A Simple HFT Simulator and Memory Management System on an FPGA

Design Document

CSEE 4840 Embedded System Design

Spring 2025

Jayden Lee-Sin (jcl2257)
Carlos Espinoza (cre2121)
Derrick Bassey (dab2266)

1. Introduction

The goal of this project is to develop a high frequency trading (HFT) simulator on an FPGA. It leverages virtual memory to handle multiple symbol order books simultaneously through dynamic resource allocation. The hardware subsystem performs order matching, memory management with page-table-based address translation, and hazard detection, while the software subsystem drives simulation data into the FPGA and collects performance metrics.

The system is decomposed into four principal components:

The HFT System implements per-symbol order books using binary heap based priority queues in hardware. A max-heap holds bids and a min-heap holds asks for each symbol. When the best bid meets or exceeds the best ask, a trade executes. In addition, we support partial fills, and trading any amount of a given symbol.

The Memory Management System provides virtualized address spaces so each symbol believes it owns all available memory. A supervisor translates virtual addresses to physical frame addresses (def frame: a group of addresses in physical memory) addresses using a page table, allocates and frees frames via a bitmap.

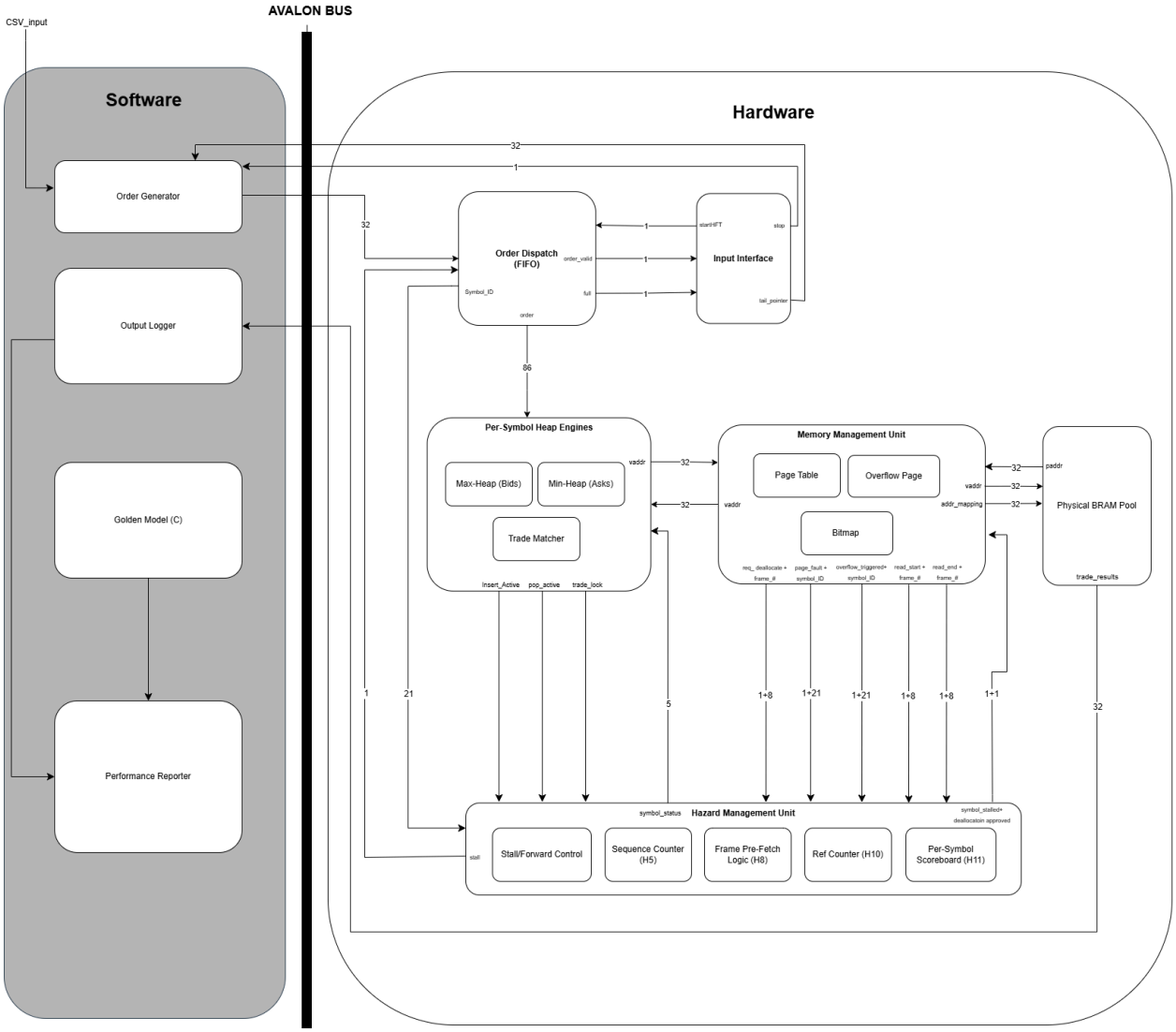
The Hazard Management System monitors the pipeline for data hazards (e.g. stale reads during heap operations), resource contention, and cross-system conflicts. It enforces correctness through per-symbol stalling, forwarding registers, and lock flags.

The Software Harness, runs on the HPS ARM processor and serves as both the data source (generating encoded order streams) and the data sink (collecting trade confirmations, timing data, and hazard logs). Additionally, a C program serves as the golden model, providing a reference for verification of the HFT system.

The target platform is the Intel DE1-SoC, which provides the Cyclone V FPGA, HPS ARM Cortex-A9 processor, and on-chip BRAM. All heap storage resides in BRAM and the HPS communicates with the FPGA fabric over the Avalon memory-mapped bus.

