endmodule
```

Listing 6: vga_display.sv: Avalon-MM character-mode VGA display peripheral.

```
/*
 * Avalon-MM backed character-mode VGA controller.
 *
 * HPS writes each character cell as {R[7:0], G[7:0], B[7:0], ASCII[7:0]}.
 * The controller renders a 40 x 30 text grid over 640 x 480 VGA.
```

```
 */
module vga_display(
    input  logic        clk,
    input  logic        reset,

    input  logic        chipselect,
    input  logic        write,
    input  logic [10:0] address,
    input  logic [31:0] writedata,

    output logic [7:0]  VGA_R, VGA_G, VGA_B,
    output logic        VGA_CLK, VGA_HS, VGA_VS,
                        VGA_BLANK_n, VGA_SYNC_n);

    logic [10:0] hcount;
    logic  [9:0] vcount;

    vga_counters counters(.clk50(clk), .*);

    // Character cells are memory-mapped so software can update the screen
    // without needing to know VGA timing.
    logic [31:0] char_buf [0:1199];

    always_ff @(posedge clk)
        if (chipselect && write && address <= 11'd1199)
            char_buf[address] <= writedata;

    // Font ROM is kept in hardware so software writes only ASCII/color cells.
    logic [7:0] font_rom [0:2047];
    initial $readmemh("rtl/font8x8.hex", font_rom);

    logic [9:0] pixel_x;
    logic [8:0] pixel_y;

    assign pixel_x = hcount[10:1];
    assign pixel_y = vcount[8:0];

    logic [5:0] char_col;
    logic [4:0] char_row;

    assign char_col = pixel_x[9:4];
    assign char_row = pixel_y[8:4];

    logic [2:0] sub_x;
    logic [2:0] sub_y;

    assign sub_x = pixel_x[3:1];
    assign sub_y = pixel_y[3:1];

    logic [10:0] char_addr;
    logic [31:0] char_cell;
    logic  [7:0] font_row_bits;
    logic        pixel_on;

    assign char_addr    = ({6'b0, char_row} << 5)
                        + ({6'b0, char_row} << 3)
                        + {5'b0, char_col};

    assign char_cell     = char_buf[char_addr];
    assign font_row_bits = font_rom[{char_cell[7:0], sub_y}];
    assign pixel_on      = font_row_bits[sub_x];

    always_comb begin
        {VGA_R, VGA_G, VGA_B} = 24'h000000;
        if (VGA_BLANK_n)
            if (pixel_on)
                {VGA_R, VGA_G, VGA_B} = {
                    char_cell[31:24],
                    char_cell[23:16],
                    char_cell[15:8]
                };
    end

endmodule


module vga_counters(
    input  logic        clk50,
    input  logic        reset,
    output logic [10:0] hcount,
    output logic  [9:0] vcount,
    output logic        VGA_CLK,
    output logic        VGA_HS,
    output logic        VGA_VS,
    output logic        VGA_BLANK_n,
    output logic        VGA_SYNC_n
);

    // 640 x 480 @ 60 Hz using a 50 MHz input clock and a 25 MHz pixel clock.
    parameter HACTIVE      = 11'd 1280,
              HFRONT_PORCH = 11'd   32,
              HSYNC        = 11'd  192,
```

```
            HBACK_PORCH  = 11'd   96,
            HTOTAL = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH;

    parameter VACTIVE      = 10'd 480,
            VFRONT_PORCH = 10'd  10,
            VSYNC        = 10'd   2,
            VBACK_PORCH  = 10'd  33,
            VTOTAL = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH;

    logic endOfLine;

    always_ff @(posedge clk50 or posedge reset)
        if (reset)          hcount <= 0;
        else if (endOfLine) hcount <= 0;
        else                hcount <= hcount + 11'd1;

    assign endOfLine = hcount == HTOTAL - 1;

    logic endOfField;

    always_ff @(posedge clk50 or posedge reset)
        if (reset)          vcount <= 0;
        else if (endOfLine)
            if (endOfField) vcount <= 0;
            else            vcount <= vcount + 10'd1;

    assign endOfField = vcount == VTOTAL - 1;

    assign VGA_HS = !( (hcount[10:8] == 3'b101) & !(hcount[7:5] == 3'b111));
    assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2 );

    assign VGA_SYNC_n = 1'b0;

    assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                         !( vcount[9]  | (vcount[8:5] == 4'b1111)  );

    assign VGA_CLK = hcount[0];

endmodule
```

Listing 7: orderbook_core.sv: Top-level Avalon-MM order book peripheral.

```
/*
 * ITCH 5.0 order book core, exposed as an Avalon-MM peripheral.
 *
 * HPS writes framed ITCH bytes into itch_mem, then starts the fabric ingest
 * path. The hardware path is:
 *
 *   buffered_ingest -> itch_parser -> hash_table -> PLA -> BBO bitmap
 */
module orderbook_core(
    input  logic         ref_clk,
    output logic         clk_out,

    input  logic         clk,
    input  logic         reset,

    input  logic [12:0]  address,
    input  logic [31:0]  writedata,
    input  logic [3:0]   byteenable,
    output logic [31:0]  readdata,
    input  logic         write,
    input  logic         read,
    input  logic         chipselect,

    output logic [6:0]   HEX0,
    output logic [6:0]   HEX1,
    output logic [6:0]   HEX2,
    output logic [6:0]   HEX3,
    output logic [6:0]   HEX4,
    output logic [6:0]   HEX5
);

    localparam int ITCH_TOTAL_BYTES = 16384;
    localparam int HALF_BYTES       = ITCH_TOTAL_BYTES / 2;
    localparam int BUF_BASE_WORD    = 512;
    localparam int ITCH_WORDS       = ITCH_TOTAL_BYTES / 4;
    localparam int LAST_BUF_WORD    = BUF_BASE_WORD + ITCH_WORDS - 1;

    localparam logic [12:0] ADDR_INGEST_CFG  = 13'd16;
    localparam logic [12:0] ADDR_INGEST_STAT = 13'd17;
    localparam logic [12:0] ADDR_BUF0_LEN    = 13'd18;
    localparam logic [12:0] ADDR_BUF1_LEN    = 13'd19;
    localparam logic [12:0] ADDR_BUF0_FLAGS  = 13'd20;
    localparam logic [12:0] ADDR_BUF1_FLAGS  = 13'd21;
    localparam logic [12:0] ADDR_PARSER_STAT = 13'd22;
    localparam logic [12:0] ADDR_PLA_CTRL    = 13'h40;
    localparam logic [12:0] ADDR_BBO_BASE    = 13'h48;
```

```
logic [31:0] reg_control;
logic [31:0] reg_scratch;
logic [31:0] reg_ingest_cfg;
logic [31:0] err_count;
logic        parse_err_sticky;
logic        run_done_sticky;

logic [15:0] reg_filter_id [0:3];
logic [4:0]  reg_filter_ctrl;
logic [23:0] reg_price_base [0:3];

logic        stock_match;
logic        stock_match_specific;
logic [1:0]  stock_id_match;

assign stock_match_specific =
    (reg_filter_ctrl[0] && (parsed_stock_locate == reg_filter_id[0])) ||
    (reg_filter_ctrl[1] && (parsed_stock_locate == reg_filter_id[1])) ||
    (reg_filter_ctrl[2] && (parsed_stock_locate == reg_filter_id[2])) ||
    (reg_filter_ctrl[3] && (parsed_stock_locate == reg_filter_id[3]));

assign stock_match = reg_filter_ctrl[4] || stock_match_specific;

always_comb begin
    if      (reg_filter_ctrl[0] && (parsed_stock_locate == reg_filter_id[0])) stock_id_match = 2'd0;
    else if (reg_filter_ctrl[1] && (parsed_stock_locate == reg_filter_id[1])) stock_id_match = 2'd1;
    else if (reg_filter_ctrl[2] && (parsed_stock_locate == reg_filter_id[2])) stock_id_match = 2'd2;
    else if (reg_filter_ctrl[3] && (parsed_stock_locate == reg_filter_id[3])) stock_id_match = 2'd3;
    else                                                                      stock_id_match = 2'd0;
end

