A Simple Virtualized HFT Simulation on an FPGA

Final Project

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 the Cyclone V FPGA. It leverages virtual memory to handle multiple symbol order books simultaneously through dynamic resource allocation. The hardware subsystem performs order matching and memory management, while the software subsystem drives simulation data into the FPGA and collects performance metrics.

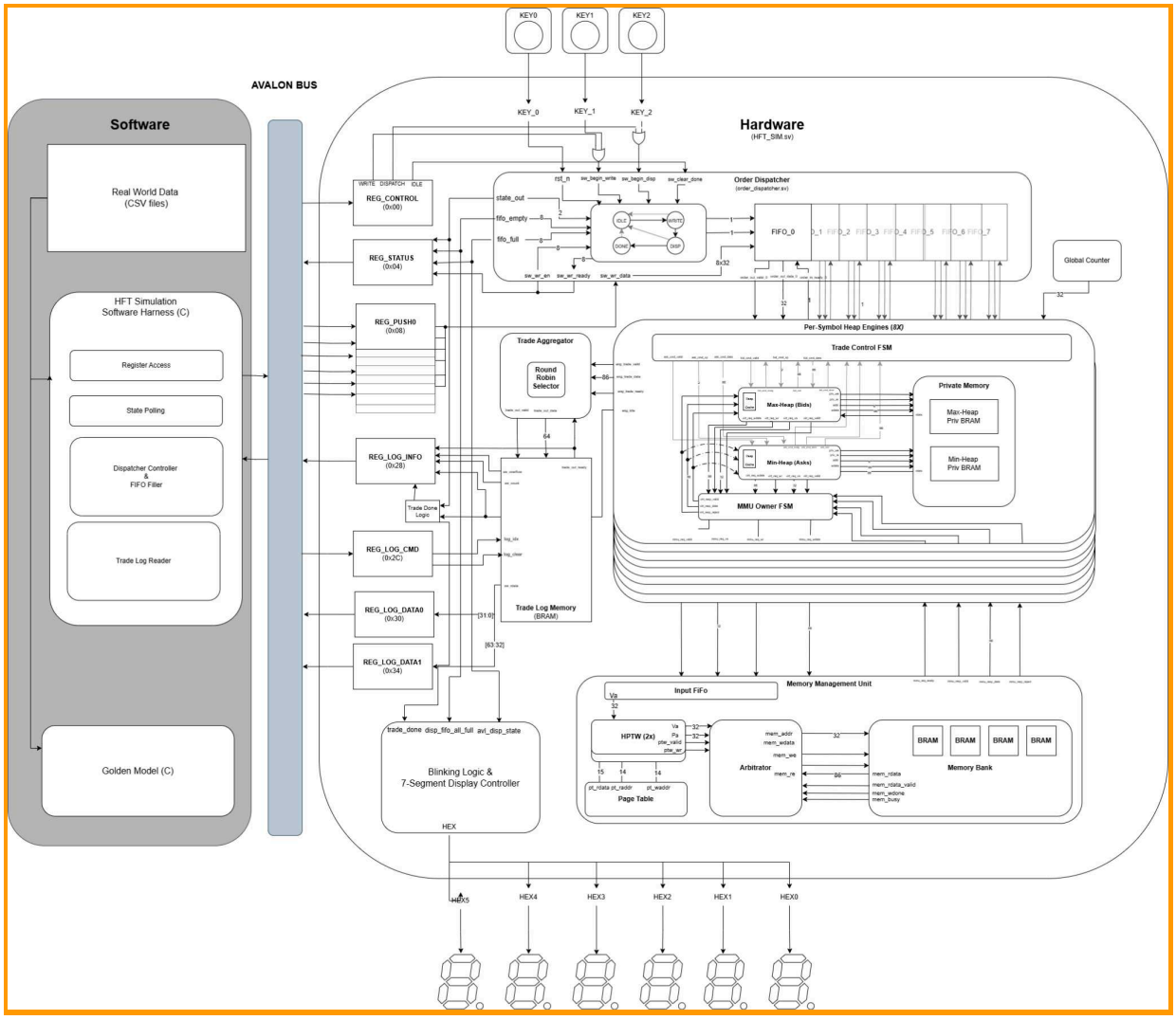
The system is decomposed into four principal components:

- **The HFT System** implements per-symbol order books using binary heap based priority queues in hardware (heap engines). For each symbol, a max-heap holds bids and a min-heap holds asks. The top 64 nodes of each heap live in a dedicated per-engine private BRAM for single-cycle access; overflow beyond 64 transparently spills into the shared MMU-managed memory described below. When the best bid meets or exceeds the best ask, a trade executes. In addition, we support partial and complete fills.
- **The Memory Management System** gives each engine a private virtualized address space backed by a shared physical memory, so heaps that overflow the on-engine BRAM transparently spill into a common pool without engines contending. A supervisor translates virtual addresses to physical frame addresses (def frame: a group of addresses in physical memory) using a page table, with per-partition monotonic page allocation.
- **The Software Harness** runs on the HPS ARM processor and serves as both the data source (generating streams of orders) and the data sink (collecting trade confirmations). The harness drives the hardware over an ioctl-based Linux kernel module and reads back the trade log after the simulation completes.
- **The C Reference Simulator (golden model)** is a software re-implementation of the same order matching and memory management algorithm that runs on the HPS. It processes the same input CSV data as the hardware pipeline and produces an independent trade list against which the FPGA output can be diffed for verification.

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, as well as intermediary data structures such as FIFOs, resides in BRAM, and the HPS communicates with the FPGA 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, trade aggregator, trade log, control and status registers, and user-interface logic, along with each functional block's submodules. The software side (HPS) contains the dataset, software harness, and golden mode.



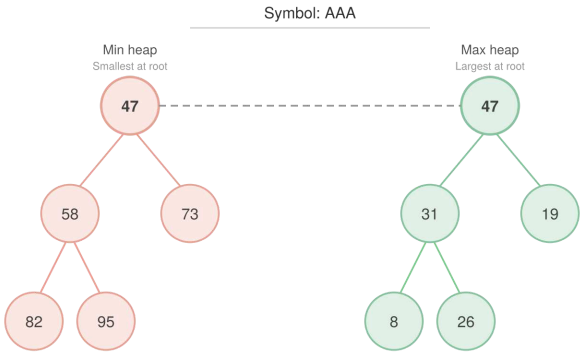
The Linux core communicates with the FPGA through a kernel driver that maps the HFT_SIM Avalon memory-mapped register space into the HPS address space and exposes it to user space as a character device. The software harness then uses this driver to write orders into the dispatcher's eight per-symbol input lanes, poll dispatcher and heap engine status bits, and read back the trade log once trading is complete.The board-level pushbuttons and displays in the block diagram were retained for demonstration purposes. The intended control path is the software-driven Avalon-MM interface described in Section 7.

3. Algorithms & Implementations

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.



Heap structure for a single symbol (min-heap asks, max-heap bids)

The heap supports four operations: insert (sift-up), peek, pop (sift-down), and edit (quantity update). All four are implemented inside a single per-heap `heap_fsm` module that exposes a uniform command interface to the trade controller:

```
None
    case (saved_op)
      OP_PUSH:   n_state = S_PUSH_LAUNCH;
      OP_POP:    n_state = S_POP_LAUNCH;
      OP_PEEK:   n_state = root_cache_valid ? S_DONE : S_PEEK_RD_ISSUE;
      OP_UPDATE: n_state = S_UPDATE_WR_ISSUE;
    endcase
```

The trade controller drives the heap_fsm through a small handshake interface:

- cmd_valid (1b): pulse from controller to start a new operation
- cmd_op (2b): which operation to perform — OP_PUSH, OP_POP, OP_PEEK, or OP_UPDATE
- cmd_data_in (86b): the order node payload, used by PUSH and UPDATE (ignored for POP/PEEK)
- cmd_done (1b): asserted by the heap_fsm when the operation completes
- cmd_root_out (86b): the root node returned by PEEK, or the popped value returned by POP

To issue a command, the controller pulses cmd_valid for one cycle with the desired cmd_op and cmd_data_in, then waits in a per-operation wait state until cmd_done rises. The heap_fsm internally walks through its sub-state machine (read parent, compare, swap, etc.) during that interval. Each operation's hardware implementation is described below.

Insert (Sift-Up). A new order is written to the next available slot (`target_idx = size`). The heap_fsm then walks up the tree, comparing the new node against its parent and swapping if the parent loses. The same `a_wins_fields` comparator is used for both heaps, parameterized by `HEAP_KIND`:

```
None
    function automatic logic a_wins_fields (
      input logic [15:0] ap, input logic [31:0] at,
      input logic [15:0] bp, input logic [31:0] bt);
      if (ap != bp)
            a_wins_fields = (HEAP_KIND == MAX_HEAP) ? (ap > bp) : (ap < bp);
      else
            a_wins_fields = (at < bt);  // earlier timestamp wins ties
    endfunction
```

For a heap of depth *d*, sift-up takes at most *d* parent reads plus *d* writes. The vast majority of pushes touch only the top 3 levels of the heap, which are kept in a per-heap register cache (see Top-3 Cache below), so most pushes complete with zero BRAM accesses.
Peek. Returns the root (index 0). When the top-3 cache holds a valid root, OP_PEEK returns it the same cycle it was issued:

```
None
    assign cmd_root_out = (saved_op == OP_PEEK)
                      ? (root_cache_valid ? root_cache : '0): popped_root;
```

If the cache is cold (heap just initialized or just emptied), the FSM issues a single BRAM/MMU read of index 0 and caches the result.

Pop (Sift-Down). The root is removed by reading the last leaf into the root slot, decrementing `size`, then sifting the new root downward. Each level reads both children, picks the winner via `a_wins_fields`, and swaps if the winner beats the current node:

```
None
    S_POP_SIFT_DECIDE:
      if (...) n_state = S_POP_SIFT_WR_UP_ISSUE;   // child wins → swap
      else     n_state = S_POP_SIFT_WR_FINAL_ISSUE; // current node settled
```

Same depth bound as sift-up. The top-3 cache reads at indices 0/1/2.

Edit (Quantity Update). Used after partial fills to reduce the surviving root's amount. Since only the amount changes (price and timestamp are unchanged), no re-sift is needed. The heap_fsm writes the new node directly to index 0 and refreshes the cache slot:

```
None
    S_UPDATE_WR_WAIT: if (priv_write_done || virt_write_done) begin
      node_cache[0]       <= saved_op_data;
      node_cache_valid[0] <= 1'b1;
    end
```

This is a single-cycle BRAM write plus a cache update.

Top-3 Node Cache

Each heap_fsm instance holds a 3-entry register cache for indices 0, 1, and 2:

```
logic [NODE_WIDTH-1:0] node_cache         [3];
logic [2:0]            node_cache_valid;
```

The cache serves two purposes. First, OP_PEEK returns `node_cache[0]` directly, eliminating BRAM read latency on the most frequent operation. Second, during sift operations, if the FSM needs to read an index ≤ 2, the cache is checked first:

```
wire mem_idx_cached = (mem_idx == 0) ? node_cache_valid[0] :
                      (mem_idx == 1) ? node_cache_valid[1] :
                      (mem_idx == 2) ? node_cache_valid[2] : 1'b0;
wire effective_read = mem_do_read && !mem_idx_cached;
```

A cache hit skips the BRAM read entirely. Combined with the fact that a typical partial-fill trade only touches indices 0/1/2 of each heap, the common path through a match emits its trade with zero BRAM accesses on either heap.

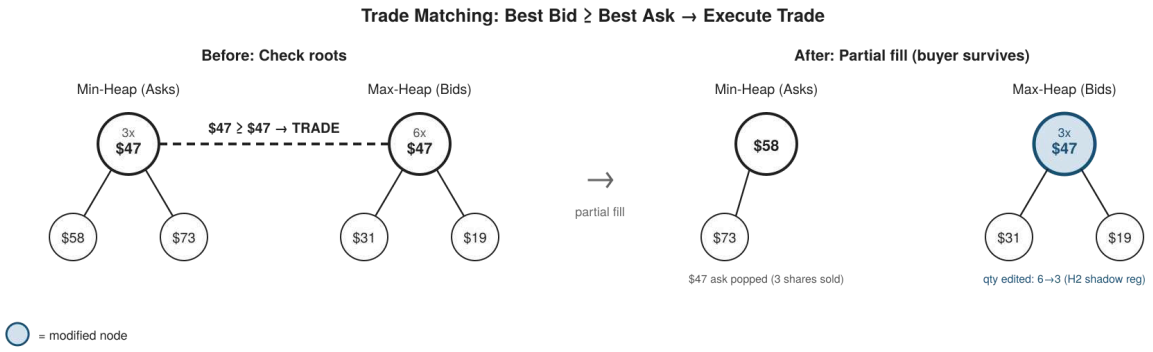
Two-Tier Storage

Each heap accesses nodes through two storage tiers:

- **Memory Indices 0–63:** a per-engine private `priv_bram` (128 entries, addressed by `{heap_kind, virt_idx[5:0]}`). Single-cycle read/write, no contention with other engines.
- **Memory Indices ≥64:** shared MMU-backed memory across four banks. The heap_fsm constructs a virtual address `{ENGINE_ID, HEAP_KIND, virt_idx}` and issues a request through the engine's MMU port. Translated through the page table, served by the appropriate bank, and returned several cycles later.

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 bid's price. The following figure illustrates a partial fill:



Partial fill trade execution

In hardware, this comparison runs combinationally inside the trade controller FSM once both roots have been snapped via OP_PEEK:

```
None
    assign trade_match           = (bid_size > 0) && (ask_size > 0)
                          && (order_price(bid_root_snap) >=
    order_price(ask_root_snap));
    assign partial_bid_remains = trade_match
                          && (order_amount(bid_root_snap) >
    order_amount(ask_root_snap));
    assign partial_ask_remains = trade_match
                          && (order_amount(ask_root_snap) >
    order_amount(bid_root_snap));
```

`T_DECIDE` then branches on these signals to pick the right pop/update sequence:

```
None
   T_DECIDE: begin
      if (!trade_match)          tn_state = T_IDLE;
      else if (partial_bid_remains) tn_state = T_POP_ASK_ISSUE;
      else if (partial_ask_remains) tn_state = T_POP_BID_ISSUE;
      else                       tn_state = T_POP_BID_ISSUE;  // equal: pop
    both
    end
```

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 (a 64-bit `TRADE_LOG_ENTRY` containing `{engine_id, amount, price, timestamp}`) is written to the trade log by way of the trade aggregator. After emission, the trade controller loops back to `T_PEEK_BID` rather than `T_IDLE`, so that a single new order that triggers multiple chained matches at the same price level is fully drained before the engine accepts its next dispatched order.

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 based on a global counter connected to all heap engines. This allows tie-breaking as two different stocks will never conflict as they are stored and managed separately.
In hardware, the global counter is a single free-running 32-bit register declared at the top level:

```
None
    logic [31:0] now_ts;
    always_ff @(posedge clk or negedge rst_n_i) begin
      if (!rst_n_i) now_ts <= 32'd0;
      else          now_ts <= now_ts + 32'd1;
    end
```

It is broadcast to all 8 symbol engines through a single `.now_ts(now_ts)` port. When an engine accepts a new order at `T_IDLE`, the trade controller samples the counter that cycle and packs it into the low 32 bits of the 86-bit node:

```
None
    if (tstate == T_IDLE && order_in_valid) begin
      pending_order <= {order_in_data[31],           // [85]       type
                        order_in_data[30:15],      // [84:69] price
                        1'b0, order_in_data[14:0], // [68:53] amount
                        SYMBOL,                    // [52:32] symbol
                        now_ts};                   // [31:0]  timestamp
    end
```

The same `a_wins_fields` comparator reads these timestamp fields directly. When prices tie, the earlier `now_ts` wins.

```
None
    function automatic logic a_wins_fields (
      input logic [15:0] ap, input logic [31:0] at,
      input logic [15:0] bp, input logic [31:0] bt);
      if (ap != bp)
            a_wins_fields = (HEAP_KIND == MAX_HEAP) ? (ap > bp) : (ap < bp);
      else
            a_wins_fields = (at < bt);  // earlier timestamp wins ties
    Endfunction
```

Because each engine processes orders for exactly one symbol and orders enter sequentially, the per-symbol order of arrival is preserved in the captured timestamps, even though all engines share the same counter.

3.4 Virtual Memory Translation

Each symbol has access to two private pages of memory, one for bids and one for asks. These pages store 64 nodes each. When a binary heap would extend past 64 nodes, it is sent to the MMU to be written into a shared pool of memory. This is done by virtualizing the address of the node. Virtual addresses are comprised of 32-bits, the top 3 bits identify which of the 8 heap engines the node originates from, the rest of the bits are ordered sequentially starting from 0.

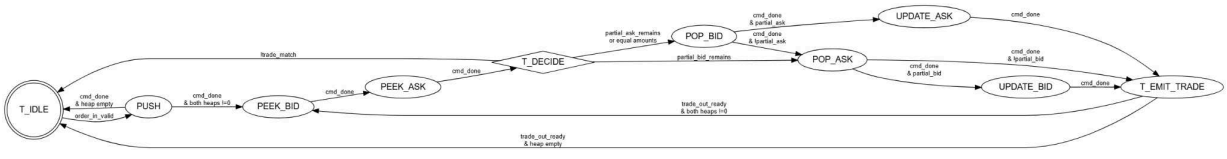
4. Heap Engine Overview

Each of the 8 per-symbol heap engines is an independent unit that processes the order stream for a single symbol. An engine receives one order at a time from its dispatch FIFO, inserts it into the appropriate side of the order book, checks for matching trades, executes any matches via partial-fill or full-fill, and emits trade records to the trade aggregator. Engines run fully in parallel and share only the MMU and trade aggregator.

A symbol engine is decomposed into three internal blocks:

- **Trade Control FSM**: the brain of the engine. Sequences the operations needed to process one order: push, peek both heaps, decide if a trade matches, execute the trade with pops/updates, and emit the trade record.
- **Two heap_fsm instances**: one max-heap for bids, one min-heap for asks. Each stores and maintains its heap invariants, exposes a simple PUSH/POP/PEEK/UPDATE command interface, and manages its own storage across a private BRAM memory and a shared MMU-backed virtual memory.
- **MMU Owner FSM**: a small arbiter that arranges the two heaps' shared accesses to the engine's single MMU port. The bid heap has priority over the ask heap when both have outstanding requests.

4.5.1 Trade Control FSM



The trade control FSM walks each accepted order through a deterministic sequence of states. The complete state machine for processing one order is:

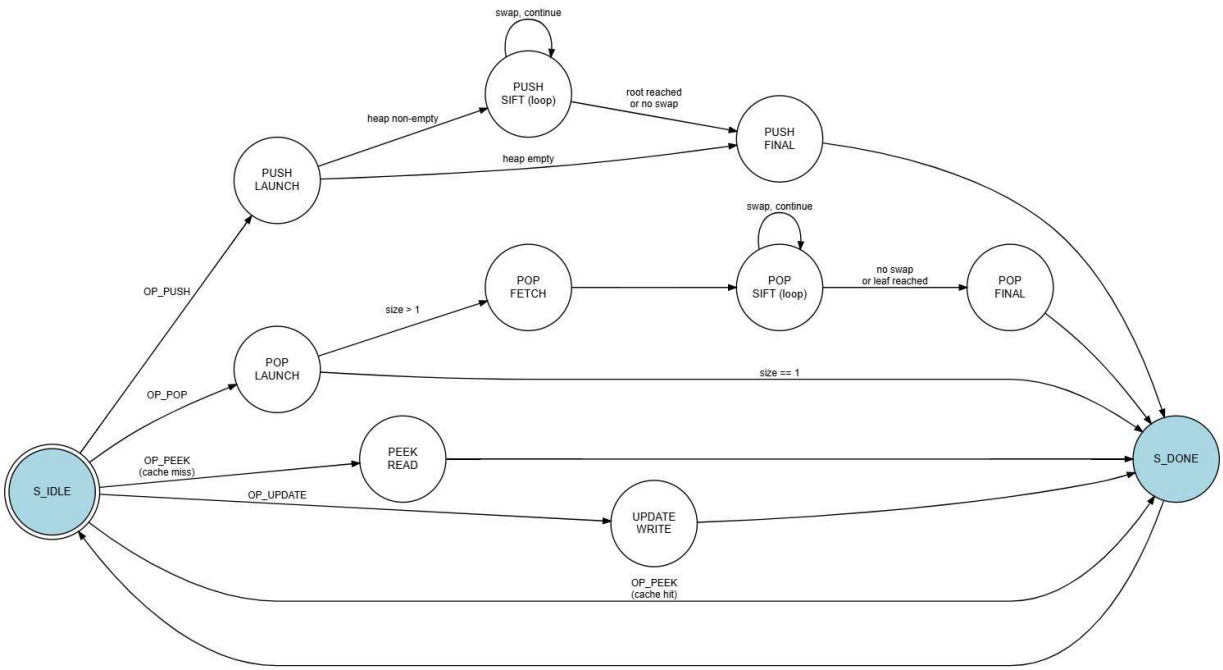
```
T_IDLE
  → on order_in_valid: capture order, route to bid or ask heap
T_PUSH_ISSUE → T_PUSH_WAIT
  → on cmd_done: if both heaps non-empty, proceed to peek; else back to T_IDLE
T_PEEK_BID_ISSUE → T_PEEK_BID_WAIT → snap bid root
T_PEEK_ASK_ISSUE → T_PEEK_ASK_WAIT → snap ask root
T_DECIDE
  → if no match (bid_price < ask_price), back to T_IDLE
  → if partial_bid_remains, pop ask then update bid
  → if partial_ask_remains, pop bid then update ask
  → if equal amounts, pop both
T_POP_* → T_UPDATE_* (when partial fill) → T_EMIT_TRADE
T_EMIT_TRADE
  → on trade_out_ready: if both heaps still non-empty, loop back to T_PEEK_BID
  → otherwise back to T_IDLE
```

The match condition is `bid_root.price >= ask_root.price`. The trade is emitted at the bid's price for `min(bid.amount, ask.amount)` shares. If both sides have equal amounts, both heap roots are popped (full fill). If one side has more, that side is updated in place with the remaining amount (partial fill). Updates do not require re-sifting because the price stays unchanged.

After emitting a trade, the FSM loops back to T_PEEK_BID rather than to T_IDLE. This drains all chained matches that one new order can trigger (e.g., a single large bid meeting multiple smaller asks at the same price). The FSM only returns to T_IDLE (and therefore only accepts the next dispatched order) when no further match is possible or one of the heaps empties.

The trade controller also drives a single status output, engine_idle, which is high whenever tstate == T_IDLE (the engine has fully drained the current order and is ready to accept the next one). The top-level AND's across all 8 engines to produce the all_engines_idle signal; combined with the order dispatcher being in its DONE state, this forms the trade_done bit that software uses to detect that the simulation has finished.

4.5.2 heap_fsm: Operations and State Machine



Each heap_fsm is a self-contained Binary Heap implementation that hides storage details from the trade controller. It exposes four operations:

Operation	Description
OP_PUSH	Insert a new node and sift it up to maintain heap order
OP_POP	Remove the root, replace it with the last leaf, and sift the new root down
OP_PEEK	Return the root node
OP_UPDATE	Overwrite the root's amount field (used after partial fills)

Each operation is issued by the trade controller via a single-cycle cmd_valid pulse with the operation code and any input data. The heap_fsm then runs through its internal state machine and signals completion with cmd_done. The trade controller waits in its wait-states until that signal fires.

Sift-up (PUSH). The new node is conceptually placed at index `size`. The FSM reads its parent, compares, and swaps if needed. This repeats up the tree until either the new node is no longer "more priority" than its parent or it reaches the root. At most $\lceil \log_2(\text{size}) \rceil$ swaps.

Sift-down (POP). The root is replaced with the last leaf, then compared against its two children. The winning child is swapped up, and the process repeats down the tree. Same depth bound as sift-up.

The comparator for a max-heap returns “a wins over b” if a.price > b.price, with a tie-break to the earlier timestamp. The min-heap reverses the price comparison. Both heap variants come from the same heap_fsm module parameterized on HEAP_KIND (0 = MAX, 1 = MIN).

4.5.3 Top-3 Node Cache

Each heap_fsm contains a small 3-entry register cache that holds the most recent values written to indices 0, 1, and 2 of the heap (the root and its two children). The cache serves two purposes:

- **Single-cycle PEEK.** When the trade controller issues OP_PEEK, the heap_fsm returns node_cache[0] immediately, without going through the BRAM read path. This eliminates the BRAM read latency for the most common operation.
- **Hit’s Cache during sift operations.** When sift-up or sift-down needs to read a node at index 0, 1, or 2, the heap_fsm checks the cache valid bit first. On hit, no BRAM read is issued and the value is taken from the cache directly. On miss, the BRAM (or MMU) read proceeds as normal.

Each write to indices 0–2 also updates the corresponding cache slot. POP invalidates slot 0 first, then writes the new root once the sift settles. The cache is cleared on heap-empty (size = 0).

4.5.4 Two-Tier Storage

Each heap accesses its nodes through a two-tier address space:

Tier	Index Range	Characteristics
Private (BRAM)	Indices 0–63	Per-engine priv_bram (128 entries split: 64 bid + 64 ask). Single-cycle read/write. No contention between engines.
Virtual (MMU)	Indices ≥64	Shared memory across four banks. Address: {ENGINE_ID, HEAP_KIND, virt_idx}. Multi-cycle MMU translation latency.

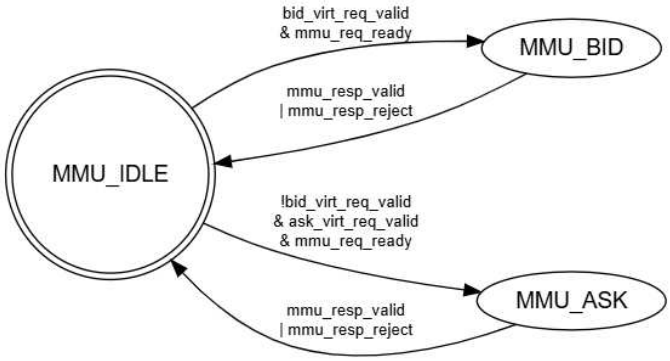
The motivation for this split is that the top of a heap is the part that gets touched most often.

By keeping the first 64 entries in dedicated private BRAM, every read and write at the top of the heap is single-cycle, while only the lower-frequency accesses to the heap’s tail pay the MMU’s translation latency.

Together with the 3-node cache, the common case for a partial-fill trade (peek root, peek root, decide, pop root, update root, emit) touches only the cache and register file, with zero BRAM or MMU traffic.

4.5.5 MMU Owner FSM

Each engine has one MMU port but two heaps that can independently issue memory requests. The MMU Owner FSM serializes their accesses. It is a three-state machine:




```
MMU_IDLE
  → if bid heap requests: MMU_BID
  → else if ask heap requests: MMU_ASK
MMU_BID  → on response (valid or reject): MMU_IDLE
MMU_ASK  → on response (valid or reject): MMU_IDLE
```

While in MMU_BID, the bid heap_fsm's request signals are connected to the engine's MMU port, and the MMU's response signals are gated through to the bid heap. The ask heap sees no response activity. MMU_ASK works symmetrically. The bid heap is given strict priority when both have an outstanding request. However, because the trade controller never issues parallel commands to both heaps (it always waits for one to finish before starting the other), in practice only one heap ever has an outstanding MMU request at a given time.

4.5.6 Trade Emission and Aggregator Backpressure

When the trade controller reaches T_EMIT_TRADE, it presents the trade record on `trade_out_valid` and `trade_out_data` (the full 86-bit ORDER node, with amount replaced by the matched quantity). The trade aggregator collects from all 8 engines via round-robin, packs the selected engine's order into the 64-bit TRADE_LOG_ENTRY format, and writes to the trade log.

If the trade log is full, the aggregator drops `trade_out_ready` low for the selected engine. The engine stays in T_EMIT_TRADE until the trade is accepted. This backpressure naturally throttles all 8 engines to a rate the trade log can absorb.

4.5.7 Putting It Together: Partial-Fill Trade

A typical partial-fill trade exercises all three blocks in sequence:

1. **Order arrives at the engine.** Trade Control FSM moves T_IDLE → T_PUSH.
2. **Trade Control issues OP_PUSH to the appropriate heap.** The heap_fsm sifts the new node up (usually only a single comparison against a cached parent), no BRAM access.
3. **Trade Control issues OP_PEEK to both heaps.** Both return immediately from their caches.
4. **Trade Control compares the two roots,** detects a match, and decides whether to pop both or pop one and update the other.
5. **Trade Control issues OP_POP and OP_UPDATE as needed.** Pops near the heap top often hit only the cache; deeper pops hit private BRAM; only operations against indices ≥64 require an MMU round-trip via the MMU Owner FSM.
6. **Trade Control emits the trade.** If both heaps remain non-empty, loops back to step 3.

This decomposition lets the high-frequency path (cache hits, private BRAM) stay single-cycle while the rare deep operations transparently fall through to shared MMU storage.

5. Memory Management Unit and Shared Memory

The MMU handles requests from the heap engines to the virtualized shared memory pool in parallel, accepting up to 8 requests at once. It translates these addresses, executes the request to memory, and then forwards the data to the correct heap engine, confirming that either a read or a write has occurred.

The MMU is comprised of:

- **The top level wrapper**: Handles routing the eight requests, instantiates the hardware page table walkers and arbiter, contains the BRAM based page table, and handles address allocation for new virtual addresses.
- **Two hardware page table walkers**: Each page table walker looks up the physical translation to the provided virtual address. If the physical address does not exist for the given virtual address, it raises a flag and asks the MMU to write to the page table, creating the address and assigning it in the same cycle.
- **The arbiter:** Receives physical addresses and, in the case of a write, data to be written. This request is then routed into a queue for one of four BRAM blocks, while the arbiter handles collisions.
- **Four memory banks**: Controls the physical memory requests for one of the four banks of memory using a FSM allowing the 86-bit nodes to be split into 3 chunks, each with a width of 32 bits. Each bank contains 60 pages which store 64 nodes each.

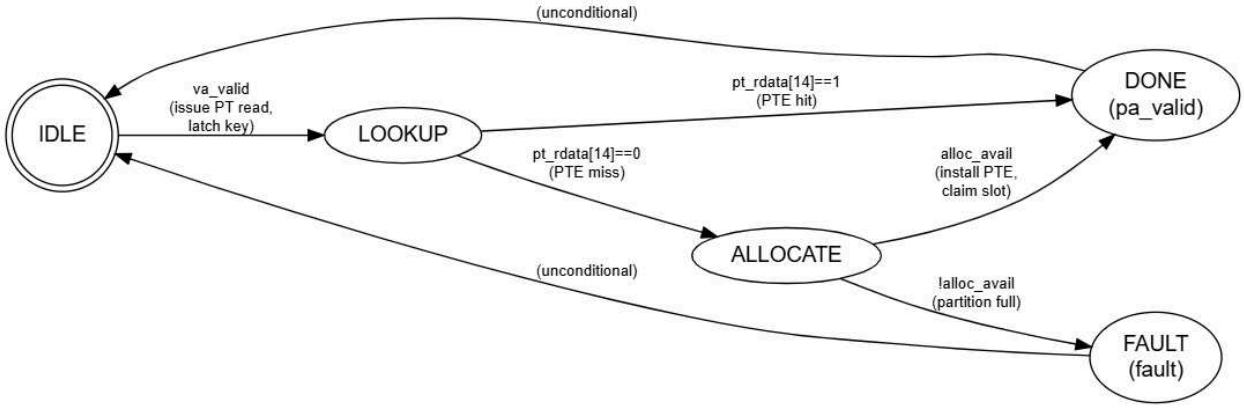
5.1 Top Level MMU wrapper

The top level interface is responsible for three main functions: handling parallel requests, allocating new physical addresses, and instantiating the page table, page table walkers and arbiter. In order to handle parallel requests, the wrapper uses a round robin algorithm, selecting from the currently raised valid requests. It selects two winners and enqueues them into one of two FiFos, each of which feeds a page table walker. These page table walkers parse the page table, also instantiated here. This page table is a single level array that has a total of 16384 entries, one for each of the possible nodes that can be stored in physical memory. This was calculated using the size of physical memory, 256 pages x 64 nodes per page. Instantiated as true dual port BRAM, allowing for two page table walkers, the page table is addressed by a 14 bit key, aptly called the page table key, which is calculated by the page table walker. If the address returns a miss, meaning the address is yet to be allocated, the page table walker informs the top level wrapper which handles allocation. Allocation is done using partitions, there are 16 partitions, each supervising 15 physical pages in memory. Each request is mapped to a partition based on the top 3 bits of the request, the engine ID, and the 11th bit of the virtual address. In other words, each heap engine can access two partitions. The top module maintains two counters per partition, one handles the node, and is incremented with each new allocation up to 64, and the other handles page, which is incremented when the node counter rolls over. If both page tables miss in the same cycle,

page table walker 0 is always given priority, this results in page table 1's request being requeued and resolving on the next cycle. This allows the hardware to allocate a new physical address in a single clock cycle, forwarding the request directly to the arbiter.

5.2 Hardware Page Table Walkers

The MMU instantiates two hardware page table walkers, these walkers can be modeled as a state machine. In the lookup state, the 14 bit page table key for a provided virtual address is formed. This page table key consists of the engine ID, formed by the top 3 bits of a virtual address, and the bottom 11 bits of the virtual address, which serve as an offset for that specific engine. The page table walker attempts to use this key to read the corresponding 15 bit value from the page table. This 15 bit result consists of an allocated or valid bit, the physical page index (8 bits), and the node index (6 bits), which also allows the page table walker to determine which memory bank the request is aimed at, storing this in the request's bank ID. If the allocated bit is high, the walker hits, meaning that the address has been allocated already, causing the walker to pad the page and node index with zeros forming a physical address and transition to the done state which forwards the physical request to the arbiter. Otherwise the state machine enters the allocate state where a new physical address is assigned by the MMU. If both page table walkers attempt to allocate the same address, page table walker 1's request is denied, moving it to the fault state, while page table walker 0's request is approved and the address is allocated.



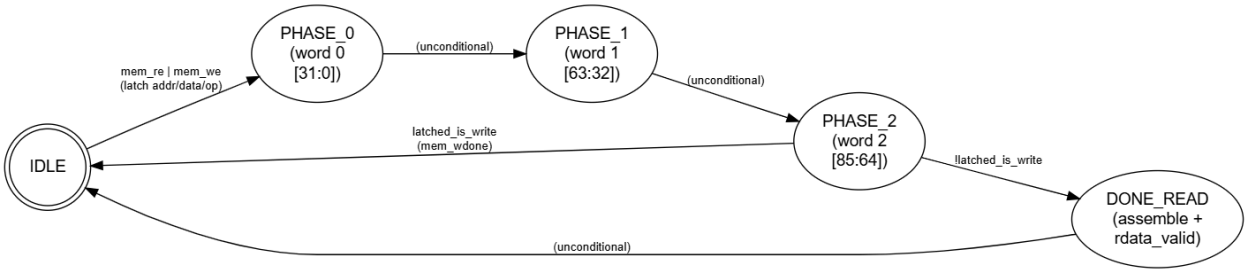
5.3 Arbiter

The arbiter receives the entire request, including both the physical and virtual address, data to be written if necessary, if the request is a read or write, if the request is valid, and the bank of memory the request should be routed to based on the bank ID. These requests are received two at a time from the page table walkers, the arbiter uses this information to insert the request into a FiFo, based on the bank ID of each request, with each FiFo draining into a memory bank. By using dual-write FiFos, the arbiter allows both page table walkers to seamlessly target the same memory bank, granted that at least 2 of the FiFo's 4 slots are free. However, if the FiFo only has one remaining slot, the winner is calculated based on round robin priority, with the loser being requeued inside of the MMU. Similarly if the targeted FiFo is full a request is requeued within the MMU. In order to synchronize the virtual address related

to each request, the arbiter also instantiates FiFos that contain the virtual address and only outputs the address back to the MMU after the corresponding memory request has been fulfilled. Interfacing directly with the memory bank, the arbiter also receives the result of memory requests, receiving not only the read data but also three critical flags. A valid flag, indicating the validity of a read, a done flag, indicating a write has been finished, and a busy flag, indicating a request is in flight. After the done or valid flag is raised and the busy flag is dropped, the arbiter forwards the resulting information to the MMU which uses the associated virtual address to return the information to the requesting heap engine.

5.4 Memory Bank

The four memory banks each control a BRAM instantiation. Each node that needs to be written or read is 86 bits, and because our M10Ks use 32-bit words, we need to sequence memory requests in sets of three. This is accomplished by the memory banks using a simple state machine. In states phase 0, phase 1, and phase 2, the state machine executes the request for the three words associated with the node, either reading or writing to physical memory. If the request is a write, the state machine returns to idle and informs the arbiter that the write has been completed. Meanwhile, if the request was a read, the memory bank assembles the complete 86 bit node and returns it to the arbiter in the done read state.



6. Resource Budgets

6.1 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**	

6.2 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).