2. System Block Diagram

The diagram below illustrates the major hardware and software components and the communication pathways between them. The hardware side contains the order dispatch unit, per-symbol heap engines, the memory management unit (MMU) with page table and bitmap allocator, and the hazard management unit with the per-symbol scoreboard. The software side (HPS) contains the order generator, output logger, golden model, and performance reporter.



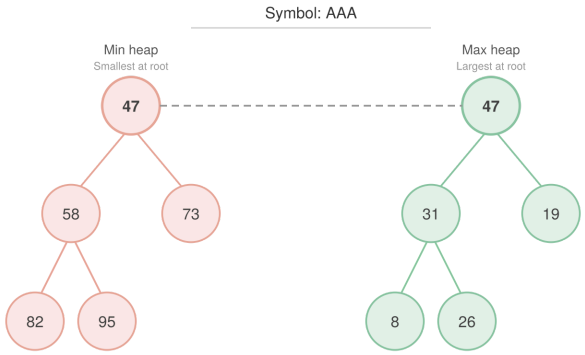
HPS ↔ FPGA (Avalon Memory-Mapped Bus): The software harness writes 3 32-bit words (86 bits total) to the input register and reads 32-bit words from the output register. A ready/valid handshake governs transfers: the harness asserts valid when data is available, the hardware asserts ready when it can accept, and a transfer occurs only when both are high.

3. Algorithms & Hazards

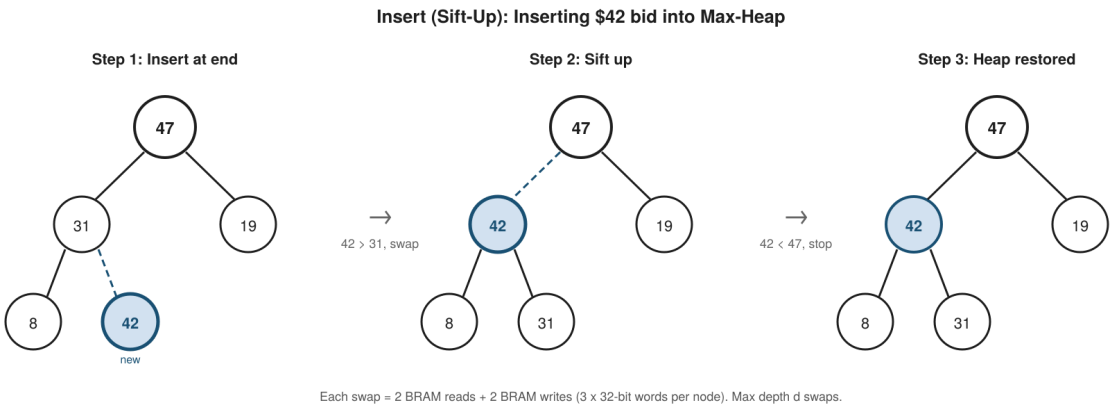
3.1 Binary Heap Operations

Each symbol maintains two binary heaps stored in BRAM: a max-heap for bids (highest price at root) and a min-heap for asks (lowest price at root). Each node contains a price (16 bits), quantity (16 bits), type flag (1 bit), symbol identifier (21 bits), and a timestamp (32 bits), totaling 86 bits per node stored in 3 consecutive 32-bit words.

The heap visualization below shows the basic structure for a single symbol, with the min-heap (asks, sellers in orange) on the left and the max-heap (bids, buyers in green) on the right. The line connecting the roots indicates a trade being executed.



The heap supports five operations: insert (sift-up), peek, pop (sift-down), edit (quantity update), and reorder. The following figure illustrates the insert operation:



Insert (Sift-Up): A new order is written to the next available slot. The node is compared with its parent and swapped upward until the heap property is restored. For a heap of depth d, this takes at most d swaps. Each swap requires two BRAM reads and two BRAM writes.

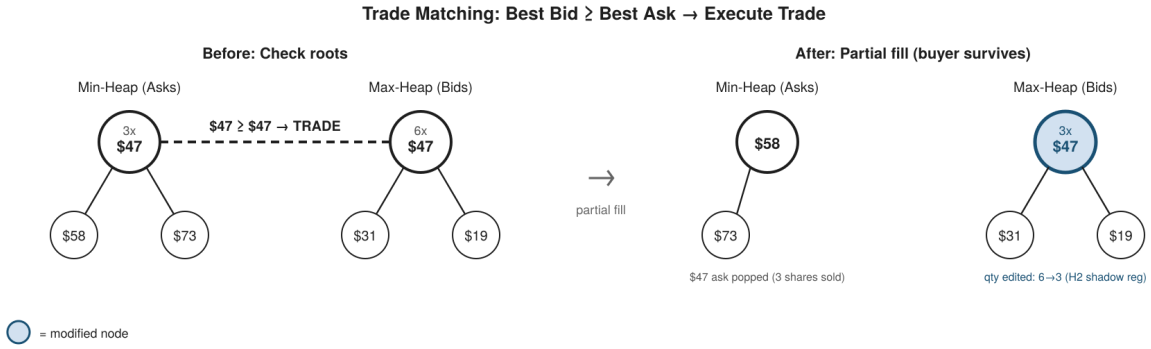
Peek: The trade matcher reads the root (index 0) of both heaps. With root caching (see Hazard H1), this is a register read with zero memory access.

Pop (Sift-Down): The root is removed by replacing it with the last element, then bubbling down by comparing with children and swapping with the higher-priority child. Also takes at most d swaps.

Edit (Quantity Update): Used during partial fills to reduce the surviving root node's quantity in place. Since only quantity changes (not price), no reordering is needed. This is a single BRAM write.