localparam logic [1:0] HT_OP_INSERT = 2'b00;
localparam logic [1:0] HT_OP_LOOKUP = 2'b01;
localparam logic [1:0] HT_OP_DELETE = 2'b10;

typedef enum logic [3:0] {
    HT_IDLE,
    HT_WAIT,
    PLA_ISSUE,
    PLA_WAIT,
    HT_REPLACE_INSERT,
    HT_WAIT2,
    PLA_ISSUE2,
    PLA_WAIT2,
    HT_EXEC_OP,
    HT_EXEC_WAIT
} ht_dispatch_state_t;

ht_dispatch_state_t ht_state;

logic        ht_op_valid;
logic [1:0]  ht_op;
logic [63:0] ht_ref_in;
logic [23:0] ht_price_in;
logic [23:0] ht_qty_in;
logic        ht_side_in;
logic [13:0] ht_sym_in;
logic        ht_ready;
logic        ht_done;
logic        ht_hit;
logic [23:0] ht_price_out;
logic [23:0] ht_qty_out;
logic        ht_side_out;
logic [13:0] ht_sym_out;

logic [7:0]  ht_msg_type;
logic [63:0] ht_order_ref;
logic [63:0] ht_new_order_ref;
logic [31:0] ht_price_raw;
logic [31:0] ht_shares_raw;
logic        ht_side_raw;
logic        ht_side_orig;
logic        ht_orig_hit;
logic [23:0] ht_lookup_price;
logic [13:0] ht_lookup_sym;
logic [23:0] ht_exec_new_qty;
logic        ht_exec_full;
logic [15:0] ht_stock_locate;
logic [1:0]  ht_stock_id;
logic        ht_pla_eligible;
logic        ht_pending;

// Stall the byte stream while a hash-table operation is pending or active.
logic ht_stall;
assign ht_stall = ht_pending || (ht_state != HT_IDLE);

// ITCH prices are in $0.0001 units; /100 gives 1-cent buckets.
logic [23:0] price_rel_raw;
logic [23:0] price_rel_out;
logic [23:0] price_bucket_raw;
logic [23:0] price_bucket_out;
```

```
logic        bucket_raw_in_window;
logic        bucket_out_in_window;

assign price_rel_raw    = ht_price_raw[23:0] - reg_price_base[ht_stock_id];
assign price_rel_out    = ht_price_out       - reg_price_base[ht_stock_id];
assign price_bucket_raw = price_rel_raw / 24'd100;
assign price_bucket_out = price_rel_out / 24'd100;

// Below-window prices unsigned-wrap, so upper-bits-zero catches both sides.
assign bucket_raw_in_window = (price_bucket_raw[23:10] == 14'h0);
assign bucket_out_in_window = (price_bucket_out[23:10] == 14'h0);

logic [31:0] oow_count [0:3];

logic [31:0] ht_ops_total;
logic [31:0] ht_miss_count;
logic        ht_last_hit;
logic [1:0]  ht_last_op;
logic [23:0] ht_last_price_out;
logic [7:0]  ht_last_qty_out;
logic [31:0] ht_stuck_cycles;
logic [31:0] ht_stuck_max;

logic [3:0]  ht_dbg_state;
logic [1:0]  ht_dbg_cur_op;
logic [63:0] ht_dbg_cur_ref;
logic [13:0] ht_dbg_probe_idx;
logic [13:0] ht_dbg_hole_idx;
logic [13:0] ht_dbg_scan_idx;
logic [7:0]  ht_dbg_chain_len;
logic        ht_dbg_rd_valid;
logic        ht_dbg_ref_match;
logic        ht_dbg_shift_needed;

logic in_reg_region, in_buf_region;
logic [11:0] buf_word_addr;
logic [31:0] read_buf_word_reg;
logic [31:0] ingest_rd_word;

assign in_reg_region = (address < BUF_BASE_WORD);
assign in_buf_region = (address >= BUF_BASE_WORD) && (address <= LAST_BUF_WORD);
assign buf_word_addr = 12'(address - BUF_BASE_WORD);

wire mem_wren = chipselect && write && in_buf_region;

// Dual-port ITCH staging memory: HPS writes port A, fabric reads port B.
altsyncram #(
    .operation_mode("BIDIR_DUAL_PORT"),
    .width_a(32),
    .widthad_a(12),
    .numwords_a(4096),
    .width_b(32),
    .widthad_b(12),
    .numwords_b(4096),
    .width_byteena_a(4),
    .width_byteena_b(1),
    .outdata_reg_a("UNREGISTERED"),
    .outdata_reg_b("UNREGISTERED"),
    .ram_block_type("M10K"),
    .byte_size(8),
    .read_during_write_mode_mixed_ports("DONT_CARE")
) itch_mem_inst (
    .clock0    (clk),
    .clock1    (clk),
    .address_a (buf_word_addr),
    .data_a    (writedata),
    .wren_a    (mem_wren),
    .byteena_a (byteenable),
    .q_a       (read_buf_word_reg),
    .address_b (ingest_mem_rd_addr[13:2]),
    .data_b    (32'h0),
    .wren_b    (1'b0),
    .byteena_b (1'b1),
    .q_b       (ingest_rd_word)
);

assign ingest_mem_rd_data = ingest_rd_word;

always_ff @(posedge clk) begin
    if (reset) begin
        reg_control      <= 32'h0;
        reg_scratch      <= 32'h0;
        reg_ingest_cfg   <= 32'h1;
        reg_filter_ctrl  <= 5'h10;
        reg_filter_id[0] <= 16'h0;
        reg_filter_id[1] <= 16'h0;
        reg_filter_id[2] <= 16'h0;
        reg_filter_id[3] <= 16'h0;
        reg_price_base[0] <= 24'h0;
        reg_price_base[1] <= 24'h0;
        reg_price_base[2] <= 24'h0;
        reg_price_base[3] <= 24'h0;
```

```
        err_count        <= 32'h0;
        parse_err_sticky <= 1'b0;
        run_done_sticky  <= 1'b0;
    end else begin
        if (chipselect && write && address == 13'h00 && writedata[4]) begin
            err_count        <= 32'h0;
            parse_err_sticky <= 1'b0;
            run_done_sticky  <= 1'b0;
        end

        if (chipselect && write) begin
            case (address)
                13'h00: reg_control       <= writedata;
                13'h03: reg_scratch       <= writedata;
                13'h06: reg_filter_id[0]  <= writedata[15:0];
                13'h07: reg_filter_id[1]  <= writedata[15:0];
                13'h08: reg_filter_id[2]  <= writedata[15:0];
                13'h09: reg_filter_id[3]  <= writedata[15:0];
                13'h0B: reg_filter_ctrl   <= writedata[4:0];
                13'h50: reg_price_base[0] <= writedata[23:0];
                13'h51: reg_price_base[1] <= writedata[23:0];
                13'h52: reg_price_base[2] <= writedata[23:0];
                13'h53: reg_price_base[3] <= writedata[23:0];
                ADDR_INGEST_CFG: reg_ingest_cfg <= writedata;
                default: ;
            endcase
        end
    end
end

/*
 * Dispatch parsed ITCH messages into the hash table and PLA/BBO path.
 * Replaces are implemented as DELETE old reference, then INSERT new
 * reference. Executes first LOOKUP the old order, then either DELETE on
 * full fill or INSERT the same reference with reduced quantity.
 */
always_ff @(posedge clk) begin
    if (reset) begin
        ht_state          <= HT_IDLE;
        ht_op_valid       <= 1'b0;
        ht_pending        <= 1'b0;
        ht_ops_total      <= 32'h0;
        ht_miss_count     <= 32'h0;
        ht_stuck_cycles   <= 32'h0;
        ht_stuck_max      <= 32'h0;
        pla_op_valid      <= 1'b0;
        pla_should_issue  <= 1'b0;
    end else begin
        ht_op_valid  <= 1'b0;
        pla_op_valid <= 1'b0;

        if (parsed_valid && stock_match) begin
            ht_msg_type      <= parsed_type;
            ht_order_ref     <= parsed_order_ref;
            ht_new_order_ref <= parsed_new_order_ref;
            ht_price_raw     <= parsed_price;
            ht_shares_raw    <= parsed_shares;
            ht_side_raw      <= parsed_side;
            ht_stock_locate  <= parsed_stock_locate;
            ht_stock_id      <= stock_id_match;
            ht_pla_eligible  <= stock_match_specific && (stock_id_match[1] == 1'b0);
            ht_pending       <= 1'b1;
        end