6.3 Memory Usage

Parameter	**Value**
Total BRAM blocks used	394
Public Memory	240
MMU	6
Order Dispatcher	56
Heap Engines	24
Trade Logger	68
Nodes per page	64
Total pages (clean address decoding)	256
Total node capacity	**16,384 nodes**

6.4 Address Bit Widths

Component	**Width**	**Description**
Heap Engine Private Pool Address	6	Assigned sequentially 0 to 63
Virtual Addresses for Shared Pool	32	3 bits for engine ID, 18 bits of padding, 11 bits of sequential assignment
Physical Addresses for Shared Pool	32	11 bits of padding, the 8 bit page index, 3 bits of padding, the 6 bit node index, 4 bits of padding
Page table key	14	3 bits for engine ID, 11 bits of sequential assignment

6.5 Logic Usage

Parameter	Value
Total ALMs used	28,456.4
Public Memory	3,390.4
MMU	3,854.3
Order Dispatcher	524.7
Heap Engines	20,364.2
Trade Logger	137.7
SoC Overhead	211

6.6 Timing

Using the slow corner at 1100mV at 85C our F_{MAX} according to Quartus was 53.85MHz.

7. Hardware-Software Interface

The final hardware-software interface is narrower and more structured than the preliminary interface used during early bring-up. For instance, the order dispatcher is split into eight fixed symbol lanes, meaning the HPS no longer needs to transmit a symbol field with every order. Likewise, because timestamps are generated inside the FPGA, software does not send timestamps to hardware. As a result, each input order is reduced to a single 32-bit dispatch word containing only type, price, and quantity. On the output side, the FPGA no longer returns trades through a streaming confirmation register. Instead, completed trades are appended to a compact indexed trade log that software reads after execution.

The top level also retains pushbutton and seven-segment display logic for the live DE1-SoC demonstration. These signals mirror the same begin-write and begin-dispatch events used by software, but they are auxiliary to the main interface. The primary control and data path is the Avalon memory-mapped interface between the HPS and the FPGA fabric

7.1 Control Register (Write, 32 bits)

Bits	Field	Width	Description
[0]	begin_write	1	Transition the dispatcher from IDLE to WRITE
[1]	begin_dispatch	1	Transition the dispatcher from WRITE to DISPATCH
[2]	clear_done	1	Transition the dispatcher from DONE back to IDLE
[31:3]	reserved	—	Reserved

Software controls the order dispatcher through a write-only control register at address **0x00**. Each asserted bit is interpreted as a pulse by the Avalon wrapper and converted into a

one-cycle internal control event.
This register is used to bracket one full simulation run. First, software places the dispatcher into WRITE so that orders may be loaded into the per-lane FIFOs. After the FIFOs have been filled, software asserts begin_dispatch to start feeding orders into the heap engines. Once the dispatcher reaches DONE and all engines have drained, software asserts clear_done to prepare the system for the next run.

7.2 Dispatcher Status Register (Read, 32 bits)

The dispatcher status register at address **0x04** exposes both the global dispatcher state and the per-lane FIFO status.

Bits	Field	Width	Description
[1:0]	state	2	Dispatcher state: 0 = IDLE, 1 = WRITE, 2 = DISPATCH, 3 = DONE
[9:2]	ready_mask	8	One bit per lane; 1 indicates the corresponding PUSH register may accept a write
[17:10]	empty_mask	8	One bit per lane; 1 indicates the corresponding FIFO is empty
[25:18]	full_mask	8	One bit per lane; 1 indicates the corresponding FIFO is full
[31:26]	reserved	6	Reserved

The ready_mask is the key flow-control signal on the input path as software write to a PUSH register is only accepted when the dispatcher is in WRITE and the corresponding ready bit is high. The harness therefore polls this register before issuing writes, and the driver also checks it before committing an order to hardware.

7.3 Per-Lane PUSH Registers (Write, 32 bits each)

Orders are written into the dispatcher through 8 per-lane PUSH registers located at addresses **0x08** through **0x24**. Each register corresponds to one symbol lane and one dispatch FIFO. Writing to a PUSH register appends one order to that lane's FIFO, provided the lane is ready.

Bits	Field	Width	Description
[14:0]	quantity	15	Unsigned order quantity
[30:15]	price	16	Unsigned order price
[31]	type	1	0 = Ask, 1 = Bid

This 32-bit format is specific to the dispatcher interface. It is smaller than the full 86-bit internal ORDER node because the symbol is implied by the selected lane and the timestamp is generated by the FPGA when the order is accepted by the symbol engine. Internally, the 15-bit quantity field is zero-extended when the full ORDER node is formed.

In the current dataset and harness, the lane mapping is fixed:

Lane	Symbol Stream
0	Apple Inc. (AAPL, shortened to APL).
1	Boston Scientific (BSX)
2	US Bonds (BUS)
3	3M Company (MMM)
4	Microsoft Corporation (MSFT, shorten to SFT)
5	Starbucks Corporation (SBUX, shorten to BUX)
6	T-Mobile US (TUS)
7	Walmart Inc. (WMT)

Note that this fixed mapping is also reused on the output side: the engine_id recorded in the trade log identifies which symbol engine produced the trade.

7.4 Trade Log Information Register (Read, 32 bits)

The trade log information register at address **0x28** exposes the status of the append-only trade log and the completion status of the engines.

Bits	Field	Width	Description
[0]	overflow	1	1 if at least one trade could not be appended because the trade log was full
[1]	data_valid	1	1 if the indexed trade entry requested through LOG_CMD is available in LOG_DATA0/1
[15:2]	count	14	Number of valid trade entries currently stored in the trade log
[16]	count_overflow	1	In case of count overflows. Never really used.
[24:17]	engine_idle_mask	8	One bit per engine; 1 indicates that engine is idle
[25]	all_engines_idle	1	1 if all eight engines are idle
[26]	trade_done	1	1 when the dispatcher is DONE and all engines are idle
[31:27]	reserved	5	Reserved

The count field reports how many trade entries have been logged so far. In the current build, the trade log depth is 8704 entries. The harness uses trade_done as its end-of-run condition, since this signal guarantees both that the dispatcher has finished issuing orders and that no engine is still draining chained matches. The engine_idle_mask and all_engines_idle signals

are used for debugging and progress monitoring because they show whether any individual engine is still active.

7.5 Trade Log Command Register (Write, 32 bits)

Software reads completed trades through an indexed command register at address **0x2C**.

Bits	Field	Width	Description
[0]	clear	1	Clear the trade log count and overflow flag
[1]	read_req	1	Request a read of the indexed trade log entry
[15:2]	index	14	Trade-log entry index
[31:16]	reserved	16	Reserved

The clear bit resets the trade log between runs. The read_req bit initiates a read of entry index. Because the trade log is BRAM-backed and read synchronously, the requested entry does not appear immediately at the read-data registers. Instead, the Avalon wrapper captures the returned entry into a shadow register, and data_valid in LOG_INFO goes high once that shadow register has been updated.

7.6 Trade Log Data Registers (Read, 64 bits total)

Once data_valid is asserted, software reads the requested trade entry from two 32-bit registers:

Address	Name	Contents
0x30	LOG_DATA0	bits [31:0] of the selected trade entry.
0x34	LOG_DATA1	bits [63:32] of the selected trade entry.

The complete 64-bit TRADE_LOG_ENTRY is packed as follows:

Bits	Field	Width	Description
[31:0]	timestamp	32	FPGA timestamp of the trad
[47:32]	price	16	Execution price
[55:48]	amount	8	Executed quantity, with the low 7 bits used in the current build
[63:56]	engine_id	8	Producing engine, with the low 3 bits used

This compact trade format is a deliberate space-saving optimization. Earlier versions of the system stored full 86-bit order nodes in the trade log, but the final design reduces the software-visible trade record to the fields required for verification and timing analysis. Since each engine is permanently associated with one symbol lane, engine_id is sufficient to identify

the source symbol during readback. The compact format also makes it possible to provision a deeper trade log within the available BRAM budget.

7.7 Linux Driver Interface

The HPS does not access the FPGA registers directly from user space. Instead, the interface is mediated by a Linux platform driver bound through the device tree. On probe, the driver locates the HFT_SIM register region, reserves it, maps it into kernel virtual memory, and registers a misc character device at /dev/hft_sim.

The driver exposes a small ioctl-based API with three groups of operations:

1. **Dispatcher control** - these operations begin the WRITE phase, begin the DISPATCH phase, clear the DONE state, push one order to a selected lane, and return the current dispatcher status.
2. **Trade-log control** - these operations clear the trade log, return current log status, and read one indexed trade entry.
3. **Status polling** - these operations allow user space to observe progress without directly manipulating physical addresses or Avalon timing.

The driver also performs basic protocol enforcement. Before pushing an order, it checks that the selected lane is valid, that the dispatcher is in WRITE, and that the corresponding ready bit is high. Then, before reading a trade-log entry, it checks that the dispatcher has reached DONE, issues the indexed read command, polls until data_valid is asserted, and then returns the two 32-bit data words to user space. A mutex serializes access so that concurrent user-space requests cannot interleave register transactions.

7.8 Software Harness Protocol

The user-space harness is responsible for supplying the workload, launching the hardware run, monitoring progress, and collecting the final trade log. In the current setup, the harness opens one CSV file per symbol lane and fills the dispatcher FIFOs in round-robin order. Each CSV contains only type, price, and quantity.

The intended software-controlled run sequence is:

1. Open /dev/hft_sim.
2. Open the eight per-lane CSV files.
3. Clear the trade log.
4. Pulse begin_write and wait until the dispatcher reports WRITE.
5. Repeatedly poll the dispatcher status and write orders into any lane whose ready bit is high.
6. Stop writing when every input file has been exhausted or when all FIFOs are full.
7. Pulse begin_dispatch and wait until the dispatcher reports DISPATCH.
8. Poll LOG_INFO until trade_done becomes 1.
9. Read LOG_INFO.count to determine how many trade entries were recorded.

10. For each entry index from 0 to *count*:
    a. Write LOG_CMD with read_req = 1 and the desired index.
    b. Wait until LOG_INFO.data_valid becomes 1.
    c. Read LOG_DATA0 and LOG_DATA1.
    d. Decode timestamp, price, amount, and engine_id.
11. Write the collected trades to disk for later comparison against the golden model.
12. Pulse clear_done to return the dispatcher to IDLE for the next run.

The harness implements the input phase in round-robin order across all eight lanes rather than draining one file at a time. This preserves fairness across symbol streams and prevents a single lane from monopolizing the software write bandwidth during the load phase. Since each lane has its own FIFO and ready bit, the harness naturally skips lanes that are temporarily full and continues writing to the remaining lanes.

During long runs, the harness may also periodically sample the dispatcher state, FIFO masks, trade-log count, and engine idle mask to produce a progress log. This is not required for correctness, but it makes it easier to determine whether the design is actively trading, stalled on one engine, or waiting for software readback.

For the live board demonstration, the same sequence can also be initiated by the buttons on the DE1-SoC board, since the hardware ORs the board-level begin-write and begin-dispatch events with the software control pulses.

7.9 Performance Measurements

The final evaluation used two performance metrics: accuracy and speed. Accuracy was determined by comparing the ordered list of trades produced by the FPGA against the ordered list of trades produced by the C golden model when both were driven by the same input dataset. In other words, correctness was judged by whether the hardware emitted the same trade sequence as the reference software model. Any missing trades, extra trades, or ordering mismatches were treated as accuracy errors.

Speed was measured from the timestamps stored in the trade log rather than from a separate counter interface. Because the FPGA assigns timestamps from a free-running global cycle counter, each trade record includes the cycle at which that trade was emitted. The execution window for a run was therefore computed as the difference between the latest trade timestamp and the earliest trade timestamp in the recorded trade list. Multiplying this cycle window by the FPGA clock period gives the elapsed hardware trading time:

$$execution\ time\ =\ (latest\ trade\ timestamp\ -\ earliest\ trade\ timestamp) \times clock\ period$$

Using this method isolates the time during which the hardware was actively producing trades. While it is possible that delays might have occurred before the earliest timestamp and after the latest timestamps, those delays would be nothing compared to the time spent actually. For

perspective, time spent trading for the provided 9680 order inputted, was about 1.5 ms. Thus, any delays would be in, at worst, a millisecond. No additional performance metrics were used in the final measurement flow.

8. 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.

8.1 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.

8.2 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 */
}
```

9. Hardware Verification

9.1 Overview

Before committing to the FPGA, the design was verified end-to-end in simulation using a SystemVerilog testbench that mimics the software stack on the HPS. This let us iterate on RTL changes without the ~1hr- synthesis + bitstream cycle, and provided an independently-checkable trade output that could be compared against our software golden model.

9.2 Testbench Implementation

The testbench (hw_test/hft_sim_csv_tb.sv) instantiates the op-level HFT_SIM module (the same Verilog Quartus synthesizes) and drives it through its Avalon memory-mapped interface, exactly as the Linux kernel module does on the real board. The TB:

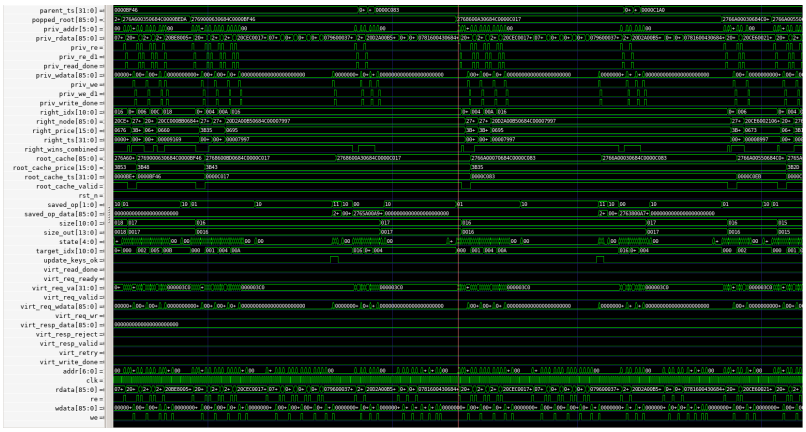
1) Reads the same per-symbol CSV files the on-board harness uses (data/AAPL.csv, data/BSX.csv, etc.).

2) Drives the Avalon bus procedurally to issue the same sequence of register writes the kernel driver performs: pulse sw_begin_write, push every order from each CSV into the corresponding PUSH<i> register, pulse sw_begin_dispatch, then poll the STATUS register for the dispatcher's DONE state.

3) Polls LOG_INFO for trade_done and reads back trade entries via the LOG_CMD / LOG_DATA0 / LOG_DATA1 registers — again, identical to the software protocol.

4) Captures every trade event to a log file so it can be diffed line-by-line against the C-reference simulator's output.

5) Does a VCD dump for waveform inspection in Verdi when needed for debugging.



9.2.1 Other Testbenches

Throughout development, several smaller testbenches were written to validate individual modules in isolation before the full system was wired together. These were progressively retired in favor of the CSV-driven full-system testbench described above, but each played a role:

- hw_test/heap_fsm_tb.sv: test for the heap_fsm module. Issued each operation (PUSH, POP, PEEK, UPDATE) against a manually-loaded heap and checked the sift-up and sift-down invariants, the top-3 node cache behavior, and the size counter under random push/pop sequences.

- hw_test/symbol_engine_tb.sv: integration test for one symbol engine. Wrapped a single symbol_engine instance, drove a handcrafted order stream into it, and observed the resulting trade events. Used to validate the trade controller FSM, the bid/ask coordination, and partial-fill handling end-to-end within one engine.

- hw_test/mmu_tb.sv: Drove translation requests from multiple simulated engine ports, checked page-table allocation behavior, response routing back to the correct engine, and reject/retry handling under the per-partition monotonic allocator.

- hw_test/trade_log_tb.sv : Used a shrunk LOG_DEPTH = 8 so overflow conditions were reachable in a few cycles. Verified the overflow latch, the count counter, and the software-side read protocol.

- hw_test/system_tb.sv: Wired the dispatcher, engines, MMU, aggregator, and trade log together with a tiny TRADE_LOG_DEPTH = 16 so the full pipeline could be exercised in seconds. Used early in development to verify cross-module signaling before the larger CSV-driven test was viable.

- hw_test/hft_sim_csv_tb.sv: The full-system, CSV-driven testbench described in §9.2. Once this was stable, it took over the verification role of every testbench above: it exercises the same modules they exercised, but at full design scale with the actual production input data and via the real Avalon protocol.

9.3 What It Tests

Because the testbench runs the actual RTL and uses the actual register-level protocol, the test path exercises:

- The order dispatcher and its 8 per-lane BRAM FIFOs
- All 8 heap engines in parallel, including the trade controller FSM, both heap_fsm instances, the MMU owner FSM, and the priv_bram private memory
- The shared MMU, including the page-table walkers, allocation logic, and the arbiter dispatching to 4 memory banks
- The trade aggregator and its round-robin selection
- The trade log memory and its software read interface
- The full Avalon-MM register layout, including all status bits the software depends on

9.4 Results

Running the testbench against the same trimmed input dataset used on the board (9,680 orders across 8 symbols), the design produced 8,422 trades, matching the C-reference simulator (sw_sim/memory_sim) exactly; Same trade count, same per-symbol distribution, same set of price/quantity/timestamp tuples in the same order. This established that the matching algorithm, partial-fill handling, heap operations, and trade-emission path are all

correct in the RTL. The testbench therefore served two purposes: it proved correctness of the design at the RTL level, and it pinpointed where to look for any remaining discrepancies between the golden model and our design.

10. Results

Running our synthesized hardware simulation, our design was able to calculate 8,604 trades. This differs by ~2.1% from our golden model which calculated 8,244 trades.

We believe this discrepancy is most likely due to a race condition in the partial filling logic or some sort of data leak in the way that the MMU connects virtual addresses resulting in trades being incorrectly associated with a heap engine.

Excluding these extra trades our hardware results exactly match our golden model, it's also important to note that this bug did not appear in our pre-synthesis testbench, and was only seen post-synthesis on the FPGA.

This loss in accuracy was more than made up for in speed. Using clock monotonic to record a nanosecond accurate measure of our golden model's runtime we recorded the simulation completing in 0.471695 seconds.

Meanwhile our hardware executed in 0.001511 seconds, a 312 times speed up, which equates to an increase in speed of 13,117% or a reduction in time of 99.68%. Given the 8,604 trades executed, we have an approximate throughput of 5,694,242 trades per second which equates to around 0.1057 trades per cycle.

11. Appendix

None

```
// HFT_SIM.sv
//
// Wires the full HFT
// includes the following...
```

```
//   - 1 order_dispatcher
//   - 8 symbol_engines
//   - 1 mmu
//   - 4 mem_banks
//   - 1 trade_aggregator
//   - 1 trade_log
//   - 1 free-running 32-bit timestamp

`include "sys_def.svh"

module HFT_SIM #(
    parameter int TRADE_LOG_DEPTH = 8704
) (
    input  logic                                    clk,
    input  logic                                    rst_n, // ignored

    // DE1-Soc Interface
    input  logic [3:0]                              KEY,
    output logic [6:0]                              HEX0, HEX1, HEX2, HEX3, HEX4,
HEX5,

    // Avalon Bus Interface
    // This will be the only way to communicate with the Linux core
    input  logic                                    chipselect,
    input  logic                                    write,
    input  logic                                    read,
    input  logic [4:0]                              address,
    input  logic [31:0]                             writedata,
    output logic [31:0]                             readdata
);

    // Free-running timestamp counter
    logic [31:0] now_ts;
    always_ff @(posedge clk or negedge rst_n_i) begin
        if (!rst_n_i) now_ts <= 32'd0;
        else        now_ts <= now_ts + 32'd1;
    end

    // Avalon/software wrapper -> dispatcher
    logic                     avl_disp_begin_write_pulse;
    logic                     avl_disp_begin_dispatch_pulse;
    logic                     avl_disp_clear_done_pulse;
    logic [`N-1:0]            avl_disp_push_en;
    logic [`N-1:0]            avl_disp_push_ready;
    DISPATCH_ORDER [`N-1:0]   avl_disp_push_data;
```

```
    logic [`N-1:0]            avl_disp_fifo_empty;
    logic [`N-1:0]            avl_disp_fifo_full;
    logic [1:0]               avl_disp_state;
    logic disp_begin_write_pulse;
    logic disp_begin_dispatch_pulse;
    logic disp_clear_done_pulse;
    logic disp_fifo_all_full;

    // Avalon/software wrapper -> trade_log
    localparam int LOG_IDX_W   = $clog2(TRADE_LOG_DEPTH);
    localparam int LOG_IDX_W_BUS  = (LOG_IDX_W > 30) ? 30 : LOG_IDX_W; // max
index bits in [31:2]
    localparam int LOG_COUNT_W = $clog2(TRADE_LOG_DEPTH + 1);

    logic                      avl_log_read_pulse;
    logic [LOG_IDX_W-1:0]      avl_log_read_index;
    logic [`TRADE_WIDTH-1:0]   avl_log_read_data;
    logic [LOG_COUNT_W-1:0]    avl_log_count;
    logic                      avl_log_overflow;
    logic                      avl_log_clear_pulse;
    logic                      trade_done;

    // Shadow register for software-visible log data
    logic [`TRADE_WIDTH-1:0]   avl_log_data_shadow;
    logic                      avl_log_data_valid;

    // trade_log read is registered:
    // write LOG_CMD(read_req+index) -> pulse sw_re
    // -> trade_log updates sw_rdata next cycle
    // -> capture sw_rdata the cycle after that
    logic [1:0]               avl_log_read_pipe;

    // Dispatcher to engines bus
    logic           [`N-1:0] order_in_valid;
    logic           [`N-1:0] order_in_ready;
    DISPATCH_ORDER  [`N-1:0] order_in_data;

    // Engines to MMU ports
    logic                      mmu_req_valid   [`N];
    logic [31:0]               mmu_req_va      [`N];
    logic                      mmu_req_wr      [`N];
    logic [`ORDER_WIDTH-1:0]   mmu_req_wdata   [`N];
    logic                      mmu_req_ready   [`N];
    logic [`ORDER_WIDTH-1:0]   mmu_resp_data   [`N];
    logic                      mmu_resp_valid  [`N];
```

```
logic                       mmu_resp_reject [`N];

// Engines to trade aggregator.
logic [`N-1:0]              eng_trade_valid;
logic [`ORDER_WIDTH-1:0]    eng_trade_data  [`N];
logic [`N-1:0]              eng_trade_ready;

// Per-engine idle status. all_engines_idle is the AND-reduce, used by
// SW (via the Avalon status register) to detect that every engine has
// drained its trade controller back to T_IDLE.
logic [`N-1:0]              eng_idle;
logic                       all_engines_idle;
assign all_engines_idle = &eng_idle;

// MMU to mem_bank ports
logic [31:0]                mem_addr        [4];
logic                       mem_we          [4];
logic                       mem_re          [4];
logic [`ORDER_WIDTH-1:0]    mem_wdata       [4];
logic [`ORDER_WIDTH-1:0]    mem_rdata       [4];
logic                       mem_rdata_valid [4];
logic                       mem_busy        [4];
logic                       mem_wdone       [4];

// Async resets
logic rst_n_raw;
logic [1:0] rst_pipe;
logic rst_n_i;

// DE1-SoC Buttons & 7-Seg Displays
logic [3:0] key_s0, key_s1, key_prev;
logic [3:0] key_press_pulse;
logic key_begin_write_pulse, key_begin_dispatch_pulse;

////////////////////////////////////////////////////////////////////////
// DE1-Soc Buttons7-Seg Displays
//
// Buttons (Active Low)
//  - KEY[0]: Async reset
//  - KEY[1]: Enable WRITE state in Order Dispatcher
//  - KEY[2]: Start / Enable DISPATCH state in Order Dispatcher
//  - KEY[3]: Nothing
//
// 7 Segment Displays:
//  -
```

```
    ////////////////////////////////////////////////////////////////////

    // Asyn resets with KEY0
    assign rst_n_raw = KEY[0];
    always_ff @(posedge clk or negedge rst_n_raw) begin
        if (!rst_n_raw) rst_pipe <= 2'b00;
        else            rst_pipe <= {rst_pipe[0], 1'b1};
    end
    assign rst_n_i = rst_pipe[1];

    // Button presse handling (for KEY1 and KEY2)
    always_ff @(posedge clk or negedge rst_n_i) begin
        if (!rst_n_i) begin
            key_s0   <= 4'hF;
            key_s1   <= 4'hF;
            key_prev <= 4'hF;
        end else begin
            key_s0   <= KEY;
            key_s1   <= key_s0;
            key_prev <= key_s1;
        end
    end
    assign key_press_pulse = key_prev & ~key_s1;
    assign key_begin_write_pulse    = key_press_pulse[1];
    assign key_begin_dispatch_pulse = key_press_pulse[2];

    // Assign transition pulses to either use software or physical button signals
    assign disp_begin_write_pulse    = avl_disp_begin_write_pulse    |
key_begin_write_pulse;
    assign disp_begin_dispatch_pulse = avl_disp_begin_dispatch_pulse |
key_begin_dispatch_pulse;
    assign disp_clear_done_pulse     = avl_disp_clear_done_pulse; // keep SW-only
for now

    ////////////////////////////////////////////////////////////////////////
    // DE1-SoC 7-Segment Displays
    //
    // Displays do the following:
    //  - Blinks "1" (for "press KEY1") while idle
    //  - Shows "0" while the dispatch FIFOs are being filled
    //  - Blinks "2" (for "press KEY2") after FIFOs get filled
    //  - Nothing while dispatching/trading
    //  - Blinks all segments when trades are done
    // Note that all segments are made to blink/display the same number.
    ////////////////////////////////////////////////////////////////////////
```

```
assign disp_fifo_all_full = &avl_disp_fifo_full;

// Blinking logic
localparam int CLK_HZ      = 50_000_000;
localparam int BLINK_HALF  = CLK_HZ/2; // 0.5 seconds
logic [$clog2(BLINK_HALF)-1:0] blink_ctr;
logic blink;
always_ff @(posedge clk or negedge rst_n_i) begin
    if (!rst_n_i) begin
        blink_ctr <= '0;
        blink     <= 1'b0;
    end else begin
        if (blink_ctr == BLINK_HALF-1) begin
            blink_ctr <= '0;
            blink     <= ~blink;
        end else begin
            blink_ctr <= blink_ctr + 1'b1;
        end
    end
end

// 7 Segment controller
localparam logic [6:0] SEG_OFF = 7'b1111111; // blank
localparam logic [6:0] SEG_ON  = 7'b0000000; // all segments on
logic [3:0] ui_digit;
logic [6:0] seg_digit;
hex7seg u_hex7(.a(ui_digit), .y(seg_digit));
always_comb begin
    // Default: blank everything
    HEX0 = SEG_OFF; HEX1 = SEG_OFF; HEX2 = SEG_OFF;
    HEX3 = SEG_OFF; HEX4 = SEG_OFF; HEX5 = SEG_OFF;
    ui_digit = 4'd0;

    if (trade_done) begin
        // Blink all segments when trading is done
        if (blink) begin
            HEX0 = SEG_ON; HEX1 = SEG_ON; HEX2 = SEG_ON;
            HEX3 = SEG_ON; HEX4 = SEG_ON; HEX5 = SEG_ON;
        end
    end else begin
        unique case (avl_disp_state)
            IDLE: begin
                // Blink "1"
                ui_digit = 4'd1;
                if (blink) begin
```

```
                    HEX0 = seg_digit; HEX1 = seg_digit; HEX2 = seg_digit;
                    HEX3 = seg_digit; HEX4 = seg_digit; HEX5 = seg_digit;
                end
            end
            WRITE: begin
                if (!disp_fifo_all_full) begin
                    // Display all 0
                    ui_digit = 4'd0;
                    HEX0 = seg_digit; HEX1 = seg_digit; HEX2 = seg_digit;
                    HEX3 = seg_digit; HEX4 = seg_digit; HEX5 = seg_digit;
                end else begin
                    // Blink "2" once full
                    ui_digit = 4'd2;
                    if (blink) begin
                        HEX0 = seg_digit; HEX1 = seg_digit; HEX2 = seg_digit;
                        HEX3 = seg_digit; HEX4 = seg_digit; HEX5 = seg_digit;
                    end
                end
            end
            default: begin
                // blank
            end
        endcase
    end
end

////////////////////////////////////////////////////////////////////////
// Translation Wrapper
//
// This decodes or encodes information so that we can communicate with
// the Linux core over the Avalon bus.
////////////////////////////////////////////////////////////////////////

// Register Map
localparam int ADDR_CONTROL   = 5'd0;  // 0x00
localparam int ADDR_STATUS    = 5'd1;  // 0x04
localparam int ADDR_PUSH0     = 5'd2;  // 0x08
localparam int ADDR_PUSH1     = 5'd3;  // 0x0C
localparam int ADDR_PUSH2     = 5'd4;  // 0x10
localparam int ADDR_PUSH3     = 5'd5;  // 0x14
localparam int ADDR_PUSH4     = 5'd6;  // 0x18
localparam int ADDR_PUSH5     = 5'd7;  // 0x1C
localparam int ADDR_PUSH6     = 5'd8;  // 0x20
localparam int ADDR_PUSH7     = 5'd9;  // 0x24
localparam int ADDR_LOG_INFO  = 5'd10; // 0x28
```

```
localparam int ADDR_LOG_CMD   = 5'd11; // 0x2C
localparam int ADDR_LOG_DATA0 = 5'd12; // 0x30
localparam int ADDR_LOG_DATA1 = 5'd13; // 0x34

// Decode Avalon writes into dispatcher and trade-log controls
always_ff @(posedge clk or negedge rst_n_i) begin
    if (!rst_n_i) begin
        avl_disp_begin_write_pulse    <= 1'b0;
        avl_disp_begin_dispatch_pulse <= 1'b0;
        avl_disp_clear_done_pulse     <= 1'b0;
        avl_disp_push_en              <= '0;

        avl_log_read_pulse            <= 1'b0;
        avl_log_read_index            <= '0;
        avl_log_clear_pulse           <= 1'b0;
        avl_log_data_shadow           <= '0;
        avl_log_data_valid            <= 1'b0;
        avl_log_read_pipe             <= 2'b00;

        for (int i = 0; i < `N; i++) begin
            avl_disp_push_data[i] <= '0;
        end

    end else begin
        // Default pulse behavior: one cycle only
        avl_disp_begin_write_pulse    <= 1'b0;
        avl_disp_begin_dispatch_pulse <= 1'b0;
        avl_disp_clear_done_pulse     <= 1'b0;
        avl_disp_push_en              <= '0;

        avl_log_read_pulse            <= 1'b0;
        avl_log_clear_pulse           <= 1'b0;

        // trade_log registered read timing:
        // 01 -> read request launched
        // 10 -> capture returned data
        if (avl_log_read_pipe[0]) begin
            avl_log_read_pipe <= 2'b10;
        end else if (avl_log_read_pipe[1]) begin
            avl_log_read_pipe   <= 2'b00;
            avl_log_data_shadow <= avl_log_read_data;
            avl_log_data_valid  <= 1'b1;
        end

        if (chipselect && write) begin
```

```
            unique case (address)
                ADDR_CONTROL: begin
                    avl_disp_begin_write_pulse    <= writedata[0];
                    avl_disp_begin_dispatch_pulse <= writedata[1];
                    avl_disp_clear_done_pulse     <= writedata[2];
                end
                ADDR_PUSH0: begin
                    if (avl_disp_push_ready[0]) begin
                        avl_disp_push_en[0]   <= 1'b1;
                        avl_disp_push_data[0] <= DISPATCH_ORDER'(writedata);
                    end
                end
                ADDR_PUSH1: begin
                    if (avl_disp_push_ready[1]) begin
                        avl_disp_push_en[1]   <= 1'b1;
                        avl_disp_push_data[1] <= DISPATCH_ORDER'(writedata);
                    end
                end
                ADDR_PUSH2: begin
                    if (avl_disp_push_ready[2]) begin
                        avl_disp_push_en[2]   <= 1'b1;
                        avl_disp_push_data[2] <= DISPATCH_ORDER'(writedata);
                    end
                end
                ADDR_PUSH3: begin
                    if (avl_disp_push_ready[3]) begin
                        avl_disp_push_en[3]   <= 1'b1;
                        avl_disp_push_data[3] <= DISPATCH_ORDER'(writedata);
                    end
                end
                ADDR_PUSH4: begin
                    if (avl_disp_push_ready[4]) begin
                        avl_disp_push_en[4]   <= 1'b1;
                        avl_disp_push_data[4] <= DISPATCH_ORDER'(writedata);
                    end
                end
                ADDR_PUSH5: begin
                    if (avl_disp_push_ready[5]) begin
                        avl_disp_push_en[5]   <= 1'b1;
                        avl_disp_push_data[5] <= DISPATCH_ORDER'(writedata);
                    end
                end
                ADDR_PUSH6: begin
                    if (avl_disp_push_ready[6]) begin
                        avl_disp_push_en[6]   <= 1'b1;
```

```
                        avl_disp_push_data[6] <= DISPATCH_ORDER'(writedata);
                    end
                end
                ADDR_PUSH7: begin
                    if (avl_disp_push_ready[7]) begin
                        avl_disp_push_en[7]   <= 1'b1;
                        avl_disp_push_data[7] <= DISPATCH_ORDER'(writedata);
                    end
                end
                ADDR_LOG_CMD: begin
                    // bit 0 = clear
                    // bit 1 = read_req
                    // bits [LOG_IDX_W_BUS+1:2] = read_index
                    if (writedata[0]) begin
                        avl_log_clear_pulse <= 1'b1;
                        avl_log_data_shadow <= '0;
                        avl_log_data_valid  <= 1'b0;
                        avl_log_read_pipe   <= 2'b00;
                    end else if (writedata[1]) begin
                        avl_log_read_index  <= writedata[LOG_IDX_W_BUS+1:2];
                        avl_log_read_pulse  <= 1'b1;
                        avl_log_data_valid  <= 1'b0;
                        avl_log_read_pipe   <= 2'b01;
                    end
                end
                default: ;
            endcase
        end
    end
end

// Decode Avalon reads from dispatcher and trade-log status/data
always_comb begin
    readdata = 32'd0;

    if (chipselect && read) begin
        case (address)
            ADDR_STATUS: readdata = {6'd0,
                           avl_disp_fifo_full,
                           avl_disp_fifo_empty,
                           avl_disp_push_ready,
                           avl_disp_state};
            ADDR_LOG_INFO: begin
                // [0]    = overflow
                // [1]    = selected log entry valid in DATA0/1/2
```

```
                // [2 +: LOG_COUNT_W] = number of valid entries in trade_log
                // [17 +: 'N]         = heap engine idle indicator
                // [25]   = All heap engines are idle signal
                // [26]   = all trades are done signal
                // [31:27] = nothing
                readdata = 32'd0;
                readdata[0] = avl_log_overflow;
                readdata[1] = avl_log_data_valid;
                readdata[2 +: LOG_COUNT_W] = avl_log_count;
                readdata[17 +: `N] = eng_idle;
                readdata[25] = all_engines_idle;
                readdata[26] = trade_done;
            end
            // Compact 64-bit trade entry packed into 2 words.
            ADDR_LOG_DATA0: readdata = avl_log_data_shadow[31:0];
            ADDR_LOG_DATA1: readdata = avl_log_data_shadow[63:32];
            default: ;
        endcase
    end
end

////////////////////////////////////////////////////////////////////////
// Module Instantiation & Connection
////////////////////////////////////////////////////////////////////////

// Order Dispatcher
order_dispatcher u_dispatcher (
    .clk              (clk),
    .rst_n            (rst_n_i),

    // Avalon/software wrapper controls
    .sw_begin_write    (disp_begin_write_pulse),
    .sw_begin_dispatch (disp_begin_dispatch_pulse),
    .sw_clear_done     (disp_clear_done_pulse),
    .sw_wr_en          (avl_disp_push_en),
    .sw_wr_data        (avl_disp_push_data),
    .sw_wr_ready       (avl_disp_push_ready),

    // FIFO state visible to Avalon/software
    .state_out         (avl_disp_state),
    .fifo_empty        (avl_disp_fifo_empty),
    .fifo_full         (avl_disp_fifo_full),

    // Heap Engine communication
    .order_in_ready    (order_in_ready),
```

```
    .order_out_valid   (order_in_valid),
    .order_out         (order_in_data)
);

// 8 Symbol Engines (one per symbol slot, ENGINE_ID = 0..7).
// SYMBOL parameter defaults to the right ASCII for the dataset.
genvar e;
generate
    for (e = 0; e < `N; e++) begin : g_engines
        symbol_engine #(.ENGINE_ID(e)) u_engine (
            .clk             (clk),
            .rst_n           (rst_n_i),
            .now_ts          (now_ts),
            .order_in_valid  (order_in_valid[e]),
            .order_in_data   (order_in_data[e]),
            .order_in_ready  (order_in_ready[e]),
            .trade_out_valid (eng_trade_valid[e]),
            .trade_out_data  (eng_trade_data[e]),
            .trade_out_ready (eng_trade_ready[e]),
            .mmu_req_valid   (mmu_req_valid[e]),
            .mmu_req_va      (mmu_req_va[e]),
            .mmu_req_wr      (mmu_req_wr[e]),
            .mmu_req_wdata   (mmu_req_wdata[e]),
            .mmu_req_ready   (mmu_req_ready[e]),
            .mmu_resp_data   (mmu_resp_data[e]),
            .mmu_resp_valid  (mmu_resp_valid[e]),
            .mmu_resp_reject (mmu_resp_reject[e]),
            .bid_size_o      (), // Leave disconnected for now
            .ask_size_o      (),
            .engine_idle     (eng_idle[e])
        );
    end
endgenerate

// MMU (8 individual client ports, 4 mem_bank ports)
mmu u_mmu (
    .clk    (clk),
    .rst_n  (rst_n_i),