3.2 Trade Matching

The trade matcher operates every clock cycle per symbol. It compares the root of the max-heap (best bid) against the root of the min-heap (best ask). If the best bid price is greater than or equal to the best ask price, a trade executes at the ask price. The following figure illustrates a partial fill:



Fill types: if bid quantity equals ask quantity, both roots are popped (full fill). If bid quantity exceeds ask quantity, the ask root is popped and the bid's quantity is reduced (partial fill, buyer survives). If the ask quantity exceeds the bid quantity, the reverse occurs. The trade confirmation is written to the output register for the software harness.

3.3 Timestamp Tie-Breaking

When two nodes share the same price, priority is given to the earlier timestamp (first-come-first-served). Timestamps are 32-bit values assigned by the FPGA upon receipt. To handle same-cycle ties, a global 8-bit sequence counter is appended, guaranteeing unique total ordering.

3.4 Virtual Memory Translation

Each symbol operates in its own virtual address space. A virtual address is composed of a virtual page number (VPN) and a page offset. The page offset indexes into a 64-node page (6 bits). The VPN is looked up in the page table, which is stored in BRAM indexed by the concatenation of the symbol ID and VPN. Each entry stores the physical frame number (PFN) and a valid bit. The page table lookup adds one cycle of latency to every memory access.

3.5 Page Allocation and Deallocation

The free-frame bitmap is a register where each bit represents one physical frame. A priority encoder finds the first set bit (free frame) in a single cycle. Allocation clears that bit, deallocation sets it back. If two requests arrive simultaneously, one is serviced immediately and the second is queued.

Three strategies handle memory exhaustion: a per-symbol cap of 128 pages, a shared overflow page with a FIFO write arbiter, and finally a hard reject that drops the order.

3.7 Hazard Detection and Resolution

The hazard management system monitors and intervenes across the entire pipeline. Hazards are categorized into HFT hazards, memory management hazards, and cross-system hazards.

ID	Hazard	Location	Solution	Latency (cycles)
H1	Stale root during heap sift-up	HFT	Speculative root forwarding register	0
H2	Stale quantity during partial fill	HFT	Quantity shadow buffer (store queue)	0
H3	Duplicate trade match on pending root	HFT	Trade-lock flag per heap	0
H4	Simultaneous insert + pop on same heap	HFT	Pop-priority serialization w/ input buffer	0–1
H5	Same-cycle timestamp tie	HFT	Sub-cycle global sequence counter	0
H6	Bitmap double-claim on concurrent alloc	Memory	Priority encoder allocator	0–1
H7	Overflow page write contention	Memory	FIFO write arbiter (4–8 entries)	Buffered
H8	Page fault mid-insert	Memory	Prefetch allocation at 75% watermark	0 (on hit)
H9	Compaction during active trade	HFT/MEM	Per-symbol compaction lock	0 (others)
H10	Deallocation during active read	HFT/MEM	Delayed free w/ reference counter	1–2
H11	Per-symbol stall implementation	HFT/MEM	Per-symbol scoreboard	0 (others)
H12	Software harness overrun	Interface	Ready/valid handshake	Buffered

HFT System Hazards

H1: Stale Root Read During Heap Sift-Up (RAW). A new order is inserted and begins bubbling through the heap. Before it settles, the trade matcher peeks at the root and sees the old value. If the in-flight node would have become the new root, the trade matcher either misses a valid trade or evaluates against an outdated price.

Solution: Speculative Root Forwarding: Maintain a single candidate root register per heap. When an insert begins, combinational logic immediately compares the new value against the current root. If the new value would win (higher for max-heap, lower for min-heap), the candidate register is updated that same cycle, before the sift-up starts. The trade matcher always reads from this register. Cost: one register, one comparator, and one mux per heap. Zero extra latency.

Resources Needed: One 86-bit candidate root register per heap (stores the full node: price, quantity, type, symbol, timestamp). One comparator per heap that compares the incoming node's price and timestamp against the current root (at minimum 48 bits wide: 16b price + 32b timestamp). One mux per heap on the trade matcher's read path to select between the actual BRAM root and the candidate register. Per symbol this is 2 registers, 2 comparators, 2 muxes (one for bid heap, one for ask heap). Total: 172 bits of register per symbol.

Difficulty: Low.

H2: Stale Quantity Read During Partial Fill Write-Back (RAW). When a partial fill occurs, the trade matcher reduces the quantity of the surviving node. This write-back takes at least one cycle. If a new ask arrives during that window and reads the old quantity, a second trade may overfill the buyer.

Solution: Forwarding with Quantity Buffer: When a partial fill begins, the updated quantity is written to a dedicated buffer in the same cycle that the trade is confirmed. Any subsequent read targeting that node checks the buffer first; if the address matches, the buffered value is forwarded directly. Once the write-back commits, the buffer clears. Same principle as a store queue in a CPU pipeline.

Resources Needed: One 16-bit quantity register per heap (stores the updated quantity). One address tag per heap to identify which node the buffer is holding (this is the heap index of the edited node, which requires 13 bits (7-bit virtual page number + 6-bit page offset)). One valid bit per heap indicating the buffer is active. One 13-bit comparator per heap on the read path that checks whether an incoming read's target address matches the buffered address. One mux per heap to forward the buffered quantity when the comparator hits. Per symbol: 2 buffers, total of 60 bits of register per symbol (2 × (16b quantity + 13b tag + 1b valid)).

Difficulty: Low-medium. The register is simple, but snooping logic adds integration complexity across the heap read path.

H3: Duplicate Trade Match on Root Pending Removal. The trade matcher runs every cycle. Once a trade is detected, a pop begins. The pop takes multiple cycles (sift-down). On the next cycle the matcher sees the same root and fires a duplicate trade.