        if (chipselect && write && address == 13'h00 && writedata[4]) begin
            ht_ops_total    <= 32'h0;
            ht_miss_count   <= 32'h0;
            ht_stuck_cycles <= 32'h0;
            ht_stuck_max    <= 32'h0;
        end else if (ht_state == HT_IDLE) begin
            ht_stuck_cycles <= 32'h0;
        end else if (ht_stuck_cycles != 32'hFFFF_FFFF) begin
            ht_stuck_cycles <= ht_stuck_cycles + 32'd1;
            if ((ht_stuck_cycles + 32'd1) > ht_stuck_max)
                ht_stuck_max <= ht_stuck_cycles + 32'd1;
        end

        case (ht_state)
        HT_IDLE: begin
            if (ht_pending && ht_ready) begin
                ht_pending <= 1'b0;

                case (ht_msg_type)
                8'h41, 8'h46: begin
                    ht_op       <= HT_OP_INSERT;
                    ht_ref_in   <= ht_order_ref;
                    ht_price_in <= ht_price_raw[23:0];
                    ht_qty_in   <= ht_shares_raw[23:0];
                    ht_side_in  <= ht_side_raw;
                    ht_sym_in   <= ht_stock_locate[13:0];
                    ht_op_valid <= 1'b1;
                    ht_state    <= HT_WAIT;
```

```
        end

        8'h44, 8'h55: begin
            ht_op       <= HT_OP_DELETE;
            ht_ref_in   <= ht_order_ref;
            ht_price_in <= 24'h0;
            ht_qty_in   <= 24'h0;
            ht_side_in  <= 1'b0;
            ht_sym_in   <= 14'h0;
            ht_op_valid <= 1'b1;
            ht_state    <= HT_WAIT;
        end

        8'h45, 8'h58, 8'h43: begin
            ht_op       <= HT_OP_LOOKUP;
            ht_ref_in   <= ht_order_ref;
            ht_price_in <= 24'h0;
            ht_qty_in   <= 24'h0;
            ht_side_in  <= 1'b0;
            ht_sym_in   <= 14'h0;
            ht_op_valid <= 1'b1;
            ht_state    <= HT_WAIT;
        end

        default: ht_state <= HT_IDLE;
        endcase
    end
end

HT_WAIT: begin
    if (ht_done) begin
        ht_last_hit       <= ht_hit;
        ht_last_op        <= ht_op;
        ht_last_price_out <= ht_price_out;
        ht_last_qty_out   <= ht_qty_out[7:0];
        ht_ops_total      <= ht_ops_total + 32'd1;
        if (!ht_hit) ht_miss_count <= ht_miss_count + 32'd1;

        case (ht_op)
        HT_OP_INSERT: begin
            pla_target_pair  <= {ht_stock_id[0], ht_side_raw};
            pla_target_price <= price_bucket_raw[9:0];
            pla_target_delta <= $signed({1'b0, ht_shares_raw[23:0]});
            pla_should_issue <= ht_pla_eligible && bucket_raw_in_window;
        end

        HT_OP_DELETE: begin
            pla_target_pair  <= {ht_stock_id[0], ht_side_out};
            pla_target_price <= price_bucket_out[9:0];
            pla_target_delta <= -$signed({1'b0, ht_qty_out[23:0]});
            pla_should_issue <= ht_pla_eligible && ht_hit && bucket_out_in_window;
            ht_side_orig     <= ht_side_out;
            ht_orig_hit      <= ht_hit;
        end

        HT_OP_LOOKUP: begin
            ht_orig_hit <= ht_hit;
            if (ht_hit) begin
                automatic logic [23:0] delta_clamped;
                delta_clamped   = (ht_qty_out > ht_shares_raw[23:0])
                                ? ht_shares_raw[23:0]
                                : ht_qty_out;
                ht_side_orig    <= ht_side_out;
                ht_lookup_price <= ht_price_out;
                ht_lookup_sym   <= ht_sym_out;
                ht_exec_new_qty <= ht_qty_out - delta_clamped;
                ht_exec_full    <= (ht_qty_out <= ht_shares_raw[23:0]);

                pla_target_pair  <= {ht_stock_id[0], ht_side_out};
                pla_target_price <= price_bucket_out[9:0];
                pla_target_delta <= -$signed({1'b0, delta_clamped});
                pla_should_issue <= ht_pla_eligible && bucket_out_in_window;
            end else begin
                pla_should_issue <= 1'b0;
            end
        end
        endcase

        ht_state <= PLA_ISSUE;
    end
end

PLA_ISSUE: begin
    if (!pla_should_issue) begin
        if (ht_msg_type == 8'h55 && ht_orig_hit)
            ht_state <= HT_REPLACE_INSERT;
        else if ((ht_msg_type == 8'h45 || ht_msg_type == 8'h58 ||
                  ht_msg_type == 8'h43) && ht_orig_hit)
            ht_state <= HT_EXEC_OP;
        else
            ht_state <= HT_IDLE;
    end else if (pla_update_ready[pla_target_pair]) begin
```

```
            pla_op_valid     <= 1'b1;
            pla_should_issue <= 1'b0;
            ht_state         <= PLA_WAIT;
        end
    end

    PLA_WAIT: begin
        if (pla_done[pla_target_pair]) begin
            if (ht_msg_type == 8'h55 && ht_orig_hit)
                ht_state <= HT_REPLACE_INSERT;
            else if ((ht_msg_type == 8'h45 || ht_msg_type == 8'h58 ||
                      ht_msg_type == 8'h43) && ht_orig_hit)
                ht_state <= HT_EXEC_OP;
            else
                ht_state <= HT_IDLE;
        end
    end

    HT_REPLACE_INSERT: begin
        if (ht_ready) begin
            ht_op       <= HT_OP_INSERT;
            ht_ref_in   <= ht_new_order_ref;
            ht_price_in <= ht_price_raw[23:0];
            ht_qty_in   <= ht_shares_raw[23:0];
            ht_side_in  <= ht_side_orig;
            ht_sym_in   <= ht_stock_locate[13:0];
            ht_op_valid <= 1'b1;
            ht_state    <= HT_WAIT2;
            ht_msg_type <= 8'h41;
        end
    end

    HT_WAIT2: begin
        if (ht_done) begin
            ht_ops_total <= ht_ops_total + 32'd1;
            if (!ht_hit) ht_miss_count <= ht_miss_count + 32'd1;

            pla_target_pair  <= {ht_stock_id[0], ht_side_orig};
            pla_target_price <= price_bucket_raw[9:0];
            pla_target_delta <= $signed({1'b0, ht_shares_raw[23:0]});
            pla_should_issue <= ht_pla_eligible && bucket_raw_in_window;
            ht_state         <= PLA_ISSUE2;
        end
    end

    PLA_ISSUE2: begin
        if (!pla_should_issue) begin
            ht_state <= HT_IDLE;
        end else if (pla_update_ready[pla_target_pair]) begin
            pla_op_valid     <= 1'b1;
            pla_should_issue <= 1'b0;
            ht_state         <= PLA_WAIT2;
        end
    end

    PLA_WAIT2: begin
        if (pla_done[pla_target_pair]) ht_state <= HT_IDLE;
    end

    HT_EXEC_OP: begin
        if (ht_ready) begin
            if (ht_exec_full) begin
                ht_op       <= HT_OP_DELETE;
                ht_ref_in   <= ht_order_ref;
                ht_price_in <= 24'h0;
                ht_qty_in   <= 24'h0;
                ht_side_in  <= 1'b0;
                ht_sym_in   <= 14'h0;
            end else begin
                ht_op       <= HT_OP_INSERT;
                ht_ref_in   <= ht_order_ref;
                ht_price_in <= ht_lookup_price;
                ht_qty_in   <= ht_exec_new_qty;
                ht_side_in  <= ht_side_orig;
                ht_sym_in   <= ht_lookup_sym;
            end
            ht_op_valid <= 1'b1;
            ht_state    <= HT_EXEC_WAIT;
        end
    end