    .req_valid_0(mmu_req_valid[0]), .req_valid_1(mmu_req_valid[1]),
    .req_valid_2(mmu_req_valid[2]), .req_valid_3(mmu_req_valid[3]),
    .req_valid_4(mmu_req_valid[4]), .req_valid_5(mmu_req_valid[5]),
    .req_valid_6(mmu_req_valid[6]), .req_valid_7(mmu_req_valid[7]),

    .req_va_0(mmu_req_va[0]), .req_va_1(mmu_req_va[1]),
```

```
.req_va_2(mmu_req_va[2]), .req_va_3(mmu_req_va[3]),
.req_va_4(mmu_req_va[4]), .req_va_5(mmu_req_va[5]),
.req_va_6(mmu_req_va[6]), .req_va_7(mmu_req_va[7]),

.req_wr_0(mmu_req_wr[0]), .req_wr_1(mmu_req_wr[1]),
.req_wr_2(mmu_req_wr[2]), .req_wr_3(mmu_req_wr[3]),
.req_wr_4(mmu_req_wr[4]), .req_wr_5(mmu_req_wr[5]),
.req_wr_6(mmu_req_wr[6]), .req_wr_7(mmu_req_wr[7]),

.req_wdata_0(mmu_req_wdata[0]), .req_wdata_1(mmu_req_wdata[1]),
.req_wdata_2(mmu_req_wdata[2]), .req_wdata_3(mmu_req_wdata[3]),
.req_wdata_4(mmu_req_wdata[4]), .req_wdata_5(mmu_req_wdata[5]),
.req_wdata_6(mmu_req_wdata[6]), .req_wdata_7(mmu_req_wdata[7]),

.req_ready_0(mmu_req_ready[0]), .req_ready_1(mmu_req_ready[1]),
.req_ready_2(mmu_req_ready[2]), .req_ready_3(mmu_req_ready[3]),
.req_ready_4(mmu_req_ready[4]), .req_ready_5(mmu_req_ready[5]),
.req_ready_6(mmu_req_ready[6]), .req_ready_7(mmu_req_ready[7]),

.resp_data_0(mmu_resp_data[0]), .resp_data_1(mmu_resp_data[1]),
.resp_data_2(mmu_resp_data[2]), .resp_data_3(mmu_resp_data[3]),
.resp_data_4(mmu_resp_data[4]), .resp_data_5(mmu_resp_data[5]),
.resp_data_6(mmu_resp_data[6]), .resp_data_7(mmu_resp_data[7]),

.resp_valid_0(mmu_resp_valid[0]), .resp_valid_1(mmu_resp_valid[1]),
.resp_valid_2(mmu_resp_valid[2]), .resp_valid_3(mmu_resp_valid[3]),
.resp_valid_4(mmu_resp_valid[4]), .resp_valid_5(mmu_resp_valid[5]),
.resp_valid_6(mmu_resp_valid[6]), .resp_valid_7(mmu_resp_valid[7]),

.resp_reject_0(mmu_resp_reject[0]), .resp_reject_1(mmu_resp_reject[1]),
.resp_reject_2(mmu_resp_reject[2]), .resp_reject_3(mmu_resp_reject[3]),
.resp_reject_4(mmu_resp_reject[4]), .resp_reject_5(mmu_resp_reject[5]),
.resp_reject_6(mmu_resp_reject[6]), .resp_reject_7(mmu_resp_reject[7]),

.mem_addr_0(mem_addr[0]), .mem_addr_1(mem_addr[1]),
.mem_addr_2(mem_addr[2]), .mem_addr_3(mem_addr[3]),

.mem_we_0(mem_we[0]), .mem_we_1(mem_we[1]),
.mem_we_2(mem_we[2]), .mem_we_3(mem_we[3]),

.mem_re_0(mem_re[0]), .mem_re_1(mem_re[1]),
.mem_re_2(mem_re[2]), .mem_re_3(mem_re[3]),

.mem_wdata_0(mem_wdata[0]), .mem_wdata_1(mem_wdata[1]),
.mem_wdata_2(mem_wdata[2]), .mem_wdata_3(mem_wdata[3]),
```

```
        .mem_rdata_0(mem_rdata[0]), .mem_rdata_1(mem_rdata[1]),
        .mem_rdata_2(mem_rdata[2]), .mem_rdata_3(mem_rdata[3]),

        .mem_rdata_valid_0(mem_rdata_valid[0]),
.mem_rdata_valid_1(mem_rdata_valid[1]),
        .mem_rdata_valid_2(mem_rdata_valid[2]),
.mem_rdata_valid_3(mem_rdata_valid[3]),

        .mem_busy_0(mem_busy[0]), .mem_busy_1(mem_busy[1]),
        .mem_busy_2(mem_busy[2]), .mem_busy_3(mem_busy[3]),

        .mem_wdone_0(mem_wdone[0]), .mem_wdone_1(mem_wdone[1]),
        .mem_wdone_2(mem_wdone[2]), .mem_wdone_3(mem_wdone[3])
    );

    // 4 Memory Banks
    genvar b;
    generate
        for (b = 0; b < 4; b++) begin : g_banks
            mem_bank #(.BANK_ID(b)) u_bank (
                .clk             (clk),
                .rst_n           (rst_n_i),
                .mem_addr        (mem_addr[b]),
                .mem_we          (mem_we[b]),
                .mem_re          (mem_re[b]),
                .mem_wdata       (mem_wdata[b]),
                .mem_rdata       (mem_rdata[b]),
                .mem_rdata_valid (mem_rdata_valid[b]),
                .mem_busy        (mem_busy[b]),
                .mem_wdone       (mem_wdone[b])
            );
        end
    endgenerate

    // Trade Output Aggregator (round-robin between 8 engines) feeds Trade Log.
    // Aggregator compacts the 86-bit engine ORDER into a 64-bit TRADE_LOG_ENTRY

    logic                       agg_trade_valid;
    logic [`TRADE_WIDTH-1:0]    agg_trade_data;
    logic                       agg_trade_ready;

    // Denote when trades are sw_clear_done
    assign trade_done = (avl_disp_state == DONE) && all_engines_idle;
```

```
    trade_aggregator #(
        .N           (`N),
        .NODE_WIDTH  (`ORDER_WIDTH),
        .TRADE_WIDTH (`TRADE_WIDTH)
    ) u_trade_agg (
        .clk             (clk),
        .rst_n           (rst_n_i),
        .eng_trade_valid (eng_trade_valid),
        .eng_trade_data  (eng_trade_data),
        .eng_trade_ready (eng_trade_ready),
        .trade_out_valid (agg_trade_valid),
        .trade_out_data  (agg_trade_data),
        .trade_out_ready (agg_trade_ready)
    );

    trade_log #(
        .NODE_WIDTH (`TRADE_WIDTH),
        .LOG_DEPTH  (TRADE_LOG_DEPTH)
    ) u_trade_log (
        .clk            (clk),
        .rst_n          (rst_n_i),
        .trade_in_valid (agg_trade_valid),
        .trade_in_data  (agg_trade_data),
        .trade_in_ready (agg_trade_ready),
        .sw_re          (avl_log_read_pulse),
        .sw_addr        (avl_log_read_index),
        .sw_rdata       (avl_log_read_data),
        .sw_count       (avl_log_count),
        .sw_overflow    (avl_log_overflow),
        .sw_clear       (avl_log_clear_pulse)
    );

endmodule
```

None

```
//////////////////////////////////////////////////////////////////////////////////
/
// Engineers: Carlos Espinoza
// Create Date: 04/07/2026
// Project Name: Virtualized High Frequence Trading (HFT) Simulator
// Design Name: FIFO BRAM
// Description:
//      FIFO BRAM module for storing dispatch orders in BRAM.
```

```
//
// Revision: 05/11/2026
//////////////////////////////////////////////////////////////////////////////////
/

module dispatch_fifo_bram #(
    parameter int WIDTH = 32,
    parameter int DEPTH = 1660,
    parameter int ADDR_W = $clog2(DEPTH)
) (
    input  logic             clk,
    input  logic             rst_n,

    // Write Interface
    input  logic             wr_en,
    input  logic [WIDTH-1:0] wr_data,
    output logic             full,

    // Dispatche/Read Interface
    input  logic             dispatch_en,
    input  logic             out_ready,
    output logic             out_valid,
    output logic [WIDTH-1:0] out_data,
    output logic             empty,

    // Clear batch
    input  logic             clear
);
    localparam int PTR_W = $clog2(DEPTH);
    localparam int CNT_W = $clog2(DEPTH + 1);

    // One BRAM-backed storage array per lane
    (* ramstyle = "no_rw_check, M10K", ram_init_file = "dispatch_fifo_zero.mif"
*)
    logic [WIDTH-1:0] mem [0:DEPTH-1];

    // Pointers:
    // wr_ptr = next location to write
    // rd_ptr = next location still in RAM (not prefetched)
    logic [PTR_W-1:0] wr_ptr, rd_ptr;

    // Number of entries still resident in RAM only
    logic [CNT_W-1:0] mem_count;
```

```
    // One read has been launched; data will appear in out_data after the BRAM
read
    logic rd_pending;
    logic [CNT_W-1:0] occupancy;
    logic             do_write, do_launch_read, do_consume;

    // Total logical occupancy includes:
    //   - entries still in RAM
    //   - one buffered output word (out_valid)
    //   - one in-flight BRAM read (rd_pending)
    assign occupancy = mem_count
                     + {{CNT_W-1{1'b0}}, out_valid}
                     + {{CNT_W-1{1'b0}}, rd_pending};
    assign empty = (occupancy == CNT_W'(0));
    assign full  = (occupancy == CNT_W'(DEPTH));
    assign do_write = wr_en && !full;

    // Hold current output stable while consumer is not ready.
    assign do_consume = dispatch_en && out_valid && out_ready;

    // Launch a BRAM read when:
    //   - we are dispatching
    //   - no read already in flight
    //   - there is still data in RAM
    //   - output register is free now, or will be freed this cycle by consume
    assign do_launch_read = dispatch_en && !rd_pending
                         && (mem_count != CNT_W'(0))
                         && (!out_valid || out_ready);

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            wr_ptr     <= '0;
            rd_ptr     <= '0;
            mem_count  <= '0;
            rd_pending <= 1'b0;
            out_valid  <= 1'b0;
            out_data   <= '0;
        end else if (clear) begin
            wr_ptr     <= '0;
            rd_ptr     <= '0;
            mem_count  <= '0;
            rd_pending <= 1'b0;
            out_valid  <= 1'b0;
            out_data   <= '0;
        end else begin
```

```
            // Write
            if (do_write) begin
                mem[wr_ptr] <= wr_data;
                wr_ptr      <= wr_ptr + PTR_W'(1);
            end

            // Do synchronous BRAM read
            if (do_launch_read) begin
                out_data   <= mem[rd_ptr];
                rd_ptr     <= rd_ptr + PTR_W'(1);
                rd_pending <= 1'b1;
            end

            // Place data in out_data register after the cycle wait
            if (rd_pending) begin
                out_valid  <= 1'b1;
                rd_pending <= 1'b0;
            end

            // Indicate data is not yet ready after engine takes it.
            if (do_consume) begin
                out_valid <= 1'b0;
            end
            // Count of entries still inside RAM storage
            unique case ({do_write, do_launch_read})
                2'b10:   mem_count <= mem_count + CNT_W'(1);
                2'b01:   mem_count <= mem_count - CNT_W'(1);
                default: mem_count <= mem_count;
            endcase
        end
    end
endmodule
```

None

```
//////////////////////////////////////////////////////////////////////////////////
/
// Engineers: Carlos Espinoza
// Create Date: 04/07/2026
// Project Name: Virtualized High Frequence Trading (HFT) Simulator
// Design Name: Input Interface & Order Dispatcher
// Description:
//      This is a simple file to define parameters and structs that will be used
//      across the various modules in this project.
```

```
//
// Revision: 05/08/2026
////////////////////////////////////////////////////////////////////////////////
/
`include "../sys_def.svh"

module order_dispatcher(
    input logic                   clk,
       input logic                rst_n,

       // Interface with software harness
       input logic                   sw_begin_write,     // Enables transition to
WRITE state
       input logic                   sw_begin_dispatch,  // Enables transition to
DISPATCH state
       input logic                   sw_clear_done,      // Enable transition to
IDLE after its done
       input logic [`N-1:0]          sw_wr_en,           // Enable writing to
FIFO tails
       input DISPATCH_ORDER [`N-1:0] sw_wr_data,         // Order to write to
FIFO tails
       output logic [`N-1:0]         sw_wr_ready,        // Signal that FIDO is
ready to write

       // FIFO state (per symbol/lane)
       output logic [1:0]            state_out,          // Current state of
Dispatcher
       output logic [`N-1:0]         fifo_empty,         // Per FIFO empty
signals
       output logic [`N-1:0]         fifo_full,          // Per FIFO full signals

       // Communication with Heap Engines
       input  logic [`N-1:0]         order_in_ready,     // Per-lane ready from
each engine: high when they can accept order
       output logic [`N-1:0]         order_out_valid,    // High when order_out
holds a real order (not trash)
       output DISPATCH_ORDER [`N-1:0]order_out          // Order presented to
engine (consumed when ready)
);

////////////////////////////////////////////////////////////////////////////////
/
    // FSM Controller
```

```
////////////////////////////////////////////////////////////////////////////////
/

    // States
    DISPATCH_STATE state, next_state;

    // FIFO wires declared up here so strict-SV simulators (Questa, VCS)
    // accept the references in the next_state logic and output assigns
    // below. The genvar block further down also drives them.
    localparam int DISP_W = $bits(DISPATCH_ORDER);
    logic [`N-1:0]     fifo_empty_i;
    logic [`N-1:0]     fifo_full_i;
    logic [`N-1:0]     fifo_out_valid_i;
    logic [DISP_W-1:0] fifo_out_bits [`N-1:0];

    // Next State Logic
    always_comb begin
        // Defaults
        next_state = state;

        // Transitions
        // Note: tansitions to DISPATCH is SW controlled to enable dipatching
        // when FIFOS have less than FIFO_SZ orders.
        unique case (state)
        IDLE:     next_state = sw_begin_write ?  WRITE:IDLE;
        WRITE:    next_state = sw_begin_dispatch ? DISPATCH:WRITE;
        DISPATCH: next_state = (&fifo_empty_i) ? DONE:DISPATCH;
        DONE:     next_state = sw_clear_done ? IDLE : DONE;
        default:; //Handled
        endcase
    end

    // Update State & FIFOs
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            state <= IDLE;
        else
            state <= next_state;
    end

    assign state_out  = state;
    assign fifo_empty = fifo_empty_i;
    assign fifo_full  = fifo_full_i;
```

```
////////////////////////////////////////////////////////////////////////////////
/
    // FIFOs
    // There are N number of FIFOs (i.e. 8), one per symbol/stock
    //  - No need to store symbol since each fifo stores only for one fifo.
    //  - No to store the timestamp since the FIFO
    // Note that FIFOs are in BRAM now, so there is a 1 cycle read delay.

////////////////////////////////////////////////////////////////////////////////
/
    genvar s;
    generate
        for (s = 0; s < `N; s++) begin : g_dispatch_fifo
            dispatch_fifo_bram #(.WIDTH (DISP_W),.DEPTH (`FIFO_SZ)) u_fifo (
                .clk         (clk),
                .rst_n       (rst_n),
                .wr_en       ((state == WRITE) && sw_wr_en[s]),
                .wr_data     (sw_wr_data[s]),
                .full        (fifo_full_i[s]),
                .dispatch_en (state == DISPATCH),
                .out_ready   ((state == DISPATCH) && order_in_ready[s]),
                .out_valid   (fifo_out_valid_i[s]),
                .out_data    (fifo_out_bits[s]),
                .empty       (fifo_empty_i[s]),
                .clear       ((state == DONE) && sw_clear_done)
            );

            assign sw_wr_ready[s]   = (state == WRITE)    && !fifo_full_i[s];
            assign order_out_valid[s] = (state == DISPATCH) &&
fifo_out_valid_i[s];
            assign order_out[s] = DISPATCH_ORDER'(fifo_out_bits[s]);
        end
    endgenerate
endmodule
```

None

```
// heap_fsm.sv: Single-heap FSM with tiered storage
//
// Drives one priority-ordered heap (max or min) for one symbol slot.
// Used in pairs by symbol_engine: a max-heap for bids, a min-heap for asks.
//
// Storage:
```

```
//   Indices [0 .. PRIVATE_NODES-1] live in a single-cycle private BRAM
//   accessed directly through the priv_* port group.
//
//   Indices [PRIVATE_NODES .. MAX_NODES-1] live in MMU-mediated virtual memory
//   accessed through the virt_* port group. ENGINE_ID is hard-wired into
//   VA[31:29] per the engine/MMU contract.
//
// Operations (cmd_op):
//   OP_PUSH    Insert cmd_data_in at the next leaf and sift up.
//   OP_POP     Extract the root, replace it with the last leaf, sift down.
//              The extracted root is presented on cmd_root_out.
//   OP_PEEK    Return the root on cmd_root_out (uses an internal cache).
//   OP_UPDATE  Overwrite the root with cmd_data_in. The caller is
//              responsible for preserving heap order (price + timestamp
//              unchanged); used for partial-fill amount edits.
//
//
// Command handshake: cmd_valid is held by the caller until cmd_done is
// observed. cmd_ready is high only while the FSM is in S_IDLE.

module heap_fsm #(
    parameter int  HEAP_KIND     = 0,        // 0 = MAX-heap (bids)
                                             // 1 = MIN-heap (asks)
    parameter int  ENGINE_ID     = 0,        // 0..7, drives VA[31:29]
    parameter int  NODE_WIDTH    = 86,
    parameter int  PRIVATE_NODES = 64,       // size of the private BRAM Memory
    parameter int  MAX_NODES     = 1088,     // 64 private + 1024 virtual; 1024
cap

    parameter int  IDX_WIDTH     = $clog2(MAX_NODES + 1)
) (
    input  logic                  clk,
    input  logic                  rst_n,

    // command interface (from the per-symbol trade controller)
    input  logic                  cmd_valid,
    input  logic [1:0]            cmd_op,
    input  logic [NODE_WIDTH-1:0] cmd_data_in,
    output logic                  cmd_ready,
    output logic                  cmd_done,
    output logic [NODE_WIDTH-1:0] cmd_root_out,
    output logic [13:0]           size_out,

    // private memory (single-cycle BRAM)
    output logic                  priv_we,
```

```
    output logic                  priv_re,
    output logic [5:0]            priv_addr,
    output logic [NODE_WIDTH-1:0] priv_wdata,
    input  logic [NODE_WIDTH-1:0] priv_rdata,

    // virtual memory (engine's MMU client port)
    output logic                  virt_req_valid,
    output logic [31:0]           virt_req_va,
    output logic                  virt_req_wr,
    output logic [NODE_WIDTH-1:0] virt_req_wdata,
    input  logic                  virt_req_ready,
    input  logic                  virt_resp_valid,
    input  logic [NODE_WIDTH-1:0] virt_resp_data,
    input  logic                  virt_resp_reject
);

    // Op-code constants and heap-kind aliases

    localparam int MAX_HEAP = 0;
    localparam int MIN_HEAP = 1;

    localparam logic [1:0] OP_PUSH   = 2'd0;
    localparam logic [1:0] OP_POP    = 2'd1;
    localparam logic [1:0] OP_PEEK   = 2'd2;
    localparam logic [1:0] OP_UPDATE = 2'd3;

    // Node-field layout inside the 86-bit payload
    //   [85]     type
    //   [84:69]  price
    //   [68:53]  amount
    //   [52:32]  symbol  (3 x 7-bit ASCII)
    //   [31:0]   timestamp
    //
    //
    // Comparator (operates on pre-extracted fields). Returns 1 iff the
    // operand identified by the "a" fields should sit closer to the root
    // than the "b" operand. .

    function automatic logic a_wins_fields (
        input logic [15:0] ap, input logic [31:0] at,
        input logic [15:0] bp, input logic [31:0] bt
    );
        if (ap != bp)
            a_wins_fields = (HEAP_KIND == MAX_HEAP) ? (ap > bp) : (ap < bp);
        else
```

```
        a_wins_fields = (at < bt);
endfunction

// Virtual-address construction
//
// VA layout:
//   [31:29]  engine_id     (hard-wired, used by MMU for routing)
//   [28:11]  reserved / zero (ignored by MMU)
//   [10]     heap_kind     (0 = bid, 1 = ask)
//   [9:0]    heap-relative virtual node index (idx - PRIVATE_NODES)


function automatic logic [31:0] make_va (input logic [IDX_WIDTH-1:0] idx);
    logic [9:0] virt_idx;
    virt_idx = idx - PRIVATE_NODES;
    make_va = {ENGINE_ID[2:0], 18'd0, HEAP_KIND[0], virt_idx};
endfunction

function automatic logic is_private (input logic [IDX_WIDTH-1:0] idx);
    is_private = (idx < PRIVATE_NODES);
endfunction

// State enum

typedef enum logic [4:0] {
    S_IDLE,
    // PUSH (sift up)
    S_PUSH_LAUNCH,
    S_PUSH_RD_PARENT_ISSUE,
    S_PUSH_RD_PARENT_WAIT,
    S_PUSH_DECIDE,
    S_PUSH_WR_DOWN_ISSUE,
    S_PUSH_WR_DOWN_WAIT,
    S_PUSH_WR_FINAL_ISSUE,
    S_PUSH_WR_FINAL_WAIT,
    // POP (extract root + sift down)
    S_POP_LAUNCH,
    S_POP_RD_ROOT_ISSUE,
    S_POP_RD_ROOT_WAIT,
    S_POP_RD_LAST_ISSUE,
    S_POP_RD_LAST_WAIT,
    S_POP_SIFT_RD_LEFT_ISSUE,
    S_POP_SIFT_RD_LEFT_WAIT,
    S_POP_SIFT_RD_RIGHT_ISSUE,
    S_POP_SIFT_RD_RIGHT_WAIT,
```

```
    S_POP_SIFT_DECIDE,
    S_POP_SIFT_WR_UP_ISSUE,
    S_POP_SIFT_WR_UP_WAIT,
    S_POP_SIFT_WR_FINAL_ISSUE,
    S_POP_SIFT_WR_FINAL_WAIT,
    // PEEK
    S_PEEK_RD_ISSUE,
    S_PEEK_RD_WAIT,
    // UPDATE
    S_UPDATE_WR_ISSUE,
    S_UPDATE_WR_WAIT,
    // common terminator
    S_DONE
} state_t;

state_t state, n_state;


logic [IDX_WIDTH-1:0]  size;
logic [IDX_WIDTH-1:0]  target_idx;     // current "hole" being filled
logic [IDX_WIDTH-1:0]  parent_idx;     // (target_idx-1) >> 1 cache
logic [IDX_WIDTH-1:0]  left_idx, right_idx;
logic [NODE_WIDTH-1:0] cur_node;       // node being placed
logic [NODE_WIDTH-1:0] parent_node;    // last parent read (push)
logic [NODE_WIDTH-1:0] left_node, right_node;
logic                  has_right;

// Top-3 node cache. Stores the last-known values of indices 0, 1, 2
// so that the trade controller's PEEK is single-cycle and the first
// sift-down step after POP can skip both child reads.
logic [NODE_WIDTH-1:0] node_cache       [3];
logic [2:0]            node_cache_valid;


wire [NODE_WIDTH-1:0]  root_cache       = node_cache[0];
wire                   root_cache_valid = node_cache_valid[0];

logic [NODE_WIDTH-1:0] saved_op_data;
logic [1:0]            saved_op;
logic [NODE_WIDTH-1:0] popped_root;



wire [15:0] cur_price     = cur_node[84:69];
wire [31:0] cur_ts        = cur_node[31:0];
```

```
wire [15:0] parent_price  = parent_node[84:69];
wire [31:0] parent_ts     = parent_node[31:0];
wire [15:0] left_price    = left_node[84:69];
wire [31:0] left_ts       = left_node[31:0];
wire [15:0] right_price   = right_node[84:69];
wire [31:0] right_ts      = right_node[31:0];

logic [15:0] cmp0_ap, cmp0_bp;
logic [15:0] cmp1_ap, cmp1_bp;
logic [31:0] cmp0_at, cmp0_bt;
logic [31:0] cmp1_at, cmp1_bt;

wire cmp0_result = a_wins_fields(cmp0_ap, cmp0_at, cmp0_bp, cmp0_bt);
wire cmp1_result = a_wins_fields(cmp1_ap, cmp1_at, cmp1_bp, cmp1_bt);

// Pop-sift

wire in_pop_sift = (state == S_POP_SIFT_RD_LEFT_ISSUE)
                || (state == S_POP_SIFT_RD_LEFT_WAIT)
                || (state == S_POP_SIFT_RD_RIGHT_ISSUE)
                || (state == S_POP_SIFT_RD_RIGHT_WAIT)
                || (state == S_POP_SIFT_DECIDE)
                || (state == S_POP_SIFT_WR_UP_ISSUE)
                || (state == S_POP_SIFT_WR_UP_WAIT);

always_comb begin
    if (in_pop_sift) begin
        cmp0_ap = left_price;   cmp0_at = left_ts;
        cmp0_bp = cur_price;    cmp0_bt = cur_ts;
    end else begin
        cmp0_ap = cur_price;    cmp0_at = cur_ts;
        cmp0_bp = parent_price; cmp0_bt = parent_ts;
    end
    cmp1_ap = right_price;
    cmp1_at = right_ts;
    cmp1_bp = cmp0_result ? left_price : cur_price;
    cmp1_bt = cmp0_result ? left_ts    : cur_ts;
end


// Aliases for next-state logic readability.
wire cur_wins_parent     = cmp0_result;   // valid in S_PUSH_DECIDE
wire left_wins_cur       = cmp0_result;   // valid in S_POP_SIFT_DECIDE
wire right_wins_combined = cmp1_result;   // valid in S_POP_SIFT_DECIDE
```

```
    wire [15:0] root_cache_price = root_cache[84:69];
    wire [31:0] root_cache_ts    = root_cache[31:0];
    wire [15:0] cmd_in_price     = cmd_data_in[84:69];
    wire [31:0] cmd_in_ts        = cmd_data_in[31:0];
    wire        update_keys_ok   = (cmd_in_price == root_cache_price)
                                && (cmd_in_ts    == root_cache_ts);

    // Sift-down child selector

    typedef enum logic [1:0] { BEST_CUR, BEST_LEFT, BEST_RIGHT } best_t;
    best_t best;
    always_comb begin
        best = BEST_CUR;
        if (left_idx < size && left_wins_cur)
            best = BEST_LEFT;
        // cmp1 is already wired to right-vs-(left if left_wins_cur else cur),
        // so right_wins_combined is the right answer regardless of best so far.
        if (has_right && right_idx < size && right_wins_combined)
            best = BEST_RIGHT;
    end

    // Memory-issue decode


    logic                  mem_do_read;
    logic                  mem_do_write;
    logic [IDX_WIDTH-1:0]  mem_idx;
    logic [NODE_WIDTH-1:0] mem_wdata;

    always_comb begin
        mem_do_read  = 1'b0;
        mem_do_write = 1'b0;
        mem_idx      = '0;
        mem_wdata    = '0;
        unique case (state)
            S_PUSH_RD_PARENT_ISSUE:    begin mem_do_read  = 1'b1; mem_idx =
parent_idx; end
            S_PUSH_WR_DOWN_ISSUE:      begin mem_do_write = 1'b1; mem_idx =
target_idx; mem_wdata = parent_node; end
            S_PUSH_WR_FINAL_ISSUE:     begin mem_do_write = 1'b1; mem_idx =
target_idx; mem_wdata = cur_node;    end
            S_POP_RD_ROOT_ISSUE:       begin mem_do_read  = 1'b1; mem_idx = '0;
end
            S_POP_RD_LAST_ISSUE:       begin mem_do_read  = 1'b1; mem_idx = size
- 1'b1; end
```

```
            S_POP_SIFT_RD_LEFT_ISSUE:  begin mem_do_read  = 1'b1; mem_idx =
left_idx;    end
            S_POP_SIFT_RD_RIGHT_ISSUE: begin mem_do_read  = 1'b1; mem_idx =
right_idx;   end
            S_POP_SIFT_WR_UP_ISSUE:    begin mem_do_write = 1'b1; mem_idx =
target_idx;
                mem_wdata = (best == BEST_LEFT) ? left_node : right_node;
            end
            S_POP_SIFT_WR_FINAL_ISSUE: begin mem_do_write = 1'b1; mem_idx =
target_idx; mem_wdata = cur_node; end
            S_PEEK_RD_ISSUE:           begin mem_do_read  = 1'b1; mem_idx = '0;
end
            S_UPDATE_WR_ISSUE:         begin mem_do_write = 1'b1; mem_idx = '0;
mem_wdata = saved_op_data; end
            default: ;
        endcase
    end

    logic mem_target_priv;
    assign mem_target_priv = is_private(mem_idx);

    // Cache hit detection. node_cache covers indices 0..2 only; on a hit
    // for a read, we skip the BRAM/MMU access entirely and feed the
    // destination register from the cache in the same cycle.
    wire mem_idx_cached = (mem_idx == 0) ? node_cache_valid[0] :
                          (mem_idx == 1) ? node_cache_valid[1] :
                          (mem_idx == 2) ? node_cache_valid[2] :
                                           1'b0;
    wire [NODE_WIDTH-1:0] mem_idx_cache_data =
                          (mem_idx == 0) ? node_cache[0] :
                          (mem_idx == 1) ? node_cache[1] :
                          (mem_idx == 2) ? node_cache[2] :
                                           '0;
    wire effective_read = mem_do_read && !mem_idx_cached;

    // private interface drivers
    assign priv_re    = (effective_read && mem_target_priv);
    assign priv_we    = (mem_do_write  && mem_target_priv);
    assign priv_addr  = mem_idx[5:0];
    assign priv_wdata = mem_wdata;

    // virtual interface drivers
    assign virt_req_valid = (effective_read || mem_do_write) && !mem_target_priv;
    assign virt_req_wr    = mem_do_write && !mem_target_priv;
    assign virt_req_va    = make_va(mem_idx);
```

```
assign virt_req_wdata = mem_wdata;

// WAIT-state completion signals
//
// Private read:  one-cycle latency, tracked by a one-cycle delay flop.
// Private write: one-cycle latency, tracked the same way.
// Virtual read:  completes on virt_resp_valid.
// Virtual write: completes on virt_resp_valid (the MMU pulses resp_valid
//                for both reads and writes; for writes resp_data is 0).
// virt_retry:    one-cycle pulse asserted when the MMU rejects the
//                request; the FSM falls back to the matching ISSUE
//                state to retry on the next cycle.

logic priv_read_done;
logic priv_write_done;
logic virt_read_done;
logic virt_write_done;
logic virt_retry;

logic priv_re_d1, priv_we_d1;
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        priv_re_d1 <= 1'b0;
        priv_we_d1 <= 1'b0;
    end else begin
        priv_re_d1 <= priv_re;
        priv_we_d1 <= priv_we;
    end
end
assign priv_read_done  = priv_re_d1;
assign priv_write_done = priv_we_d1;
assign virt_read_done  = virt_resp_valid;
assign virt_write_done = virt_resp_valid;
assign virt_retry      = virt_resp_reject;

// Next-state logic

always_comb begin
    n_state = state;
    unique case (state)
        // IDLE
        S_IDLE: begin
            if (cmd_valid) begin
                unique case (cmd_op)
                    OP_PUSH:   n_state = S_PUSH_LAUNCH;
```

```
                        OP_POP:    n_state = S_POP_LAUNCH;
                        OP_PEEK:   begin
                            if (root_cache_valid) n_state = S_DONE;
                            else                  n_state = S_PEEK_RD_ISSUE;
                        end
                        OP_UPDATE: begin
                            // Reject if root cache is cold or the new node's
                            // price/timestamp don't match the current root.
                            if (root_cache_valid && update_keys_ok) n_state =
S_UPDATE_WR_ISSUE;
                            else                                    n_state =
S_DONE;
                        end
                        default:   n_state = S_DONE;
                    endcase
                end
            end

            // PUSH
            S_PUSH_LAUNCH: begin
                if (size == 0) n_state = S_PUSH_WR_FINAL_ISSUE;
                else           n_state = S_PUSH_RD_PARENT_ISSUE;
            end
            S_PUSH_RD_PARENT_ISSUE: begin
                if (mem_idx_cached)          n_state = S_PUSH_DECIDE;
                else if (mem_target_priv)    n_state = S_PUSH_RD_PARENT_WAIT;
                else if (virt_req_ready)     n_state = S_PUSH_RD_PARENT_WAIT;
            end
            S_PUSH_RD_PARENT_WAIT: begin
                if (priv_read_done || virt_read_done) n_state = S_PUSH_DECIDE;
                else if (virt_retry)                  n_state =
S_PUSH_RD_PARENT_ISSUE;
            end
            S_PUSH_DECIDE: begin
                if (cur_wins_parent) n_state = S_PUSH_WR_DOWN_ISSUE;
                else                 n_state = S_PUSH_WR_FINAL_ISSUE;
            end
            S_PUSH_WR_DOWN_ISSUE: begin
                if (mem_target_priv)         n_state = S_PUSH_WR_DOWN_WAIT;
                else if (virt_req_ready)     n_state = S_PUSH_WR_DOWN_WAIT;
            end
            S_PUSH_WR_DOWN_WAIT: begin
                if (priv_write_done || virt_write_done) begin
                    if (parent_idx == 0) n_state = S_PUSH_WR_FINAL_ISSUE;
                    else                 n_state = S_PUSH_RD_PARENT_ISSUE;
```

```
    end else if (virt_retry) begin
        n_state = S_PUSH_WR_DOWN_ISSUE;
    end
end
S_PUSH_WR_FINAL_ISSUE: begin
    if (mem_target_priv)         n_state = S_PUSH_WR_FINAL_WAIT;
    else if (virt_req_ready)     n_state = S_PUSH_WR_FINAL_WAIT;
end
S_PUSH_WR_FINAL_WAIT: begin
    if (priv_write_done || virt_write_done) n_state = S_DONE;
    else if (virt_retry) n_state = S_PUSH_WR_FINAL_ISSUE;
end

// POP
S_POP_LAUNCH: begin
    if (size == 0)              n_state = S_DONE;
    else if (root_cache_valid) begin
        if (size == 1) n_state = S_DONE;
        else           n_state = S_POP_RD_LAST_ISSUE;
    end
    else                        n_state = S_POP_RD_ROOT_ISSUE;
end
S_POP_RD_ROOT_ISSUE: begin
    if (mem_target_priv)         n_state = S_POP_RD_ROOT_WAIT;
    else if (virt_req_ready)     n_state = S_POP_RD_ROOT_WAIT;
end
S_POP_RD_ROOT_WAIT: begin
    if (priv_read_done || virt_read_done) begin
        if (size == 1) n_state = S_DONE;
        else           n_state = S_POP_RD_LAST_ISSUE;
    end else if (virt_retry) n_state = S_POP_RD_ROOT_ISSUE;
end
S_POP_RD_LAST_ISSUE: begin
    if (mem_idx_cached)          n_state = S_POP_SIFT_RD_LEFT_ISSUE;
    else if (mem_target_priv)    n_state = S_POP_RD_LAST_WAIT;
    else if (virt_req_ready)     n_state = S_POP_RD_LAST_WAIT;
end
S_POP_RD_LAST_WAIT: begin
    if (priv_read_done || virt_read_done)
        n_state = S_POP_SIFT_RD_LEFT_ISSUE;
    else if (virt_retry) n_state = S_POP_RD_LAST_ISSUE;
end
S_POP_SIFT_RD_LEFT_ISSUE: begin
    if (left_idx >= size)        n_state = S_POP_SIFT_WR_FINAL_ISSUE;
    else if (mem_idx_cached)     n_state = S_POP_SIFT_RD_RIGHT_ISSUE;
```

```
                else if (mem_target_priv)    n_state = S_POP_SIFT_RD_LEFT_WAIT;
                else if (virt_req_ready)     n_state = S_POP_SIFT_RD_LEFT_WAIT;
            end
            S_POP_SIFT_RD_LEFT_WAIT: begin
                if (priv_read_done || virt_read_done) n_state =
S_POP_SIFT_RD_RIGHT_ISSUE;
                else if (virt_retry) n_state = S_POP_SIFT_RD_LEFT_ISSUE;
            end
            S_POP_SIFT_RD_RIGHT_ISSUE: begin
                if (right_idx >= size)       n_state = S_POP_SIFT_DECIDE;
                else if (mem_idx_cached)     n_state = S_POP_SIFT_DECIDE;
                else if (mem_target_priv)    n_state = S_POP_SIFT_RD_RIGHT_WAIT;
                else if (virt_req_ready)     n_state = S_POP_SIFT_RD_RIGHT_WAIT;
            end
            S_POP_SIFT_RD_RIGHT_WAIT: begin
                if (priv_read_done || virt_read_done) n_state =
S_POP_SIFT_DECIDE;
                else if (virt_retry) n_state = S_POP_SIFT_RD_RIGHT_ISSUE;
            end
            S_POP_SIFT_DECIDE: begin
                if (best == BEST_CUR) n_state = S_POP_SIFT_WR_FINAL_ISSUE;
                else                  n_state = S_POP_SIFT_WR_UP_ISSUE;
            end
            S_POP_SIFT_WR_UP_ISSUE: begin
                if (mem_target_priv)         n_state = S_POP_SIFT_WR_UP_WAIT;
                else if (virt_req_ready)     n_state = S_POP_SIFT_WR_UP_WAIT;
            end
            S_POP_SIFT_WR_UP_WAIT: begin
                if (priv_write_done || virt_write_done) n_state =
S_POP_SIFT_RD_LEFT_ISSUE;
                else if (virt_retry) n_state = S_POP_SIFT_WR_UP_ISSUE;
            end
            S_POP_SIFT_WR_FINAL_ISSUE: begin
                if (mem_target_priv)         n_state = S_POP_SIFT_WR_FINAL_WAIT;
                else if (virt_req_ready)     n_state = S_POP_SIFT_WR_FINAL_WAIT;
            end
            S_POP_SIFT_WR_FINAL_WAIT: begin
                if (priv_write_done || virt_write_done) n_state = S_DONE;
                else if (virt_retry) n_state = S_POP_SIFT_WR_FINAL_ISSUE;
            end

