FPGA HFT Order Book Core

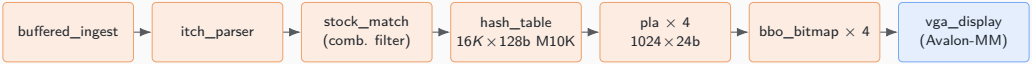
ITCH 5.0 + 16K-slot hash table + hardware BBO @ 100 MHz on DE1-SoC

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.

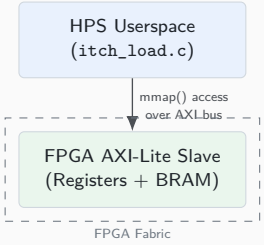
```
altpll #(
  .bandwidth_type ("AUTO"),
  .compensate_clock ("CLK0"),
  .clk0_divide_by (1),
  .clk0_duty_cycle (50),
  .clk0_multiply_by (2),
  .clk0_phase_shift ("0"),
  .inclk0_input_frequency(20000),
  .intended_device_family("Cyclone V"),
  .lpm_type ("altpll"),
  .operation_mode ("NORMAL"),
  .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.

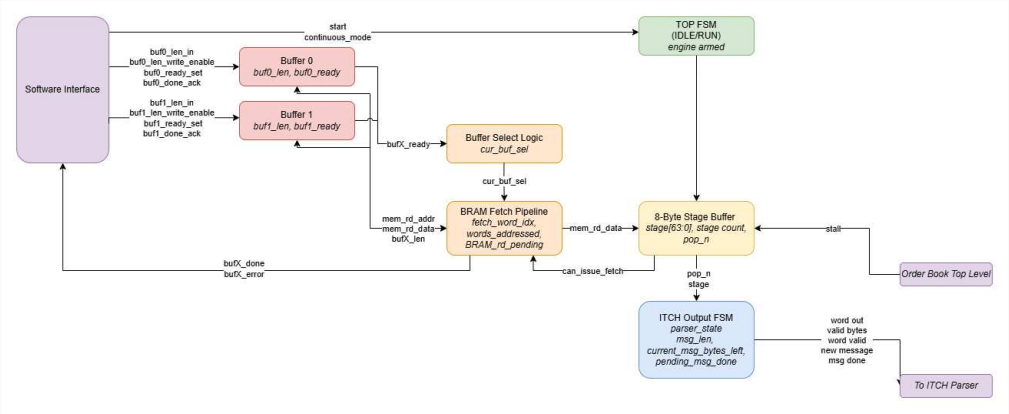
Word interface to the parser.

- `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

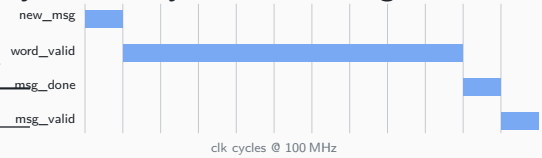


itch_parser — 32-bit word interface, 64-byte buffer

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.

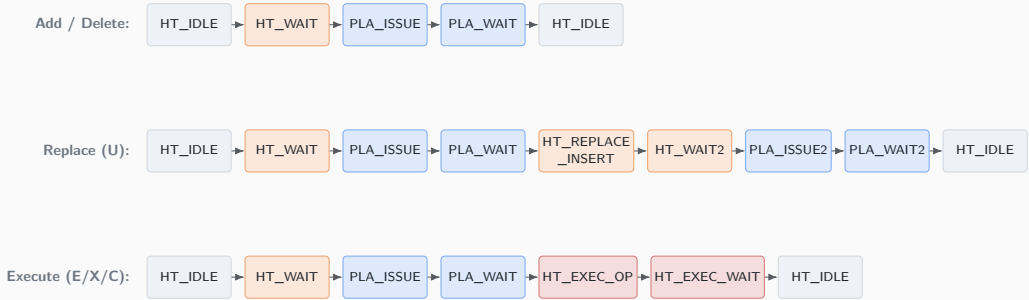
Field	Offset	Decoded for
type	0	all
stock_locate	1..2	all
order_ref	11..18	A/F/E/X/D/U/C
shares	19..22	A/F/E/X/U/C
side	19	A/F (‘B’/‘S’)
price	23 or 32	A/F (23); U new price (32)
new_order_ref	19..26	U

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

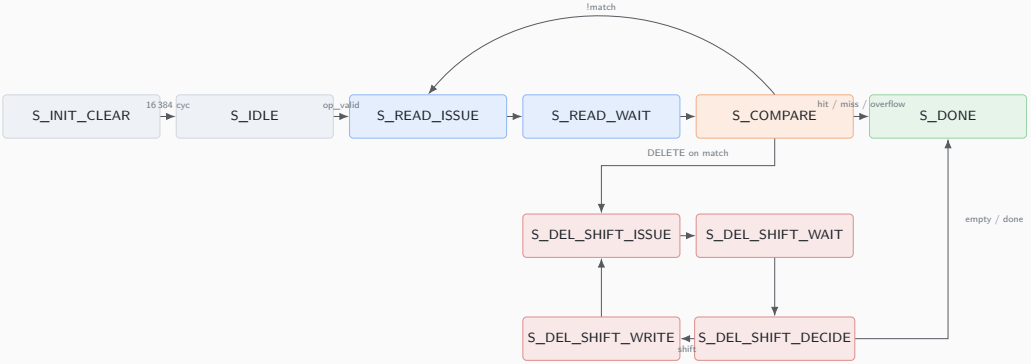
- 16 384 slots, 128 bits each, M10K.
- Hash = 14-bit XOR fold of `order_ref`: `ref[13:0] ^ ref[27:14] ^ ref[41:28]`.
- Linear probing, `MAX_CHAIN`=64; overflow pulses on exceed.