    HT_EXEC_WAIT: begin
        if (ht_done) begin
            ht_ops_total <= ht_ops_total + 32'd1;
            if (!ht_hit) ht_miss_count <= ht_miss_count + 32'd1;
            ht_state <= HT_IDLE;
        end
    end

    default: ht_state <= HT_IDLE;
    endcase
end
```

```
end

assign ingest_start_pulse        = chipselect && write && (address == 13'h00) && writedata[0];
assign ingest_clear_status_pulse = chipselect && write && (address == 13'h00) && writedata[4];
assign ingest_buf0_len_we        = chipselect && write && (address == ADDR_BUF0_LEN);
assign ingest_buf1_len_we        = chipselect && write && (address == ADDR_BUF1_LEN);
assign ingest_buf0_ready_set     = chipselect && write && (address == ADDR_BUF0_FLAGS) && writedata[0];
assign ingest_buf1_ready_set     = chipselect && write && (address == ADDR_BUF1_FLAGS) && writedata[0];
assign ingest_buf0_done_ack      = chipselect && write && (address == ADDR_BUF0_FLAGS) && writedata[1];
assign ingest_buf1_done_ack      = chipselect && write && (address == ADDR_BUF1_FLAGS) && writedata[1];

always_comb begin
    if (chipselect && read) begin
        if (in_buf_region) begin
            readdata = read_buf_word_reg;
        end else begin
            case (address)
                13'h00: readdata = reg_control;
                13'h01: readdata = {23'h0, parse_err_sticky, 3'h0, 1'b1, 4'h0};
                13'h02: readdata = parsed_msg_count;
                13'h03: readdata = reg_scratch;
                13'h04: readdata = 32'hCAFE_4840;
                13'h06: readdata = {16'h0, reg_filter_id[0]};
                13'h07: readdata = {16'h0, reg_filter_id[1]};
                13'h0B: readdata = {27'h0, reg_filter_ctrl};
                ADDR_INGEST_CFG:  readdata = reg_ingest_cfg;
                ADDR_INGEST_STAT: readdata = {8'h0, ingest_current_msg_left,
                                              ingest_error_code, ingest_active_buf,
                                              ingest_busy, ingest_armed, reg_ingest_cfg[0]};

                13'h1c: readdata = {17'b0, ht_load};
                13'd29: readdata = {24'h0, ht_max_probe};
                13'd30: readdata = {23'h0, ht_last_hit, ht_last_op,
                                    ht_state, ht_ready, ht_done};
                13'd31: readdata = {ht_last_price_out, ht_last_qty_out};
                13'd32: readdata = ht_ops_total;
                13'd33: readdata = ht_miss_count;
                13'h22: readdata = {ht_op_valid, ht_done, ht_ready, ht_pending,
                                    ht_dbg_state, ht_state,
                                    ht_dbg_rd_valid, ht_dbg_ref_match,
                                    ht_dbg_shift_needed, 1'b0,
                                    ht_msg_type, ht_dbg_cur_op, 6'h0};
                13'h23: readdata = ht_stuck_cycles;
                13'h24: readdata = ht_dbg_cur_ref[31:0];
                13'h25: readdata = ht_dbg_cur_ref[63:32];
                13'h26: readdata = {ht_dbg_chain_len, 10'h0, ht_dbg_probe_idx};
                13'h27: readdata = ht_stuck_max;
                13'h28: readdata = {2'h0, ht_dbg_hole_idx, 2'h0, ht_dbg_scan_idx};

                13'h48, 13'h49, 13'h4a, 13'h4b:
                    readdata = {21'h0, bbo_best_valid[address[1:0]],
                                bbo_best_price[address[1:0]]};

                13'h50: readdata = {8'h0, reg_price_base[0]};
                13'h51: readdata = {8'h0, reg_price_base[1]};
                13'h54: readdata = oow_count[0];
                13'h55: readdata = oow_count[1];
                default: readdata = 32'hDEAD_BEEF;
            endcase
        end
    end else begin
        readdata = 32'h0;
    end
end

buffered_ingest #(
    .HALF_BUFFER_BYTES (HALF_BYTES)
) u_buffered_ingest (
    .clk             (clk),
    .reset           (reset),
    .stall           (ht_stall),
    .start           (ingest_start_pulse),
    .continuous_mode (reg_ingest_cfg[1]),
    .clear_status    (ingest_clear_status_pulse),
    .mem_rd_addr     (ingest_mem_rd_addr),
    .mem_rd_data     (ingest_mem_rd_data),
    .word_out        (ingest_word),
    .valid_bytes     (ingest_valid_bytes),
    .word_valid      (ingest_word_valid),
    .msg_done        (ingest_msg_done),
    .new_message     (ingest_new_message),
    .busy            (ingest_busy),
    .armed           (ingest_armed),
    .active_buf      (ingest_active_buf)
);

itch_parser u_itch_parser (
    .clk           (clk),
    .reset         (reset),
    .word_in       (ingest_word),
    .valid_bytes   (ingest_valid_bytes),
    .word_valid    (reg_ingest_cfg[0] & ingest_word_valid),
```

```
    .new_message   (reg_ingest_cfg[0] & ingest_new_message),
    .msg_done      (reg_ingest_cfg[0] & ingest_msg_done),
    .msg_valid     (parsed_valid),
    .msg_type      (parsed_type),
    .stock_locate  (parsed_stock_locate),
    .order_ref     (parsed_order_ref),
    .price         (parsed_price),
    .shares        (parsed_shares),
    .side          (parsed_side),
    .new_order_ref (parsed_new_order_ref),
    .msg_count     (parsed_msg_count)
);

hash_table #(
    .SLOT_BITS (14),
    .NUM_SLOTS (16384),
    .MAX_CHAIN (64)
) u_hash_table (
    .clk       (clk),
    .rst       (reset),
    .op_valid  (ht_op_valid),
    .op        (ht_op),
    .ref_in    (ht_ref_in),
    .price_in  (ht_price_in),
    .qty_in    (ht_qty_in),
    .side_in   (ht_side_in),
    .sym_in    (ht_sym_in),
    .ready     (ht_ready),
    .done      (ht_done),
    .hit       (ht_hit),
    .price_out (ht_price_out),
    .qty_out   (ht_qty_out),
    .side_out  (ht_side_out),
    .sym_out   (ht_sym_out),
    .ht_load   (ht_load),
    .max_probe (ht_max_probe),
    .dbg_state        (ht_dbg_state),
    .dbg_cur_op       (ht_dbg_cur_op),
    .dbg_cur_ref      (ht_dbg_cur_ref),
    .dbg_probe_idx    (ht_dbg_probe_idx),
    .dbg_hole_idx     (ht_dbg_hole_idx),
    .dbg_scan_idx     (ht_dbg_scan_idx),
    .dbg_chain_len    (ht_dbg_chain_len),
    .dbg_rd_valid     (ht_dbg_rd_valid),
    .dbg_ref_match    (ht_dbg_ref_match),
    .dbg_shift_needed (ht_dbg_shift_needed)
);

logic               pla_op_valid;
logic [1:0]         pla_target_pair;
logic [9:0]         pla_target_price;
logic signed [24:0] pla_target_delta;

logic [3:0]  pla_update_ready;
logic [3:0]  pla_done;
logic [3:0]  bbo_best_valid;
logic [9:0]  bbo_best_price [0:3];

genvar gi;
generate
    for (gi = 0; gi < 4; gi++) begin : g_pla_pairs
        logic        set_valid_w;
        logic        set_active_w;
        logic [9:0]  set_price_w;

        pla #(
            .PRICE_BITS (10),
            .QTY_BITS   (24)
        ) u_pla (
            .clk          (clk),
            .rst          (reset),
            .update_valid (pla_op_valid && (pla_target_pair == gi[1:0])),
            .update_ready (pla_update_ready[gi]),
            .update_price (pla_target_price),
            .update_delta (pla_target_delta),
            .done         (pla_done[gi]),
            .set_valid    (set_valid_w),
            .set_active   (set_active_w),
            .set_price    (set_price_w)
        );

        bbo_bitmap #(
            .PRICE_BITS  (10),
            .GROUP_BITS  (5),
            .WITHIN_BITS (5),
            .IS_MAX      ((gi % 2) == 0)
        ) u_bbo (
            .clk        (clk),
            .rst        (reset),
            .set_valid  (set_valid_w),
            .set_active (set_active_w),
            .set_price  (set_price_w),
```

```
                .best_valid (bbo_best_valid[gi]),
                .best_price (bbo_best_price[gi])
            );
        end
    endgenerate

    /*
     * The PLL is declared inside this peripheral so Platform Designer sees a
     * clock source that can be looped back into the core's Avalon clock input.
     */
    wire [4:0] pll_clk_w;
    wire       pll_locked_w;

    altpll #(
        .clk0_divide_by         (1),
        .clk0_multiply_by       (2),
        .inclk0_input_frequency (20000),
        .intended_device_family ("Cyclone␣V"),
        .operation_mode         ("NORMAL"),
        .port_clk0              ("PORT_USED"),
        .port_inclk0            ("PORT_USED"),
        .port_locked            ("PORT_USED"),
        .width_clock            (5)
    ) u_pll_50_to_100 (
        .inclk  ({1'b0, ref_clk}),
        .clk    (pll_clk_w),
        .locked (pll_locked_w)
    );

    assign clk_out = pll_clk_w[0];

endmodule
```

C.2 Software

Listing 8: itch_arm.c: Software reference implementation.