            // PEEK
            S_PEEK_RD_ISSUE: begin
                if (mem_target_priv)         n_state = S_PEEK_RD_WAIT;
                else if (virt_req_ready)     n_state = S_PEEK_RD_WAIT;
```

```
        end
        S_PEEK_RD_WAIT: begin
            if (priv_read_done || virt_read_done) n_state = S_DONE;
            else if (virt_retry) n_state = S_PEEK_RD_ISSUE;
        end

        // UPDATE
        S_UPDATE_WR_ISSUE: begin
            if (mem_target_priv)         n_state = S_UPDATE_WR_WAIT;
            else if (virt_req_ready)     n_state = S_UPDATE_WR_WAIT;
        end
        S_UPDATE_WR_WAIT: begin
            if (priv_write_done || virt_write_done) n_state = S_DONE;
            else if (virt_retry) n_state = S_UPDATE_WR_ISSUE;
        end

        // DONE
        S_DONE: n_state = S_IDLE;

        default: n_state = S_IDLE;
    endcase
end

// Sequential state and register updates

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state            <= S_IDLE;
        size             <= '0;
        target_idx       <= '0;
        parent_idx       <= '0;
        left_idx         <= '0;
        right_idx        <= '0;
        has_right        <= 1'b0;
        cur_node         <= '0;
        parent_node      <= '0;
        left_node        <= '0;
        right_node       <= '0;
        saved_op_data    <= '0;
        saved_op         <= 2'd0;
        popped_root      <= '0;
        for (int i = 0; i < 3; i++) node_cache[i] <= '0;
        node_cache_valid <= 3'b000;
    end else begin
        state <= n_state;
```

```
            unique case (state)
                S_IDLE: if (cmd_valid) begin
                    saved_op      <= cmd_op;
                    saved_op_data <= cmd_data_in;
                    if (cmd_op == OP_PUSH) begin
                        cur_node   <= cmd_data_in;
                        target_idx <= size;
                        parent_idx <= (size == 0) ? '0 : (size - 1) >> 1;
                    end
                    if (cmd_op == OP_POP) begin
                        target_idx <= '0;
                        if (node_cache_valid[0]) popped_root <= node_cache[0];
                    end
                end

                // PUSH bookkeeping
                S_PUSH_RD_PARENT_ISSUE: begin
                    if (mem_idx_cached) parent_node <= mem_idx_cache_data;
                end
                S_PUSH_RD_PARENT_WAIT: begin
                    if (priv_read_done || virt_read_done) begin
                        if (priv_read_done) parent_node <= priv_rdata;
                        else                parent_node <= virt_resp_data;
                    end
                end
                S_PUSH_WR_DOWN_WAIT: if (priv_write_done || virt_write_done)
begin
                    // We wrote parent_node down to target_idx; cache it if
top-3.
                    if (target_idx < 3) begin
                        node_cache[target_idx]       <= parent_node;
                        node_cache_valid[target_idx] <= 1'b1;
                    end
                    target_idx <= parent_idx;
                    parent_idx <= (parent_idx == 0) ? '0 : (parent_idx - 1) >> 1;
                end
                S_PUSH_WR_FINAL_WAIT: if (priv_write_done || virt_write_done)
begin
                    size <= size + 1'b1;
                    // cur_node now lives at target_idx; cache it if top-3.
                    if (target_idx < 3) begin
                        node_cache[target_idx]       <= cur_node;
                        node_cache_valid[target_idx] <= 1'b1;
                    end
```

```
end

// POP bookkeeping
S_POP_LAUNCH: begin
    // size=1 with cached root jumps straight to S_DONE; the
    // heap goes empty, drop all cache slots.
    if (size == 1 && node_cache_valid[0]) begin
        size             <= '0;
        node_cache_valid <= 3'b000;
    end
end
S_POP_RD_ROOT_WAIT: begin
    if (priv_read_done || virt_read_done) begin
        if (priv_read_done) popped_root <= priv_rdata;
        else                popped_root <= virt_resp_data;
        if (size == 1) begin
            size             <= '0;
            node_cache_valid <= 3'b000;
        end
    end
end
S_POP_RD_LAST_ISSUE: if (mem_idx_cached) begin
    cur_node            <= mem_idx_cache_data;
    target_idx          <= '0;
    left_idx            <= 1;
    right_idx           <= 2;
    size                <= size - 1'b1;
    node_cache_valid[0] <= 1'b0;
    if (size <= 2) node_cache_valid[1] <= 1'b0;
    if (size <= 3) node_cache_valid[2] <= 1'b0;
end
S_POP_RD_LAST_WAIT: begin
    if (priv_read_done || virt_read_done) begin
        if (priv_read_done) cur_node <= priv_rdata;
        else                cur_node <= virt_resp_data;
        target_idx       <= '0;
        left_idx         <= 1;
        right_idx        <= 2;
        size             <= size - 1'b1;
        // Root is being replaced via the upcoming sift-down.
        // Slots 1 and 2 stay valid only if they remain in the
        // post-decrement heap.
        node_cache_valid[0] <= 1'b0;
        if (size <= 2) node_cache_valid[1] <= 1'b0;
        if (size <= 3) node_cache_valid[2] <= 1'b0;
```

```
                    end
                end
                S_POP_SIFT_RD_LEFT_ISSUE: begin
                    if (left_idx < size && mem_idx_cached)
                        left_node <= mem_idx_cache_data;
                end
                S_POP_SIFT_RD_LEFT_WAIT: begin
                    if (priv_read_done || virt_read_done) begin
                        if (priv_read_done) left_node <= priv_rdata;
                        else                left_node <= virt_resp_data;
                    end
                end
                S_POP_SIFT_RD_RIGHT_ISSUE: begin
                    if (right_idx >= size) has_right <= 1'b0;
                    else if (mem_idx_cached) begin
                        right_node <= mem_idx_cache_data;
                        has_right  <= 1'b1;
                    end
                end
                S_POP_SIFT_RD_RIGHT_WAIT: begin
                    if (priv_read_done || virt_read_done) begin
                        if (priv_read_done) right_node <= priv_rdata;
                        else                right_node <= virt_resp_data;
                        has_right <= 1'b1;
                    end
                end
                S_POP_SIFT_WR_UP_WAIT: if (priv_write_done || virt_write_done)
begin
                    // We wrote (best==LEFT ? left_node : right_node) to
target_idx.
                    if (target_idx < 3) begin
                        node_cache[target_idx]       <= (best == BEST_LEFT) ?
left_node : right_node;
                        node_cache_valid[target_idx] <= 1'b1;
                    end
                    if (best == BEST_LEFT) begin
                        target_idx <= left_idx;
                        left_idx   <= (left_idx  << 1) + 1'b1;
                        right_idx  <= (left_idx  << 1) + 2'd2;
                    end else begin
                        target_idx <= right_idx;
                        left_idx   <= (right_idx << 1) + 1'b1;
                        right_idx  <= (right_idx << 1) + 2'd2;
                    end
                end
```

```
                S_POP_SIFT_WR_FINAL_WAIT: if (priv_write_done || virt_write_done)
begin
                    if (target_idx < 3) begin
                        node_cache[target_idx]       <= cur_node;
                        node_cache_valid[target_idx] <= 1'b1;
                    end
                end

                // PEEK bookkeeping
                S_PEEK_RD_WAIT: begin
                    if (priv_read_done || virt_read_done) begin
                        if (priv_read_done) node_cache[0] <= priv_rdata;
                        else                node_cache[0] <= virt_resp_data;
                        node_cache_valid[0] <= 1'b1;
                    end
                end

                // UPDATE bookkeeping
                S_UPDATE_WR_WAIT: if (priv_write_done || virt_write_done) begin
                    node_cache[0]       <= saved_op_data;
                    node_cache_valid[0] <= 1'b1;
                end

                default: ;
            endcase
        end
    end

    // Outputs

    assign cmd_ready    = (state == S_IDLE);
    assign cmd_done     = (state == S_DONE);
    assign size_out     = size;
    assign cmd_root_out = (saved_op == OP_PEEK)
                          ? (root_cache_valid ? root_cache : '0)
                          : popped_root;

endmodule
```

```
None
// symbol_engine.sv
//
// Wraps everything required for one of the eight symbol slots in the HFT
```

```
// pipeline.
//
// Contents:
//   - heap_fsm: two instances, bid (max-heap) and ask (min-heap)
//   - priv_bram: two instances, one 64-node BRAM behind each heap_fsm
//   - trade_ctrl: orchestrates push, peek, decide, pop/update, emit-trade
//   - mmu_mux: 2:1 single-in-flight mux of the two heaps onto the engine's
//     single MMU client port
//

module symbol_engine #(
    parameter int               ENGINE_ID  = 0,
    parameter int               NODE_WIDTH = 86,
    parameter logic [20:0]      SYMBOL     =
        (ENGINE_ID == 0) ? {7'd65, 7'd80, 7'd76} :   // APL  (AAPL)
        (ENGINE_ID == 1) ? {7'd66, 7'd83, 7'd88} :   // BSX
        (ENGINE_ID == 2) ? {7'd66, 7'd85, 7'd83} :   // BUS
        (ENGINE_ID == 3) ? {7'd77, 7'd77, 7'd77} :   // MMM
        (ENGINE_ID == 4) ? {7'd83, 7'd70, 7'd84} :   // SFT  (MSFT)
        (ENGINE_ID == 5) ? {7'd66, 7'd85, 7'd88} :   // BUX  (SBUX)
        (ENGINE_ID == 6) ? {7'd84, 7'd85, 7'd83} :   // TUS
        (ENGINE_ID == 7) ? {7'd87, 7'd77, 7'd84} :   // WMT
                           21'd0
) (
    input  logic                 clk,
    input  logic                 rst_n,

    // free-running timestamp counter
    input  logic [31:0]          now_ts,

    // Payload is a 32-bit DISPATCH_ORDER:
    // bit[31] = type, [30:15] = price, [14:0] = quantity.
    input  logic                 order_in_valid,
    input  logic [31:0]          order_in_data,
    output logic                 order_in_ready,

    // trade egress
    output logic                 trade_out_valid,
    output logic [NODE_WIDTH-1:0] trade_out_data,
    input  logic                 trade_out_ready,

    // MMU client port (one of mmu.sv's eight)
    output logic                 mmu_req_valid,
    output logic [31:0]          mmu_req_va,
    output logic                 mmu_req_wr,
```

```
    output logic [NODE_WIDTH-1:0] mmu_req_wdata,
    input  logic                  mmu_req_ready,
    input  logic [NODE_WIDTH-1:0] mmu_resp_data,
    input  logic                  mmu_resp_valid,
    input  logic                  mmu_resp_reject,

    // status (to hazard unit / harness)
    output logic [13:0]           bid_size_o,
    output logic [13:0]           ask_size_o,
    // High when the trade controller is back to T_IDLE with no pending op.
    // ANDed across all 8 engines at the top level to detect drain.
    output logic                  engine_idle
);

    function automatic logic        order_type   (input logic [NODE_WIDTH-1:0]
n);
        order_type = n[85];
    endfunction
    function automatic logic [15:0] order_amount (input logic [NODE_WIDTH-1:0]
n);
        order_amount = n[68:53];
    endfunction
    function automatic logic [15:0] order_price  (input logic [NODE_WIDTH-1:0]
n);
        order_price = n[84:69];
    endfunction
    function automatic logic [NODE_WIDTH-1:0] with_amount (
        input logic [NODE_WIDTH-1:0] n,
        input logic [15:0]           new_amt
    );
        with_amount = {n[85:69], new_amt, n[52:0]};
    endfunction

    // Per-heap virtual-memory buses

    logic                  bid_virt_req_valid, ask_virt_req_valid;
    logic [31:0]           bid_virt_req_va,    ask_virt_req_va;
    logic                  bid_virt_req_wr,    ask_virt_req_wr;
    logic [NODE_WIDTH-1:0] bid_virt_req_wdata, ask_virt_req_wdata;
    logic                  bid_virt_req_ready, ask_virt_req_ready;
    logic                  bid_virt_resp_valid,  ask_virt_resp_valid;
    logic [NODE_WIDTH-1:0] bid_virt_resp_data,   ask_virt_resp_data;
    logic                  bid_virt_resp_reject, ask_virt_resp_reject;

    // Per-heap private-memory buses
```

```
logic                  bid_priv_we, bid_priv_re;
logic [5:0]            bid_priv_addr;
logic [NODE_WIDTH-1:0] bid_priv_wdata, bid_priv_rdata;
logic                  ask_priv_we, ask_priv_re;
logic [5:0]            ask_priv_addr;
logic [NODE_WIDTH-1:0] ask_priv_wdata, ask_priv_rdata;

// Per-heap command buses (driven by trade_ctrl)

logic                  bid_cmd_valid, ask_cmd_valid;
logic [1:0]            bid_cmd_op,    ask_cmd_op;
logic [NODE_WIDTH-1:0] bid_cmd_data,  ask_cmd_data;
logic                  bid_cmd_ready, ask_cmd_ready;
logic                  bid_cmd_done,  ask_cmd_done;
logic [NODE_WIDTH-1:0] bid_root,      ask_root;
logic [13:0]           bid_size,      ask_size;

// Heap FSMs

heap_fsm #(
    .HEAP_KIND (0),       // MAX
    .ENGINE_ID (ENGINE_ID)
) u_bid (
    .clk             (clk),
    .rst_n           (rst_n),
    .cmd_valid       (bid_cmd_valid),
    .cmd_op          (bid_cmd_op),
    .cmd_data_in     (bid_cmd_data),
    .cmd_ready       (bid_cmd_ready),
    .cmd_done        (bid_cmd_done),
    .cmd_root_out    (bid_root),
    .size_out        (bid_size),
    .priv_we         (bid_priv_we),
    .priv_re         (bid_priv_re),
    .priv_addr       (bid_priv_addr),
    .priv_wdata      (bid_priv_wdata),
    .priv_rdata      (bid_priv_rdata),
    .virt_req_valid  (bid_virt_req_valid),
    .virt_req_va     (bid_virt_req_va),
    .virt_req_wr     (bid_virt_req_wr),
    .virt_req_wdata  (bid_virt_req_wdata),
    .virt_req_ready  (bid_virt_req_ready),
    .virt_resp_valid (bid_virt_resp_valid),
    .virt_resp_data  (bid_virt_resp_data),
```

```
    .virt_resp_reject(bid_virt_resp_reject)
);

heap_fsm #(
    .HEAP_KIND (1),       // MIN
    .ENGINE_ID (ENGINE_ID)
) u_ask (
    .clk             (clk),
    .rst_n           (rst_n),
    .cmd_valid       (ask_cmd_valid),
    .cmd_op          (ask_cmd_op),
    .cmd_data_in     (ask_cmd_data),
    .cmd_ready       (ask_cmd_ready),
    .cmd_done        (ask_cmd_done),
    .cmd_root_out    (ask_root),
    .size_out        (ask_size),
    .priv_we         (ask_priv_we),
    .priv_re         (ask_priv_re),
    .priv_addr       (ask_priv_addr),
    .priv_wdata      (ask_priv_wdata),
    .priv_rdata      (ask_priv_rdata),
    .virt_req_valid  (ask_virt_req_valid),
    .virt_req_va     (ask_virt_req_va),
    .virt_req_wr     (ask_virt_req_wr),
    .virt_req_wdata  (ask_virt_req_wdata),
    .virt_req_ready  (ask_virt_req_ready),
    .virt_resp_valid (ask_virt_resp_valid),
    .virt_resp_data  (ask_virt_resp_data),
    .virt_resp_reject(ask_virt_resp_reject)
);

// Shared private BRAM. Address top bit selects the heap:
// 0 for bid (lower 64 entries), 1 for ask (upper 64 entries).

logic                  priv_we, priv_re;
logic [6:0]            priv_addr;
logic [NODE_WIDTH-1:0] priv_wdata;
logic [NODE_WIDTH-1:0] priv_rdata;

logic ask_active;
assign ask_active = ask_priv_we | ask_priv_re;

assign priv_we    = bid_priv_we | ask_priv_we;
assign priv_re    = bid_priv_re | ask_priv_re;
assign priv_addr  = ask_active ? {1'b1, ask_priv_addr}
```

```
                                : {1'b0, bid_priv_addr};
assign priv_wdata = ask_active ? ask_priv_wdata : bid_priv_wdata;

assign bid_priv_rdata = priv_rdata;
assign ask_priv_rdata = priv_rdata;

priv_bram #(.WIDTH(NODE_WIDTH), .DEPTH(128)) u_priv (
    .clk   (clk),
    .we    (priv_we),
    .re    (priv_re),
    .addr  (priv_addr),
    .wdata (priv_wdata),
    .rdata (priv_rdata)
);

// MMU port mux (single-in-flight, fixed bid > ask priority)

typedef enum logic [1:0] { MMU_IDLE, MMU_BID, MMU_ASK } mmu_owner_t;
mmu_owner_t mmu_owner;

logic bid_grant, ask_grant;
assign bid_grant = (mmu_owner == MMU_IDLE) && bid_virt_req_valid;
assign ask_grant = (mmu_owner == MMU_IDLE) && !bid_virt_req_valid
                                           && ask_virt_req_valid;

always_comb begin
    mmu_req_valid = 1'b0;
    mmu_req_va    = '0;
    mmu_req_wr    = 1'b0;
    mmu_req_wdata = '0;
    unique case (mmu_owner)
        MMU_IDLE: begin
            if (bid_grant) begin
                mmu_req_valid = bid_virt_req_valid;
                mmu_req_va    = bid_virt_req_va;
                mmu_req_wr    = bid_virt_req_wr;
                mmu_req_wdata = bid_virt_req_wdata;
            end else if (ask_grant) begin
                mmu_req_valid = ask_virt_req_valid;
                mmu_req_va    = ask_virt_req_va;
                mmu_req_wr    = ask_virt_req_wr;
                mmu_req_wdata = ask_virt_req_wdata;
            end
        end
        default: ;
```

```
    endcase
end

assign bid_virt_req_ready = bid_grant && mmu_req_ready;
assign ask_virt_req_ready = ask_grant && mmu_req_ready;

assign bid_virt_resp_valid  = (mmu_owner == MMU_BID) && mmu_resp_valid;
assign bid_virt_resp_reject = (mmu_owner == MMU_BID) && mmu_resp_reject;
assign bid_virt_resp_data   = mmu_resp_data;
assign ask_virt_resp_valid  = (mmu_owner == MMU_ASK) && mmu_resp_valid;
assign ask_virt_resp_reject = (mmu_owner == MMU_ASK) && mmu_resp_reject;
assign ask_virt_resp_data   = mmu_resp_data;

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) mmu_owner <= MMU_IDLE;
    else begin
        unique case (mmu_owner)
            MMU_IDLE: begin
                if (bid_grant && mmu_req_ready)      mmu_owner <= MMU_BID;
                else if (ask_grant && mmu_req_ready) mmu_owner <= MMU_ASK;
            end
            MMU_BID, MMU_ASK: begin
                if (mmu_resp_valid || mmu_resp_reject) mmu_owner <= MMU_IDLE;
            end
            default: mmu_owner <= MMU_IDLE;
        endcase
    end
end

// Trade controller
//
// For each ingested order the FSM runs:
//   1. latch the order
//   2. push it into the bid or ask heap
//   3. peek both roots
//   4. decide:
//        no trade               : back to idle
//        exact match            : pop bid, pop ask, emit
//        partial (bid > ask)    : pop ask, update bid, emit
//        partial (ask > bid)    : pop bid, update ask, emit
// After a trade emits, the FSM loops back to the peek phase if both
// heaps remain non-empty (cascade trades).

typedef enum logic [4:0] {
    T_IDLE,
```

```
        T_PUSH_ISSUE,        T_PUSH_WAIT,
        T_PEEK_BID_ISSUE,    T_PEEK_BID_WAIT,
        T_PEEK_ASK_ISSUE,    T_PEEK_ASK_WAIT,
        T_DECIDE,
        T_POP_BID_ISSUE,     T_POP_BID_WAIT,
        T_POP_ASK_ISSUE,     T_POP_ASK_WAIT,
        T_UPDATE_BID_ISSUE,  T_UPDATE_BID_WAIT,
        T_UPDATE_ASK_ISSUE,  T_UPDATE_ASK_WAIT,
        T_EMIT_TRADE
    } tstate_t;

    tstate_t                tstate, tn_state;
    logic [NODE_WIDTH-1:0]  pending_order;
    logic                   pending_is_ask;
    logic [NODE_WIDTH-1:0]  bid_root_snap, ask_root_snap;
    logic [NODE_WIDTH-1:0]  trade_event_data;
    logic                   trade_event_pending;

    // Latched trade decision.
    logic                   latched_partial_bid, latched_partial_ask;

    localparam logic [1:0] OP_PUSH = 2'd0, OP_POP = 2'd1,
                           OP_PEEK = 2'd2, OP_UPDATE = 2'd3;

    // command-bus drivers
    always_comb begin
        bid_cmd_valid = 1'b0; bid_cmd_op = OP_PEEK; bid_cmd_data = '0;
        ask_cmd_valid = 1'b0; ask_cmd_op = OP_PEEK; ask_cmd_data = '0;
        unique case (tstate)
            T_PUSH_ISSUE: begin
                if (pending_is_ask) begin
                    ask_cmd_valid = 1'b1; ask_cmd_op = OP_PUSH; ask_cmd_data =
pending_order;
                end else begin
                    bid_cmd_valid = 1'b1; bid_cmd_op = OP_PUSH; bid_cmd_data =
pending_order;
                end
            end
            T_PUSH_WAIT: begin
                if (pending_is_ask) ask_cmd_valid = 1'b1;
                else                bid_cmd_valid = 1'b1;
            end
            T_PEEK_BID_ISSUE, T_PEEK_BID_WAIT: begin
                bid_cmd_valid = 1'b1; bid_cmd_op = OP_PEEK;
            end
```

```
        T_PEEK_ASK_ISSUE, T_PEEK_ASK_WAIT: begin
            ask_cmd_valid = 1'b1; ask_cmd_op = OP_PEEK;
        end
        T_POP_BID_ISSUE, T_POP_BID_WAIT: begin
            bid_cmd_valid = 1'b1; bid_cmd_op = OP_POP;
        end
        T_POP_ASK_ISSUE, T_POP_ASK_WAIT: begin
            ask_cmd_valid = 1'b1; ask_cmd_op = OP_POP;
        end
        T_UPDATE_BID_ISSUE, T_UPDATE_BID_WAIT: begin
            bid_cmd_valid = 1'b1; bid_cmd_op = OP_UPDATE;
            bid_cmd_data  = with_amount(bid_root_snap,
                              order_amount(bid_root_snap)
                            - order_amount(ask_root_snap));
        end
        T_UPDATE_ASK_ISSUE, T_UPDATE_ASK_WAIT: begin
            ask_cmd_valid = 1'b1; ask_cmd_op = OP_UPDATE;
            ask_cmd_data  = with_amount(ask_root_snap,
                              order_amount(ask_root_snap)
                            - order_amount(bid_root_snap));
        end
        default: ;
    endcase
end

// trade decision (combinational on snapped roots)
logic trade_match, partial_bid_remains, partial_ask_remains;
assign trade_match         = (bid_size > 0) && (ask_size > 0)
                          && (order_price(bid_root_snap)
                              >= order_price(ask_root_snap));
assign partial_bid_remains = trade_match
                          && (order_amount(bid_root_snap)
                              > order_amount(ask_root_snap));
assign partial_ask_remains = trade_match
                          && (order_amount(ask_root_snap)
                              > order_amount(bid_root_snap));

// next-state logic
always_comb begin
    tn_state = tstate;
    unique case (tstate)
        T_IDLE: if (order_in_valid) tn_state = T_PUSH_ISSUE;

        T_PUSH_ISSUE: tn_state = T_PUSH_WAIT;
        T_PUSH_WAIT:  if ((pending_is_ask && ask_cmd_done)
```

```
                    || (!pending_is_ask && bid_cmd_done)) begin
            if (bid_size > 0 && ask_size > 0) tn_state = T_PEEK_BID_ISSUE;
            else                              tn_state = T_IDLE;
        end

        T_PEEK_BID_ISSUE: tn_state = T_PEEK_BID_WAIT;
        T_PEEK_BID_WAIT:  if (bid_cmd_done) tn_state = T_PEEK_ASK_ISSUE;
        T_PEEK_ASK_ISSUE: tn_state = T_PEEK_ASK_WAIT;
        T_PEEK_ASK_WAIT:  if (ask_cmd_done) tn_state = T_DECIDE;

        T_DECIDE: begin
            if (!trade_match)             tn_state = T_IDLE;
            else if (partial_bid_remains) tn_state = T_POP_ASK_ISSUE;
            else if (partial_ask_remains) tn_state = T_POP_BID_ISSUE;
            else                          tn_state = T_POP_BID_ISSUE;
        end

        T_POP_BID_ISSUE: tn_state = T_POP_BID_WAIT;
        T_POP_BID_WAIT:  if (bid_cmd_done) begin
            if (latched_partial_ask) tn_state = T_UPDATE_ASK_ISSUE;
            else                     tn_state = T_POP_ASK_ISSUE;
        end
        T_POP_ASK_ISSUE: tn_state = T_POP_ASK_WAIT;
        T_POP_ASK_WAIT:  if (ask_cmd_done) begin
            if (latched_partial_bid) tn_state = T_UPDATE_BID_ISSUE;
            else                     tn_state = T_EMIT_TRADE;
        end
        T_UPDATE_BID_ISSUE: tn_state = T_UPDATE_BID_WAIT;
        T_UPDATE_BID_WAIT:  if (bid_cmd_done) tn_state = T_EMIT_TRADE;
        T_UPDATE_ASK_ISSUE: tn_state = T_UPDATE_ASK_WAIT;
        T_UPDATE_ASK_WAIT:  if (ask_cmd_done) tn_state = T_EMIT_TRADE;

        T_EMIT_TRADE: if (trade_out_ready) begin
            if (bid_size > 0 && ask_size > 0) tn_state = T_PEEK_BID_ISSUE;
            else                              tn_state = T_IDLE;
        end
        default: tn_state = T_IDLE;
    endcase
end

// sequential bookkeeping
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        tstate              <= T_IDLE;
        pending_order       <= '0;
```

```
            pending_is_ask      <= 1'b0;
            bid_root_snap       <= '0;
            ask_root_snap       <= '0;
            trade_event_data    <= '0;
            trade_event_pending <= 1'b0;
            latched_partial_bid <= 1'b0;
            latched_partial_ask <= 1'b0;
        end else begin
            tstate <= tn_state;

            if (tstate == T_IDLE && order_in_valid) begin
                // Build the 86-bit in-heap node from the 32-bit dispatched
                // order plus a sampled timestamp and the engine's symbol
                pending_order <= {order_in_data[31],         // [85]    type
                                  order_in_data[30:15],      // [84:69] price
                                  1'b0, order_in_data[14:0], // [68:53] amount
(zero-ext)
                                  SYMBOL,                    // [52:32] symbol
                                  now_ts};                   // [31:0]  timestamp
                pending_is_ask <= !order_in_data[31];
            end
            if (tstate == T_PEEK_BID_WAIT && bid_cmd_done) bid_root_snap <=
bid_root;
            if (tstate == T_PEEK_ASK_WAIT && ask_cmd_done) ask_root_snap <=
ask_root;

            if (tstate == T_DECIDE && trade_match) begin
                trade_event_data <= with_amount(bid_root_snap,
                                        (order_amount(bid_root_snap)
                                         < order_amount(ask_root_snap))
                                        ? order_amount(bid_root_snap)
                                        : order_amount(ask_root_snap));
                trade_event_pending <= 1'b1;
                latched_partial_bid <= partial_bid_remains;
                latched_partial_ask <= partial_ask_remains;
            end
            if (tstate == T_EMIT_TRADE && trade_out_ready) begin
                trade_event_pending <= 1'b0;
            end
        end
    end

    // Outputs

    assign order_in_ready  = (tstate == T_IDLE);
```

```
    assign trade_out_valid = (tstate == T_EMIT_TRADE) && trade_event_pending;
    assign trade_out_data  = trade_event_data;
    assign bid_size_o      = bid_size;
    assign ask_size_o      = ask_size;
    assign engine_idle     = (tstate == T_IDLE);

endmodule


// priv_bram

module priv_bram #(
    parameter int WIDTH = 86,
    parameter int DEPTH = 64
) (
    input  logic                    clk,
    input  logic                    we,
    input  logic                    re,
    input  logic [$clog2(DEPTH)-1:0] addr,
    input  logic [WIDTH-1:0]        wdata,
    output logic [WIDTH-1:0]        rdata
);
    (* ramstyle = "M10K", ram_init_file = "priv_bram_zero.mif" *)
    logic [WIDTH-1:0] mem [0:DEPTH-1];

    always_ff @(posedge clk) begin
        if (we) mem[addr] <= wdata;
        if (re) rdata     <= mem[addr];
    end
endmodule
```

None

```
module HPTW #(
    parameter LOW_FIRST = 1'b1
) (
    input  logic         clk,
    input  logic         rst_n,

    input  logic         va_valid,
    input  logic [31:0]  va,

    output logic         pa_valid,
    output logic [31:0]  pa,
```

```
    output logic [1:0]   bank_id,
    output logic         fault,
    output logic         busy,

    output logic [13:0]  pt_raddr,
    input  logic [14:0]  pt_rdata,

    output logic         pt_we,
    output logic [13:0]  pt_waddr,
    output logic [14:0]  pt_wdata,

    input  logic         alloc_avail,
    input  logic [7:0]   alloc_page_in,
    input  logic [5:0]   alloc_node_in,

    output logic         alloc,
    output logic [7:0]   alloc_page_idx,
    output logic [5:0]   alloc_node_idx
);
    logic [13:0] pt_key;
    assign pt_key = {va[31:29], va[10:0]};

    typedef enum logic [2:0] {
        IDLE,
        LOOKUP,
        ALLOCATE,
        DONE,
        FAULT
    } state_t;

    state_t state, n_state;

    logic [13:0] latched_key;

    logic       lookup_valid_bit;
    logic [7:0] lookup_page_idx;
    logic [5:0] lookup_node_idx;
    assign lookup_valid_bit = pt_rdata[14];
    assign lookup_page_idx  = pt_rdata[13:6];
    assign lookup_node_idx  = pt_rdata[5:0];

    logic [7:0] selected_visible_page;
    logic [5:0] selected_node;
    always_comb begin
        if (state == LOOKUP) begin
```

```
            selected_visible_page = lookup_page_idx;
            selected_node         = lookup_node_idx;
        end else begin
            selected_visible_page = alloc_page_in;
            selected_node         = alloc_node_in;
        end
    end

    logic [7:0] selected_pa_page;
    assign selected_pa_page = selected_visible_page + 8'd16;

    logic [31:0] computed_pa;
    assign computed_pa = {11'd0, selected_pa_page, 3'd0, selected_node, 4'd0};

    logic [1:0] computed_bank_id;
    always_comb begin
        if      (selected_visible_page < 8'd60)  computed_bank_id = 2'd0;
        else if (selected_visible_page < 8'd120) computed_bank_id = 2'd1;
        else if (selected_visible_page < 8'd180) computed_bank_id = 2'd2;
        else                                     computed_bank_id = 2'd3;
    end

    always_comb begin
        n_state = state;
        unique case (state)
            IDLE:     if (va_valid) n_state = LOOKUP;
            LOOKUP:   if (lookup_valid_bit) n_state = DONE; else n_state =
ALLOCATE;
            ALLOCATE: if (alloc_avail) n_state = DONE; else n_state = FAULT;
            DONE:     n_state = IDLE;
            FAULT:    n_state = IDLE;
        endcase
    end

    logic [31:0] pa_reg;
    logic        pa_valid_reg;
    logic [1:0]  bank_id_reg;
    logic        fault_reg;

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            state        <= IDLE;
            latched_key  <= 14'd0;
            pa_reg       <= 32'd0;
            pa_valid_reg <= 1'b0;
```

```
            bank_id_reg  <= 2'd0;
            fault_reg    <= 1'b0;
        end else begin
            state        <= n_state;
            pa_valid_reg <= 1'b0;
            fault_reg    <= 1'b0;
            if (state == IDLE && va_valid)
                latched_key <= pt_key;

            if (n_state == DONE) begin
                pa_reg       <= computed_pa;
                bank_id_reg  <= computed_bank_id;
                pa_valid_reg <= 1'b1;
            end

            if (n_state == FAULT)
                fault_reg <= 1'b1;
        end
    end

    always_comb begin
        if (state == IDLE && va_valid) pt_raddr = pt_key;
        else                           pt_raddr = latched_key;
    end

    assign pt_we    = (state == ALLOCATE) && alloc_avail;
    assign pt_waddr = latched_key;
    assign pt_wdata = {1'b1, alloc_page_in, alloc_node_in};

    assign alloc          = (state == ALLOCATE) && alloc_avail;
    assign alloc_page_idx = alloc_page_in;
    assign alloc_node_idx = alloc_node_in;

    assign pa       = pa_reg;
    assign pa_valid = pa_valid_reg;
    assign bank_id  = bank_id_reg;
    assign fault    = fault_reg;
    assign busy     = (state != IDLE);
endmodule
```

None

```
module arbiter(
       input logic      clk,
```

```
input logic      rst_n,

 input logic           ptw0_valid,
input logic [31:0]  ptw0_va,
input logic [31:0]  ptw0_pa,
input logic [1:0]   ptw0_bank_id,
input logic      ptw0_wr,
input logic [85:0]  ptw0_wdata,

input logic      ptw1_valid,
input logic [31:0]  ptw1_va,
input logic [31:0]  ptw1_pa,
input logic [1:0]   ptw1_bank_id,
input logic      ptw1_wr,
input logic [85:0]  ptw1_wdata,

output logic     ptw0_accept,
output logic     ptw0_reject,
output logic     ptw1_accept,
output logic     ptw1_reject,

output logic [85:0] rdata_0,
output logic [31:0] rdata_va_0,
output logic     rdata_valid_0,

output logic [85:0] rdata_1,
output logic [31:0] rdata_va_1,
output logic     rdata_valid_1,

output logic [85:0] rdata_2,
output logic [31:0] rdata_va_2,
output logic     rdata_valid_2,

output logic [85:0] rdata_3,
output logic [31:0] rdata_va_3,
output logic     rdata_valid_3,

output logic [31:0] mem_addr_0,
output logic     mem_we_0,
output logic     mem_re_0,
output logic [85:0] mem_wdata_0,
input logic  [85:0] mem_rdata_0,
input logic      mem_rdata_valid_0,
input logic      mem_wdone_0,
input logic      mem_busy_0,
```

```
      output logic [31:0] mem_addr_1,
      output logic      mem_we_1,
      output logic      mem_re_1,
      output logic [85:0] mem_wdata_1,
      input logic [85:0]  mem_rdata_1,
      input logic       mem_rdata_valid_1,
      input logic       mem_wdone_1,
      input logic       mem_busy_1,

      output logic [31:0] mem_addr_2,
      output logic      mem_we_2,
      output logic      mem_re_2,
      output logic [85:0] mem_wdata_2,
      input logic  [85:0] mem_rdata_2,
      input logic       mem_rdata_valid_2,
      input logic       mem_wdone_2,
      input logic       mem_busy_2,

      output logic [31:0] mem_addr_3,
      output logic      mem_we_3,
      output logic      mem_re_3,
      output logic [85:0] mem_wdata_3,
      input logic  [85:0] mem_rdata_3,
      input logic       mem_rdata_valid_3,
      input logic       mem_wdone_3,
      input logic       mem_busy_3
);

    logic [2:0]   fifo_count   [4];
    logic [3:0]   fifo_full;
    logic [3:0]   fifo_almost_full;
    logic [3:0]   fifo_empty;
    logic [150:0] fifo_rd_data [4];
    logic [3:0]   fifo_rd_en;

    assign fifo_full[0]        = (fifo_count[0] == 3'd4);
    assign fifo_full[1]        = (fifo_count[1] == 3'd4);
    assign fifo_full[2]        = (fifo_count[2] == 3'd4);
    assign fifo_full[3]        = (fifo_count[3] == 3'd4);

    assign fifo_almost_full[0] = (fifo_count[0] == 3'd3);
    assign fifo_almost_full[1] = (fifo_count[1] == 3'd3);
    assign fifo_almost_full[2] = (fifo_count[2] == 3'd3);
    assign fifo_almost_full[3] = (fifo_count[3] == 3'd3);
```