Solution: Trade-Lock Flag Per Heap: Set a single-bit trade-in-progress flag when a trade is initiated. The trade matcher skips any heap with this flag set. Clear it once the pop completes. Cost: one flip-flop and one AND gate per heap.

Resources Needed: One 1-bit flip-flop per heap. One AND gate per heap in the trade matcher's comparison logic (trade is valid only when neither heap's lock flag is set). Per symbol: 2 flip-flops, 2 AND gates. Total: 2 bits of register per symbol.

Difficulty: Very low.

H4: Simultaneous Insert and Pop on Same Heap. If an insert (sift-up) and a pop (sift-down) target the same heap simultaneously, both move nodes and can corrupt the heap structure.

Solution: Priority Operation Serialization: Give pops strict priority since they result from confirmed trades. If both arrive in the same cycle, buffer the insert in a small per-heap input register for one cycle. At most one cycle of latency on the insert, far cheaper than dual-ported heap logic.

Resources Needed: One 86-bit input buffer register per heap (stores the full order node waiting to be inserted). One 1-bit valid flag per heap indicating the buffer is occupied. A small FSM or priority signal per heap (2–3 bits of state) that arbitrates:

if pop is active and insert arrives, latch insert into buffer and assert valid;

when pop finishes, drain the buffer.

Per symbol: 2 buffers, total of 178 bits of register per symbol (2 × (86b node + 1b valid + 2b FSM state)).

Difficulty: Low.

H5: Ambiguous Priority When Same-Cycle Orders Share Price and Timestamp. Two orders arrive in the same clock cycle with identical prices. The oldest-first tie-breaking rule becomes ambiguous.

Solution: Sub-Cycle Sequence Counter: Append a small 4–8 bit sequence number to each timestamp via a global counter that increments with every accepted order. Comparison logic checks price first, then timestamp+sequence. Collisions are impossible in practice.

Resources Needed: One 8-bit global counter (increments every time an order is accepted, wraps around). The heap comparator must be widened to include the sequence field — comparison becomes 56 bits (16b price + 32b timestamp + 8b sequence) instead of 48 bits (16b price + 32b timestamp). This is a one-time change to the comparator logic, not per-symbol. Total: 8 bits of global register, plus slightly wider comparators in every heap.

Difficulty: Very low.

Memory Management Hazards

H6: Two Symbols Claiming the Same Free Frame in the Same Cycle. Two symbols need new pages simultaneously. Both read the bitmap, both see the same free bit, both claim it. One allocation silently overwrites the other.

Solution: Allocator via Priority Encoder: A priority encoder returns the first free bit. Find + clear completes in a single cycle. If two requests arrive simultaneously, one is serviced and the other queued for the next cycle. This would cause a one-cycle delay, but would ensure the value we get in the end is correct.

Resources Needed: One 256-bit priority encoder on the free-frame bitmap. Could be implemented as 8-bit sub-encoders feeding into a second level to meet timing. One 1-entry request queue (21-bit symbol ID + 7-bit VPN = 28 bits) to hold the second request when two

arrive simultaneously. Control logic (2–3 bit FSM) to manage the queue drain on the next cycle. Total: 256-bit encoder, 28 bits of queue register, 3 bits of FSM state.

Difficulty: Low-medium. Priority encoders are well-understood, but a wide bitmap may need a tree-structured encoder to meet timing.

H7: Multiple Symbols Writing to the Shared Overflow Page in the Same Cycle. Multiple symbols hit their page limit simultaneously and all try to write to the shared overflow page, corrupting data.

Solution: Overflow Write Arbiter with FIFO: Place a 4–8 entry FIFO in front of the overflow page. Symbols enqueue writes, the arbiter drains one per cycle. If the FIFO fills, hard reject triggers. The FIFO buys extra cycles for trades to free memory to avoid dropping orders.

Resources Needed: One FIFO buffer, 4–8 entries deep, each entry 86 bits wide. FIFO control logic: read pointer, write pointer, and full/empty flags (approximately 8–10 bits of state for an 8-entry FIFO). One round-robin or priority arbiter FSM (3–4 bits of state) that selects which symbol enqueues when multiple requests happen simultaneously. One 1-bit hard reject signal output, asserted when the FIFO full flag is set and the overflow page itself has no remaining capacity. Total: 688 bits of register for an 8-entry FIFO (8 × 86b), plus approximately 15 bits of control state.

Difficulty: Medium.

H8: Heap Insert Stalling on Synchronous Page Allocation. An insert causes the heap to grow beyond allocated pages. Allocation takes at least one cycle, leaving the insert stuck mid-operation.

Solution: Prefetch Allocation: Track page fullness per symbol. When a page crosses 75% capacity, proactively allocate the next frame in the background. The prefetched frame sits in a standby register per symbol. When the page fills, the new frame is already mapped.

Resources Needed: One 6-bit page fullness counter per active symbol (counts 0–64 nodes in the current page). One threshold comparator per symbol (checks if counter > 48, which is 75% of 64). One 8-bit standby frame register per symbol (stores the prefetched physical frame number). One 1-bit valid flag per symbol indicating a prefetched frame is available. Background allocation control logic shared across symbols (FSM, approximately 4 bits of state) that issues prefetch requests to the bitmap allocator during idle cycles. Per symbol: 15 bits of register (6b counter + 8b standby PFN + 1b valid).

Difficulty: Medium-high.

Cross-System Hazards

H9: Frame Freed by Deallocation While Another Module Is Still Reading. A pop removes the last node from a page, triggering deallocation, but a concurrent insert is still reading from that page.

Solution: Delayed Free with Reference Counter: Each frame gets a 2–3 bit saturating counter tracking active readers. Increment on read start, decrement on completion. The memory manager only frees when the counter reaches zero. Typically a 1–2 cycle delay.