```
#define _POSIX_C_SOURCE 200809L
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <time.h>

#define HT_SLOTS    16384u
#define HT_MAX_PROBE   64u
#define N_BUCKETS    1024u
#define HALF_WINDOW 51200u

typedef struct {
    uint8_t  occupied;
    uint64_t order_ref;
    uint32_t price;
    uint32_t qty;
    uint8_t  side;
    uint8_t  stock_id;
} ht_entry;

static ht_entry ht[HT_SLOTS];
static uint32_t book[2][2][N_BUCKETS];
static uint32_t base_price[2];
static int      best_bid_b[2] = {-1, -1};
static int      best_ask_b[2] = {-1, -1};

static uint16_t hash_key(uint64_t r) {
    uint16_t a = (r >>  0) & 0x3FFFu;
    uint16_t b = (r >> 14) & 0x3FFFu;
    uint16_t c = (r >> 28) & 0x3FFFu;
    return (uint16_t)((a ^ b ^ c) & (HT_SLOTS - 1));
}

static int ht_find(uint64_t r) {
    uint16_t k = hash_key(r);
    for (uint32_t i = 0; i < HT_MAX_PROBE; i++) {
        uint32_t s = (k + i) & (HT_SLOTS - 1);
        if (!ht[s].occupied) return -1;
        if (ht[s].order_ref == r) return (int)s;
    }
    return -1;
}

static void apply_delta(uint8_t stock, uint8_t side, int bucket, int32_t delta) {
    if (bucket < 0) return; // Ignore updates outside the tracked cent-window
    uint32_t before = book[stock][side][bucket];
    int64_t after = (int64_t)before + delta;
```

```
    book[stock][side][bucket] = (after < 0) ? 0 : (uint32_t)after;

    if (side == 0) { // Bid
        if (after > 0 && bucket > best_bid_b[stock]) best_bid_b[stock] = bucket;
        else if (after == 0 && bucket == best_bid_b[stock]) {
            best_bid_b[stock] = -1;
            for (int b = bucket - 1; b >= 0; b--)
                if (book[stock][0][b]) { best_bid_b[stock] = b; break; }
        }
    } else { // Ask
        if (after > 0 && (best_ask_b[stock] < 0 || bucket < best_ask_b[stock])) best_ask_b[stock] = bucket;
        else if (after == 0 && bucket == best_ask_b[stock]) {
            best_ask_b[stock] = -1;
            for (int b = bucket + 1; b < N_BUCKETS; b++)
                if (book[stock][1][b]) { best_ask_b[stock] = b; break; }
        }
    }
}

int main(int argc, char **argv) {
    // ... mmap session.bin ...

    // Resolve Locate IDs from Stock Directory messages at file start
    resolve_symbols(data, sz, syms, n_syms, ids);

    // Set baseline prices to center the initial order in the 1024-cent window
    scan_first_prices(data, sz, ids, n_syms, firsts);
    for (int s = 0; s < n_syms; s++) {
        base_price[s] = (firsts[s] >= HALF_WINDOW) ? firsts[s] - HALF_WINDOW : 0;
    }

    size_t offset = 0;
    while (offset < sz) {
        // ... Chunked framing logic ...
        while (i < chunk_end) {
            uint16_t mlen = be16(data + i);
            const uint8_t *p = data + i + 2;
            total_msgs++;

            int sid = resolve_sid(p);
            if (sid < 0) { i += 2 + mlen; continue; }
            msgs++;

            switch (p[0]) {
                case 'A': case 'F': {
                    uint64_t ref = be64(p + 11);
                    uint32_t qty = be32(p + 20), price = be32(p + 32);
                    ht_insert(ref, price, qty, (p[19]=='S'), sid);
                    apply_delta(sid, (p[19]=='S'), (price - base_price[sid])/100, qty);
                    break;
                }
                case 'D': {
                    int s = ht_find(be64(p + 11));
                    if (s >= 0) {
                        apply_delta(ht[s].stock_id, ht[s].side, (ht[s].price - base_price[ht[s].stock_id])/100, -ht[s].qty);
                        ht_delete(s);
                    }
                    break;
                }
                case 'U': { // Replace
                    int s = ht_find(be64(p+11));
                    if (s >= 0) {
                        apply_delta(ht[s].stock_id, ht[s].side, (ht[s].price - base_price[ht[s].stock_id])/100, -ht[s].qty);
                        ht_delete(s);
                        ht_insert(be64(p+19), be32(p+31), be32(p+27), ht[s].side, sid);
                        apply_delta(sid, ht[s].side, (be32(p+31) - base_price[sid])/100, be32(p+27));
                    }
                    break;
                }
                case 'E': case 'C': case 'X': // Executes
                    execute_ref(be64(p + 11), be32(p + 19));
                    break;
            }
            i += 2 + mlen;
        }
        offset += chunk_size;
    }
    // Write results to /tmp/hft_arm.txt for comparison
    return 0;
}
```

Listing 9: itch_load.c: FPGA Ingest and MMIO management.

```
#define ORDERBOOK_PHYS_BASE   0xFF200000u
#define ORDERBOOK_BUF_BASE_WORD 512u
#define HALF_BYTES              8192u

int main(int argc, char **argv) {
    // Access hardware via /dev/mem mapping
    void *map = mmap(NULL, 0x8000, PROT_READ|PROT_WRITE, MAP_SHARED, memfd, ORDERBOOK_PHYS_BASE);
```

```
    g_mm = (volatile uint32_t *)map;

    mm_write(W_CONTROL, 0x10);   // Clear sticky errors
    mm_write(W_INGEST_CFG, 0x3); // Enable engine + continuous mode

    // Program hardware Locate ID filters
    for (int i = 0; i < 4; i++) mm_write(W_FILTER_ID0 + i, filter_ids[i]);
    mm_write(W_FILTER_CTRL, ctrl_mask);

    size_t offset = 0;
    int use_buf1 = 0;
    while (offset < sz) {
        // ... Chunk framing logic ...

        // Wait for hardware to finish processing the previous buffer half
        uint32_t flags_addr = use_buf1 ? W_BUF1_FLAGS : W_BUF0_FLAGS;
        while ((mm_read(flags_addr) & 3u) != 0);

        // Copy chunk to dual-port RAM via AXI-Lite bridge
        volatile uint32_t *buf_words = (uint32_t*)((char*)map + (ORDERBOOK_BUF_BASE_WORD*4));
        if (use_buf1) buf_words += HALF_BYTES/4;
        memcpy((void*)buf_words, file_data + offset, chunk_size);

        mm_write(use_buf1 ? W_BUF1_LEN : W_BUF0_LEN, chunk_size);

        // Handshake: Set READY bit and arm the FSM on the first buffer
        mm_write(flags_addr, 1u);
        if (!started) { mm_write(W_CONTROL, 0x1u); started = 1; }

        // Poll for bit 2 (DONE) for cycle-accurate latency tracking
        clock_gettime(CLOCK_MONOTONIC, &c0);
        while ((mm_read(flags_addr) & 4u) == 0);
        clock_gettime(CLOCK_MONOTONIC, &c1);
        accumulated_fpga_us += elapsed_us(c0, c1);

        offset += chunk_size;
        use_buf1 = !use_buf1;
    }
    // Poll global busy bit until internal pipeline is empty
    while ((mm_read(W_INGEST_STAT) & 4u) != 0);

    // Dump final BBO and throughput results to /tmp/hft_fpga.txt
    return 0;
}
```

Listing 10: vga_writer.c: Live VGA status display.