```
    logic same_bank_collision;
    assign same_bank_collision = ptw0_valid && ptw1_valid && (ptw0_bank_id ==
ptw1_bank_id);

    logic rr_priority;
    always_ff @(posedge clk or negedge rst_n) begin
       if (!rst_n) begin
            rr_priority <= 1'b0;
       end else if (same_bank_collision && fifo_almost_full[ptw0_bank_id]) begin
            rr_priority <= ~rr_priority;
        end
    end

    logic ptw0_target_full;
    logic ptw1_target_full;
    logic shared_target_almost_full;

    assign ptw0_target_full          = fifo_full[ptw0_bank_id];
    assign ptw1_target_full          = fifo_full[ptw1_bank_id];
    assign shared_target_almost_full = fifo_almost_full[ptw0_bank_id];

    logic ptw0_can_enqueue, ptw1_can_enqueue;

    always_comb begin
        ptw0_can_enqueue = 1'b0;
        ptw1_can_enqueue = 1'b0;

        if (same_bank_collision) begin
            if (ptw0_target_full) begin
                ptw0_can_enqueue = 1'b0;
                ptw1_can_enqueue = 1'b0;
            end else if (shared_target_almost_full) begin
                ptw0_can_enqueue = (rr_priority == 1'b0);
                ptw1_can_enqueue = (rr_priority == 1'b1);
            end else begin
                ptw0_can_enqueue = 1'b1;
                ptw1_can_enqueue = 1'b1;
            end
        end else begin
            ptw0_can_enqueue = ptw0_valid && !ptw0_target_full;
            ptw1_can_enqueue = ptw1_valid && !ptw1_target_full;
        end
    end
```

```
assign ptw0_accept = ptw0_can_enqueue;
assign ptw0_reject = ptw0_valid && !ptw0_can_enqueue;
assign ptw1_accept = ptw1_can_enqueue;
assign ptw1_reject = ptw1_valid && !ptw1_can_enqueue;

logic [150:0] entry_from_ptw0;
logic [150:0] entry_from_ptw1;
assign entry_from_ptw0 = {ptw0_va, ptw0_wr, ptw0_pa, ptw0_wdata};
assign entry_from_ptw1 = {ptw1_va, ptw1_wr, ptw1_pa, ptw1_wdata};

logic ptw0_writes_b0, ptw1_writes_b0;
logic ptw0_writes_b1, ptw1_writes_b1;
logic ptw0_writes_b2, ptw1_writes_b2;
logic ptw0_writes_b3, ptw1_writes_b3;

assign ptw0_writes_b0 = ptw0_can_enqueue && (ptw0_bank_id == 2'd0);
assign ptw1_writes_b0 = ptw1_can_enqueue && (ptw1_bank_id == 2'd0);
assign ptw0_writes_b1 = ptw0_can_enqueue && (ptw0_bank_id == 2'd1);
assign ptw1_writes_b1 = ptw1_can_enqueue && (ptw1_bank_id == 2'd1);
assign ptw0_writes_b2 = ptw0_can_enqueue && (ptw0_bank_id == 2'd2);
assign ptw1_writes_b2 = ptw1_can_enqueue && (ptw1_bank_id == 2'd2);
assign ptw0_writes_b3 = ptw0_can_enqueue && (ptw0_bank_id == 2'd3);
assign ptw1_writes_b3 = ptw1_can_enqueue && (ptw1_bank_id == 2'd3);

dual_write_fifo #(.WIDTH(151), .DEPTH(4)) u_fifo_0 (
    .clk(clk), .rst_n(rst_n),
    .wr0_en(ptw0_writes_b0), .wr0_data(entry_from_ptw0),
    .wr1_en(ptw1_writes_b0), .wr1_data(entry_from_ptw1),
    .rd_en(fifo_rd_en[0]),    .rd_data(fifo_rd_data[0]),
    .empty(fifo_empty[0]),    .full(),
    .count(fifo_count[0])
);

dual_write_fifo #(.WIDTH(151), .DEPTH(4)) u_fifo_1 (
    .clk(clk), .rst_n(rst_n),
    .wr0_en(ptw0_writes_b1), .wr0_data(entry_from_ptw0),
    .wr1_en(ptw1_writes_b1), .wr1_data(entry_from_ptw1),
    .rd_en(fifo_rd_en[1]),    .rd_data(fifo_rd_data[1]),
    .empty(fifo_empty[1]),    .full(),
    .count(fifo_count[1])
);

dual_write_fifo #(.WIDTH(151), .DEPTH(4)) u_fifo_2 (
    .clk(clk), .rst_n(rst_n),
    .wr0_en(ptw0_writes_b2), .wr0_data(entry_from_ptw0),
```

```
        .wr1_en(ptw1_writes_b2), .wr1_data(entry_from_ptw1),
        .rd_en(fifo_rd_en[2]),    .rd_data(fifo_rd_data[2]),
        .empty(fifo_empty[2]),    .full(),
        .count(fifo_count[2])
    );

    dual_write_fifo #(.WIDTH(151), .DEPTH(4)) u_fifo_3 (
        .clk(clk), .rst_n(rst_n),
        .wr0_en(ptw0_writes_b3), .wr0_data(entry_from_ptw0),
        .wr1_en(ptw1_writes_b3), .wr1_data(entry_from_ptw1),
        .rd_en(fifo_rd_en[3]),    .rd_data(fifo_rd_data[3]),
        .empty(fifo_empty[3]),    .full(),
        .count(fifo_count[3])
    );

    logic [31:0] e_va_0,  e_va_1,  e_va_2,  e_va_3;
    logic        e_wr_0,  e_wr_1,  e_wr_2,  e_wr_3;
    logic [31:0] e_pa_0,  e_pa_1,  e_pa_2,  e_pa_3;
    logic [85:0] e_wd_0,  e_wd_1,  e_wd_2,  e_wd_3;

    assign {e_va_0, e_wr_0, e_pa_0, e_wd_0} = fifo_rd_data[0];
    assign {e_va_1, e_wr_1, e_pa_1, e_wd_1} = fifo_rd_data[1];
    assign {e_va_2, e_wr_2, e_pa_2, e_wd_2} = fifo_rd_data[2];
    assign {e_va_3, e_wr_3, e_pa_3, e_wd_3} = fifo_rd_data[3];

    logic [85:0] resp_buf_data [4];
    logic [31:0] resp_buf_va   [4];
    logic [3:0]  resp_buf_valid;

    logic        va_full  [4];
    logic        va_empty [4];
    logic [31:0] va_dout  [4];
    logic        va_pop   [4];

    logic  dispatch_0, dispatch_1, dispatch_2, dispatch_3;
    assign dispatch_0 = !fifo_empty[0] && !mem_busy_0 && !resp_buf_valid[0] &&
!va_full[0];
    assign dispatch_1 = !fifo_empty[1] && !mem_busy_1 && !resp_buf_valid[1] &&
!va_full[1];
    assign dispatch_2 = !fifo_empty[2] && !mem_busy_2 && !resp_buf_valid[2] &&
!va_full[2];
    assign dispatch_3 = !fifo_empty[3] && !mem_busy_3 && !resp_buf_valid[3] &&
!va_full[3];

    assign fifo_rd_en[0] = dispatch_0;
```

```
assign fifo_rd_en[1] = dispatch_1;
assign fifo_rd_en[2] = dispatch_2;
assign fifo_rd_en[3] = dispatch_3;

assign mem_addr_0  = e_pa_0;
assign mem_wdata_0 = e_wd_0;
assign mem_we_0    = dispatch_0 &&  e_wr_0;
assign mem_re_0    = dispatch_0 && !e_wr_0;

assign mem_addr_1  = e_pa_1;
assign mem_wdata_1 = e_wd_1;
assign mem_we_1    = dispatch_1 &&  e_wr_1;
assign mem_re_1    = dispatch_1 && !e_wr_1;

assign mem_addr_2  = e_pa_2;
assign mem_wdata_2 = e_wd_2;
assign mem_we_2    = dispatch_2 &&  e_wr_2;
assign mem_re_2    = dispatch_2 && !e_wr_2;

assign mem_addr_3  = e_pa_3;
assign mem_wdata_3 = e_wd_3;
assign mem_we_3    = dispatch_3 &&  e_wr_3;
assign mem_re_3    = dispatch_3 && !e_wr_3;

assign va_pop[0] = (mem_rdata_valid_0 || mem_wdone_0) && !resp_buf_valid[0];
assign va_pop[1] = (mem_rdata_valid_1 || mem_wdone_1) && !resp_buf_valid[1];
assign va_pop[2] = (mem_rdata_valid_2 || mem_wdone_2) && !resp_buf_valid[2];
assign va_pop[3] = (mem_rdata_valid_3 || mem_wdone_3) && !resp_buf_valid[3];

tracking_fifo #(.WIDTH(32), .DEPTH(8)) u_va_fifo_0 (
    .clk(clk), .rst_n(rst_n),
    .push(dispatch_0), .din(e_va_0),
    .pop(va_pop[0]),   .dout(va_dout[0]),
    .full(va_full[0]), .empty(va_empty[0])
);

tracking_fifo #(.WIDTH(32), .DEPTH(8)) u_va_fifo_1 (
    .clk(clk), .rst_n(rst_n),
    .push(dispatch_1), .din(e_va_1),
    .pop(va_pop[1]),   .dout(va_dout[1]),
    .full(va_full[1]), .empty(va_empty[1])
);

tracking_fifo #(.WIDTH(32), .DEPTH(8)) u_va_fifo_2 (
    .clk(clk), .rst_n(rst_n),
```

```
        .push(dispatch_2), .din(e_va_2),
        .pop(va_pop[2]),   .dout(va_dout[2]),
        .full(va_full[2]), .empty(va_empty[2])
    );

    tracking_fifo #(.WIDTH(32), .DEPTH(8)) u_va_fifo_3 (
        .clk(clk), .rst_n(rst_n),
        .push(dispatch_3), .din(e_va_3),
        .pop(va_pop[3]),   .dout(va_dout[3]),
        .full(va_full[3]), .empty(va_empty[3])
    );

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            resp_buf_data[0]  <= 86'd0;
            resp_buf_va[0]    <= 32'd0;
            resp_buf_valid[0] <= 1'b0;
            resp_buf_data[1]  <= 86'd0;
            resp_buf_va[1]    <= 32'd0;
            resp_buf_valid[1] <= 1'b0;
            resp_buf_data[2]  <= 86'd0;
            resp_buf_va[2]    <= 32'd0;
            resp_buf_valid[2] <= 1'b0;
            resp_buf_data[3]  <= 86'd0;
            resp_buf_va[3]    <= 32'd0;
            resp_buf_valid[3] <= 1'b0;
        end else begin
            if ((mem_rdata_valid_0 || mem_wdone_0) && !resp_buf_valid[0]) begin
                resp_buf_data[0]  <= mem_rdata_valid_0 ? mem_rdata_0 : 86'd0;
                resp_buf_va[0]    <= va_dout[0]; // Fetch the oldest VA from
tracking FIFO
                resp_buf_valid[0] <= 1'b1;
            end else if (resp_buf_valid[0]) begin
                resp_buf_valid[0] <= 1'b0;
            end

            if ((mem_rdata_valid_1 || mem_wdone_1) && !resp_buf_valid[1]) begin
                resp_buf_data[1]  <= mem_rdata_valid_1 ? mem_rdata_1 : 86'd0;
                resp_buf_va[1]    <= va_dout[1]; // Fetch the oldest VA from
tracking FIFO
                resp_buf_valid[1] <= 1'b1;
            end else if (resp_buf_valid[1]) begin
                resp_buf_valid[1] <= 1'b0;
            end
```

```
            if ((mem_rdata_valid_2 || mem_wdone_2) && !resp_buf_valid[2]) begin
                resp_buf_data[2]  <= mem_rdata_valid_2 ? mem_rdata_2 : 86'd0;
                resp_buf_va[2]    <= va_dout[2]; // Fetch the oldest VA from
tracking FIFO
                resp_buf_valid[2] <= 1'b1;
            end else if (resp_buf_valid[2]) begin
                resp_buf_valid[2] <= 1'b0;
            end

            if ((mem_rdata_valid_3 || mem_wdone_3) && !resp_buf_valid[3]) begin
                resp_buf_data[3]  <= mem_rdata_valid_3 ? mem_rdata_3 : 86'd0;
                resp_buf_va[3]    <= va_dout[3]; // Fetch the oldest VA from
tracking FIFO
                resp_buf_valid[3] <= 1'b1;
            end else if (resp_buf_valid[3]) begin
                resp_buf_valid[3] <= 1'b0;
            end
        end
    end

    assign rdata_0       = resp_buf_data[0];
    assign rdata_va_0    = resp_buf_va[0];
    assign rdata_valid_0 = resp_buf_valid[0];

    assign rdata_1       = resp_buf_data[1];
    assign rdata_va_1    = resp_buf_va[1];
    assign rdata_valid_1 = resp_buf_valid[1];

    assign rdata_2       = resp_buf_data[2];
    assign rdata_va_2    = resp_buf_va[2];
    assign rdata_valid_2 = resp_buf_valid[2];

    assign rdata_3       = resp_buf_data[3];
    assign rdata_va_3    = resp_buf_va[3];
    assign rdata_valid_3 = resp_buf_valid[3];

endmodule

module dual_write_fifo #(
    parameter int WIDTH = 8,
    parameter int DEPTH = 4
) (
    input  logic                          clk,
    input  logic                          rst_n,
```

```
    input  logic                          wr0_en,
    input  logic [WIDTH-1:0]              wr0_data,

    input  logic                          wr1_en,
    input  logic [WIDTH-1:0]              wr1_data,

    input  logic                          rd_en,
    output logic [WIDTH-1:0]              rd_data,

    output logic                          empty,
    output logic                          full,
    output logic [$clog2(DEPTH+1)-1:0]    count
);

    localparam int PTR_WIDTH = $clog2(DEPTH);
    localparam int CNT_WIDTH = $clog2(DEPTH + 1);

    logic [WIDTH-1:0]     mem [DEPTH];
    logic [PTR_WIDTH-1:0] wr_ptr, rd_ptr;
    logic [CNT_WIDTH-1:0] count_reg;

    assign empty   = (count_reg == 0);
    assign full    = (count_reg == DEPTH);
    assign count   = count_reg;
    assign rd_data = mem[rd_ptr];

    logic do_wr0, do_wr1, do_read;
    assign do_wr0  = wr0_en && (count_reg < DEPTH);
    assign do_wr1  = wr1_en && ((count_reg + (do_wr0 ? 1 : 0)) < DEPTH);
    assign do_read = rd_en && !empty;

    logic [1:0] num_writes;
    assign num_writes = {1'b0, do_wr0} + {1'b0, do_wr1};

    logic [PTR_WIDTH-1:0] wr_ptr_p1, wr_ptr_p2;
    assign wr_ptr_p1 = (wr_ptr == DEPTH-1) ? '0 : wr_ptr + 1'b1;
    assign wr_ptr_p2 = (wr_ptr == DEPTH-1) ? 1'b1 :
                       (wr_ptr == DEPTH-2) ? '0   :
                                             wr_ptr + 2'd2;

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            wr_ptr    <= '0;
            rd_ptr    <= '0;
            count_reg <= '0;
```

```
        end else begin
           if (do_wr0) begin
                          mem[wr_ptr] <= wr0_data;
           end

            if (do_wr1) begin
                if (do_wr0) mem[wr_ptr_p1] <= wr1_data;
                else        mem[wr_ptr]    <= wr1_data;
            end

            unique case (num_writes)
                2'd0: wr_ptr <= wr_ptr;
                2'd1: wr_ptr <= wr_ptr_p1;
                2'd2: wr_ptr <= wr_ptr_p2;
                default: wr_ptr <= wr_ptr;
            endcase

            if (do_read) begin
                rd_ptr <= (rd_ptr == DEPTH-1) ? '0 : rd_ptr + 1'b1;
            end

            count_reg <= count_reg + num_writes - {1'b0, do_read};
        end
    end

endmodule

module tracking_fifo #(
    parameter int WIDTH = 32,
    parameter int DEPTH = 8
) (
    input  logic             clk,
    input  logic             rst_n,

    input  logic             push,
    input  logic [WIDTH-1:0] din,

    input  logic             pop,
    output logic [WIDTH-1:0] dout,

    output logic             full,
    output logic             empty
);

    localparam int PTR_WIDTH = $clog2(DEPTH);
```

```
    localparam int CNT_WIDTH = $clog2(DEPTH + 1);

    (* ramstyle = "MLAB" *)
    logic [WIDTH-1:0]     mem [DEPTH];
    logic [PTR_WIDTH-1:0] wr_ptr, rd_ptr;
    logic [CNT_WIDTH-1:0] count;

    assign full  = (count == DEPTH);
    assign empty = (count == 0);
    assign dout  = mem[rd_ptr];

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            wr_ptr <= '0;
            rd_ptr <= '0;
            count  <= '0;
        end else begin
            if (push && !full) begin
                mem[wr_ptr] <= din;
                wr_ptr <= (wr_ptr == DEPTH-1) ? '0 : wr_ptr + 1'b1;
            end

            if (pop && !empty) begin
                rd_ptr <= (rd_ptr == DEPTH-1) ? '0 : rd_ptr + 1'b1;
            end

            case ({push && !full, pop && !empty})
                2'b10: count <= count + 1'b1;
                2'b01: count <= count - 1'b1;
                default: count <= count;
            endcase
        end
    end
endmodule
```

None

```
module mem_bank #(
    parameter int BANK_ID = 0
) (
    input logic     clk,
    input logic     rst_n,

    input logic [31:0]     mem_addr,
```

```
    input logic     mem_we,
    input logic     mem_re,
    input logic [85:0]     mem_wdata,

    output logic [85:0] mem_rdata,
    output logic    mem_rdata_valid,
    output logic    mem_wdone,
    output logic    mem_busy
);
    localparam int FIRST_PAGE_FOR_BANK = 16 + BANK_ID * 60;

    logic [18:0] global_page;
    logic [5:0]  local_page_idx;
    logic [5:0]  node_idx;

    assign global_page    = mem_addr[31:13];
    assign local_page_idx = global_page[5:0] - FIRST_PAGE_FOR_BANK[5:0];
    assign node_idx       = mem_addr[9:4];

    typedef enum logic [2:0] {
        IDLE,
       PHASE_0,
        PHASE_1,
        PHASE_2,
        DONE_READ
    } state_t;

    state_t state, n_state;

    logic        latched_is_write;
    logic [5:0]  latched_page;
    logic [5:0]  latched_node;
    logic [85:0] latched_wdata;

    always_comb begin
        n_state = state;
        unique case (state)
            IDLE: begin
                if (mem_re || mem_we)
                    n_state = PHASE_0;
            end
           PHASE_0:   n_state = PHASE_1;
            PHASE_1:   n_state = PHASE_2;
            PHASE_2:   n_state = latched_is_write ? IDLE : DONE_READ;
            DONE_READ: n_state = IDLE;
```

```
    endcase
end

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        state            <= IDLE;
        latched_is_write <= 1'b0;
        latched_page     <= 6'd0;
        latched_node     <= 6'd0;
        latched_wdata    <= 86'd0;
    end else begin
        state <= n_state;
        if (state == IDLE && (mem_re || mem_we)) begin
            latched_is_write <= mem_we;
            latched_page     <= local_page_idx;
            latched_node     <= node_idx;
            latched_wdata    <= mem_wdata;
        end
    end
end

logic [7:0]  bram_waddr, bram_raddr;
logic [31:0] bram_wdata;

always_comb begin
    bram_waddr  = '0;
    bram_raddr  = '0;
    bram_wdata  = 32'd0;

    unique case (state)
       PHASE_0: begin
          bram_waddr  = {latched_node, 2'd0};
          bram_raddr  = {latched_node, 2'd0};
          bram_wdata  = latched_wdata[31:0];
       end
        PHASE_1: begin
            bram_waddr = {latched_node, 2'd1};
            bram_raddr = {latched_node, 2'd1};
            bram_wdata = latched_wdata[63:32];
        end
        PHASE_2: begin
            bram_waddr = {latched_node, 2'd2};
           bram_raddr = {latched_node, 2'd2};
            bram_wdata = {10'd0, latched_wdata[85:64]};
        end
```

```
        default: ;
    endcase
end

logic [59:0] bram_we;
logic [59:0] bram_re;
logic [31:0] bram_rdata [60];

always_comb begin
    bram_we = '0;
    bram_re = '0;

    if((state == PHASE_0) || (state == PHASE_1) || (state == PHASE_2)) begin
      if (latched_is_write) begin
          bram_we[latched_page] = 1'b1;
      end else begin
          bram_re[latched_page] = 1'b1;
      end
   end
end

logic [31:0] active_rdata;
assign active_rdata = bram_rdata[latched_page];

logic [85:0] read_assemble;

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        read_assemble <= 86'd0;
    end else if (!latched_is_write) begin
       if (state == PHASE_1) begin
         read_assemble[31:0] <= active_rdata;
       end
        if (state == PHASE_2) begin
            read_assemble[63:32]  <= active_rdata;
        end
        if (state == DONE_READ) begin
            read_assemble[85:64] <= active_rdata[21:0];
        end
    end
end

assign mem_busy = (state != IDLE);

always_ff @(posedge clk or negedge rst_n) begin
```

```
        if (!rst_n) begin
            mem_rdata       <= 86'd0;
            mem_rdata_valid <= 1'b0;
           mem_wdone    <= 1'b0;
        end else begin
            mem_rdata_valid <= 1'b0;
           mem_wdone    <= 1'b0;

           if (state == DONE_READ && !latched_is_write) begin
                mem_rdata       <= {active_rdata[21:0], read_assemble[63:0]};
                mem_rdata_valid <= 1'b1;
            end

           if (state == PHASE_2 && latched_is_write) begin
              mem_wdone <= 1'b1;
           end
        end
    end

    genvar p;
    generate
        for (p = 0; p < 60; p++) begin : gen_brams
            bram_dp_256x32 u_bram (
                .clk     (clk),

                .we    (bram_we[p]),
                .re    (bram_re[p]),
              .waddr (bram_waddr),
                .raddr (bram_raddr),
                .wdata (bram_wdata),
                .rdata (bram_rdata[p])
            );
        end
    endgenerate

endmodule


module bram_dp_256x32 (
    input  logic        clk,

    input  logic        we,
    input  logic        re,
    input  logic [7:0]  waddr,
    input  logic [7:0]  raddr,
```

```
    input  logic [31:0] wdata,
    output logic [31:0] rdata
);
    (* ramstyle = "no_rw_check, M10K", ram_init_file = "mem_bank_zero.mif" *)
    logic [31:0] mem [0:255];

    always_ff @(posedge clk) begin
        if (we) mem[waddr] <= wdata;
        if (re) rdata      <= mem[raddr];
    end

endmodule
```

None

```
module mmu (
    input  logic        clk,
    input  logic        rst_n,

    input  logic        req_valid_0, req_valid_1, req_valid_2, req_valid_3,
                        req_valid_4, req_valid_5, req_valid_6, req_valid_7,
    input  logic [31:0] req_va_0, req_va_1, req_va_2, req_va_3,
                        req_va_4, req_va_5, req_va_6, req_va_7,
    input  logic        req_wr_0, req_wr_1, req_wr_2, req_wr_3,
                        req_wr_4, req_wr_5, req_wr_6, req_wr_7,
    input  logic [85:0] req_wdata_0, req_wdata_1, req_wdata_2, req_wdata_3,
                        req_wdata_4, req_wdata_5, req_wdata_6, req_wdata_7,

    output logic        req_ready_0, req_ready_1, req_ready_2, req_ready_3,
                        req_ready_4, req_ready_5, req_ready_6, req_ready_7,

    output logic [85:0] resp_data_0, resp_data_1, resp_data_2, resp_data_3,
                        resp_data_4, resp_data_5, resp_data_6, resp_data_7,
    output logic        resp_valid_0, resp_valid_1, resp_valid_2, resp_valid_3,
                        resp_valid_4, resp_valid_5, resp_valid_6, resp_valid_7,
    output logic        resp_reject_0, resp_reject_1, resp_reject_2,
resp_reject_3,
                        resp_reject_4, resp_reject_5, resp_reject_6,
resp_reject_7,

    output logic [31:0] mem_addr_0, mem_addr_1, mem_addr_2, mem_addr_3,
    output logic        mem_we_0,   mem_we_1,   mem_we_2,   mem_we_3,
    output logic        mem_re_0,   mem_re_1,   mem_re_2,   mem_re_3,
    output logic [85:0] mem_wdata_0, mem_wdata_1, mem_wdata_2, mem_wdata_3,
```

```
    input  logic [85:0] mem_rdata_0, mem_rdata_1, mem_rdata_2, mem_rdata_3,
    input  logic        mem_rdata_valid_0, mem_rdata_valid_1,
                        mem_rdata_valid_2, mem_rdata_valid_3,
    input  logic        mem_busy_0, mem_busy_1, mem_busy_2, mem_busy_3,
    input  logic    mem_wdone_0, mem_wdone_1, mem_wdone_2, mem_wdone_3
);

    logic        req_valid [8];
    logic [31:0] req_va    [8];
    logic        req_wr    [8];
    logic [85:0] req_wdata [8];
    logic        req_ready [8];

    assign req_valid[0] = req_valid_0; assign req_valid[1] = req_valid_1;
    assign req_valid[2] = req_valid_2; assign req_valid[3] = req_valid_3;
    assign req_valid[4] = req_valid_4; assign req_valid[5] = req_valid_5;
    assign req_valid[6] = req_valid_6; assign req_valid[7] = req_valid_7;

    assign req_va[0] = req_va_0; assign req_va[1] = req_va_1;
    assign req_va[2] = req_va_2; assign req_va[3] = req_va_3;
    assign req_va[4] = req_va_4; assign req_va[5] = req_va_5;
    assign req_va[6] = req_va_6; assign req_va[7] = req_va_7;

    assign req_wr[0] = req_wr_0; assign req_wr[1] = req_wr_1;
    assign req_wr[2] = req_wr_2; assign req_wr[3] = req_wr_3;
    assign req_wr[4] = req_wr_4; assign req_wr[5] = req_wr_5;
    assign req_wr[6] = req_wr_6; assign req_wr[7] = req_wr_7;

    assign req_wdata[0] = req_wdata_0; assign req_wdata[1] = req_wdata_1;
    assign req_wdata[2] = req_wdata_2; assign req_wdata[3] = req_wdata_3;
    assign req_wdata[4] = req_wdata_4; assign req_wdata[5] = req_wdata_5;
    assign req_wdata[6] = req_wdata_6; assign req_wdata[7] = req_wdata_7;

    assign req_ready_0 = req_ready[0]; assign req_ready_1 = req_ready[1];
    assign req_ready_2 = req_ready[2]; assign req_ready_3 = req_ready[3];
    assign req_ready_4 = req_ready[4]; assign req_ready_5 = req_ready[5];
    assign req_ready_6 = req_ready[6]; assign req_ready_7 = req_ready[7];

    logic [2:0] rr_pointer;

    logic [3:0] fifo_in_count [2];
    logic       fifo_in_full  [2];
    logic       fifo_in_empty [2];

    assign fifo_in_full[0] = (fifo_in_count[0] >= 4'd7);
```

```
assign fifo_in_full[1] = (fifo_in_count[1] >= 4'd7);

logic [2:0] first_winner_idx, second_winner_idx;
logic       first_winner_found, second_winner_found;

always_comb begin
    first_winner_found  = 1'b0;
    second_winner_found = 1'b0;
    first_winner_idx    = 3'd0;
    second_winner_idx   = 3'd0;

    for (int i = 0; i < 8; i++) begin
        logic [2:0] idx;
        idx = rr_pointer + i[2:0];

        if (req_valid[idx]) begin
            if (!first_winner_found) begin
                first_winner_idx   = idx;
                first_winner_found = 1'b1;
            end else if (!second_winner_found) begin
                second_winner_idx   = idx;
                second_winner_found = 1'b1;
            end
        end
    end
end

logic first_to_fifo1;
assign first_to_fifo1 = (fifo_in_count[1] < fifo_in_count[0]);

logic                     requeue_to_0, requeue_to_1;
logic                     new_can_write_fifo0, new_can_write_fifo1;
assign new_can_write_fifo0 = !requeue_to_0;
assign new_can_write_fifo1 = !requeue_to_1;

logic first_target_has_space, second_target_has_space;
assign first_target_has_space  = first_to_fifo1
    ? (!fifo_in_full[1] && new_can_write_fifo1)
    : (!fifo_in_full[0] && new_can_write_fifo0);
assign second_target_has_space = first_to_fifo1
    ? (!fifo_in_full[0] && new_can_write_fifo0)
    : (!fifo_in_full[1] && new_can_write_fifo1);

logic accept_first, accept_second;
assign accept_first  = first_winner_found  && first_target_has_space;
```

```
assign accept_second = second_winner_found && second_target_has_space &&
                       accept_first;

always_comb begin
    for (int i = 0; i < 8; i++) req_ready[i] = 1'b0;
    if (accept_first)  req_ready[first_winner_idx]  = 1'b1;
    if (accept_second) req_ready[second_winner_idx] = 1'b1;
end

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n)
        rr_pointer <= 3'd0;
    else if (accept_first && accept_second)
        rr_pointer <= second_winner_idx + 3'd1;
    else if (accept_first)
        rr_pointer <= first_winner_idx + 3'd1;
end

localparam int IN_FIFO_WIDTH = 32 + 1 + 86;

logic [IN_FIFO_WIDTH-1:0] fifo_in_wr_data [2];
logic [IN_FIFO_WIDTH-1:0] fifo_in_rd_data [2];
logic                     fifo_in_wr_en   [2];
logic                     fifo_in_rd_en   [2];

logic [IN_FIFO_WIDTH-1:0] entry_first, entry_second;
assign entry_first  = {req_va[first_winner_idx],
                       req_wr[first_winner_idx],
                       req_wdata[first_winner_idx]};
assign entry_second = {req_va[second_winner_idx],
                       req_wr[second_winner_idx],
                       req_wdata[second_winner_idx]};

logic [IN_FIFO_WIDTH-1:0] requeue_data_0, requeue_data_1;

always_comb begin
    fifo_in_wr_en[0]   = 1'b0;
    fifo_in_wr_en[1]   = 1'b0;
    fifo_in_wr_data[0] = '0;
    fifo_in_wr_data[1] = '0;

    if (requeue_to_0) begin
        fifo_in_wr_en[0]   = 1'b1;
        fifo_in_wr_data[0] = requeue_data_0;
    end
```

```
    if (requeue_to_1) begin
        fifo_in_wr_en[1]   = 1'b1;
        fifo_in_wr_data[1] = requeue_data_1;
    end

    if (accept_first) begin
        if (first_to_fifo1 && new_can_write_fifo1) begin
            fifo_in_wr_en[1]   = 1'b1;
            fifo_in_wr_data[1] = entry_first;
        end else if (!first_to_fifo1 && new_can_write_fifo0) begin
            fifo_in_wr_en[0]   = 1'b1;
            fifo_in_wr_data[0] = entry_first;
        end
    end
    if (accept_second) begin
        if (first_to_fifo1 && new_can_write_fifo0) begin
            fifo_in_wr_en[0]   = 1'b1;
            fifo_in_wr_data[0] = entry_second;
        end else if (!first_to_fifo1 && new_can_write_fifo1) begin
            fifo_in_wr_en[1]   = 1'b1;
            fifo_in_wr_data[1] = entry_second;
        end
    end
end

fifo #(.WIDTH(IN_FIFO_WIDTH)) u_in_fifo_0 (
    .clk(clk), .rst_n(rst_n),
    .wr_en(fifo_in_wr_en[0]), .wr_data(fifo_in_wr_data[0]),
    .rd_en(fifo_in_rd_en[0]), .rd_data(fifo_in_rd_data[0]),
    .empty(fifo_in_empty[0]),
    .count(fifo_in_count[0])
);

fifo #(.WIDTH(IN_FIFO_WIDTH)) u_in_fifo_1 (
    .clk(clk), .rst_n(rst_n),
    .wr_en(fifo_in_wr_en[1]), .wr_data(fifo_in_wr_data[1]),
    .rd_en(fifo_in_rd_en[1]), .rd_data(fifo_in_rd_data[1]),
    .empty(fifo_in_empty[1]),
    .count(fifo_in_count[1])
);

logic [31:0] ptw0_in_va, ptw1_in_va;
logic        ptw0_in_wr, ptw1_in_wr;
logic [85:0] ptw0_in_wdata, ptw1_in_wdata;
```

```
assign {ptw0_in_va, ptw0_in_wr, ptw0_in_wdata} = fifo_in_rd_data[0];
assign {ptw1_in_va, ptw1_in_wr, ptw1_in_wdata} = fifo_in_rd_data[1];

logic        ptw0_pa_valid, ptw1_pa_valid;
logic [31:0] ptw0_pa,       ptw1_pa;
logic [1:0]  ptw0_bank_id,  ptw1_bank_id;
logic        ptw0_fault,    ptw1_fault;
logic        ptw0_busy,     ptw1_busy;

logic ptw0_va_valid_in, ptw1_va_valid_in;
assign ptw0_va_valid_in = !fifo_in_empty[0] && !ptw0_busy;
assign ptw1_va_valid_in = !fifo_in_empty[1] && !ptw1_busy;

assign fifo_in_rd_en[0] = ptw0_va_valid_in;
assign fifo_in_rd_en[1] = ptw1_va_valid_in;

logic [31:0] ptw0_pipe_va, ptw1_pipe_va;
logic        ptw0_pipe_wr, ptw1_pipe_wr;
logic [85:0] ptw0_pipe_wdata, ptw1_pipe_wdata;

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        ptw0_pipe_va    <= 32'd0;
        ptw0_pipe_wr    <= 1'b0;
        ptw0_pipe_wdata <= 86'd0;
        ptw1_pipe_va    <= 32'd0;
        ptw1_pipe_wr    <= 1'b0;
        ptw1_pipe_wdata <= 86'd0;
    end else begin
        if (ptw0_va_valid_in) begin
            ptw0_pipe_va    <= ptw0_in_va;
            ptw0_pipe_wr    <= ptw0_in_wr;
            ptw0_pipe_wdata <= ptw0_in_wdata;
        end
        if (ptw1_va_valid_in) begin
            ptw1_pipe_va    <= ptw1_in_va;
            ptw1_pipe_wr    <= ptw1_in_wr;
            ptw1_pipe_wdata <= ptw1_in_wdata;
        end
    end
end

// Per-partition write-pointer allocator. 16 partitions, one per
// (engine_id[2:0], heap_kind) tuple. Each partition owns 15 contiguous
// physical pages x 64 nodes = 960 unique slots.
```

```
logic [3:0] partition_page_off [16];   // 0..14
logic [5:0] partition_node_off [16];   // 0..63
logic       partition_full     [16];   // 1 once all 960 slots consumed

// Decode each PTW's target partition from its piped VA (same key the
// page_table is indexed on: top engine_id + heap_kind bit).
logic [3:0] ptw0_partition, ptw1_partition;
assign ptw0_partition = {ptw0_pipe_va[31:29], ptw0_pipe_va[10]};
assign ptw1_partition = {ptw1_pipe_va[31:29], ptw1_pipe_va[10]};

// Absolute visible-page index = partition * 15 + per-partition offset.
// Encoded via a precomputed base lookup.
logic [7:0] partition_base [16];
initial begin
    for (int p = 0; p < 16; p++) partition_base[p] = 8'(p * 15);
end

logic [7:0] ptw0_alloc_page_avail, ptw1_alloc_page_avail;
logic [5:0] ptw0_alloc_node_avail, ptw1_alloc_node_avail;
logic       ptw0_alloc_avail,      ptw1_alloc_avail;

assign ptw0_alloc_page_avail = partition_base[ptw0_partition]
                             + {4'd0, partition_page_off[ptw0_partition]};
assign ptw0_alloc_node_avail = partition_node_off[ptw0_partition];
assign ptw0_alloc_avail      = !partition_full[ptw0_partition];

assign ptw1_alloc_page_avail = partition_base[ptw1_partition]
                             + {4'd0, partition_page_off[ptw1_partition]};
assign ptw1_alloc_node_avail = partition_node_off[ptw1_partition];
assign ptw1_alloc_avail      = !partition_full[ptw1_partition];

logic       ptw0_alloc, ptw1_alloc;
logic [7:0] ptw0_alloc_page, ptw1_alloc_page;
logic [5:0] ptw0_alloc_node, ptw1_alloc_node;

logic same_alloc_collision;
assign same_alloc_collision = ptw0_alloc && ptw1_alloc
                           && (ptw0_partition == ptw1_partition);

logic ptw1_alloc_eff;
assign ptw1_alloc_eff = ptw1_alloc && !same_alloc_collision;

// Advance the write pointer for each allocating partition.
always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
```