Resources Needed: One 2-bit saturating counter per physical frame (supports up to 3 concurrent readers, which is sufficient since at most one insert and one pop can read the same frame simultaneously). 256 counters total, one per frame, consuming 512 bits of register. Increment and decrement control signals routed from every module that issues BRAM reads. One zero-detect comparator per frame (2-input NOR on the 2-bit counter) that feeds the deallocation-approved signal back to the MMU.

Difficulty: Medium.

H10: Per-Symbol Stall Implementation. The hazards above require stalling individual symbols without blocking the entire pipeline. This hazard entry describes the centralized mechanism that makes per-symbol stalling work.

Solution: Per-Symbol Scoreboard: A small register file with one entry per active symbol. Each entry has status bits: inserting, popping, page-fault pending, compacting, overflow. The pipeline checks the scoreboard before dispatching. If the target symbol is blocked, only that operation stalls. Similar to a reservation station in an out-of-order processor.

Resources Needed: A register file with one 5-bit entry per active symbol (inserting, popping, page-fault pending, compacting, overflow). For up to 8 active symbols this is 40 bits of register. One read port for the dispatch unit (indexed by symbol ID, returns 5 status bits). Write access from heap engines (set/clear inserting and popping bits),and from the MMU (set/clear page-fault and overflow bits). OR together all 5 status bits per entry to produce a single "symbol stalled" signal for the dispatch unit. If using the full 21-bit symbol ID, a small CAM or lookup table (3-bit index × 8 entries) maps symbol IDs to scoreboard indices. Total: 40 bits of status register + 168 bits of symbol-to-index mapping (8 entries × 21 bits) = 208 bits.

Difficulty: Medium-high.

H11: Software Harness Sending Orders Faster Than Hardware Can Accept. Without backpressure, orders are silently dropped or written into memory that is not ready, causing corruption.

Solution: Ready/Valid Handshake: The harness asserts valid when data is available. The hardware asserts a ready signal when it can accept. A transfer occurs only when both are high. The harness waits when ready drops.

Resources Needed: One 1-bit ready output register (driven by the dispatch unit, deasserted when the pipeline is full or a stall is active). One 1-bit valid input (directly from the Avalon bus write signal). One AND gate that gates the data latch: order data is latched into the input interface only when ready AND valid are both high. Total: 1 bit of register, 1 AND gate.

Difficulty: Very low.

4. Resource Budgets

4.1 Order Node Sizing

Field	Width	Range
Type (Ask/Bid)	1 bit	0 or 1
Price	16 bits	0 – 65,536
Quantity	16 bits	0 – 65,536
Symbol	21 bits	3 ASCII uppercase (7b each)
Timestamp	32 bits	0 – 4,294,967,296
Total	**86 bits**	

4.2 BRAM Word Packing

Using 32-bit words: 86/32 = 2.69, rounded up to 3 words per node (96 bits, 10 unused). Each BRAM block holds 256 words, yielding 85 nodes per block. For clean addressing we use 64 nodes per page (192 words per page).

4.3 Memory Capacity

Parameter	Value
Total BRAM blocks available	397 (Cyclone V)
Blocks reserved for non-heap (page table, scoreboard, etc.)	10
Blocks available for heaps	387
Nodes per page	64
Total pages (clean address decoding)	256
Total node capacity	**16,384 nodes**
Max pages per symbol	128 (half of total)
Overflow pool	1 page (64 nodes)

4.4 Address Bit Widths

Component	Width	Derivation
Page offset (node index)	6 bits	log2(64) = 6
Virtual page number	7 bits	log2(128) max pages per symbol
Physical frame number	8 bits	log2(256) total pages
Word offset within node	2 bits	3 words per node, rounded to 4
Free-frame bitmap width	256 bits	1 bit per physical frame

4.5 Hazard Unit Resource Estimates

Component	Size Per Instance	Instances
Root candidate register	86 bits	2 per symbol (bid + ask)
Quantity shadow buffer	16 bits + address tag	2 per symbol
Trade-lock flag	1 bit	2 per symbol
Insert buffer register	86 bits	2 per symbol
Sequence counter	8 bits	1 global
Scoreboard entry	5 status bits	1 per active symbol
Reference counter per frame	2–3 bits	256 (one per frame)
Overflow FIFO	86 bits per entry	4–8 entries (1 global)
Page fullness counter	6 bits	1 per active symbol

5. Hardware/Software Interface

The HPS ARM processor communicates with the FPGA fabric through memory-mapped registers accessible via the Avalon bus.

5.1 Input Order Register (Write, 64 bits)

Bits	Field	Width	Description
[0]	type	1	0 = Bid, 1 = Ask
[16:1]	price	16	Order price (unsigned)
[32:17]	quantity	16	Number of shares
[53:33]	symbol	21	3 ASCII uppercase letters, 7 bits each
[63:54]	reserved	10	Reserved, write as 0

5.2 Control/Status Register (Read/Write, 32 bits)

Bits	Field	Access	Description
[0]	fifo_tail	R	Integers indicating index of FIFO tail
[1]	fifo_stop	R	1 = indicating FIFO is full
[2]	trade_avail	R	1 = trade confirmation available in output register
[3]	trade_ack	W	Set to 1 to acknowledge receipt of trade data
[4]	sim_active	R	1 = simulation running
[5]	sim_start	W	Write 1 to begin simulation

[6]	sim_reset	W	Write 1 to reset all hardware state
[7]	overflow_flag	R	1 = overflow pool accessed since last reset
[8]	reject_flag	R	1 = at least one order hard-rejected
[31:9]	reserved	—	Reserved