```
#define LW_PHYS_BASE  0xFF200000u
#define CHAR_BUF_WOFF (0x8000u / 4u)

// Packing: {Red[7:0], Green[7:0], Blue[7:0], ASCII[7:0]}
#define CELL(r,g,b,ch) ((((r)<<24)|((g)<<16)|((b)<<8)) | (uint8_t)(ch))

static void vc_put(int row, int col, char c, uint32_t color) {
    cbuf[row * 40 + col] = (color & 0xFFFFFF00u) | (uint8_t)c;
}

static void draw_buf(int row, int idx, uint32_t flags, uint32_t len) {
    // Decode hardware status bits for visual feedback
    bool error  = (flags >> 3) & 1, done = (flags >> 2) & 1;
    bool active = (flags >> 1) & 1, ready = (flags >> 0) & 1;

    uint32_t color = error ? C_RED : active ? C_GREEN : done ? C_YELLOW : C_DIM;
    snprintf(line, 40, "BUF%d␣[%s]␣%u␣bytes", idx, active?"ACTV":done?"DONE":"IDLE", len);
    vc_puts_fill(row, line, color);
}

int main(int argc, char **argv) {
    // Map both orderbook registers (0x0000) and character buffer (0x8000)
    void *map = mmap(NULL, 0x10000, PROT_READ|PROT_WRITE, MAP_SHARED, memfd, LW_PHYS_BASE);
    regs = (volatile uint32_t *)map;
    cbuf = regs + CHAR_BUF_WOFF;

    while (g_run) {
        // Read the most recent Golden Model results from /tmp/
        read_result("/tmp/hft_arm.txt", &arm_r);
        read_result("/tmp/hft_fpga.txt", &fpga_r);

        // Display Hardware stats directly from AXI-Lite registers
        snprintf(line, 40, "MSG␣CNT␣:␣%u␣␣␣ERR:␣%u", reg_rd(W_MSG_COUNT), reg_rd(W_ERR_COUNT));
        vc_puts_fill(3, line, C_WHITE);

        // Live Speedup calculation
        if (arm_r.present && fpga_r.present) {
            double sx = (double)arm_r.us / (double)fpga_r.us;
            int matches = count_matching_bbo(&arm_r, &fpga_r);
            snprintf(line, 40, "SPEEDUP␣:␣%.2fx␣␣MATCH:␣%d/2", sx, matches);
            vc_puts_fill(22, line, sx >= 1.0 ? C_GREEN : C_YELLOW);
        }
```

```
        // Display last parsed message (from parser's debug registers)
        uint32_t msg1 = reg_rd(W_PARSED_MSG1);
        snprintf(line, 40, "TYPE␣'%c'␣␣SIDE␣%c␣␣LOC␣%u", (msg1>>16)&0xFF, (msg1>>24)?'S':'B', msg1&0xFFFF);
        vc_puts_fill(26, line, C_WHITE);

        usleep(100000); // UI Refresh Rate
    }
}
```

C.3 Testing Code

Listing 11: generate_robust_itch.py: Synthetic ITCH stress-stream generator.

```
import argparse
import os
import struct
from collections import deque
from pathlib import Path

SYMBOLS = {'RKLB': 1, 'FLNC': 2, 'NOISE': 99}

DEFAULT_MESSAGES = 1_000_000
DEFAULT_CYCLES = 1_000
DEFAULT_TARGET_DEPTH = 200
CHUNK_MSGS = 27_594

REF_BASE = {'RKLB': 0, 'FLNC': 8_192, 'NOISE': 9_000_000}
COLLISION_INDEX_BASE = {'RKLB': 0, 'FLNC': 8_192}
REFS_PER_STOCK = 8_192
BAD_REF_BASE = 1 << 60


def wrap(payload):
    return struct.pack('>H', len(payload)) + payload


def pad_symbol(stock):
    return stock.encode('ascii').ljust(8, b'␣')[:8]


def build_stock_directory():
    directory = bytearray()
    for stock, locate in SYMBOLS.items():
        payload = struct.pack(
            '>cHH6s8sccIcc2sccccIc',
            b'R', locate, 0, b'\x00' * 6, pad_symbol(stock),
            b'Q', b'␣', 100, b'N', b'C', b'␣␣', b'P',
            b'␣', b'␣', b'␣', b'N', 0, b'N',
        )
        directory.extend(wrap(payload))
    return bytes(directory)


def build_header(msg_type, stock, ref):
    return struct.pack('>cHH6sQ', msg_type, SYMBOLS[stock], 0, b'\x00' * 6, ref)


def build_add(stock, ref, price, qty, side):
    return wrap(build_header(b'A', stock, ref) +
                struct.pack('>cI8sI', side, qty, pad_symbol(stock), price))


def build_delete(stock, ref):
    return wrap(build_header(b'D', stock, ref))


def build_execute(stock, ref, qty):
    return wrap(build_header(b'E', stock, ref) + struct.pack('>I', qty))


def build_replace(stock, old_ref, new_ref, price, qty):
    return wrap(build_header(b'U', stock, old_ref) +
                struct.pack('>QII', new_ref, qty, price))


def build_noise_pair():
    buy = build_add('NOISE', REF_BASE['NOISE'], 10_000, 100, b'B')
    sell = build_add('NOISE', REF_BASE['NOISE'] + 1, 10_100, 100, b'S')
    return buy, sell, buy + sell


def write_noise_messages(f, count, noise_index, noise_msgs):
```

```
    if count <= 0:
        return noise_index

    buy, sell, pair = noise_msgs

    if noise_index % 2:
        f.write(sell)
        noise_index += 1
        count -= 1

    while count >= CHUNK_MSGS:
        f.write(pair * (CHUNK_MSGS // 2))
        noise_index += CHUNK_MSGS
        count -= CHUNK_MSGS

    if count >= 2:
        pairs = count // 2
        f.write(pair * pairs)
        noise_index += pairs * 2
        count -= pairs * 2

    if count:
        f.write(buy)
        noise_index += 1

    return noise_index


def tracked_positions(total_messages, tracked_count):
    if tracked_count == 0:
        return []
    return [(i * total_messages) // tracked_count for i in range(tracked_count)]


def clustered_ref(index, bucket):
    bucket &= 0x3FFF
    index &= 0x3FFF
    return bucket | (index << 14) | (index << 28)


def next_ref(state, stock, collision_mode='none', collision_bucket=0):
    if state[stock]['refs_used'] >= REFS_PER_STOCK:
        raise ValueError(f'{stock} exhausted its {REFS_PER_STOCK}-ref range')

    if collision_mode == 'clustered':
        index = COLLISION_INDEX_BASE[stock] + state[stock]['refs_used']
        ref = clustered_ref(index, collision_bucket)
    else:
        ref = state[stock]['next_ref']
        if ref >= REF_BASE[stock] + REFS_PER_STOCK:
            raise ValueError(f'{stock} exhausted its collision-safe {REFS_PER_STOCK}-ref range')
        state[stock]['next_ref'] += 1

    state[stock]['refs_used'] += 1
    return ref


def add_order(events, active, state, stats, stock, side, price, cycle, qty=100,
              collision_mode='none', collision_bucket=0):
    ref = next_ref(state, stock, collision_mode, collision_bucket)
    events.append(build_add(stock, ref, price, qty, side))
    active[stock][side].append({
        'ref': ref,
        'price': price,
        'qty': qty,
        'birth_cycle': cycle,
    })
    stats['adds'] += 1
    return ref


def delete_deque_index(items, index):
    items.rotate(-index)
    removed = items.popleft()
    items.rotate(index)
    return removed


def select_execute_index(orders, cycle, policy, min_age, cursor):
    effective_min_age = max(1, min_age) if policy == 'aged' else min_age
    eligible = [
        i for i, order in enumerate(orders)
        if cycle - order['birth_cycle'] >= effective_min_age
    ]
    if not eligible:
        return None

    if policy == 'newest':
        return eligible[-1]
    if policy in ('oldest', 'aged'):
        return eligible[0]
    if policy == 'round-robin':
```