```
        for (int p = 0; p < 16; p++) begin
            partition_page_off[p] <= 4'd0;
            partition_node_off[p] <= 6'd0;
            partition_full[p]     <= 1'b0;
        end
    end else begin
        if (ptw0_alloc) begin
            if (partition_node_off[ptw0_partition] == 6'd63) begin
                partition_node_off[ptw0_partition] <= 6'd0;
                if (partition_page_off[ptw0_partition] == 4'd14)
                    partition_full[ptw0_partition] <= 1'b1;
                else
                    partition_page_off[ptw0_partition] <=
                        partition_page_off[ptw0_partition] + 4'd1;
            end else begin
                partition_node_off[ptw0_partition] <=
                    partition_node_off[ptw0_partition] + 6'd1;
            end
        end
        if (ptw1_alloc_eff) begin
            if (partition_node_off[ptw1_partition] == 6'd63) begin
                partition_node_off[ptw1_partition] <= 6'd0;
                if (partition_page_off[ptw1_partition] == 4'd14)
                    partition_full[ptw1_partition] <= 1'b1;
                else
                    partition_page_off[ptw1_partition] <=
                        partition_page_off[ptw1_partition] + 4'd1;
            end else begin
                partition_node_off[ptw1_partition] <=
                    partition_node_off[ptw1_partition] + 6'd1;
            end
        end
    end
end

logic [13:0] ptw0_pt_raddr, ptw1_pt_raddr;
logic [14:0] ptw0_pt_rdata, ptw1_pt_rdata;
logic        ptw0_pt_we,    ptw1_pt_we;
logic [13:0] ptw0_pt_waddr, ptw1_pt_waddr;
logic [14:0] ptw0_pt_wdata, ptw1_pt_wdata;

logic ptw1_pt_we_eff;
assign ptw1_pt_we_eff = ptw1_pt_we && !same_alloc_collision;

(* ramstyle = "M10K, no_rw_check", ram_init_file = "page_table_zero.mif" *)
```

```
logic [14:0] page_table [0:16383];

always_ff @(posedge clk) begin
    if (ptw0_pt_we && (ptw0_pt_raddr == ptw0_pt_waddr)) begin
        page_table[ptw0_pt_raddr] <= ptw0_pt_wdata;
    end

    ptw0_pt_rdata <= page_table[ptw0_pt_raddr];

    if (ptw1_pt_we_eff && (ptw1_pt_raddr == ptw1_pt_waddr)) begin
        page_table[ptw1_pt_raddr] <= ptw1_pt_wdata;
    end

    ptw1_pt_rdata <= page_table[ptw1_pt_raddr];
end

HPTW #(.LOW_FIRST(1'b1)) u_ptw0 (
    .clk                (clk),
    .rst_n              (rst_n),
    .va_valid           (ptw0_va_valid_in),
    .va                 (ptw0_in_va),
    .pa_valid           (ptw0_pa_valid),
    .pa                 (ptw0_pa),
    .bank_id            (ptw0_bank_id),
    .fault              (ptw0_fault),
    .busy               (ptw0_busy),
    .pt_raddr           (ptw0_pt_raddr),
    .pt_rdata           (ptw0_pt_rdata),
    .pt_we              (ptw0_pt_we),
    .pt_waddr           (ptw0_pt_waddr),
    .pt_wdata           (ptw0_pt_wdata),
    .alloc_avail        (ptw0_alloc_avail),
    .alloc_page_in      (ptw0_alloc_page_avail),
    .alloc_node_in      (ptw0_alloc_node_avail),
    .alloc              (ptw0_alloc),
    .alloc_page_idx     (ptw0_alloc_page),
    .alloc_node_idx     (ptw0_alloc_node)
);

HPTW #(.LOW_FIRST(1'b0)) u_ptw1 (
    .clk                (clk),
    .rst_n              (rst_n),
    .va_valid           (ptw1_va_valid_in),
    .va                 (ptw1_in_va),
    .pa_valid           (ptw1_pa_valid),
```

```
    .pa                 (ptw1_pa),
    .bank_id            (ptw1_bank_id),
    .fault              (ptw1_fault),
    .busy               (ptw1_busy),
    .pt_raddr           (ptw1_pt_raddr),
    .pt_rdata           (ptw1_pt_rdata),
    .pt_we              (ptw1_pt_we),
    .pt_waddr           (ptw1_pt_waddr),
    .pt_wdata           (ptw1_pt_wdata),
    .alloc_avail        (ptw1_alloc_avail),
    .alloc_page_in      (ptw1_alloc_page_avail),
    .alloc_node_in      (ptw1_alloc_node_avail),
    .alloc              (ptw1_alloc),
    .alloc_page_idx     (ptw1_alloc_page),
    .alloc_node_idx     (ptw1_alloc_node)
);

logic ptw1_pa_valid_eff, ptw1_fault_eff;
assign ptw1_pa_valid_eff = ptw1_pa_valid && !same_alloc_collision;
assign ptw1_fault_eff    = ptw1_fault    || same_alloc_collision;

logic        arb_ptw0_valid, arb_ptw1_valid;
logic [31:0] arb_ptw0_va,    arb_ptw1_va;
logic [31:0] arb_ptw0_pa,    arb_ptw1_pa;
logic [1:0]  arb_ptw0_bank,  arb_ptw1_bank;
logic        arb_ptw0_wr,    arb_ptw1_wr;
logic [85:0] arb_ptw0_wdata, arb_ptw1_wdata;
logic        arb_ptw0_accept, arb_ptw0_reject;
logic        arb_ptw1_accept, arb_ptw1_reject;

assign arb_ptw0_valid = ptw0_pa_valid;
assign arb_ptw0_va    = ptw0_pipe_va;
assign arb_ptw0_pa    = ptw0_pa;
assign arb_ptw0_bank  = ptw0_bank_id;
assign arb_ptw0_wr    = ptw0_pipe_wr;
assign arb_ptw0_wdata = ptw0_pipe_wdata;

assign arb_ptw1_valid = ptw1_pa_valid_eff;
assign arb_ptw1_va    = ptw1_pipe_va;
assign arb_ptw1_pa    = ptw1_pa;
assign arb_ptw1_bank  = ptw1_bank_id;
assign arb_ptw1_wr    = ptw1_pipe_wr;
assign arb_ptw1_wdata = ptw1_pipe_wdata;

logic [85:0] arb_rdata_0, arb_rdata_1, arb_rdata_2, arb_rdata_3;
```

```
    logic [31:0] arb_rdata_va_0, arb_rdata_va_1, arb_rdata_va_2, arb_rdata_va_3;
    logic        arb_rdata_valid_0, arb_rdata_valid_1, arb_rdata_valid_2,
arb_rdata_valid_3;

    assign requeue_to_0   = arb_ptw0_reject;
    assign requeue_to_1   = arb_ptw1_reject;
    assign requeue_data_0 = {ptw0_pipe_va, ptw0_pipe_wr, ptw0_pipe_wdata};
    assign requeue_data_1 = {ptw1_pipe_va, ptw1_pipe_wr, ptw1_pipe_wdata};

    arbiter u_arbiter (
        .clk               (clk),
        .rst_n             (rst_n),

        .ptw0_valid        (arb_ptw0_valid),
        .ptw0_va           (arb_ptw0_va),
        .ptw0_pa           (arb_ptw0_pa),
        .ptw0_bank_id      (arb_ptw0_bank),
        .ptw0_wr           (arb_ptw0_wr),
        .ptw0_wdata        (arb_ptw0_wdata),

        .ptw1_valid        (arb_ptw1_valid),
        .ptw1_va           (arb_ptw1_va),
        .ptw1_pa           (arb_ptw1_pa),
        .ptw1_bank_id      (arb_ptw1_bank),
        .ptw1_wr           (arb_ptw1_wr),
        .ptw1_wdata        (arb_ptw1_wdata),

        .ptw0_accept       (arb_ptw0_accept),
        .ptw0_reject       (arb_ptw0_reject),
        .ptw1_accept       (arb_ptw1_accept),
        .ptw1_reject       (arb_ptw1_reject),

        .rdata_0           (arb_rdata_0),
        .rdata_va_0        (arb_rdata_va_0),
        .rdata_valid_0     (arb_rdata_valid_0),
        .rdata_1           (arb_rdata_1),
        .rdata_va_1        (arb_rdata_va_1),
        .rdata_valid_1     (arb_rdata_valid_1),
        .rdata_2           (arb_rdata_2),
        .rdata_va_2        (arb_rdata_va_2),
        .rdata_valid_2     (arb_rdata_valid_2),
        .rdata_3           (arb_rdata_3),
        .rdata_va_3        (arb_rdata_va_3),
        .rdata_valid_3     (arb_rdata_valid_3),
```

```
    .mem_addr_0        (mem_addr_0),
    .mem_we_0          (mem_we_0),
    .mem_re_0          (mem_re_0),
    .mem_wdata_0       (mem_wdata_0),
    .mem_rdata_0       (mem_rdata_0),
    .mem_rdata_valid_0 (mem_rdata_valid_0),
   .mem_wdone_0    (mem_wdone_0),
    .mem_busy_0        (mem_busy_0),

    .mem_addr_1        (mem_addr_1),
    .mem_we_1          (mem_we_1),
    .mem_re_1          (mem_re_1),
    .mem_wdata_1       (mem_wdata_1),
    .mem_rdata_1       (mem_rdata_1),
    .mem_rdata_valid_1 (mem_rdata_valid_1),
   .mem_wdone_1    (mem_wdone_1),
    .mem_busy_1        (mem_busy_1),

    .mem_addr_2        (mem_addr_2),
    .mem_we_2          (mem_we_2),
    .mem_re_2          (mem_re_2),
    .mem_wdata_2       (mem_wdata_2),
    .mem_rdata_2       (mem_rdata_2),
    .mem_rdata_valid_2 (mem_rdata_valid_2),
   .mem_wdone_2    (mem_wdone_2),
    .mem_busy_2        (mem_busy_2),

    .mem_addr_3        (mem_addr_3),
    .mem_we_3          (mem_we_3),
    .mem_re_3          (mem_re_3),
    .mem_wdata_3       (mem_wdata_3),
    .mem_rdata_3       (mem_rdata_3),
    .mem_rdata_valid_3 (mem_rdata_valid_3),
   .mem_wdone_3    (mem_wdone_3),
    .mem_busy_3        (mem_busy_3)
);

logic [85:0] resp_data    [8];
logic        resp_valid   [8];

always_comb begin
    for (int i = 0; i < 8; i++) begin
        resp_data[i]  = 86'd0;
        resp_valid[i] = 1'b0;
    end
```

```
    if (arb_rdata_valid_0) begin
        resp_data[arb_rdata_va_0[31:29]]  = arb_rdata_0;
        resp_valid[arb_rdata_va_0[31:29]] = 1'b1;
    end
    if (arb_rdata_valid_1) begin
        resp_data[arb_rdata_va_1[31:29]]  = arb_rdata_1;
        resp_valid[arb_rdata_va_1[31:29]] = 1'b1;
    end
    if (arb_rdata_valid_2) begin
        resp_data[arb_rdata_va_2[31:29]]  = arb_rdata_2;
        resp_valid[arb_rdata_va_2[31:29]] = 1'b1;
    end
    if (arb_rdata_valid_3) begin
        resp_data[arb_rdata_va_3[31:29]]  = arb_rdata_3;
        resp_valid[arb_rdata_va_3[31:29]] = 1'b1;
    end
end

assign resp_data_0  = resp_data[0];  assign resp_valid_0 = resp_valid[0];
assign resp_data_1  = resp_data[1];  assign resp_valid_1 = resp_valid[1];
assign resp_data_2  = resp_data[2];  assign resp_valid_2 = resp_valid[2];
assign resp_data_3  = resp_data[3];  assign resp_valid_3 = resp_valid[3];
assign resp_data_4  = resp_data[4];  assign resp_valid_4 = resp_valid[4];
assign resp_data_5  = resp_data[5];  assign resp_valid_5 = resp_valid[5];
assign resp_data_6  = resp_data[6];  assign resp_valid_6 = resp_valid[6];
assign resp_data_7  = resp_data[7];  assign resp_valid_7 = resp_valid[7];

logic resp_reject [8];

always_comb begin
    for (int i = 0; i < 8; i++) resp_reject[i] = 1'b0;

    if (ptw0_fault) begin
        resp_reject[ptw0_pipe_va[31:29]] = 1'b1;
    end
    if (ptw1_fault_eff) begin
        resp_reject[ptw1_pipe_va[31:29]] = 1'b1;
    end
end

assign resp_reject_0 = resp_reject[0];
assign resp_reject_1 = resp_reject[1];
assign resp_reject_2 = resp_reject[2];
assign resp_reject_3 = resp_reject[3];
```

```
    assign resp_reject_4 = resp_reject[4];
    assign resp_reject_5 = resp_reject[5];
    assign resp_reject_6 = resp_reject[6];
    assign resp_reject_7 = resp_reject[7];

endmodule

module fifo #(
    parameter int WIDTH = 8
) (
    input  logic             clk,
    input  logic             rst_n,

    input  logic             wr_en,
    input  logic [WIDTH-1:0] wr_data,

    input  logic             rd_en,
    output logic [WIDTH-1:0] rd_data,

    output logic             empty,
    output logic [3:0]       count
);

    logic [WIDTH-1:0] mem [8];
    logic [2:0]       wr_ptr, rd_ptr;
    logic [3:0]       count_reg;
    logic             full;

    assign empty   = (count_reg == 4'd0);
    assign full    = (count_reg == 4'd8);
    assign count   = count_reg;
    assign rd_data = mem[rd_ptr];

    logic do_write, do_read;
    assign do_write = wr_en && !full;
    assign do_read  = rd_en && !empty;

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            wr_ptr    <= 3'd0;
            rd_ptr    <= 3'd0;
            count_reg <= 4'd0;
        end else begin
            if (do_write) begin
                mem[wr_ptr] <= wr_data;
```

```
                wr_ptr      <= wr_ptr + 3'd1;
            end
            if (do_read) begin
                rd_ptr <= rd_ptr + 3'd1;
            end
            unique case ({do_write, do_read})
                2'b10:   count_reg <= count_reg + 4'd1;
                2'b01:   count_reg <= count_reg - 4'd1;
                default: count_reg <= count_reg;
            endcase
        end
    end

endmodule
```

None

```
// trade_aggregator: Round-robin merge of N heap engine streams.
//
// Each engine drives its own valid/data/ready handshake. The aggregator
// scans engines starting from a round-robin pointer; the first engine
// found with a valid trade wins this cycle; all others see ready=0.
// On a successful handshake, the Round Robin pointer advances to the engine
// afterthe winner so no engine starves.
//
// For the trade log we only keep:
//     - engine_id   (the winning engine's index, 3b -> low byte)
//     - amount      (low 7b of the ORDER's amount field; trades cap <128)
//     - price       (16b)
//     - timestamp   (32b)


module trade_aggregator #(
    parameter int N           = 8,
    parameter int NODE_WIDTH  = 86,  // engine-side trade width (ORDER struct)
    parameter int TRADE_WIDTH = 64   // aggregator-output / trade_log width
) (
    input  logic                   clk,
    input  logic                   rst_n,

    input  logic [N-1:0]           eng_trade_valid,
    input  logic [NODE_WIDTH-1:0]  eng_trade_data  [N],
    output logic [N-1:0]           eng_trade_ready,
```

```
    output logic                   trade_out_valid,
    output logic [TRADE_WIDTH-1:0] trade_out_data,
    input  logic                   trade_out_ready
);

    localparam int PTR_W = $clog2(N);

    logic [PTR_W-1:0] rr_ptr;

    logic [PTR_W-1:0] selected;
    logic             any_valid;

        selected  = rr_ptr;
        any_valid = 1'b0;
        for (int i = 0; i < N; i++) begin
            logic [PTR_W-1:0] idx;
            idx = rr_ptr + i[PTR_W-1:0];
            if (eng_trade_valid[idx] && !any_valid) begin
                selected  = idx;
                any_valid = 1'b1;
            end
        end
    end

    // Compact the winner's ORDER into a 64-bit TRADE_LOG_ENTRY.
        logic [NODE_WIDTH-1:0] winner_order;
    assign winner_order   = eng_trade_data[selected];

    assign trade_out_valid = any_valid;
    assign trade_out_data  = { {(8-PTR_W){1'b0}}, selected,           // [63:56]
engine_id
                               1'b0, winner_order[59:53],             // [55:48]
amount[6:0]
                               winner_order[84:69],                   // [47:32]
price
                               winner_order[31:0] };                  // [31:0]
timestamp

    always_comb begin
        eng_trade_ready = '0;
        if (any_valid) eng_trade_ready[selected] = trade_out_ready;
    end

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n)
```

```
            rr_ptr <= '0;
        else if (any_valid && trade_out_ready)
            rr_ptr <= selected + 1'b1;
    end

endmodule
```

None

```
// trade_log.sv
//
// Stores every NODE_WIDTH-bit trade event the trade_aggregator emits in a
// linear, append-only memory. Software collects trades by reading
// mem[0 .. sw_count-1] and then pulses sw_clear to recycle the log.
//
// Default NODE_WIDTH is 64
//
//   trade_in_ready falls when the log is full. The aggregator stalls
//   until software clears the log. If a trade tries to push while the
//   log is full, sw_overflow latches high so software can detect a
//   missed trade after the fact.

module trade_log #(
    parameter int NODE_WIDTH = 64,
    parameter int LOG_DEPTH  = 8704,
    parameter int CNT_WIDTH  = $clog2(LOG_DEPTH + 1),
    parameter int ADDR_WIDTH = $clog2(LOG_DEPTH)
) (
    input  logic                  clk,
    input  logic                  rst_n,

    // HW write port (from trade_aggregator)
    input  logic                  trade_in_valid,
    input  logic [NODE_WIDTH-1:0] trade_in_data,
    output logic                  trade_in_ready,

    // SW read port (1-cycle registered read latency)
    input  logic                  sw_re,
    input  logic [ADDR_WIDTH-1:0] sw_addr,
    output logic [NODE_WIDTH-1:0] sw_rdata,

    // Status / control
    output logic [CNT_WIDTH-1:0]  sw_count,
    output logic                  sw_overflow,
```

```
    input  logic                 sw_clear
);

    (* ramstyle = "M10K", ram_init_file = "trade_log_zero.mif" *)
    logic [NODE_WIDTH-1:0] mem [0:LOG_DEPTH-1];
    logic [ADDR_WIDTH-1:0] wr_ptr;
    logic [CNT_WIDTH-1:0]  count_reg;
    logic                  overflow_reg;

    logic accept_write;
    assign accept_write = trade_in_valid && (count_reg <
LOG_DEPTH[CNT_WIDTH-1:0]);

    assign trade_in_ready = (count_reg < LOG_DEPTH[CNT_WIDTH-1:0]);
    assign sw_count       = count_reg;
    assign sw_overflow    = overflow_reg;

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            wr_ptr       <= '0;
            count_reg    <= '0;
            overflow_reg <= 1'b0;
            sw_rdata     <= '0;
        end else begin
            if (sw_re) sw_rdata <= mem[sw_addr];

            if (sw_clear) begin
                wr_ptr       <= '0;
                count_reg    <= '0;
                overflow_reg <= 1'b0;
            end else begin
                if (accept_write) begin
                    mem[wr_ptr] <= trade_in_data;
                    wr_ptr      <= wr_ptr + 1'b1;
                    count_reg   <= count_reg + 1'b1;
                end
                if (trade_in_valid && !accept_write) begin
                    overflow_reg <= 1'b1;
                end
            end
        end
    end

endmodule
```

C/C++

```
//////////////////////////////////////////////////////////////////////////////////
/
// Engineers: Carlos Espinoza
// Create Date: 04/07/2026
// Project Name: Virtualized High Frequence Trading (HFT) Simulator
// Design Name: Device Drivers
// Description:
//      This is the device drivers for the Virtualize HFT Device.
//
// Revision: 05/08/2026
//////////////////////////////////////////////////////////////////////////////////
/

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/types.h>
#include <linux/mutex.h>
#include <linux/delay.h>
#include "HFT_drivers.h"

#define DRIVER_NAME "hft_sim"

// Device registers
#define REG_CONTROL     0x00
#define REG_STATUS      0x04
#define REG_PUSH_BASE   0x08
#define REG_PUSH(n)     (REG_PUSH_BASE + 4 * (n)) // We are using 32-bit words
#define REG_LOG_INFO    0x28
#define REG_LOG_CMD     0x2C
#define REG_LOG_DATA0   0x30
#define REG_LOG_DATA1   0x34
#define REG_ADDR(x)     (dev.virtbase + (x))

// Control and status bit defs
```

```
#define CTRL_BEGIN_WRITE                BIT(0)
#define CTRL_BEGIN_DISPATCH             BIT(1)
#define CTRL_CLEAR_DONE                 BIT(2)
#define STATUS_STATE_MASK               0x00000003
#define STATUS_READY_MASK               0x000003FC
#define STATUS_EMPTY_MASK               0x0003FC00
#define STATUS_FULL_MASK                0x03FC0000
#define STATUS_READY_SHIFT              2
#define STATUS_EMPTY_SHIFT              10
#define STATUS_FULL_SHIFT               18
#define HFT_STATE_IDLE                  0
#define HFT_STATE_WRITE                 1
#define HFT_STATE_DISPATCH              2
#define HFT_STATE_DONE                  3
#define LOG_INFO_OVERFLOW_BIT           0
#define LOG_INFO_DATA_VALID_BIT         1
#define LOG_INFO_COUNT_SHIFT            2
#define LOG_INFO_COUNT_MASK             (0x7FFFu << LOG_INFO_COUNT_SHIFT) /* 15b
mask covers count for depth up to 32767 (current TRADE_LOG_DEPTH = 8704) */
#define LOG_INFO_ENGINE_IDLE_SHIFT      17
#define LOG_INFO_ENGINE_IDLE_MASK       (0xFFu << LOG_INFO_ENGINE_IDLE_SHIFT)
#define LOG_INFO_ALL_ENGINES_IDLE_BIT   25
#define LOG_INFO_TRADE_DONE_BIT         26

// Information about HFT_SIM device
struct hft_dev {
    struct resource res;            // Resource: registers
    void __iomem *virtbase;         // Where registers can be accessed in memory
    struct mutex lock;
} dev;

////////////////////////////////////////////////////////////////////////
// Helper Functions
////////////////////////////////////////////////////////////////////////

// Constructs Orders to match what Order Dispatcher expects:
// [type [31]| price [30:15] | quantity [14:0]]
static inline __u32 hft_disp_pack_order_word(const struct hft_disp_order *o){
    return ((((__u32)(o->type & 0x1)) << 31) |
           ((((__u32)(o->price & 0xFFFF)) << 15) |
           (((__u32)(o->quantity & 0x7FFF));
}

// Write Order to specific buffer in hardware
static int hft_disp_push_order_hw(__u32 lane, const struct hft_disp_order *o){
```

```
    __u32 raw, ready_mask;

    // Check lane of buffer we are writing to
    if (lane >= 8)
        return -EINVAL;

    // Check that type and quantity fit
    if (o->type > 1 || o->quantity > 0x7FFF)
        return -EINVAL;

    // Check dipatcher status and lanes are ready to write
    raw = ioread32(REG_ADDR(REG_STATUS));
    if ((raw & STATUS_STATE_MASK) != HFT_STATE_WRITE)
        return -EAGAIN;
    ready_mask = (raw & STATUS_READY_MASK) >> STATUS_READY_SHIFT;
    if (!(ready_mask & BIT(lane)))
        return -EBUSY;

    // Write order word into PUSH lane register
    iowrite32(hft_disp_pack_order_word(o), REG_ADDR(REG_PUSH(lane)));
    return 0;
}

// Read trade log
static int hft_log_read_entry_hw(struct hft_log_entry *entry){
    __u32 status, state;
    __u32 info, count;
    int timeout_us = 1000;

    // Check dispatcher STATUS for DONE state
    status = ioread32(REG_ADDR(REG_STATUS));
    state  = status & STATUS_STATE_MASK;
    if (state != HFT_STATE_DONE)
        return -EAGAIN;

    // Read LOG_INFO for: count, overflow, data_valid, and engine idle bits
    info  = ioread32(REG_ADDR(REG_LOG_INFO));
    count = (info & LOG_INFO_COUNT_MASK) >> LOG_INFO_COUNT_SHIFT;
    if (entry->index >= count)
        return -EINVAL;

    // Issue trade log read request: bit1=read_req, bits[...]=index<<2
    iowrite32(BIT(1) | (entry->index << 2), REG_ADDR(REG_LOG_CMD));

    // Wait for valid data:
```

```
    // LOG_INFO[1] = data_valid
    do {
        info = ioread32(REG_ADDR(REG_LOG_INFO));
        if (info & BIT(LOG_INFO_DATA_VALID_BIT))
            break;
        udelay(1);
    } while (--timeout_us);
    if (!(info & BIT(LOG_INFO_DATA_VALID_BIT)))
        return -ETIMEDOUT;

    // Read trade logs payload words
    entry->word0 = ioread32(REG_ADDR(REG_LOG_DATA0));
    entry->word1 = ioread32(REG_ADDR(REG_LOG_DATA1));

    return 0;
}

////////////////////////////////////////////////////////////////////////
// User API
////////////////////////////////////////////////////////////////////////

// Handle ioctl() calls from user
static long hft_ioctl(struct file *f, unsigned int cmd, unsigned long arg){
    void __user *user_arg = (void __user *)arg;
    struct hft_disp_push_req req;
    struct hft_disp_status st;
    struct hft_log_info li;
    struct hft_log_entry le;
    __u32 disp_status_raw;
    __u32 log_info_raw;
    long ret = 0;

    mutex_lock(&dev.lock);

    // Determine what operation to perform
    switch (cmd) {
        case HFT_IOC_DISP_BEGIN_WRITE:{
            // Write dispatcher CONTROL: begin_write pulse
            iowrite32(CTRL_BEGIN_WRITE, REG_ADDR(REG_CONTROL));
            break;
        }

        case HFT_IOC_DISP_BEGIN_DISPATCH:{
            // Write dispatcher CONTROL: begin_dispatch pulse
            iowrite32(CTRL_BEGIN_DISPATCH, REG_ADDR(REG_CONTROL));
```

```
            break;
        }

        case HFT_IOC_DISP_CLEAR_DONE:{
            // Write dispatcher CONTROL: clear_done pulse
            iowrite32(CTRL_CLEAR_DONE, REG_ADDR(REG_CONTROL));
            break;
        }

        case HFT_IOC_LOG_CLEAR:{
            // Write LOG_CMD: clear pulse (bit0)
            iowrite32(BIT(0), REG_ADDR(REG_LOG_CMD));
            break;
        }

        case HFT_IOC_DISP_PUSH_ORDER:{
            if (copy_from_user(&req, user_arg, sizeof(req))) {
                ret = -EFAULT;
                break;
            }

            ret = hft_disp_push_order_hw(req.lane, &req.order);
            break;
        }

        case HFT_IOC_DISP_GET_STATUS: {
            // Read dispatcher status
            disp_status_raw = ioread32(REG_ADDR(REG_STATUS));
            pr_info("HFT REG_STATUS raw=0x%08x\n", disp_status_raw);

            // Unpack signals
            st.state      = disp_status_raw & STATUS_STATE_MASK;
            st.ready_mask = (disp_status_raw & STATUS_READY_MASK) >>
STATUS_READY_SHIFT;
            st.empty_mask = (disp_status_raw & STATUS_EMPTY_MASK) >>
STATUS_EMPTY_SHIFT;
            st.full_mask  = (disp_status_raw & STATUS_FULL_MASK)  >>
STATUS_FULL_SHIFT;

            if (copy_to_user(user_arg, &st, sizeof(st)))
                ret = -EFAULT;
            break;
        }

        case HFT_IOC_LOG_GET_INFO: {
```

```
            // Probe: allow read regardless of dispatcher state so we can see
            // engine_idle_mask / all_engines_idle / trade_done while the
            // dispatcher is stuck in DISPATCH. count may be stale until DONE,
            // but the engine status bits come straight from the HW and are
            // always valid.
            log_info_raw = ioread32(REG_ADDR(REG_LOG_INFO));
            li.overflow         = !!(log_info_raw & BIT(LOG_INFO_OVERFLOW_BIT));
            li.count            = (log_info_raw & LOG_INFO_COUNT_MASK) >>
LOG_INFO_COUNT_SHIFT;
            li.engine_idle_mask = (log_info_raw & LOG_INFO_ENGINE_IDLE_MASK) >>
LOG_INFO_ENGINE_IDLE_SHIFT;
            li.all_engines_idle = !!(log_info_raw &
BIT(LOG_INFO_ALL_ENGINES_IDLE_BIT));
            li.trade_done       = !!(log_info_raw &
BIT(LOG_INFO_TRADE_DONE_BIT));

            if (copy_to_user(user_arg, &li, sizeof(li)))
                ret = -EFAULT;
            break;
        }

        case HFT_IOC_LOG_READ_ENTRY:{
            if (copy_from_user(&le, user_arg, sizeof(le))) {
                ret = -EFAULT;
                break;
            }

            ret = hft_log_read_entry_hw(&le);
            if (ret)
                break;

            if (copy_to_user(user_arg, &le, sizeof(le)))
                ret = -EFAULT;
            break;
        }

        default:{
            ret = -EINVAL;
            break;
        }
    }

    mutex_unlock(&dev.lock);
    return ret;
}
```

```
// The operations our device knows how to do
static const struct file_operations hft_fops = {
    .owner          = THIS_MODULE,
    .unlocked_ioctl = hft_ioctl,
};

// Information about our device for the "misc" framework -- like a char dev
static struct miscdevice hft_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name  = DRIVER_NAME,
    .fops  = &hft_fops,
};

/////////////////////////////////////////////////////////////////////////
// Driver Probe and Remove
/////////////////////////////////////////////////////////////////////////

// Initialization code: get resources (registers) and display a welcome message
static int __init hft_probe(struct platform_device *pdev){
    int ret;

    pr_info(DRIVER_NAME ": probe called for %s\n", dev_name(&pdev->dev));

    mutex_init(&dev.lock);

    ret = misc_register(&hft_misc_device);
    if (ret) {
        pr_err(DRIVER_NAME ": misc_register failed: %d\n", ret);
        return ret;
    }

    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        pr_err(DRIVER_NAME ": of_address_to_resource failed: %d\n", ret);
        ret = -ENOENT;
        goto out_deregister;
    }

    pr_info(DRIVER_NAME ": resource start=%pa size=%lu\n",
        &dev.res.start, (unsigned long)resource_size(&dev.res));

    if (!request_mem_region(dev.res.start, resource_size(&dev.res), DRIVER_NAME))
{
        pr_err(DRIVER_NAME ": request_mem_region failed\n");
```

```
        ret = -EBUSY;
        goto out_deregister;
    }

    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (!dev.virtbase) {
        pr_err(DRIVER_NAME ": of_iomap failed\n");
        ret = -ENOMEM;
        goto out_release_mem;
    }

    pr_info(DRIVER_NAME ": probe successful\n");
    return 0;
out_release_mem:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&hft_misc_device);
    return ret;
}

// Clean-up code: release resources
static int hft_remove(struct platform_device *pdev){
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&hft_misc_device);
    return 0;
}

/////////////////////////////////////////////////////////////////////////
// Driver Probe and Remove
/////////////////////////////////////////////////////////////////////////

// Which "compatible" string(s) to search for in the Device Tree
#ifdef CONFIG_OF
static const struct of_device_id hft_of_match[] = {
    { .compatible = "csee4840,hft_sim-1.0" },
    { },
};
MODULE_DEVICE_TABLE(of, hft_of_match);
#endif

// Information for registering ourselves as a "platform" driver
static struct platform_driver hft_driver = {
    .driver = {
        .name = DRIVER_NAME,
```

```
        .owner = THIS_MODULE,
        .of_match_table = of_match_ptr(hft_of_match),
    },
    .remove = __exit_p(hft_remove),
};

// Called when the module is loaded: set things up
static int __init hft_init(void){
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&hft_driver, hft_probe);
}

// Calball when the module is unloaded: release resources
static void __exit hft_exit(void){
    platform_driver_unregister(&hft_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(hft_init);
module_exit(hft_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Carlos Espinoza");
MODULE_DESCRIPTION("HFT_SIM Driver");
```

C/C++

```
#ifndef _HFT_DRIVER_H
#define _HFT_DRIVER_H
#include <linux/ioctl.h>
#include <linux/types.h>
#define HFT_NUM_LANES 8

// Structs for Interacting with Dispatcher
struct hft_disp_order {
    __u8  type;      // 0 or 1
    __u16 price;     // 16 bits
    __u16 quantity;  // low 15 bits
};

struct hft_disp_push_req {
    __u32 lane;  // 0..7
    struct hft_disp_order order;
};
```

```
struct hft_disp_status {
    __u32 state;       // IDLE/WRITE/DISPATCH/DONE
    __u32 ready_mask;  // per-lane write ready
    __u32 empty_mask;  // per-lane fifo empty
    __u32 full_mask;   // per-lane fifo full
};

// Structs for Interacting with Trade Logger
struct hft_log_info {
    __u32 count;
    __u32 overflow;
    __u32 engine_idle_mask;
    __u32 all_engines_idle;
    __u32 trade_done;
};

struct hft_log_entry {
    __u32 index;   // input
    __u32 word0;   // output: [31:0]
    __u32 word1;   // output: [63:32]
};

// Communication with Order Dispatcher / Trade Logger
#define HFT_DRIVERS_MAGIC 'h'

#define HFT_IOC_DISP_BEGIN_WRITE     _IO (HFT_DRIVERS_MAGIC, 1)
#define HFT_IOC_DISP_BEGIN_DISPATCH  _IO (HFT_DRIVERS_MAGIC, 2)
#define HFT_IOC_DISP_CLEAR_DONE      _IO (HFT_DRIVERS_MAGIC, 3)
#define HFT_IOC_DISP_PUSH_ORDER      _IOW(HFT_DRIVERS_MAGIC, 4, struct
hft_disp_push_req)
#define HFT_IOC_DISP_GET_STATUS      _IOR(HFT_DRIVERS_MAGIC, 5, struct
hft_disp_status)

#define HFT_IOC_LOG_GET_INFO         _IOR (HFT_DRIVERS_MAGIC, 6, struct
hft_log_info)
#define HFT_IOC_LOG_READ_ENTRY       _IOWR(HFT_DRIVERS_MAGIC, 7, struct
hft_log_entry)
#define HFT_IOC_LOG_CLEAR            _IO  (HFT_DRIVERS_MAGIC, 8)