5.3 Trade Output Register (Read, 64 bits)

Bits	Field	Width	Description
[15:0]	exec_price	16	Trade execution price
[31:16]	exec_qty	16	Number of shares traded
[52:32]	symbol	21	Symbol identifier
[53]	fill_type	1	0 = full fill, 1 = partial fill
[63:54]	reserved	10	Reserved

5.4 Performance Counter Registers (Read, 32 bits each)

Offset	Name	Description
0x10	cycle_count	Total clock cycles since sim_start
0x14	trade_count	Total trades executed
0x18	mem_reads	Total BRAM read operations
0x1C	mem_writes	Total BRAM write operations
0x20	hazard_stalls	Total cycles lost to hazard stalls
0x24	hard_rejects	Number of orders hard-rejected

5.5 Software Harness Protocol

1. Write 1 to *sim_reset*. Wait for *sim_active* to read 0.
2. Write 1 to *sim_start*.
3. For each order:
    a. Stall harness until *ready* is set to 1.
    b. Write 64-bit order
    c. Set valid_in to 1.
4. Concurrently continuously read *trade_avail*. When 1, read trade output, log it, write *trade_ack*.
5. After all orders are submitted, continue reading *trade_avail* until there are no more trades.
6. Read performance counters. Print final report.

5.6 Driver Interface

The order data structure in software mirrors the hardware register layout:

```
C/C++
#include <linux/types.h>

typedef struct {
        __u8 type:1;   // 1 = Ask, 0 = Bid
        __u8 symbol[3]; // 3 chars
        __u16 price;
        __u16 amount;
        __u32 timestamp;
} Order;
```

Please note that the following C code is preliminary and multiple things have been omitted or abstracted to make it easier to understand. *This code is bound to change as we make realizations about the hardware we are developing.*

Symbol Byte Encoding

Each symbol is three ASCII uppercase characters (7 bits each, 21 bits total). The MSB of each symbol byte is repurposed to carry control signals, avoiding the need for separate control register writes per field:

symbol[0] MSB: Validity bit. Set to 1 when the full order word is ready for the hardware to latch.

symbol[1] MSB: Type bit. Carries the ask/bid flag (1 = Ask, 0 = Bid).

symbol[2] MSB: Flip-bit for transfer detection. Toggled on each new order so the hardware can detect a new write even if the data is identical to the previous order.

Since ASCII uppercase letters (A–Z = 0x41–0x5A) never use bit 7, packing control signals into the MSBs is safe and eliminates extra register writes.

Register Write

The Linux driver on the HPS side communicates with the FPGA hardware registers using memory-mapped I/O calls (iowrite/ioread):

```
C/C++
void __iomem *fifo_tail_base;
void __iomem *ctrl_base;

// Write offsets
#define META      0    // Valid + Type + Flip + Symbols
#define PRC_AMT   1    // Price + Ammount
#define TIMESTAMP 2    // Time Stamp

// Read offset
#define CTRL_STATUS_W  0   // temp
```

```
#define CTRL_TAIL_W    1

#define REG32(base, w) ((void __iomem *)((u8 __iomem *)(base) + ((w) * 4)))
#define STOP_MASK         (1u << 0)

// Read stop signal
static inline bool read_stop(void __iomem *ctrl_base){
    u32 st = ioread32(REG32(ctrl_base, CTRL_STATUS_W));
    return (st & STOP_MASK) != 0;
}

// Read tail idex of FIFO
static inline u32 read_tail_index(void __iomem *ctrl_base){
    return ioread32(REG32(ctrl_base, CTRL_TAIL_W)); // Unsigned int
}

// Write order as a set of 12 bytes
static void write_order(void __iomem *fifo_tail_base, const Order *o){
        // Pack Symbols, Type, Valid bit and Flip-bit
        __u8 b0 = ((1 & 0x1) << 7) | (o->symbol[0] & 0x7f);
        __u8 b1 = ((o->type  & 0x1) << 7) | (o->symbol[1] & 0x7f);
        __u8 b2 = ((0 & 0x1) << 7) | (o->symbol[2] & 0x7f);
        __u8 b3 = 0;
        __u32 meta = ((__u32)b0) |
                 ((__u32)b1 << 8) |
                 ((__u32)b2 << 16) |
                 ((__u32)b3 << 24);

       // Pack Price and Ammount
       __u32 prc_amt = ((__u32)o->price) | ((__u32)o->amount << 16);

       //  Write to buffers
       iowrite32(meta, REG32(fifo_tail_base, META));
              iowrite32(prc_amt, REG32(fifo_tail_base, PRC_AMT));
              iowrite32(o->timestamp, REG32(fifo_tail_base, TIMESTAMP));
}

static bool write_order_and_wait_tail_advance(void __iomem *fifo_tail_base,
                                              void __iomem *ctrl_base,
                                              const Order *o){
    u32 tail_before = read_tail_index(ctrl_base);

    // Write data to tail
    write_order(fifo_tail_base, o);

    // Wait for hardware to update tail or raises stop signal
    for (int i = 0; i < 1000000; i++) {
        if (read_stop(ctrl_base))
            return false;

        u32 tail_after = read_tail_index(ctrl_base);
```

```
        if (tail_after != tail_before)
            return true;
    }
    return false; // timed out (i.e. write never committed)
}
```

Simulation Loop

The simulation reads orders from a pre-generated CSV file and writes them to the Order Dispatch in hardware, one at a time, until the FIFO is filled (indicated by a stop signal):

```C/C++
FILE *fp = fopen("data.csv", "r");
if (!fp) {
       // handle error
       return;
}

// Read line by line
char line[256];
while (!read_stop() && fgets(line, sizeof(line), fp)) {
       // Check if FIFO is full
       if (read_stop(ctrl_base))
              break;

       // Parse line
       // Function assumed to handle errors
       // and invalid data
       Order o;
       parse_csv_line(line, &o);

       // Write to fifo
       if (!write_order_and_wait_tail_advance(fifo_tail_base, ctrl_base, &o))
        break;  // stop raised or timeout
}
fclose(fp);
```

Control & Trade Register Reads

Note that while the code below mostly focuses on the readings of the control and trade registers, there are also some writes being performed.