- 1-cycle BRAM read; the FSM bridges with `S_READ_ISSUE` → `S_READ_WAIT` → `S_COMPARE`.
- At reset, `S_INIT_CLEAR` walks all 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

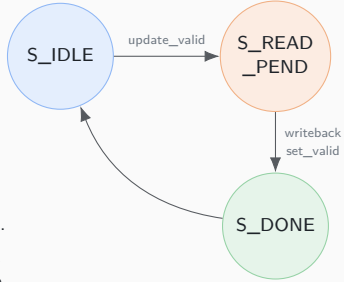
4 banks = 2 stocks × {bid, ask}. Each bank is 1024 × 24b in M10K (`(* ramstyle="M10K" *)`). Bucket = `((price − PRICE_BASE) / 100)[9:0]` — ITCH price is in $0.0001, /100 gives 1-cent buckets; Quartus emits multiply-by-reciprocal. HPS sets `PRICE_BASE = first_price − $5.12` so the live quote sits mid-window.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

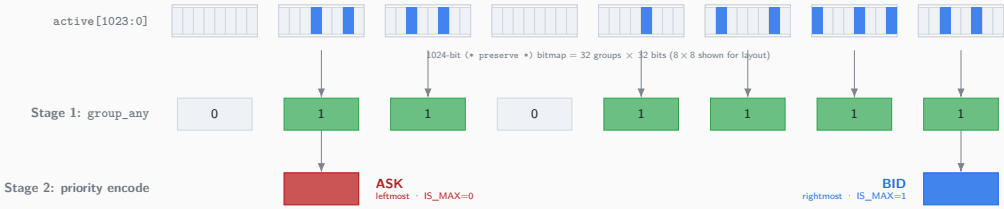
FSM (3 states)



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

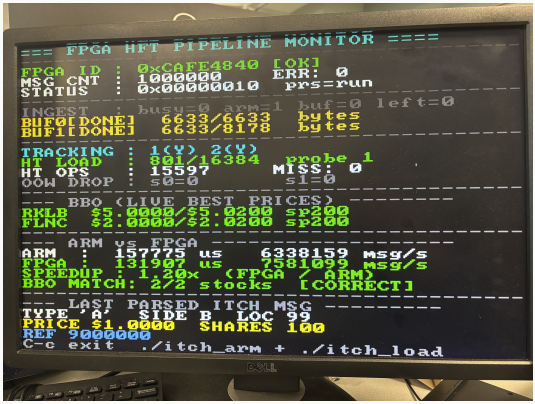
Details in Appendix

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.



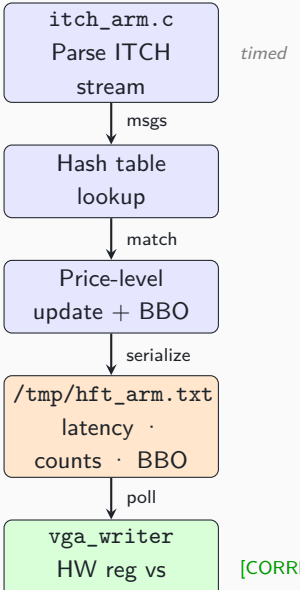
The rasterizer bridges the dual-port AXI-Lite BRAM to the VGA output region in real-time, requiring no clock-domain crossings as both the HPS bridge and VGA pixel clock (derived) share the fabric clock.

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

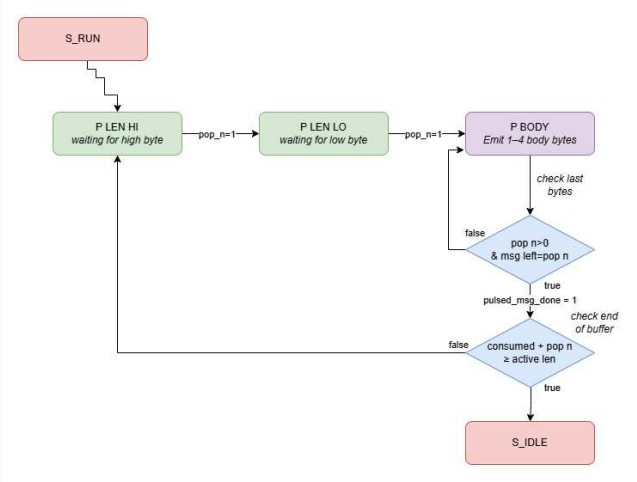
```
// 1. Tiling: convert pixel (x,y) to tile (col,row) by shifting right 4 (/16)
logic [5:0] char_col = pixel_x[9:4];
logic [4:0] char_row = pixel_y[8:4];

// 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
```