```
        return eligible[cursor % len(eligible)]

    raise ValueError(f'unknown execute policy {policy!r}')


def execute_order(events, active, stats, execute_cursor, stock, side, cycle,
                  qty=25, full_fill=False, policy='newest', min_age=0):
    orders = active[stock][side]
    if not orders:
        stats['skipped_executes'] += 1
        return False

    key = (stock, side)
    index = select_execute_index(orders, cycle, policy, min_age,
                                 execute_cursor.get(key, 0))
    if index is None:
        stats['skipped_executes'] += 1
        return False

    order = orders[index]
    exec_qty = order['qty'] if full_fill else min(qty, order['qty'])
    events.append(build_execute(stock, order['ref'], exec_qty))
    order['qty'] -= exec_qty
    stats['executes'] += 1
    execute_cursor[key] = execute_cursor.get(key, 0) + 1

    if order['qty'] == 0:
        stats['full_fill_executes'] += 1
        delete_deque_index(orders, index)
    else:
        stats['partial_executes'] += 1

    return True


def replace_oldest(events, active, state, stats, stock, side, price, cycle, qty=100,
                   collision_mode='none', collision_bucket=0):
    if not active[stock][side]:
        return

    old = active[stock][side].popleft()
    new_ref = next_ref(state, stock, collision_mode, collision_bucket)
    events.append(build_replace(stock, old['ref'], new_ref, price, qty))
    active[stock][side].append({
        'ref': new_ref,
        'price': price,
        'qty': qty,
        'birth_cycle': cycle,
    })
    stats['replaces'] += 1


def delete_oldest(events, active, stats, stock, side):
    if active[stock][side]:
        old = active[stock][side].popleft()
        events.append(build_delete(stock, old['ref']))
        stats['deletes'] += 1


def delete_best(events, active, stats, stock, side):
    orders = active[stock][side]
    if not orders:
        return

    if side == b'B':
        index = max(range(len(orders)), key=lambda i: orders[i]['price'])
    else:
        index = min(range(len(orders)), key=lambda i: orders[i]['price'])

    old = delete_deque_index(orders, index)
    events.append(build_delete(stock, old['ref']))
    stats['deletes'] += 1


def should_full_fill_execute(execute_index, full_fill_exec_ratio):
    if full_fill_exec_ratio <= 0:
        return False
    if full_fill_exec_ratio >= 100:
        return True
    return (execute_index % 100) < full_fill_exec_ratio


def build_miss_events(bad_delete_count, bad_execute_count):
    events = []
    stats = {'bad_deletes': bad_delete_count, 'bad_executes': bad_execute_count}

    for i in range(bad_delete_count):
        stock = 'RKLB' if i % 2 == 0 else 'FLNC'
        events.append(build_delete(stock, BAD_REF_BASE + i))

    for i in range(bad_execute_count):
        stock = 'RKLB' if i % 2 == 0 else 'FLNC'
```

```
        events.append(build_execute(stock, BAD_REF_BASE + bad_delete_count + i, 25))

    return events, stats


def interleave_events(primary, extra):
    if not extra:
        return primary

    total = len(primary) + len(extra)
    extra_positions = set(tracked_positions(total, len(extra)))
    result = []
    primary_index = 0
    extra_index = 0

    for out_index in range(total):
        if out_index in extra_positions and extra_index < len(extra):
            result.append(extra[extra_index])
            extra_index += 1
        else:
            result.append(primary[primary_index])
            primary_index += 1

    return result


def triangle_offset(cycle, period, amplitude):
    if period <= 1 or amplitude == 0:
        return 0

    phase = (cycle % period) / period
    if phase < 0.5:
        return int(-amplitude + (4 * amplitude * phase))
    return int((3 * amplitude) - (4 * amplitude * phase))


def price_pair(spec, cycle, cycles, price_mode):
    base = spec['base']
    spread = spec['spread']

    if price_mode == 'trend':
        improvement = (cycle * spread) // max(1, cycles - 1)
        return base - spread + improvement, base + 200 + spread - improvement

    if price_mode == 'wave':
        period = max(8, cycles // 4)
        offset = triangle_offset(cycle, period, spread // 2)
        return base + offset, base + offset + 200

    raise ValueError(f'unknown price mode {price_mode!r}')


def build_churn_events(cycles, target_depth, replace_interval, include_replaces,
                       execute_qty, full_fill_exec_ratio, collision_mode,
                       collision_bucket, bad_delete_count, bad_execute_count,
                       execute_policy, execute_min_age, price_mode,
                       bbo_churn_demo, bbo_churn_interval):
    events = []
    active = {
        stock: {b'B': deque(), b'S': deque()}
        for stock in ('RKLB', 'FLNC')
    }
    state = {
        stock: {'next_ref': REF_BASE[stock], 'refs_used': 0}
        for stock in ('RKLB', 'FLNC')
    }
    stats = {
        'adds': 0,
        'deletes': 0,
        'executes': 0,
        'partial_executes': 0,
        'full_fill_executes': 0,
        'skipped_executes': 0,
        'replaces': 0,
        'bad_deletes': 0,
        'bad_executes': 0,
    }
    specs = {
        'RKLB': {'base': 50_000, 'spread': 10_000},
        'FLNC': {'base': 20_000, 'spread': 10_000},
    }

    execute_index = 0
    execute_cursor = {}

    for cycle in range(cycles):
        for stock in ('RKLB', 'FLNC'):
            bid_price, ask_price = price_pair(specs[stock], cycle, cycles, price_mode)

            add_order(events, active, state, stats, stock, b'B', bid_price, cycle,
                      collision_mode=collision_mode, collision_bucket=collision_bucket)
```

```
            if include_replaces and cycle and cycle % replace_interval == 0:
                replace_oldest(events, active, state, stats, stock, b'B',
                               bid_price, cycle, 100, collision_mode, collision_bucket)

            if execute_order(events, active, stats, execute_cursor, stock, b'B',
                             cycle, execute_qty,
                             should_full_fill_execute(execute_index, full_fill_exec_ratio),
                             execute_policy, execute_min_age):
                execute_index += 1

            add_order(events, active, state, stats, stock, b'S', ask_price, cycle,
                      collision_mode=collision_mode, collision_bucket=collision_bucket)

            if include_replaces and cycle and cycle % replace_interval == 0:
                replace_oldest(events, active, state, stats, stock, b'S',
                               ask_price, cycle, 100, collision_mode, collision_bucket)

            if execute_order(events, active, stats, execute_cursor, stock, b'S',
                             cycle, execute_qty,
                             should_full_fill_execute(execute_index, full_fill_exec_ratio),
                             execute_policy, execute_min_age):
                execute_index += 1

            if bbo_churn_demo and cycle and cycle % bbo_churn_interval == 0:
                delete_best(events, active, stats, stock, b'B')
                delete_best(events, active, stats, stock, b'S')

            while len(active[stock][b'B']) > target_depth:
                delete_oldest(events, active, stats, stock, b'B')
            while len(active[stock][b'S']) > target_depth:
                delete_oldest(events, active, stats, stock, b'S')

    miss_events, miss_stats = build_miss_events(bad_delete_count, bad_execute_count)
    stats.update(miss_stats)
    events = interleave_events(events, miss_events)

    return events, active, state, stats


def main():
    args = parse_args()

    repo_root = Path(__file__).resolve().parents[1]
    out_file = Path(args.output).expanduser() if args.output else \
        repo_root / 'sw' / 'userspace' / 'input' / 'robust_itch.bin'
    if not out_file.is_absolute():
        out_file = repo_root / out_file
    out_file = out_file.resolve()

    directory = build_stock_directory()
    tracked_events, active, state, stats = build_churn_events(
        args.cycles,
        args.target_depth,
        args.replace_interval,
        args.include_replaces,
        args.execute_qty,
        args.full_fill_exec_ratio,
        args.collision_mode,
        args.collision_bucket,
        args.bad_delete_count,
        args.bad_execute_count,
        args.execute_policy,
        args.execute_min_age,
        args.price_mode,
        args.bbo_churn_demo,
        args.bbo_churn_interval,
    )