C/C++

```
// Control/Status offsets
#define CTRL_STATUS_W        0  // temp

// CTRL_STATUS Masks (not offsets)
#define READY_MASK           (1u << 0)
#define TRADE_AVAIL_MASK     (1u << 1)
#define TRADE_ACK_MASK       (1u << 2)  // W
#define SIM_ACTIVE_MASK      (1u << 3)
#define SIM_START_MASK       (1u << 4)  // W
#define SIM_RESET_MASK       (1u << 5)  // W
#define OVERFLOW_MASK        (1u << 6)
#define REJECT_FLAG_MASK     (1u << 7)

// Trade output register offsets
#define TRADE_OUT_LO_W       0
#define TRADE_OUT_HI_W       1

typedef struct {
    Order ord;
    __u8  fill_type;   // 0 = full fill, 1 = partial fill
} TradeConfirm;

// Read for trade available
static bool read_trade_avail(void __iomem *ctrl_base){
    u32 st = ioread32(REG32(ctrl_base, CTRL_STATUS_W));
    return (st & TRADE_AVAIL_MASK) != 0;
}

// Read for sim active signal
static bool read_sim_active(void __iomem *ctrl_base){
    u32 st = ioread32(REG32(ctrl_base, CTRL_STATUS_W));
    return (st & SIM_ACTIVE_MASK) != 0;
}

// Ack trade after reading trade output reg
// Pulse for now
static void write_trade_ack(void __iomem *ctrl_base){
    iowrite32(TRADE_ACK_MASK, REG32(ctrl_base, CTRL_STATUS_W));
    iowrite32(0,              REG32(ctrl_base, CTRL_STATUS_W));
}

// Reads and parse the 64-bit trade output (LO/HI).
static TradeConfirm read_trade(void __iomem *trade_out_base){
    TradeConfirm t;
```

```
    // Read raw 64-bit trade value from the two 32-bit regs
    u32 lo = ioread32(REG32(trade_out_base, TRADE_OUT_LO_W));
    u32 hi = ioread32(REG32(trade_out_base, TRADE_OUT_HI_W));
    __u64 raw = ((__u64)hi << 32) | lo;

    // Parse fields from raw
    __u16 exec_price = (raw >> 0)  & 0xFFFF;
    __u16 exec_qty   = (raw >> 16) & 0xFFFF;
    __u32 sym21      = (raw >> 32) & 0x1FFFFF;  // 21-bit symbol (3x7-bit)
    __u8  fill_type  = (raw >> 53) & 0x1;

    t.ord.type = 0;        // Not needed here
    t.ord.price = exec_price;
    t.ord.amount = exec_qty;
    t.ord.timestamp = 0;   // Not needed here, again
    // Note that we have no null terminator here:
    // Add it before printing
    t.ord.symbol[0] = (sym21 >> 0)  & 0x7F;
    t.ord.symbol[1] = (sym21 >> 7)  & 0x7F;
    t.ord.symbol[2] = (sym21 >> 14) & 0x7F;
    t.fill_type = fill_type;
    return t;
}

// Wait until a trade is available, read it, and ack it
static bool read_trade_and_ack(void __iomem *ctrl_base, void __iomem
                               *trade_out_base,TradeConfirm *out){
    for (int i = 0; i < 1000000; i++) {
        if (!read_sim_active(ctrl_base))
            return false;

        if (read_trade_avail(ctrl_base)) {
            *out = read_trade(trade_out_base);
            write_trade_ack(ctrl_base);
            return true;
        }
    }
    return false; // timed out
}
```

Performance Measurements

C/C++

```
#define PERF_CYCLE_W         0
#define PERF_TRADE_W         1
#define PERF_MEM_READS_W     2
#define PERF_MEM_WRITES_W    3
#define PERF_HAZARD_W        4
#define PERF_HARD_REJ_W      5

// Struct for counters
typedef struct {
    u32 cycle_count;
    u32 trade_count;
    u32 mem_reads;
    u32 mem_writes;
    u32 hazard_stalls;
    u32 hard_rejects;
} PerfCounters;

// Read counters only
static inline PerfCounters perf_read(void __iomem *perf_base){
    PerfCounters p;
    p.cycle_count   = ioread32(REG32(perf_base, PERF_CYCLE_W));
    p.trade_count   = ioread32(REG32(perf_base, PERF_TRADE_W));
    p.mem_reads     = ioread32(REG32(perf_base, PERF_MEM_READS_W));
    p.mem_writes    = ioread32(REG32(perf_base, PERF_MEM_WRITES_W));
    p.hazard_stalls = ioread32(REG32(perf_base, PERF_HAZARD_W));
    p.hard_rejects  = ioread32(REG32(perf_base, PERF_HARD_REJ_W));
    return p;
}

static void report_end_of_sim(void __iomem *ctrl_base, void __iomem *perf_base){
    // Read status and performance metrics
    u32 st = ioread32(REG32(ctrl_base, CTRL_STATUS_W));
    PerfCounters p = perf_read(perf_base);

    // Print values
    printf("=== END OF SIM REPORT ===\n");
    fprintf("status=0x%08x overflow=%u reject_flag=%u\n",
            st,
            !!(st & OVERFLOW_MASK),
            !!(st & REJECT_FLAG_MASK));

    fprintf("cycle_count   : %u\n", p.cycle_count);
    fprintf("trade_count   : %u\n", p.trade_count);
```

```
    fprintf("mem_reads     : %u\n", p.mem_reads);
    fprintf("mem_writes    : %u\n", p.mem_writes);
    fprintf("hazard_stalls : %u\n", p.hazard_stalls);
    fprintf("hard_rejects  : %u\n", p.hard_rejects);

    if (p.trade_count)
        fprintf("avg cycles/trade: %u\n", p.cycle_count / p.trade_count);
}
```

6. Software Golden Model

The software golden model is a C implementation of the same order matching and memory management logic that runs on the FPGA. It runs on the HPS ARM processor and processes the same input data as the hardware pipeline. After simulation completes, the harness compares the golden model's output against the hardware's trade confirmations on a trade-by-trade basis. Any mismatch is flagged with the order index, expected values, and actual values for debugging.