#endif
```

C/C++

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <stdarg.h>
#include <errno.h>
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <signal.h>

#include "HFT_drivers.h"

#define HFT_ALL_LANES_MASK ((1u << HFT_NUM_LANES) - 1u)
#define HFT_STATE_IDLE     0
#define HFT_STATE_WRITE    1
#define HFT_STATE_DISPATCH 2
#define HFT_STATE_DONE     3

// Error macros
#define HFT_ERR_RESET (-ECONNRESET)

// Driver
static int hft_sim_fd = -1;

// Ctrl+C handling
static volatile sig_atomic_t g_stop = 0;
static void on_sigint(int signo) {
    (void)signo;
    g_stop = 1;
}

// Internal Structs
typedef struct {
    FILE *lane_csv[HFT_NUM_LANES];   // One CSV stream per lane (see below
correspondence)
    bool  lane_eof[HFT_NUM_LANES];   // True if EOF reached on that lane
} LaneCSVReader;

////////////////////////////////////////////////////////////////////////
// Helper Functions
////////////////////////////////////////////////////////////////////////
```

```c
///////// Time/Sleep /////////
static uint64_t now_ms(void) {
    struct timespec ts;
    if (clock_gettime(CLOCK_MONOTONIC, &ts) != 0) {
        return 0;
    }
    return (uint64_t)ts.tv_sec * 1000ull + (uint64_t)ts.tv_nsec / 1000000ull;
}

static int sleep_ms(unsigned ms) {
    struct timespec ts;
    ts.tv_sec  = ms / 1000u;
    ts.tv_nsec = (long)(ms % 1000u) * 1000000l;
    while (nanosleep(&ts, &ts) != 0) {
        if (errno == EINTR) continue;
        return -errno;
    }
    return 0;
}

static int sleep_us(unsigned us) {
    struct timespec ts;
    ts.tv_sec  = us / 1000000u;
    ts.tv_nsec = (long)(us % 1000000u) * 1000l;
    while (nanosleep(&ts, &ts) != 0) {
        if (errno == EINTR) continue;
        return -errno;
    }
    return 0;
}

///////// CSV Helpers /////////
static void close_all_csvs(LaneCSVReader *r) {
    if (!r) return;

    // Close all CSVs
    for (int i = 0; i < HFT_NUM_LANES; i++) {
        if (r->lane_csv[i]) {
            fclose(r->lane_csv[i]);
            r->lane_csv[i] = NULL;
        }
        r->lane_eof[i] = false;
    }
}
```

```c
static int open_all_csvs(LaneCSVReader *r, const char *data_dir) {
    if (!r || !data_dir) return -EINVAL;

    // All files that will be used
    static const char *lane_file[HFT_NUM_LANES] = {
        "AAPL.csv", "BSX.csv", "BUS.csv", "MMM.csv",
        "MSFT.csv", "SBUX.csv", "TUS.csv", "WMT.csv"
    };

    // Try to open CSV files
    for (int lane = 0; lane < HFT_NUM_LANES; lane++) {
        char path[512];
        snprintf(path, sizeof(path), "%s/%s", data_dir, lane_file[lane]);

        // Try to open file
        r->lane_csv[lane] = fopen(path, "r");

        // If opening fails, close all files and print error message.
        if (!r->lane_csv[lane]) {
            fprintf(stderr, "ERROR: cannot open lane %d CSV '%s': %s\n",
                    lane, path, strerror(errno));
            close_all_csvs(r);
            return -ENOENT;
        }

        r->lane_eof[lane] = false;
    }
    return 0;
}

static void csv_rewind_all(LaneCSVReader *csv) {
    if (!csv) return;
    for (int i = 0; i < HFT_NUM_LANES; i++) {
        if (csv->lane_csv[i]) {
            rewind(csv->lane_csv[i]);
            clearerr(csv->lane_csv[i]);
        }
        csv->lane_eof[i] = false;
    }
}

///////// Order Dispatcher States /////////
// Get Status of Order Dispatcher
int hft_disp_get_status(int fd, struct hft_disp_status *st) {
```

```
    if (!st) return -EINVAL;
    if (ioctl(fd, HFT_IOC_DISP_GET_STATUS, st) < 0) return -errno;
    return 0;
}

// Wait for a specific state in the Order Dispatcher
int hft_disp_wait_state(int fd, uint32_t want_state, int timeout_ms, int poll_ms,
bool abort_on_idle) {
    if (poll_ms <= 0) poll_ms = 1;
    uint64_t deadline = (timeout_ms < 0) ? 0 : (now_ms() + (uint64_t)timeout_ms);

    for (;;) {
        // Get state
        struct hft_disp_status st;
        int rc = hft_disp_get_status(fd, &st);
        if (rc) return rc;

        // Check for desire state
        if (st.state == want_state) return 0;

        // Check if board was reseted: Dispatcher went into IDLE all the sudden
        if (abort_on_idle && st.state == HFT_STATE_IDLE && want_state !=
HFT_STATE_IDLE)
            return HFT_ERR_RESET;

        // Check for timeout
        if (timeout_ms >= 0 && now_ms() >= deadline) return -ETIMEDOUT;
        (void)sleep_ms((unsigned)poll_ms);
    }
}

// Signals and wait until Order Dispatcher has transitioned to DISPATCHT
int hft_transition_to_dispatch(int fd, int timeout_ms) {
    // Check that dispatcher is in WRITE (otherwise DISPATCH pulse is ignored).
    int rc = hft_disp_wait_state(fd, HFT_STATE_WRITE, timeout_ms, 1, false);
    if (rc) return rc;

    // Signal transition
    rc = ioctl(fd, HFT_IOC_DISP_BEGIN_DISPATCH);
    if (rc < 0) return -errno;

    // Wait until DISPATCH is observed.
    return hft_disp_wait_state(fd, HFT_STATE_DISPATCH, timeout_ms, /*poll_ms=*/1,
true);
}
```

```c
// Waits for the Dispatcher to transition to either DISPATCH or DONE
// Only needed when using the buttons for the demo.
int hft_disp_wait_dispatch_or_done(int fd, int timeout_ms, int poll_ms) {
    if (poll_ms <= 0) poll_ms = 1;
    uint64_t deadline = (timeout_ms < 0) ? 0 : (now_ms() + (uint64_t)timeout_ms);

    for (;;) {
        struct hft_disp_status st;
        int rc = hft_disp_get_status(fd, &st);
        if (rc) return rc;

        if (st.state == HFT_STATE_DISPATCH || st.state == HFT_STATE_DONE)
            return 0;

        if (st.state == HFT_STATE_IDLE)
            return HFT_ERR_RESET;

        if (timeout_ms >= 0 && now_ms() >= deadline)
            return -ETIMEDOUT;

        (void)sleep_ms((unsigned)poll_ms);
    }
}

// Signals and wait until Order Dispatcher has transitioned to IDLE
// This is once it has transitioned to DONE by itself
int hft_transition_done_to_idle(int fd, int timeout_ms) {
    int rc = hft_disp_wait_state(fd, HFT_STATE_DONE, timeout_ms, 1, false);
    if (rc) return rc;

    // Indicate to Order Dispatcher to transition to IDLE state
    rc = ioctl(fd, HFT_IOC_DISP_CLEAR_DONE);
    if (rc < 0) return -errno;

    return hft_disp_wait_state(fd, HFT_STATE_IDLE, timeout_ms, 1, false);
}

// Simple herlper fucntion to check if all FIFOs are full
static bool hft_all_fifos_full_mask(uint32_t full_mask) {
    return (full_mask & HFT_ALL_LANES_MASK) == HFT_ALL_LANES_MASK;
}

///////// Pushing Orders into FIFOs /////////
// Pushes an order into the specified lane (lane = fifo)
```

```c
int hft_disp_push_order(int fd, uint32_t lane, const struct hft_disp_order *o) {
    if (!o) return -EINVAL;

    struct hft_disp_push_req req;
    memset(&req, 0, sizeof(req));
    req.lane  = lane;
    req.order = *o;

    if (ioctl(fd, HFT_IOC_DISP_PUSH_ORDER, &req) < 0) return -errno;
    return 0;
}

// Push that waits for lane ready (or until timeout).
int hft_disp_push_order_blocking(int fd, uint32_t lane, const struct
hft_disp_order *o,
                            int timeout_ms, int poll_us) {
    if (poll_us <= 0) poll_us = 200; // 0.2ms default
    uint64_t deadline = (timeout_ms < 0) ? 0 : (now_ms() + (uint64_t)timeout_ms);

    for (;;) {
        int rc = hft_disp_push_order(fd, lane, o);
        if (rc == 0) return 0;

        // Typical transient errors from driver:
        //  -EAGAIN: not in WRITE state
        //  -EBUSY : lane not ready (FIFO full)
        if (rc != -EAGAIN && rc != -EBUSY) return rc;

        if (timeout_ms >= 0 && now_ms() >= deadline) return -ETIMEDOUT;
        (void)sleep_us((unsigned)poll_us);
    }
}

// Check if there is new order to push for specified lane
static int lane_csv_next_order(void *ctx, int lane, struct hft_disp_order *out){
    LaneCSVReader *r = (LaneCSVReader *)ctx;
    // Check lane reader is not empty
    if (!r || !out) return -EINVAL;
    // Check for null pointers
    if ((unsigned)lane >= HFT_NUM_LANES) return -EINVAL;
    // Check lane
    if (r->lane_eof[lane]) return 0;

    char line[256];
```

```
    // Loop until there is ususable data in a row
    for (;;) {
        // Read line in csv
        if (!fgets(line, sizeof(line), r->lane_csv[lane])) {
            r->lane_eof[lane] = true;
            return 0; // EOF
        }

        // Skip header or blank lines
        if (line[0] == '\n' || line[0] == '\r') continue;
        if (!strncmp(line, "Type,", 5)) continue;

        char type_s[16];
        unsigned price_u, qty_u;

        // Parse lines
        // Expected format: <BID/ASK>, <Price>, <Quantity>
        if (sscanf(line, " %15[^,],%u,%u", type_s, &price_u, &qty_u) != 3)
            return -EINVAL;

        // Map type string to bit value
        if (!strcmp(type_s, "ASK")) out->type = 0;
        else if (!strcmp(type_s, "BID")) out->type = 1;
        else return -EINVAL;

        // Range checks for price and quantity
        if (price_u > 0xFFFF) return -EINVAL;
        if (qty_u > 0x7FFF)  return -EINVAL;

        // Fill output struct
        out->price = (uint16_t)price_u;
        out->quantity = (uint16_t)qty_u;

        return 1; // successfuly produced an order
    }
}

// Callback: produce next order for a given lane.
// Return:
//   1  -> *out filled with next order
//   0  -> EOF/no-more-orders for that lane
//  <0  -> error
typedef int (*hft_next_order_fn)(void *ctx, int lane, struct hft_disp_order
*out);
```

```
// Writes round-robin to fifo0-fifo7 until:
//   - all FIFOs are full, OR
//   - every lane's source is exhausted, OR
//   - timeout hits (timeout applies to overall loop; use -1 for no timeout)
int hft_write_orders_round_robin(int fd, hft_next_order_fn next_order, void *ctx,
int timeout_ms){
    if (!next_order) return -EINVAL;

    uint64_t deadline = (timeout_ms < 0) ? 0 : (now_ms() + (uint64_t)timeout_ms);

    bool lane_done[HFT_NUM_LANES] = { false };

    for (;;) {
        // Stop if all lanes are done
        bool all_done = true;
        for (int i = 0; i < HFT_NUM_LANES; i++) all_done &= lane_done[i];
        if (all_done) return 0;

        struct hft_disp_status st;
        int rc = hft_disp_get_status(fd, &st);
        if (rc) return rc;

        // Stop if all FIFOs full
        if (hft_all_fifos_full_mask(st.full_mask)) return 0;

        // Check if board was reseted: dispatcher when back to IDLE
        if (st.state == HFT_STATE_IDLE) return HFT_ERR_RESET;

        // Check if Order Dispatcher in WRITE state
        if (st.state != HFT_STATE_WRITE)
            return -EAGAIN;

        for (int lane = 0; lane < HFT_NUM_LANES; lane++) {
            if (lane_done[lane]) continue;
            if (st.full_mask & (1u << lane)) continue;

            struct hft_disp_order o;
            int have = next_order(ctx, lane, &o);
            if (have < 0) return have;
            if (have == 0) { lane_done[lane] = true; continue; }

            rc = hft_disp_push_order_blocking(fd, (uint32_t)lane, &o,
                                              /*timeout_ms=*/2000,
                                              /*poll_us=*/200);
            if (rc) return rc;
```

```
            if (timeout_ms >= 0 && now_ms() >= deadline) return -ETIMEDOUT;

            // Refresh status to avoid pushing into lanes that just filled
            rc = hft_disp_get_status(fd, &st);
            if (rc) return rc;
            if (hft_all_fifos_full_mask(st.full_mask)) return 0;
        }
    }
}

/////////// Reading Trade Logs /////////
// Get Trade log data
int hft_log_get_info(int fd, struct hft_log_info *li) {
    if (!li) return -EINVAL;
    if (ioctl(fd, HFT_IOC_LOG_GET_INFO, li) < 0) return -errno;
    return 0;
}

// Reads up to `max_entries` into `entries`.
// On success, *out_count is filled with the number read.
int hft_log_read_all(int fd, struct hft_log_entry *entries,
                     uint32_t max_entries, uint32_t *out_count, uint32_t
*out_overflow) {
    if (!entries || !out_count) return -EINVAL;

    struct hft_log_info li;
    int rc = hft_log_get_info(fd, &li);
    if (rc) return rc;

    if (out_overflow) *out_overflow = li.overflow;

    uint32_t n = li.count;
    if (n > max_entries) n = max_entries;

    for (uint32_t i = 0; i < n; i++) {
        struct hft_log_entry le;
        memset(&le, 0, sizeof(le));
        le.index = i;

        if (ioctl(fd, HFT_IOC_LOG_READ_ENTRY, &le) < 0) return -errno;
        entries[i] = le;
    }

    *out_count = n;
```

```c
    return 0;
}

// Writes formatted output to both the given file stream and stdout, so probe
// output is visible on the terminal AND captured in the progress log.
static void tee_printf(FILE *fp, const char *fmt, ...) {
    va_list args;
    if (fp) {
        va_start(args, fmt);
        vfprintf(fp, fmt, args);
        va_end(args);
    }
    va_start(args, fmt);
    vfprintf(stdout, fmt, args);
    va_end(args);
}

// Prints specified mask for all lanes: fifo full, fifo empty, or engine idle
static void print_lane_mask(FILE *fp, const char *label, uint32_t mask) {
    tee_printf(fp, "%s:", label);
    for (int lane = 0; lane < HFT_NUM_LANES; lane++) {
        tee_printf(fp, " %d=%c", lane, (mask & (1u << lane)) ? '1' : '0');
    }
    tee_printf(fp, "\n");
}

// Prints the state of the Dispatcher
static const char *disp_state_str(uint32_t state) {
    switch (state) {
        case HFT_STATE_IDLE:     return "IDLE";
        case HFT_STATE_WRITE:    return "WRITE";
        case HFT_STATE_DISPATCH: return "DISPATCH";
        case HFT_STATE_DONE:     return "DONE";
        default:                 return "UNKNOWN";
    }
}

//
static void hft_print_progress_snapshot(FILE *fp, int fd) {
    struct hft_disp_status st;
    int rc_st = hft_disp_get_status(fd, &st);

    struct hft_log_info li;
    int rc_li = hft_log_get_info(fd, &li);
```

```
    if (rc_st == 0) {
        tee_printf(fp, "[progress] Dispatcher state=%s (%u)\n",
disp_state_str(st.state), st.state);
        print_lane_mask(fp, "[progress] FIFO empty_mask", st.empty_mask);
        print_lane_mask(fp, "[progress] FIFO full_mask ", st.full_mask);
        print_lane_mask(fp, "[progress] FIFO ready_mask", st.ready_mask);
    } else {
        tee_printf(fp, "[progress] Dispatcher status unavailable (rc=%d)\n",
rc_st);
    }

    if (rc_li == 0) {
        tee_printf(fp, "[progress] Trade log count=%u overflow=%u
trade_done=%u\n",
               li.count, li.overflow, li.trade_done);
        tee_printf(fp, "[progress] Engines: all_idle=%u idle_mask=0x%08x\n",
               li.all_engines_idle, li.engine_idle_mask);
    } else if (rc_li == -EAGAIN) {
        tee_printf(fp, "[progress] Trade logger info not ready yet (rc=%d)\n",
rc_li);
    } else {
        tee_printf(fp, "[progress] Trade logger info unavailable (rc=%d)\n",
rc_li);
    }
    fflush(stdout);
}

// Checks that HFT_SIM is done trading by
// - Checking Order Dispatcher is the DONE state
// - All Heap Engines are in the IDLE state
static int hft_log_wait_trade_done(FILE *progress_fp, int fd, int timeout_ms, int
poll_ms,
                                   struct hft_log_info *out_final){
    if (poll_ms <= 0) poll_ms = 1;

    uint64_t deadline = (timeout_ms < 0) ? 0 : (now_ms() + (uint64_t)timeout_ms);

    // Throttle printing so we don't spam the console.
    const uint64_t print_period_ms = 250;
    uint64_t next_print_ms = 0;

    for (;;) {
        // Handle logging being interrupted nicely
        if (g_stop) {
            tee_printf(progress_fp, "[progress] interrupted (SIGINT)\n");
```

```
            if (progress_fp) fflush(progress_fp);
            fflush(stdout);
            return -EINTR;
        }

        // Detect if board was reseted: Dispatcher back to IDLE
        struct hft_disp_status st;
        int rc_st = hft_disp_get_status(fd, &st);
        if (rc_st) return rc_st;
        if (st.state == HFT_STATE_IDLE) return HFT_ERR_RESET;

        // Get trade log info
        struct hft_log_info li = {0};
        int rc = hft_log_get_info(fd, &li);

        uint64_t t = now_ms();
        if (t >= next_print_ms) {
            if (progress_fp) {
                hft_print_progress_snapshot(progress_fp, fd);
                fflush(progress_fp);
            }
            next_print_ms = t + print_period_ms;
        }

        if (rc == 0) {
            if (li.trade_done) {
                if (out_final) *out_final = li;
                return 0;
            }
        } else if (rc != -EAGAIN) {
            // Real error
            return rc;
        }

        if (timeout_ms >= 0 && now_ms() >= deadline)
            return -ETIMEDOUT;

        (void)sleep_ms((unsigned)poll_ms);
    }
}

// Prints each trade log
// The following is assumed about the data:
// [63:56] engine_id (only low 3 bits used)
// [55:48] quantity  (only low 7 bits used)
```

```
// [47:32] price     (unsigned int, fixed-point with 2 decimal digits)
// [31:0]  timestamp (unsigned int)
static void hft_log_dump_entries(FILE *fp, const struct hft_log_entry *e,
uint32_t n){
    for (uint32_t i = 0; i < n; i++) {
        uint32_t ts = e[i].word0;
        uint32_t w1 = e[i].word1;

        uint32_t engine_id   = (w1 >> 24) & 0x7u;     // only need low 3 bits
        uint32_t quantity    = (w1 >> 16) & 0x7Fu;    // only need low 7 bits
        uint32_t price_cents = (w1 >> 0)  & 0xFFFFu;  // 16-bit price

        // Convert cents to string with 2 digits after decimal.
        // Example: 12345 -> "123.45"
        uint32_t dollars = price_cents / 100u;
        uint32_t cents   = price_cents % 100u;

        fprintf(fp, "%u,0x%08x,0x%08x,%u,%u,%u.%02u,%u\n",
                i, e[i].word0, e[i].word1,
                engine_id, quantity, dollars, cents, ts);
    }
}

////////////////////////////////////////////////////////////////////////
// Main Function - Actual harness
// Lanes/FIFO corresponding to each symbol (from symbol_engine.sv):
// 0 - AAPL (aka APL)
// 1 - BSX
// 2 - BUS
// 3 - MMM
// 4 - MSFT (aka SFT)
// 5 - SBUX (aks BUX)
// 6 - TUS
// 7 - WMT
////////////////////////////////////////////////////////////////////////
int main(){
    // Install Ctrl+C handler to flush/close logs cleanly.
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = on_sigint;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    (void)sigaction(SIGINT, &sa, NULL);

    FILE *progress_fp = fopen("hft_progress.txt", "w");
```

```
    if (!progress_fp) {
        fprintf(stderr, "WARNING: could not open hft_progress.txt for writing:
%s\n", strerror(errno));
    }
    // Change if data dir changes
    const char *data_dir = "../data";

    // Check for device drivers
    static const char filename[] = "/dev/hft_sim";
    if ( (hft_sim_fd = open(filename, O_RDWR)) == -1) {
        fprintf(stderr, "Error: could not open %s\n", filename);
        return -1;
    }
    printf("Starting HFT Simulator Harness.\n");

    // Get CSV files
    LaneCSVReader csv = {0};
    int rc = open_all_csvs(&csv, data_dir); // Error messages already handled
    if (rc) return 1;

    for (;;) {
        // On CTL+C
        if (g_stop) break;

        // Rewind CSV
        csv_rewind_all(&csv);

        // Clear counter and overflow of Trade logs
        if (ioctl(hft_sim_fd, HFT_IOC_LOG_CLEAR) < 0) return -errno;

        // Transition Order Dispatcher to WRITE
        //if (ioctl(hft_sim_fd, HFT_IOC_DISP_BEGIN_WRITE) < 0) return -errno;
        rc = hft_disp_wait_state(hft_sim_fd, HFT_STATE_WRITE, -1, 10, false); //
No timeout
        if (rc == HFT_ERR_RESET) { printf("Reset detected. Restarting.\n");
continue; }
        if (rc) { fprintf(stderr, "Error: wait for WRITE failed rc=%d\n", rc);
return 1; }

        // Fill FIFOs round-robin from CSVs
        printf("Writing Data to Order Dispatcher.\n");
        rc = hft_write_orders_round_robin(hft_sim_fd, lane_csv_next_order, &csv,
30000);
        if (rc == HFT_ERR_RESET) { printf("Reset detected during write.
Restarting.\n"); continue; }
```

```
        if (rc) { fprintf(stderr, "ERROR: write_orders_round_robin failed: %d\n",
rc); return 1;}

        // Debug: Check state before start dispatching
        struct hft_disp_status st = {0};
        rc = hft_disp_get_status(hft_sim_fd, &st);
        if (rc) {
            fprintf(stderr, "Error: get status before dispatch failed rc=%d\n",
rc);
            close_all_csvs(&csv);
            if (progress_fp) fclose(progress_fp);
            if (hft_sim_fd >= 0) close(hft_sim_fd);
            return 1;
        }
        fprintf(stderr, "Before DISPATCH: state=%u ready=0x%02x empty=0x%02x
full=0x%02x\n",
                st.state, st.ready_mask, st.empty_mask, st.full_mask);

        // Transition Order Dispatcher to DISPATCH
        // Or for IDLE (possible since we are using buttons. )
        rc = hft_disp_wait_dispatch_or_done(hft_sim_fd, -1, 1);
        if (rc == HFT_ERR_RESET) { printf("Reset detected before dispatch.
Restarting.\n"); continue; }
        if (rc) { fprintf(stderr, "Error: wait for DISPATCH/DONE failed rc=%d\n",
rc); return 1; }

        rc = hft_disp_get_status(hft_sim_fd, &st);
        if (rc) {
            fprintf(stderr, "Error: get status after dispatch wait failed
rc=%d\n", rc);
            close_all_csvs(&csv);
            if (progress_fp) fclose(progress_fp);
            if (hft_sim_fd >= 0) close(hft_sim_fd);
            return 1;
        }
        if (st.state == HFT_STATE_DISPATCH)
            printf("Virtualized HFT Simulator has started.\n");
        else
            printf("Dispatcher already reached DONE before SW observed
DISPATCH.\n");

        // Wait until trading is completely finished
        struct hft_log_info final_li = {0};
        rc = hft_log_wait_trade_done(progress_fp, hft_sim_fd, 600000, 2,
&final_li);
```

```
        if (rc == HFT_ERR_RESET) { printf("Reset detected during dispatch.
Restarting.\n"); continue; }
        if (rc == -EINTR) { fprintf(stderr, "Interrupted (Ctrl+C). Exiting
cleanly.\n"); close_all_csvs(&csv); return 0; }
        if (rc) { fprintf(stderr, "ERROR: timed out / failed waiting for
trade_done: %d\n", rc); close_all_csvs(&csv); return 1; }

        // Read all trade log entries
        printf("Trading done.\n");
        printf("Trade log count (trades recorded): %u\n", final_li.count);
        printf("Trade log overflow: %u\n", final_li.overflow);
        uint32_t n = final_li.count;
        if (n > 8704) n = 8704; // safety (matches RTL depth)
        if (n == 0) {
            printf("No trades recorded.\n");
            close_all_csvs(&csv);
            return 0;
        }

        struct hft_log_entry *logbuf = calloc(n, sizeof(*logbuf));
        if (!logbuf) {
            fprintf(stderr, "ERROR: calloc failed\n");
            close_all_csvs(&csv);
            return 1;
        }

        uint32_t got = 0, overflow = 0;
        rc = hft_log_read_all(hft_sim_fd, logbuf, n, &got, &overflow);
        if (rc) {
            fprintf(stderr, "ERROR: hft_log_read_all failed: %d\n", rc);
            free(logbuf);
            close_all_csvs(&csv);
            return 1;
        }

        // Dump to a text file
        printf("Read back %u trade log entries (overflow=%u)\n", got, overflow);

        FILE *trades_fp = fopen("hft_trades.txt", "w");
        if (!trades_fp) {
            fprintf(stderr, "ERROR: could not open hft_trades.txt for writing:
%s\n", strerror(errno));
            free(logbuf);
            close_all_csvs(&csv);
            return 1;
```

```
        }

        fprintf(trades_fp,
"index,word0,word1,engine_id,quantity,price,timestamp\n");
        hft_log_dump_entries(trades_fp, logbuf, got);

        fclose(trades_fp);

        free(logbuf);

        // For now, run once.
        break;
    }

    // Clean up
    close_all_csvs(&csv);
    if (progress_fp) fclose(progress_fp);
    if (hft_sim_fd >= 0) close(hft_sim_fd);

    return 0;
}
```

C/C++

```
#ifndef _HEAP_H_
#define _HEAP_H_

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define DEFAULT_MODE 1
#define DEFAULT_ORDERS 100
#define MAX_PRICE 100
#define MAX_AMOUNT 10
#define MAX_ORDERS 1500000
#define TRIM_TIME 0xFFFFFFFFFFFF

struct MemoryManager;
struct OrderBook;
struct SimStats;

typedef void *(*HeapCmpFunc)(const void *a, const void *b);
```

```
void free_node_memory(struct MemoryManager *mm, struct OrderBook *ob, struct
SimStats *stats);

typedef struct {
	short price;
	uint32_t timestamp;
	short amount;
	int type; //1 = Ask 0 = Bid - This will be 1 bit in hardware
	char symbol[4]; //3 chars + null terminator
} Order;

typedef struct {
	void **data;
	int size;
	int capacity;
	HeapCmpFunc cmp;
	struct MemoryManager *mm;
	struct OrderBook *ob;
	struct SimStats *stats;
} Heap;

Order *create_order(int price, int amount, int type, const char *symbol) {
	Order *o = malloc(sizeof(Order));
	o->price = price;
	struct timespec ts;
	clock_gettime(CLOCK_MONOTONIC, &ts);
	o->timestamp = (uint32_t) ts.tv_nsec;
	o->amount = amount;
	o->type = type;
	strncpy(o->symbol, symbol, 3);
	o->symbol[3] = '\0';
	return o;
}

Heap *create_heap(int capacity, HeapCmpFunc cmp) {
	Heap *h = malloc(sizeof(Heap));
	h->data = malloc(sizeof(void *) * capacity);
	h->size = 0;
	h->capacity = capacity;
	h->cmp = cmp;
	h->mm = NULL;
	h->ob = NULL;
	h->stats = NULL;
	return h;
```

```
}

static int rand_range(int min, int max) {
	return min + rand() % (max - min + 1);
}

static void swap(void **a, void **b) {
	void *tmp = *a;
		*a = *b;
		*b = tmp;
}

static void sift_up(Heap *h, int i) {
	while (i > 0) {
		int parent = (i - 1) / 2;
		if (h->cmp(h->data[i], h->data[parent]) == h->data[i]) {
			swap(&h->data[i], &h->data[parent]);
			i = parent;
		} else {
			break;
		}
	}
}

static int is_expired(const Order *o, uint32_t now) {
		return (now - o->timestamp) > TRIM_TIME;
}

static void sift_down(Heap *h, int i) {
	struct timespec ts;
	clock_gettime(CLOCK_MONOTONIC, &ts);
	uint32_t now = (uint32_t)ts.tv_nsec;

	while (i < h->size) {

		int best = i;
		int left = 2 * i + 1;
		int right = 2 * i + 2;

		if (left < h->size && is_expired((Order *)h->data[left], now)) {
			h->data[left] = h->data[--h->size];
			if(h->mm) {
				free_node_memory(h->mm, h->ob, h->stats);
			}
			continue;
```

```
            }
            if (right < h->size && is_expired((Order *)h->data[right], now)) {
                  h->data[right] = h->data[--h->size];
                  if(h->mm) {
                        free_node_memory(h->mm, h->ob, h->stats);
                  }
                  continue;
            }

            if(left < h->size && h->cmp(h->data[left], h->data[best]) ==
h->data[left]) {
                  best = left;
            }
            if(right < h->size && h->cmp(h->data[right], h->data[best]) ==
h->data[right]) {
                  best = right;
            }
            if(best != i) {
                  swap(&h->data[i], &h->data[best]);
                  i = best;
            } else {
                  break;
            }
      }
}

void update(Heap *h, int new_amount) {
      ((Order *)h->data[0])->amount = new_amount;
}

void push(Heap *h, void *val) {
      if(h->size == h->capacity) {
            h->capacity *= 2;
            h->data = realloc(h->data, sizeof(void *) * h->capacity);
      }
      h->data[h->size] = val;
      sift_up(h, h->size);
      h->size++;
}

void *pop(Heap *h) {
      void *top = h->data[0];
      h->data[0] = h->data[--h->size];
      sift_down(h, 0);
      return top;
```

```
}

void *peek(Heap *h) {
	return h->data[0];
}

void free_order(Order *o) {
	free(o);
}

void free_heap(Heap *h) {
	free(h->data);
	free(h);
}

void *min_cmp(const void *a, const void *b) {
	if(((Order *)a)->price < ((Order *)b)->price) {
		return (void *)a;
	} else if(((Order *)a)->price == ((Order *)b)->price) {
		if(((Order *)a)->timestamp < ((Order *)b)->timestamp) {
			return (void *)a;
		} else if(((Order *)a)->timestamp == ((Order *)b)->timestamp) {
			if(((Order *)a)->amount > ((Order *)b)->amount) {
				return (void *)a;
			} else {
				return (void *)b;
			}
		} else {
			return (void *)b;
		}
	} else {
		return (void *)b;
	}
}

void *max_cmp(const void *a, const void *b) {
	if(((Order *)a)->price > ((Order *)b)->price) {
		return (void *)a;
	} else if(((Order *)a)->price == ((Order *)b)->price) {
		if(((Order *)a)->timestamp < ((Order *)b)->timestamp) {
			return (void *)a;
		} else if(((Order *)a)->timestamp == ((Order *)b)->timestamp) {
			if(((Order *)a)->amount > ((Order *)b)->amount) {
				return (void *)a;
			} else {
```

```
                        return (void *)b;
                }
        } else {
                return (void *)b;
        }
    } else {
        return (void *)b;
    }
}

static int check_for_trade(Heap *asks, Heap *bids) {

    if(asks->size == 0 || bids->size == 0) {
        return 0;
    }

    Order *bid = (Order *)peek(bids);
    Order *ask = (Order *)peek(asks);

    if(bid->price >= ask->price) {
        if(bid->amount == ask->amount) {
            struct timespec ts;
            clock_gettime(CLOCK_MONOTONIC, &ts);
            printf("A trade has been executed at %d! Sold %d shares of %s
at $%d.\n",
                        (uint32_t) ts.tv_nsec,
                        ask->amount,
                        ask->symbol,
                        bid->price
            );
            pop(bids);
            pop(asks);
            return 1;
        } else {
            if(bid->amount > ask->amount) {
                struct timespec ts;
                clock_gettime(CLOCK_MONOTONIC, &ts);
                printf("A partial fill has been executed at %d! Sold
%d shares of %s at $%d. A bid for %d shares remains.\n",
                                (uint32_t) ts.tv_nsec,
                                ask->amount,
                                ask->symbol,
                                    bid->price,
                                bid->amount - ask->amount
                );
```

```
                        update(bids, bid->amount - ask->amount);
                        pop(asks);
                        return 1;
                } else {
                        struct timespec ts;
                        clock_gettime(CLOCK_MONOTONIC, &ts);
                        printf("A partial fill has been executed at %d! Sold
%d shares of %s at $%d. A ask of %d shares remains.\n",
                                        (uint32_t) ts.tv_nsec,
                                        bid->amount,
                                        ask->symbol,
                                                bid->price,
                                        ask->amount - bid->amount
                        );

                        update(asks, ask->amount - bid->amount);
                        pop(bids);
                        return 1;
                }
        }
        } else {
                return 0;
        }
}


#endif
```

C/C++

```
#include <stdio.h>
#include <time.h>
#include <string.h>
#include "mmu.h"

void free_node_memory(struct MemoryManager *mm, struct OrderBook *ob, struct
SimStats *stats) {
        if (ob->mem.nodes_in_curr_page > 0) {
                ob->mem.nodes_in_curr_page--;
                if (ob->mem.page_count > 0) {
                        int frame = translate(ob, ob->mem.page_count - 1, stats);
                        if (frame >= 0) {
                                mm->frames[frame].node_count--;
```

```
                        if (mm->frames[frame].node_count == 0) {
                                free_frame(mm, ob, ob->mem.page_count - 1,
stats);
                                ob->mem.page_count--;
                                if (ob->mem.page_count > 0) {
                                        ob->mem.nodes_in_curr_page = PAGE_SIZE;
                                } else {
                                        ob->mem.nodes_in_curr_page = 0;
                                }
                        }
                }
        }
}

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, mm, &stats);
                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(order_list[i]->type) {
                        printf("Ask Submitted for %s; Price: $%d, Amount %d,
Timestamp: %d\n",
                                        symbol,
                                order_list[i]->price,
```

```
                                order_list[i]->amount,
                                order_list[i]->timestamp);
                } else {
                        printf("Bid Submitted for %s; Price: $%d, Amount %d,
Timestamp: %d\n",
                                        symbol,
                                order_list[i]->price,
                                order_list[i]->amount,
                                order_list[i]->timestamp);
                }

                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);
                        }
                }
        }

        print_sim_stats(ex, mm, &stats);

        for(int i = 0; i < orders; i++) {
                free_order(order_list[i]);
        }
        free(order_list);
        free_exchange(ex);
        free(mm);
}

static void manual_execution() {
        Exchange *ex = create_exchange(16);
        MemoryManager *mm = create_memory_manager();
        SimStats stats = {0};

        int orders = 0;
        char order;
        int price;
        int amount;
        char type[16];
        char symbol[8];

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

        while (1) {
                if (orders >= MAX_ORDERS) {
```

```
                    printf("Maximum number of orders received, ending
simulation\n");
                    break;
             }

             printf("Would you like to submit an order? (Y/N)\n");
             scanf(" %c", &order);
             while (getchar() != '\n');
             if (order == 'Y' || order == 'y') {
                    printf("Format: (ask/bid) (symbol) (price) (amount)\n");
                    scanf("%s %s %d %d", type, symbol, &price, &amount);
                    while (getchar() != '\n');
                    symbol[3] = '\0';
                    int is_ask = -1;

                    if (strcmp(type, "Ask") == 0 || strcmp(type, "ask") == 0) {
                           is_ask = 1;
                    } else if (strcmp(type, "Bid") == 0 || strcmp(type, "bid") ==
0) {
                           is_ask = 0;
                    }

                    if (is_ask < 0) {
                           printf("Please specify ask or bid\n");
                           continue;
                    }

                    order_list[orders] = create_order(price, amount, is_ask,
symbol);
                    OrderBook *ob = find_or_create_book(ex, symbol, mm, &stats);

                    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 (is_ask) {
                           printf("Ask Submitted for %s; Price: $%d, Amount %d,
Timestamp: %d\n",
                                  symbol, price, amount,
order_list[orders]->timestamp);
                    } else {
```

```
                        printf("Bid Submitted for %s; Price: $%d, Amount %d,
Timestamp: %d\n",
                                symbol, price, amount,
order_list[orders]->timestamp);
                }

                if (mem_aware_insert(ob, mm, order_list[orders], sym_id,
&stats)) {
                        while (check_for_trade_multi(ob, &stats)) {
                                ob->trades++;
                                post_trade_cleanup(mm, ob, &stats);
                        }
                }
                orders++;

        } else if (order == 'N' || order == 'n') {
                printf("Ending simulation\n");
                break;
        } else {
                printf("Please submit Y or N\n");
        }
    }

    print_sim_stats(ex, mm, &stats);

    for (int i = 0; i < orders; i++) {
        free_order(order_list[i]);
    }

    free(order_list);
    free_exchange(ex);
    free(mm);
}

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;
    }
```

```
	int orders = 0;
	Order **order_list = malloc(sizeof(Order *) * MAX_ORDERS);
	char line[256];
	char type[16];
	char symbol[8];
	int price, amount;

	fgets(line, sizeof(line), fp);
	if (strstr(line, "ask") == NULL && strstr(line, "bid") == NULL &&
	    strstr(line, "Ask") == NULL && strstr(line, "Bid") == NULL) {
	} else {
		rewind(fp);
	}

	while (fgets(line, sizeof(line), fp) && orders < MAX_ORDERS) {
		if (sscanf(line, "%[^,],%[^,],%d,%d", type, symbol, &price,
&amount) != 4) {
			printf("Skipping malformed line: %s", line);
			continue;
		}

		symbol[3] = '\0';

		int is_ask = -1;
		if (strcmp(type, "Ask") == 0 || strcmp(type, "ASK") == 0)
			is_ask = 1;
		else if (strcmp(type, "Bid") == 0 || strcmp(type, "BID") == 0)
			is_ask = 0;

		if (is_ask < 0) {
			printf("Skipping invalid type: %s\n", type);
			continue;
		}

		order_list[orders] = create_order(price, amount, is_ask, symbol);
		OrderBook *ob = find_or_create_book(ex, symbol, mm, &stats);

		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 (is_ask)
                  printf("Ask Submitted for %s; Price: $%d, Amount %d,
Timestamp: %d\n",
                        symbol, price, amount, order_list[orders]->timestamp);
            else
                  printf("Bid Submitted for %s; Price: $%d, Amount %d,
Timestamp: %d\n",
                        symbol, price, amount, order_list[orders]->timestamp);

            if (mem_aware_insert(ob, mm, order_list[orders], sym_id, &stats)) {
                  while (check_for_trade_multi(ob, &stats)) {
                        ob->trades++;
                        post_trade_cleanup(mm, ob, &stats);
                  }
            }
            orders++;
      }

      fclose(fp);
      print_sim_stats(ex, mm, &stats);

      for (int i = 0; i < orders; i++)
            free_order(order_list[i]);
      free(order_list);
      free_exchange(ex);
      free(mm);

}

int main(int argc, char *argv[]) {
      int mode = DEFAULT_MODE;
       int orders = DEFAULT_ORDERS;
      for(int i = 1; i < argc; i++) {
            if(strcmp(argv[i], "--mode") == 0) {
                  mode = atoi(argv[++i]);
            } else if(strcmp(argv[i], "--orders") == 0) {
                  orders = atoi(argv[++i]);
            }
      }

      if(mode != 1 && mode != 0 && mode != 2) {
            printf("Usage: mode must be either 1 (automatic) or 0 (manual)
\n");
            return 1;
      }
```

```
	if(orders < 2) {
		printf("Usage: At least 2 orders must be submitted \n");
		return 1;
	}

	if(mode == 1) {
		automatic_execution(orders);
	} else if(mode == 0) {
		manual_execution();
	} else if(mode == 2) {
		if(argc > 2) {
			csv_execution(argv[argc - 1]);
		} else {
			printf("Usage: supply a .csv file as the last arguement\n");
			return 1;
		}
	}

	return 0;
}
```