    total_messages = args.messages
    if args.tracked_fraction is not None:
        total_messages = (
            (len(tracked_events) * 100 + args.tracked_fraction - 1) //
            args.tracked_fraction
        )
        total_messages = max(total_messages, len(tracked_events))

    if len(tracked_events) > total_messages:
        raise SystemExit(
            f'Churn scenario has {len(tracked_events)} tracked messages, '
            f'which exceeds total messages={total_messages}.'
        )

    os.makedirs(out_file.parent, exist_ok=True)
    noise_msgs = build_noise_pair()
    positions = tracked_positions(total_messages, len(tracked_events))

    msgs_written = 0
    tracked_written = 0
    noise_written = 0

    with open(out_file, 'wb') as f:
        f.write(directory)
```

```
        for position, event in zip(positions, tracked_events):
            noise_count = position - msgs_written
            noise_written = write_noise_messages(f, noise_count, noise_written, noise_msgs)
            msgs_written += noise_count

            f.write(event)
            tracked_written += 1
            msgs_written += 1

        remaining = total_messages - msgs_written
        noise_written = write_noise_messages(f, remaining, noise_written, noise_msgs)
        msgs_written += remaining

    expected_ht_ops = (
        stats['adds'] +
        stats['deletes'] +
        (2 * stats['executes']) +
        (2 * stats['replaces']) +
        stats['bad_deletes'] +
        stats['bad_executes']
    )
    expected_misses = stats['bad_deletes'] + stats['bad_executes']

    print(f'Messages␣written␣after␣directory:␣{msgs_written}')
    print(f'Tracked␣churn␣written:␣{tracked_written}')
    print(f'NOISE␣written:␣{noise_written}')
    print(f'Expected␣HT_OPS_TOTAL␣with␣RKLB/FLNC␣filters:␣{expected_ht_ops}')
    print(f'Expected␣HT_MISS_COUNT␣with␣RKLB/FLNC␣filters:␣{expected_misses}')

    for stock in ('RKLB', 'FLNC'):
        live = len(active[stock][b'B']) + len(active[stock][b'S'])
        refs_used = state[stock]['refs_used']
        print(f'{stock}:␣live_orders={live},␣refs_used={refs_used}')


if __name__ == '__main__':
    main()
```

Listing 12: generate_full_itch.py: Baseline large ITCH stream generator.

```
import argparse
import os
import struct
from pathlib import Path

SYMBOLS = {'RKLB': 1, 'FLNC': 2, 'NOISE': 99}

TARGET_MSGS = 56_512_727
CHUNK_MSGS = 27_594
TRACKED_PER_SYMBOL = 4_000


def wrap(payload):
    return struct.pack('>H', len(payload)) + payload


def pad_symbol(stock):
    return stock.encode('ascii').ljust(8, b'␣')[:8]


def build_stock_directory():
    directory = bytearray()
    for stock, locate in SYMBOLS.items():
        payload = struct.pack(
            '>cHH6s8sccIcc2sccccIc',
            b'R',
            locate,
            0,
            b'\x00' * 6,
            pad_symbol(stock),
            b'Q',
            b'␣',
            100,
            b'N',
            b'C',
            b'␣␣',
            b'P',
            b'␣',
            b'␣',
            b'␣',
            b'N',
            0,
            b'N',
        )
        directory.extend(wrap(payload))
    return bytes(directory)
```

```
def build_add(stock, price, side, ref, qty=100):
    payload = struct.pack(
        '>cHH6sQcI8sI',
        b'A',
        SYMBOLS[stock],
        0,
        b'\x00' * 6,
        ref,
        side,
        qty,
        pad_symbol(stock),
        price,
    )
    return wrap(payload)


def build_tracked_messages(count_per_symbol):
    messages = []
    specs = (
        ('RKLB', 0, 50_000, 10_000),
        ('FLNC', 8_192, 20_000, 10_000),
    )
    levels = count_per_symbol // 2

    for level in range(levels):
        for stock, ref_base, price_base, spread in specs:
            improvement = (level * spread) // max(1, levels - 1)
            buy_price = price_base - spread + improvement
            sell_price = price_base + 200 + spread - improvement

            buy_ref = ref_base + (2 * level)
            sell_ref = ref_base + (2 * level) + 1

            messages.append(build_add(stock, buy_price, b'B', buy_ref))
            messages.append(build_add(stock, sell_price, b'S', sell_ref))

    return messages


def build_noise_pair():
    buy = build_add('NOISE', 10_000, b'B', 9_000_000)
    sell = build_add('NOISE', 10_100, b'S', 9_000_001)
    return buy, sell, buy + sell


def write_noise_messages(f, count, noise_index, noise_msgs):
    if count <= 0:
        return noise_index

    buy, sell, pair = noise_msgs

    if noise_index % 2:
        f.write(sell)
        noise_index += 1
        count -= 1

    while count >= CHUNK_MSGS:
        f.write(pair * (CHUNK_MSGS // 2))
        noise_index += CHUNK_MSGS
        count -= CHUNK_MSGS

    if count >= 2:
        pairs = count // 2
        f.write(pair * pairs)
        noise_index += pairs * 2
        count -= pairs * 2

    if count:
        f.write(buy)
        noise_index += 1

    return noise_index


def tracked_positions(total_messages, tracked_count):
    if tracked_count == 0:
        return []
    return [(i * total_messages) // tracked_count for i in range(tracked_count)]


def parse_args():
    parser = argparse.ArgumentParser(
        description='Generate␣a␣large,␣length-prefixed␣ITCH␣binary␣test␣stream.'
    )
    parser.add_argument(
        '-o',
        '--output',
        default=None,
        help='Output␣path.␣Defaults␣to␣sw/userspace/input/full_itch.bin.'
    )
    parser.add_argument(
```

```
        '--messages',
        type=int,
        default=TARGET_MSGS,
        help='Number of Add Order messages after the Stock Directory header.'
    )
    parser.add_argument(
        '--tracked-per-symbol',
        type=int,
        default=TRACKED_PER_SYMBOL,
        help='Balanced Add Order messages to generate for each tracked symbol.'
    )
    return parser.parse_args()


def main():
    args = parse_args()

    if args.tracked_per_symbol <= 0:
        raise SystemExit('--tracked-per-symbol must be positive.')
    if args.tracked_per_symbol % 2:
        raise SystemExit('--tracked-per-symbol must be even so buy/sell counts stay balanced.')
    if args.tracked_per_symbol > 8_192:
        raise SystemExit('--tracked-per-symbol must be <= 8192 to keep refs collision-safe.')

    repo_root = Path(__file__).resolve().parents[1]
    if args.output:
        out_file = Path(args.output).expanduser()
        if not out_file.is_absolute():
            out_file = repo_root / out_file
    else:
        out_file = repo_root / 'sw' / 'userspace' / 'input' / 'full_itch.bin'
    out_file = out_file.resolve()

    os.makedirs(out_file.parent, exist_ok=True)

    directory = build_stock_directory()
    tracked_msgs = build_tracked_messages(args.tracked_per_symbol)
    noise_msgs = build_noise_pair()
    positions = tracked_positions(args.messages, len(tracked_msgs))

    tracked_idx = 0
    msgs_written = 0
    noise_idx = 0

    with open(out_file, 'wb') as f:
        f.write(directory)

        for position, tracked_msg in zip(positions, tracked_msgs):
            noise_count = position - msgs_written
            noise_idx = write_noise_messages(f, noise_count, noise_idx, noise_msgs)
            msgs_written += noise_count

            f.write(tracked_msg)
            tracked_idx += 1
            msgs_written += 1

        remaining = args.messages - msgs_written
        noise_idx = write_noise_messages(f, remaining, noise_idx, noise_msgs)
        msgs_written += remaining

    print(f'Done: {msgs_written} Add messages plus {len(SYMBOLS)} Stock Directory messages.')
    print(f'RKLB/FLNC placed: {tracked_idx} / {len(tracked_msgs)}')
    print(f'NOISE placed: {noise_idx}')


if __name__ == '__main__':
    main()
```