The simulation supports two modes selected via command-line arguments: automatic (--mode 1) generates a specified number of random orders, and manual (--mode 0) accepts interactive input from the user. The automatic mode is the primary tool for verification at scale.

6.1 Multi-Symbol Simulation (automatic_execution)

The multi-symbol simulation uses an Exchange struct to manage multiple OrderBook instances (one per symbol) and a MemoryManager to handle page allocation and deallocation across symbols. Six test symbols (AAA through FFF) are used. For each order, the simulation finds or creates the order book for that symbol, attempts a memory-aware insert, and then loops on the trade matcher until no more trades can execute. After each trade, post_trade_cleanup frees any pages that are now empty, and trim reclaims heap capacity.

C/C++

```
static void automatic_execution(int orders) {
    Exchange *ex = create_exchange(16);
    MemoryManager *mm = create_memory_manager();
    SimStats stats = {0};

    const char *symbols[] = {"AAA","BBB","CCC","DDD","EEE","FFF"};
    int num_symbols = 6;

    Order **order_list = malloc(sizeof(Order *) * orders);
    for(int i = 0; i < orders; i++) {
        const char *symbol = symbols[rand_range(0, num_symbols-1)];
```

```
        order_list[i] = create_order(
            rand_range(1, MAX_PRICE),
            rand_range(1, MAX_AMOUNT),
            rand_range(0, 1), symbol);
        OrderBook *ob = find_or_create_book(ex, symbol);
        int sym_id = 0;
        for(int j = 0; j < ex->cnt; j++) {
            if(strcmp(ex->books[j]->symbol, symbol) == 0) {
                sym_id = j; break;
            }
        }
        // Insert and attempt trade
        if (mem_aware_insert(ob, mm, order_list[i], sym_id,
&stats)) {
            while(check_for_trade_multi(ob, &stats)) {
                ob->trades++;
                post_trade_cleanup(mm, ob, &stats);
            }
        }
        trim(ob->asks, mm, ob, &stats);
        trim(ob->bids, mm, ob, &stats);
    }
    print_sim_stats(ex, mm, &stats);
    /* cleanup omitted for brevity */
}
```

The golden model mirrors the hardware's algorithmic flow: orders are inserted into the appropriate heap via sift-up, the trade matcher compares roots after each insertion, partial fills reduce surviving node quantities via in-place edits, and the memory manager tracks page allocation and deallocation via a bitmap. The key difference is that the software runs sequentially, so hazards that exist in the pipelined hardware (stale root reads, concurrent insert/pop conflicts) do not arise. This means any trade-for-trade mismatch between the golden model and the hardware output points to a hazard that was not properly handled.

6.2 CSV-Driven Simulation

The CSV execution mode is the primary method for running repeatable, deterministic simulations. Instead of generating random orders at runtime, the simulation reads a pre-generated CSV file containing a sequence of orders with known prices, quantities, types, and symbols. This is critical for verification because the same CSV can be fed to both the

golden model and the hardware pipeline, guaranteeing identical input for trade-by-trade comparison.

Each row of the CSV encodes one order in the format: symbol, type (ask/bid), price, amount. The simulation parses each line, constructs an Order struct, and processes it through the matching engine. Because the order sequence is fixed, the expected output is fully deterministic and any difference between the golden model's trades and the hardware's trades would be a bug/issue.

C/C++

```
static void csv_execution(const char *filename) {

    Exchange *ex = create_exchange(16);

    MemoryManager *mm = create_memory_manager();

    SimStats stats = {0};

    FILE *fp = fopen(filename, "r");

    if (!fp) {

        printf("Error: could not open %s\n", filename);

        return;

    }


    char line[256];

    char symbol[4];

    char type_str[4];

    int price, amount;

    int order_count = 0;

    Order **order_list = malloc(sizeof(Order *) * MAX_ORDERS);

    while (fgets(line, sizeof(line), fp)) {

        sscanf(line, "%3s,%3s,%d,%d", symbol, type_str, &price,
&amount);
```

```
        int type = (strcmp(type_str, "ask") == 0 ||
                    strcmp(type_str, "Ask") == 0) ? 1 : 0;
        order_list[order_count] = create_order(price, amount, type,
symbol);
        OrderBook *ob = find_or_create_book(ex, symbol);
        int sym_id = 0;
        for (int j = 0; j < ex->cnt; j++) {
            if (strcmp(ex->books[j]->symbol, symbol) == 0) {
                sym_id = j; break;
            }
        }
        if (mem_aware_insert(ob, mm, order_list[order_count],
sym_id, &stats)) {
            while (check_for_trade_multi(ob, &stats)) {
                ob->trades++;
                post_trade_cleanup(mm, ob, &stats);
            }
        }
        trim(ob->asks, mm, ob, &stats);
        trim(ob->bids, mm, ob, &stats);
        order_count++;
    }
    fclose(fp);
    print_sim_stats(ex, mm, &stats);
```

```
    /* cleanup omitted for brevity */
}
```