C/C++

```
#ifndef _MMU_H_
#define _MMU_H_

#include "heap.h"
#define PAGE_SIZE 64
#define MAX_PAGES 256
#define MAX_PAGES_PER_SYMBOL 128
#define BITMAP_WORDS ((MAX_PAGES + 63) / 64)
#define OVERFLOW_PAGE_SIZE 8
#define PAGE_WALK_CYCLES 3
#define OVERFLOW_PENALTY_CYCLES 2

typedef struct {
	int frame_index;
	int owner_symbol;
	int node_count;
} PhysicalFrame;

typedef struct {
	int data[OVERFLOW_PAGE_SIZE];
```

```
	int cnt;
} OverflowPage;

typedef struct MemoryManager {
	PhysicalFrame frames[MAX_PAGES];
	uint64_t free_bitmap[BITMAP_WORDS]; //change type to best represent max
number of pages
	int hard_rejects;
	OverflowPage overflow;
} MemoryManager;

typedef struct {
	int virtual_pages[MAX_PAGES_PER_SYMBOL];
	int page_count;
	int nodes_in_curr_page;
} SymbolMemory;

typedef struct OrderBook {
	char symbol[8];
	Heap *asks;
	Heap *bids;
	int trades;
	SymbolMemory mem;
	int trade_in_progress;
	int insert_pending;
} OrderBook;

typedef struct {
	OrderBook **books;
	int cnt;
	int cap;
} Exchange;

typedef struct SimStats {
	int total_trades;
	int total_reads;
	int total_writes;
	int overflow_accesses;
	int hard_rejects;
	int hazards;
	int overflow_max;
	long long total_cycles;
} SimStats;

MemoryManager *create_memory_manager(void) {
```

```
	MemoryManager *mm = malloc(sizeof(MemoryManager));

	for (int i = 0; i < MAX_PAGES; i++) {
		mm->frames[i].frame_index = i;
		mm->frames[i].owner_symbol = -1;
		mm->frames[i].node_count = 0;
	}

	for(int i = 0; i < BITMAP_WORDS; i++) {
		mm->free_bitmap[i] = ~0ULL;
	}

	mm->hard_rejects = 0;
	mm->overflow.cnt = 0;

	return mm;
}

int translate(OrderBook *ob, int page_index, SimStats *stats) {
	stats->total_cycles += PAGE_WALK_CYCLES;
	if(page_index < ob->mem.page_count) {
		return ob->mem.virtual_pages[page_index];
	}
	return -1;
}

int allocate_frame(MemoryManager *mm, int symbol_id) {
	for (int i = 0; i < MAX_PAGES; i++) {
		int word = i/64;
		int bit  = i%64;
		if (mm->free_bitmap[word] & (1ULL << bit)) {
			mm->free_bitmap[word] &= ~(1ULL << bit);
			mm->frames[i].owner_symbol = symbol_id;
			mm->frames[i].node_count = 0;
			return i;
		}
	}
	return -1;
}

void free_frame(MemoryManager *mm, OrderBook *ob, int page_index, SimStats
*stats) {
	int frame = translate(ob, page_index, stats);
	if (frame < 0) {
		return;
```

```
        }

        int word = frame/64;
        int bit  = frame%64;
        mm->free_bitmap[word] |= (1ULL << bit);
        mm->frames[frame].owner_symbol = -1;
        mm->frames[frame].node_count = 0;
}

Exchange *create_exchange(int cap) {
        Exchange *ex = malloc(sizeof(Exchange));
        ex->books = malloc(sizeof(OrderBook *) * cap);
        ex->cnt = 0;
        ex->cap = cap;
        return ex;
}

OrderBook *create_order_book(const char *symbol, MemoryManager *mm, SimStats
*stats) {
        OrderBook *ob = malloc(sizeof(OrderBook));
        strncpy(ob->symbol, symbol, 3);
        ob->symbol[3] = '\0';
        ob->asks = create_heap(10, min_cmp);
        ob->bids = create_heap(10, max_cmp);
        ob->asks->mm = mm;
        ob->asks->ob = ob;
        ob->asks->stats = stats;
        ob->bids->mm = mm;
        ob->bids->ob = ob;
        ob->bids->stats = stats;
        ob->trades = 0;
        ob->trade_in_progress = 0;
        ob->insert_pending = 0;
        ob->mem.page_count = 0;
        ob->mem.nodes_in_curr_page = 0;
        return ob;
}

OrderBook *find_or_create_book(Exchange *ex, const char *symbol, MemoryManager
*mm, SimStats *stats) {
        for (int i = 0; i < ex->cnt; i++) {
                if (strcmp(ex->books[i]->symbol, symbol) == 0) {
                        return ex->books[i];
                }
        }
```

```
	if (ex->cnt >= ex->cap) {
		ex->cap *= 2;
		ex->books = realloc(ex->books, sizeof(OrderBook *) * ex->cap);
	}

	OrderBook *ob = create_order_book(symbol, mm, stats);
	ex->books[ex->cnt++] = ob;
	return ob;
}

void free_exchange(Exchange *ex) {
	for (int i = 0; i < ex->cnt; i++) {
		free_heap(ex->books[i]->asks);
		free_heap(ex->books[i]->bids);
		free(ex->books[i]);
	}
	free(ex->books);
	free(ex);
}

int mem_aware_insert(OrderBook *ob, MemoryManager *mm, Order *o, int sym_id,
SimStats *stats) {
	if (ob->mem.page_count > 0 && ob->mem.nodes_in_curr_page < PAGE_SIZE) {
//room in curr page
		int frame = translate(ob, ob->mem.page_count-1, stats);
		if (frame < 0) {
			return 0;
		}

		ob->mem.nodes_in_curr_page++;
		mm->frames[frame].node_count++;
		stats->total_writes++;

		if (o->type) {
			push(ob->asks, o);
		} else {
			push(ob->bids, o);
		}

		return 1;
	}


	if (ob->mem.page_count < MAX_PAGES_PER_SYMBOL) { //new page
```

```
        int frame = allocate_frame(mm, sym_id);

        if (frame >= 0) {
                ob->mem.virtual_pages[ob->mem.page_count++] = frame;
                ob->mem.nodes_in_curr_page = 1;
                mm->frames[frame].node_count = 1;
                stats->total_cycles += PAGE_WALK_CYCLES;
                stats->total_writes++;

                if (o->type) {
                        push(ob->asks, o);
                } else {
                        push(ob->bids, o);
                }

                return 1;
        }
}

if (mm->overflow.cnt < OVERFLOW_PAGE_SIZE) { //use overflow page
        mm->overflow.cnt++;
        stats->total_cycles += PAGE_WALK_CYCLES + OVERFLOW_PENALTY_CYCLES;
        stats->overflow_accesses++;
        stats->total_writes++;

        printf("[OVERFLOW] %s using shared overflow (%d/%d) +%d cycles\n",
                        ob->symbol,
                                mm->overflow.cnt,
                                OVERFLOW_PAGE_SIZE,
                        PAGE_WALK_CYCLES + OVERFLOW_PENALTY_CYCLES);

        if (o->type) {
                push(ob->asks, o);
        } else {
                push(ob->bids, o);
        }

        if(mm->overflow.cnt > stats->overflow_max) {
                stats->overflow_max = mm->overflow.cnt;
        }

        return 1;
}

//hard reject
```

```
	mm->hard_rejects++;
	stats->total_cycles += 1;

	printf("[HARD REJECT] %s order rejected: price=%d amount=%d\n",
			ob->symbol, o->price, o->amount);
	return 0;
}

int check_for_trade_multi(OrderBook *ob, SimStats *stats) {
	FILE *fptr = fopen("results.txt", "a");

	if(fptr == NULL) {
		perror("File error");
		exit(1);
	}

	if (ob->asks->size == 0 || ob->bids->size == 0) {
		return 0;
	}

	int ask_frame = translate(ob, 0, stats);
	int bid_frame = translate(ob, 0, stats);
	(void)ask_frame;
	(void)bid_frame;

	Order *bid = (Order *) peek(ob->bids);
	Order *ask = (Order *) peek(ob->asks);

	stats->total_reads += 2;

	if (bid->price >= ask->price) {
		if (bid->amount == ask->amount) {
			struct timespec ts;
			clock_gettime(CLOCK_MONOTONIC, &ts);
			printf("A trade has been executed at %d! Sold %d shares of %s
at $%d.\n",
				(uint32_t) ts.tv_nsec,
				bid->amount,
					ob->symbol,
					bid->price);

			fprintf(fptr, "A trade has been executed at %d! Sold %d
shares of %s at $%d.\n",
				(uint32_t) ts.tv_nsec,
				bid->amount,
```

```
                                ob->symbol,
                                bid->price);
                        pop(ob->bids);
                        pop(ob->asks);
                        stats->total_trades++;
                        fclose(fptr);
                        return 1;
                } else if (bid->amount > ask->amount) {
                        struct timespec ts;
                        clock_gettime(CLOCK_MONOTONIC, &ts);
                        printf("A partial fill has been executed at %d! Sold %d
shares of %s at $%d. A bid for %d shares remains.\n",
                                (uint32_t) ts.tv_nsec,
                                ask->amount,
                                        ob->symbol,
                                        bid->price,
                                bid->amount - ask->amount);
                        fprintf(fptr, "A partial fill has been executed at %d! Sold
%d shares of %s at $%d. A bid for %d shares remains.\n",
                                (uint32_t) ts.tv_nsec,
                                ask->amount,
                                        ob->symbol,
                                        bid->price,
                                bid->amount - ask->amount);
                        update(ob->bids, bid->amount - ask->amount);
                        pop(ob->asks);
                        stats->total_trades++;
                        stats->total_writes++;
                        fclose(fptr);
                        return 1;
                } else {
                        struct timespec ts;
                        clock_gettime(CLOCK_MONOTONIC, &ts);
                        printf("A partial fill has been executed at %d! Sold %d
shares of %s at $%d. A ask of %d shares remains.\n",
                                (uint32_t) ts.tv_nsec,
                                bid->amount,
                                        ob->symbol,
                                        bid->price,
                                ask->amount - bid->amount);

                        fprintf(fptr, "A partial fill has been executed at %d! Sold
%d shares of %s at $%d. A ask of %d shares remains.\n",
                                (uint32_t) ts.tv_nsec,
                                bid->amount,
```

```
                                ob->symbol,
                                bid->price,
                        ask->amount - bid->amount);
                update(ob->asks, ask->amount - bid->amount);
                pop(ob->bids);
                stats->total_trades++;
                stats->total_writes++;
                fclose(fptr);
                return 1;
            }
        } else {
            fclose(fptr);
            return 0;
        }
}

void post_trade_cleanup(MemoryManager *mm, OrderBook *ob, SimStats *stats) {
        if (mm->overflow.cnt > 0) {
            mm->overflow.cnt--;
        }

        if (ob->mem.nodes_in_curr_page > 0) {
            ob->mem.nodes_in_curr_page--;
            if (ob->mem.page_count > 0) {
                int frame = translate(ob, ob->mem.page_count - 1, stats);

                if (frame >= 0) {
                    mm->frames[frame].node_count--;

                    if(mm->frames[frame].node_count == 0) {
                        free_frame(mm, ob, ob->mem.page_count-1,
stats);

                        ob->mem.page_count--;
                        if(ob->mem.page_count > 0) {
                            ob->mem.nodes_in_curr_page = PAGE_SIZE;
                        } else {
                            ob->mem.nodes_in_curr_page = 0;
                        }
                    }
                }
            }
        }
}
```

```
void print_sim_stats(Exchange *ex, MemoryManager *mm, SimStats *stats) {
	printf("\nPer-Symbol Breakdown:\n");
	for (int i = 0; i < ex->cnt; i++) {
		OrderBook *ob = ex->books[i];
		printf("%s: %d trades, %d open asks, %d open bids, %d pages
used\n",
			ob->symbol,
				ob->trades,
				ob->asks->size,
				ob->bids->size,
			ob->mem.page_count);
	}

	printf("\nOverflow:\n");
	printf("Accesses:       %d\n", stats->overflow_accesses);
	printf("Current usage:  %d/%d\n", mm->overflow.cnt, OVERFLOW_PAGE_SIZE);
	printf("Maximum usage: %d/%d\n", stats->overflow_max, OVERFLOW_PAGE_SIZE);
	printf("Penalty cycles: %d\n", stats->overflow_accesses *
(PAGE_WALK_CYCLES + OVERFLOW_PENALTY_CYCLES));

	printf("\nTotals:\n");
	printf("Hard rejects: %d\n", mm->hard_rejects);
	printf("Trades:       %d\n", stats->total_trades);
	printf("Reads:        %d\n", stats->total_reads);
	printf("Writes:       %d\n", stats->total_writes);
	printf("Cycles:       %lld\n", stats->total_cycles);
}

#endif
```

C/C++

```
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include "heap.h"

void free_node_memory(struct MemoryManager *mm, struct OrderBook *ob, struct
SimStats *stats) {
	(void)mm; (void)ob; (void)stats;
}
```

```
static void automatic_execution(int orders) {
	Heap *asks = create_heap(10, min_cmp);
	Heap *bids = create_heap(10, max_cmp);

	int trades = 0;

	Order **order_list = malloc(sizeof(Order *) * orders);
	for(int i = 0; i < orders; i++) {
		order_list[i] = create_order(rand_range(1, MAX_PRICE),
rand_range(1, MAX_AMOUNT), rand_range(0, 1), "AAA");
		if(order_list[i]->type) {
			push(asks, order_list[i]);
			printf("Ask Submitted; Price: $%d, Amount: %d, Timestamp:
%d\n", order_list[i]->price, order_list[i]->amount, order_list[i]->timestamp);
			if(check_for_trade(asks, bids)) {
				trades++;
			}
		} else {
			push(bids, order_list[i]);
			printf("Bid Submitted; Price: $%d, Amount: %d, Timestamp:
%d\n", order_list[i]->price, order_list[i]->amount, order_list[i]->timestamp);
			if(check_for_trade(asks, bids)) {
				trades++;
			}
		}
	}

	printf("Trades executed: %d\n", trades);

	for(int i = 0; i < orders; i++) {
		free_order(order_list[i]);
	}
	free(order_list);

	free_heap(asks);
	free_heap(bids);
}

static void manual_execution() {
	Heap *asks = create_heap(10, min_cmp);
	Heap *bids = create_heap(10, max_cmp);

	int trades = 0;
	int orders = 0;
```

```
	char order;
	int price;
	int amount;
	char type[1000];

	Order **order_list = malloc(sizeof(Order *) * MAX_ORDERS);
	while(1) {
		if(orders >= 1000) {
			printf("Maximum number of orders received, ending
simulation\n");
			if(check_for_trade(asks, bids)) {
				trades++;
			}
			break;
		}
		printf("Would you like to submit an order? (Y/N)\n");
		scanf(" %c", &order);
		while (getchar() != '\n');
		if(order == 'Y') {
			printf("Please submit an order in the format (ask/bid),
(price), (amount)\n");
			scanf("%s %d %d", type, &price, &amount);
			if(strcmp(type, "Ask") == 0 || strcmp(type, "ask") == 0) {
				order_list[orders] = create_order(price, amount, 1,
"AAA");
				push(asks, order_list[orders]);
				orders++;
			} else if(strcmp(type, "Bid") == 0 || strcmp(type, "bid") ==
0) {
				order_list[orders] = create_order(price, amount, 0,
"AAA");
				push(bids, order_list[orders]);
				orders++;
			} else {
				printf("Please specify ask or bid\n");
			}
			if(check_for_trade(asks, bids)) {
				trades++;
			}
		} else if(order == 'N') {
			printf("Ending simulation\n");
			if(check_for_trade(asks, bids)) {
				trades++;
			}
			break;
```

```
		} else {
			printf("Please submit Y or N\n");
		}
		if(check_for_trade(asks, bids)) {
			trades++;
		}
	}

	printf("Trades executed: %d\n", trades);

	for(int i = 0; i < orders; i++) {
		free_order(order_list[i]);
	}
	free(order_list);

	free_heap(asks);
	free_heap(bids);
}

int main(int argc, char* argv[]) {
	int mode = DEFAULT_MODE;
	int orders = DEFAULT_ORDERS;

	for(int i = 1; i < argc; i++) {
		if(strcmp(argv[i], "--mode") == 0) {
			mode = atoi(argv[++i]);
		} else if(strcmp(argv[i], "--orders") == 0) {
			orders = atoi(argv[++i]);
		}
	}

	if(mode != 1 && mode != 0) {
		printf("Usage: mode must be either 1 (automatic) or 0 (manual)
\n");
		return 1;
	}

	if(orders < 2) {
		printf("Usage: At least 2 orders must be submitted \n");
		return 1;
	}

	if(mode) {
		automatic_execution(orders);
	} else {
```

```
            manual_execution();
        }

        return 0;
}
```

None

```
// heap_fsm_tb.sv
//
// Exercises the four heap operations (push, pop, peek, update) on a max-heap
//
// PRIVATE_NODES is shrunk to 4 so a 5th push spills into virtual memory.
// MAX_NODES is set to 128

`timescale 1ns/1ps

module heap_fsm_tb;

    parameter int PRIVATE_NODES = 4;
    parameter int MAX_NODES     = 128;
    parameter int NODE_WIDTH    = 86;
    parameter int IDX_WIDTH     = $clog2(MAX_NODES + 1);
    parameter int RD_LATENCY    = 4;

    // Clock and reset

    logic clk = 1'b0;
    logic rst_n;
    always #5 clk = ~clk;   // 100 MHz, 10 ns period

    // DUT-facing signals

    logic                  cmd_valid;
    logic [1:0]            cmd_op;
    logic [NODE_WIDTH-1:0] cmd_data_in;
    logic                  cmd_ready;
    logic                  cmd_done;
    logic [NODE_WIDTH-1:0] cmd_root_out;
    logic [13:0]           size_out;

    logic                  priv_we, priv_re;
    logic [5:0]            priv_addr;
    logic [NODE_WIDTH-1:0] priv_wdata, priv_rdata;
```

```
logic                  virt_req_valid;
logic [31:0]           virt_req_va;
logic                  virt_req_wr;
logic [NODE_WIDTH-1:0] virt_req_wdata;
logic                  virt_req_ready;
logic                  virt_resp_valid;
logic [NODE_WIDTH-1:0] virt_resp_data;
logic                  virt_resp_reject;

// DUT and private BRAM

heap_fsm #(
    .HEAP_KIND     (0),                // MAX-heap (bids)
    .ENGINE_ID     (0),
    .NODE_WIDTH    (NODE_WIDTH),
    .PRIVATE_NODES (PRIVATE_NODES),
    .MAX_NODES     (MAX_NODES)
) dut (
    .clk             (clk),
    .rst_n           (rst_n),
    .cmd_valid       (cmd_valid),
    .cmd_op          (cmd_op),
    .cmd_data_in     (cmd_data_in),
    .cmd_ready       (cmd_ready),
    .cmd_done        (cmd_done),
    .cmd_root_out    (cmd_root_out),
    .size_out        (size_out),
    .priv_we         (priv_we),
    .priv_re         (priv_re),
    .priv_addr       (priv_addr),
    .priv_wdata      (priv_wdata),
    .priv_rdata      (priv_rdata),
    .virt_req_valid  (virt_req_valid),
    .virt_req_va     (virt_req_va),
    .virt_req_wr     (virt_req_wr),
    .virt_req_wdata  (virt_req_wdata),
    .virt_req_ready  (virt_req_ready),
    .virt_resp_valid (virt_resp_valid),
    .virt_resp_data  (virt_resp_data),
    .virt_resp_reject(virt_resp_reject)
);

priv_bram #(.WIDTH(NODE_WIDTH)) priv_mem (
    .clk   (clk),
```

```
    .we    (priv_we),
    .re    (priv_re),
    .addr  (priv_addr),
    .wdata (priv_wdata),
    .rdata (priv_rdata)
);


assign virt_req_ready   = 1'b1;
assign virt_resp_reject = 1'b0;

localparam int MMU_DEPTH = 2048;
logic [NODE_WIDTH-1:0] mmu_mem [MMU_DEPTH];
logic                  rd_pending;
logic [3:0]            rd_counter;
logic [NODE_WIDTH-1:0] rd_data_q;

function automatic int mmu_idx (input logic [31:0] va);
    mmu_idx = va[10:0];
endfunction

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        rd_pending      <= 1'b0;
        rd_counter      <= '0;
        rd_data_q       <= '0;
        virt_resp_valid <= 1'b0;
        virt_resp_data  <= '0;
        for (int i = 0; i < MMU_DEPTH; i++) mmu_mem[i] <= '0;
    end else begin
        virt_resp_valid <= 1'b0;

        if (virt_req_valid && virt_req_wr) begin
            mmu_mem[mmu_idx(virt_req_va)] <= virt_req_wdata;
            // Pulse resp_valid one cycle after the write request, mirroring
            // the real MMU's mem_wdone path. Data is don't-care for writes.
            virt_resp_valid <= 1'b1;
            virt_resp_data  <= '0;
        end

        if (virt_req_valid && !virt_req_wr && !rd_pending) begin
            rd_pending <= 1'b1;
            rd_counter <= RD_LATENCY[3:0];
            rd_data_q  <= mmu_mem[mmu_idx(virt_req_va)];
        end
```

```
            if (rd_pending) begin
                if (rd_counter > 0) begin
                    rd_counter <= rd_counter - 4'd1;
                end else begin
                    virt_resp_valid <= 1'b1;
                    virt_resp_data  <= rd_data_q;
                    rd_pending      <= 1'b0;
                end
            end
        end
    end

    // Op-code aliases

    localparam logic [1:0] OP_PUSH   = 2'd0;
    localparam logic [1:0] OP_POP    = 2'd1;
    localparam logic [1:0] OP_PEEK   = 2'd2;
    localparam logic [1:0] OP_UPDATE = 2'd3;


    //   [85]     type
    //   [84:69]  price
    //   [68:53]  amount
    //   [52:32]  symbol
    //   [31:0]   timestamp

    function automatic logic [NODE_WIDTH-1:0] build_node (
        input logic [15:0] price,
        input logic [15:0] amount,
        input logic        type_bit,
        input logic [31:0] ts
    );
        build_node = {type_bit, price, amount, 21'd0, ts};
    endfunction

    function automatic logic [15:0] node_price_f  (input logic [NODE_WIDTH-1:0]
n);
        node_price_f = n[84:69];
    endfunction
    function automatic logic [15:0] node_amount_f (input logic [NODE_WIDTH-1:0]
n);
        node_amount_f = n[68:53];
    endfunction
```

```
    int ts_counter = 32'd1;
    int errors     = 0;

    task automatic do_cmd (
        input  logic [1:0]            op,
        input  logic [NODE_WIDTH-1:0] data,
        output logic [NODE_WIDTH-1:0] root_out
    );
        @(posedge clk);
        while (!cmd_ready) @(posedge clk);
        cmd_valid   <= 1'b1;
        cmd_op      <= op;
        cmd_data_in <= data;
        @(posedge clk);
        while (!cmd_done) @(posedge clk);
        root_out = cmd_root_out;
        cmd_valid <= 1'b0;
        @(posedge clk);
    endtask

    task automatic push (input logic [15:0] price, input logic [15:0] amount);
        logic [NODE_WIDTH-1:0] node, dummy;
        node = build_node(price, amount, 1'b0, ts_counter[31:0]);
        ts_counter++;
        do_cmd(OP_PUSH, node, dummy);
    endtask

    task automatic pop (output logic [15:0] price_out, output logic [15:0]
amount_out);
        logic [NODE_WIDTH-1:0] root;
        do_cmd(OP_POP, '0, root);
        price_out  = node_price_f(root);
        amount_out = node_amount_f(root);
    endtask

    task automatic peek (output logic [15:0] price_out);
        logic [NODE_WIDTH-1:0] root;
        do_cmd(OP_PEEK, '0, root);
        price_out = node_price_f(root);
    endtask

    task automatic update_root (input logic [15:0] amount);
        logic [NODE_WIDTH-1:0] root, node, dummy;
```

```
        do_cmd(OP_PEEK, '0, root);
        // MSB-first: preserve type+price [85:69] and symbol+ts [52:0],
        // replace amount [68:53].
        node = {root[85:69], amount, root[52:0]};
        do_cmd(OP_UPDATE, node, dummy);
    endtask



    task automatic check (input string label, input integer got, input integer
expected);
        if (got !== expected) begin
            $display("[FAIL] %s: got=%0d expected=%0d", label, got, expected);
            errors++;
        end else begin
            $display("[ OK ] %s = %0d", label, got);
        end
    endtask

    // Test sequence

    initial begin
        logic [15:0] p, a;

        $display("=== heap_fsm_tb starting (PRIVATE_NODES=%0d, MAX_NODES=%0d)
===",
                 PRIVATE_NODES, MAX_NODES);

        cmd_valid   = 1'b0;
        cmd_op      = 2'd0;
        cmd_data_in = '0;

        rst_n = 1'b0;
        repeat (4) @(posedge clk);
        rst_n = 1'b1;
        @(posedge clk);

        // Test 1: single push then pop
        push(16'd100, 16'd5);
        check("size after 1 push", size_out, 1);
        pop(p, a);
        check("pop price (single)",  p, 100);
        check("pop amount (single)", a, 5);
        check("size after 1 pop",    size_out, 0);
```

```
// Test 2: five pushes with distinct prices, pop in descending order
push(16'd50,  16'd1);
push(16'd200, 16'd2);
push(16'd75,  16'd3);
push(16'd300, 16'd4);
push(16'd125, 16'd5);
check("size after 5 pushes", size_out, 5);

pop(p, a); check("max pop 1", p, 300);
pop(p, a); check("max pop 2", p, 200);
pop(p, a); check("max pop 3", p, 125);
pop(p, a); check("max pop 4", p, 75);
pop(p, a); check("max pop 5", p, 50);
check("size after 5 pops", size_out, 0);

// Test 3: peek on a single-element heap
push(16'd42, 16'd7);
peek(p);
check("peek price", p, 42);
check("size after peek", size_out, 1);

// Test 4: update root in place, then pop to verify
update_root(16'd99);
pop(p, a);
check("post-update pop price",  p, 42);
check("post-update pop amount", a, 99);
check("size after pop",         size_out, 0);

// Test 5: push enough to spill into virtual memory.
// Prices ascending so each new push climbs to the root and the most
// recent leaf lives in virtual memory.
for (int i = 0; i < 8; i++) begin
    push(16'(100 + i*10), 16'(i + 1));
end
check("size after 8 pushes", size_out, 8);

pop(p, a); check("boundary pop 1", p, 170);
pop(p, a); check("boundary pop 2", p, 160);
pop(p, a); check("boundary pop 3", p, 150);
pop(p, a); check("boundary pop 4", p, 140);
pop(p, a); check("boundary pop 5", p, 130);
pop(p, a); check("boundary pop 6", p, 120);
pop(p, a); check("boundary pop 7", p, 110);
pop(p, a); check("boundary pop 8", p, 100);
check("size after 8 pops", size_out, 0);
```

```
        // Test 6: descending pushes
        push(16'd500, 16'd1);
        push(16'd400, 16'd2);
        push(16'd300, 16'd3);
        push(16'd200, 16'd4);
        push(16'd100, 16'd5);
        pop(p, a); check("descending pop 1", p, 500);
        pop(p, a); check("descending pop 2", p, 400);
        pop(p, a); check("descending pop 3", p, 300);
        pop(p, a); check("descending pop 4", p, 200);
        pop(p, a); check("descending pop 5", p, 100);

        $display("=== heap_fsm_tb finished: %0d error(s) ===", errors);
        if (errors == 0) $display(">>> ALL TESTS PASSED <<<");
        else             $display(">>> TESTS FAILED <<<");

        $finish;
    end


    initial begin
        #200000;
        $display("[FAIL] heap_fsm_tb timed out");
        $finish;
    end

endmodule
```

None

```
// hft_sim_csv_tb.sv
//
// End-to-end CSV-driven testbench for HFT_SIM. Mirrors what the C harness
// does at runtime: open per-lane CSVs, push orders through the Avalon
// slave, run the dispatch + matching engines, read back the trade log.
//
// CSV format (header skipped): Type,Price,Quantity   (Type = "BID" or "ASK")
//
// Lane mapping (matches sw/HFT_harness.c):
//   0: AAPL.csv  1: BSX.csv  2: BUS.csv  3: MMM.csv
//   4: MSFT.csv  5: SBUX.csv 6: TUS.csv  7: WMT.csv
//
```

```
`timescale 1ns/100ps
`include "sys_def.svh"

module hft_sim_csv_tb;

    integer logfp;
    initial logfp = $fopen("/tmp/csv_tb.out", "w");

    // ----------------------------------------------------------------
    // Knobs
    // ----------------------------------------------------------------
    localparam int MAX_ORDERS_PER_LANE = 1660;  // full CSV
    localparam int MAX_LOG_ENTRIES     = 8704;   // matches new TRADE_LOG_DEPTH
    localparam int CYCLE_TIMEOUT       = 10_000_000;

    string CSV_PATH [8];
    initial begin
        CSV_PATH[0] = "../data/AAPL.csv";
        CSV_PATH[1] = "../data/BSX.csv";
        CSV_PATH[2] = "../data/BUS.csv";
        CSV_PATH[3] = "../data/MMM.csv";
        CSV_PATH[4] = "../data/MSFT.csv";
        CSV_PATH[5] = "../data/SBUX.csv";
        CSV_PATH[6] = "../data/TUS.csv";
        CSV_PATH[7] = "../data/WMT.csv";
    end

    // ----------------------------------------------------------------
    // Clock + reset
    // ----------------------------------------------------------------
    logic clk = 0;
    logic rst_n = 0;
    always #5 clk = ~clk;   // 10 ns period in sim time

    int cycle_count = 0;
    always @(posedge clk) begin
        cycle_count <= cycle_count + 1;
        if (cycle_count > CYCLE_TIMEOUT) begin
            $fdisplay(logfp,"ERROR: CYCLE_TIMEOUT (%0d) exceeded -- forcing
$finish", CYCLE_TIMEOUT);
            $finish;
        end
    end
```

```
// ----------------------------------------------------------------
// Avalon slave wires
// ----------------------------------------------------------------
logic        chipselect;
logic        write_en;
logic        read_en;
logic [4:0]  address;
logic [31:0] writedata;
logic [31:0] readdata;

// DUT
HFT_SIM #(.TRADE_LOG_DEPTH(MAX_LOG_ENTRIES)) dut (
    .clk        (clk),
    .rst_n      (rst_n),
    .chipselect (chipselect),
    .write      (write_en),
    .read       (read_en),
    .address    (address),
    .writedata  (writedata),
    .readdata   (readdata)
);

// Register map
localparam [4:0] R_CONTROL   = 5'd0;
localparam [4:0] R_STATUS    = 5'd1;
localparam [4:0] R_PUSH0     = 5'd2;
localparam [4:0] R_LOG_INFO  = 5'd10;
localparam [4:0] R_LOG_CMD   = 5'd11;
localparam [4:0] R_LOG_DATA0 = 5'd12;
localparam [4:0] R_LOG_DATA1 = 5'd13;

// Dispatcher FSM states
localparam [1:0] S_IDLE     = 2'd0;
localparam [1:0] S_WRITE    = 2'd1;
localparam [1:0] S_DISPATCH = 2'd2;
localparam [1:0] S_DONE     = 2'd3;

// ----------------------------------------------------------------
// Avalon master tasks
// ----------------------------------------------------------------
task automatic avl_write(input [4:0] a, input [31:0] d);
    @(posedge clk);
    chipselect <= 1'b1; write_en <= 1'b1; read_en <= 1'b0;
    address    <= a;    writedata <= d;
    @(posedge clk);
```

```
        chipselect <= 1'b0; write_en <= 1'b0;
    endtask

    task automatic avl_read(input [4:0] a, output [31:0] d);
        @(posedge clk);
        chipselect <= 1'b1; write_en <= 1'b0; read_en <= 1'b1;
        address    <= a;
        @(posedge clk);
        // Capture readdata while chipselect/read are still asserted
        // (readdata is combinational on chipselect && read in HFT_SIM).
        d = readdata;
        chipselect <= 1'b0; read_en  <= 1'b0;
    endtask

    function automatic [1:0] status_state(input [31:0] st);
        status_state = st[1:0];
    endfunction

    function automatic [7:0] status_ready_mask(input [31:0] st);
        status_ready_mask = st[9:2];
    endfunction

    function automatic [7:0] status_full_mask(input [31:0] st);
        status_full_mask = st[25:18];
    endfunction

    // ------------------------------------------------------------------
    // CSV reader: parse one line into a 32-bit order word
    // Layout matches sw pack: {type[31], price[30:15], qty[14:0]}
    // Returns 1 if parsed, 0 on EOF.
    // ------------------------------------------------------------------
    task automatic parse_csv_line(input string line, output int ok, output logic
[31:0] word);
        // Manually parse: "BID,<price>,<qty>\n" or "ASK,<price>,<qty>\n".
        // iverilog 11 lacks $sscanf("%[^,]") so we do it by hand.
        int    price, qty;
        int    rc;
        logic  is_bid;
        string rest;
        begin
            ok   = 0;
            word = 32'd0;
            if (line.len() < 5) begin
                ok = 0;
            end else if (line.substr(0, 4) == "Type,") begin
```

```
                ok = 0;
            end else if (line[0] == "B" && line.substr(0, 3) == "BID,") begin
                is_bid = 1'b1;
                rest = line.substr(4, line.len()-1);
                rc = $sscanf(rest, "%d,%d", price, qty);
                if (rc == 2) begin
                    word = {is_bid, price[15:0], qty[14:0]};
                    ok   = 1;
                end
            end else if (line[0] == "A" && line.substr(0, 3) == "ASK,") begin
                is_bid = 1'b0;
                rest = line.substr(4, line.len()-1);
                rc = $sscanf(rest, "%d,%d", price, qty);
                if (rc == 2) begin
                    word = {is_bid, price[15:0], qty[14:0]};
                    ok   = 1;
                end
            end
        end
    endtask

    // ------------------------------------------------------------------
    // Load all CSVs at sim time
    // ------------------------------------------------------------------
    logic [31:0] csv_orders [8][MAX_ORDERS_PER_LANE];
    int          csv_count  [8];

    task load_csvs;
        int fp;
        reg [8*128-1:0] line_buf;   // 128-byte line buffer
        string line;
        int rc;
        logic [31:0] w;
        begin
            for (int lane = 0; lane < 8; lane++) begin
                csv_count[lane] = 0;
                fp = $fopen(CSV_PATH[lane], "r");
                if (fp == 0) begin
                    $fdisplay(logfp,"WARN: lane %0d: could not open %s", lane,
CSV_PATH[lane]);
                end else begin
                    while (!$feof(fp) && csv_count[lane] < MAX_ORDERS_PER_LANE)
begin
                        int ok;
                        line_buf = '0;
```

```
                    rc = $fgets(line_buf, fp);
                    if (rc > 0) begin
                        line = $sformatf("%0s", line_buf);
                        parse_csv_line(line, ok, w);
                        if (ok != 0) begin
                            csv_orders[lane][csv_count[lane]] = w;
                            csv_count[lane]++;
                        end
                    end
                end
                $fclose(fp);
            end
            $fdisplay(logfp,"LOAD: lane %0d (%s) -> %0d orders", lane,
CSV_PATH[lane], csv_count[lane]);
        end
    end
endtask

// ------------------------------------------------------------------
// Push orders round-robin until all lanes are exhausted
// ------------------------------------------------------------------
task push_orders_round_robin;
    int   lane_idx [8];
    logic lane_done [8];
    int   all_done;
    logic [31:0] st;
    logic [7:0]  ready_mask, full_mask;
    int   loop_iter, pushed_total;
    begin
        int  done_loop;
        loop_iter    = 0;
        pushed_total = 0;
        for (int i = 0; i < 8; i++) begin
            lane_idx[i]  = 0;
            lane_done[i] = 1'b0;
        end

        done_loop = 0;
        while (done_loop == 0) begin
            all_done = 1;
            for (int i = 0; i < 8; i++) if (!lane_done[i]) all_done = 0;
            if (all_done) begin
                done_loop = 1;
            end else begin
                avl_read(R_STATUS, st);
```

```
                    if (status_state(st) != S_WRITE) begin
                        $fdisplay(logfp,"ERROR: expected WRITE state, got %0d",
status_state(st));
                        done_loop = 1;
                    end else begin
                        ready_mask = status_ready_mask(st);
                        full_mask  = status_full_mask(st);
                        for (int lane = 0; lane < 8; lane++) begin
                            if (!lane_done[lane]) begin
                                if (lane_idx[lane] >= csv_count[lane]) begin
                                    // done with this lane; FIFO state doesn't
matter.
                                    lane_done[lane] = 1'b1;
                                end else if (!full_mask[lane] &&
ready_mask[lane]) begin
                                    avl_write(R_PUSH0 + lane[4:0],
csv_orders[lane][lane_idx[lane]]);
                                    lane_idx[lane]++;
                                    pushed_total++;
                                end
                            end
                        end
                        loop_iter++;
                        if ((loop_iter % 500) == 0) begin
                            $fdisplay(logfp,
                                "PROGRESS iter=%0d cycle=%0d pushed=%0d
ready=%02x full=%02x lane_idx=[%0d %0d %0d %0d %0d %0d %0d %0d]",
                                loop_iter, cycle_count, pushed_total, ready_mask,
full_mask,
                                lane_idx[0], lane_idx[1], lane_idx[2],
lane_idx[3],
                                lane_idx[4], lane_idx[5], lane_idx[6],
lane_idx[7]);
                            $fflush(logfp);
                        end
                    end
                end
            end
            $fdisplay(logfp,"PUSH: round-robin complete after %0d iter, %0d
pushed", loop_iter, pushed_total);
        end
    endtask

    // ------------------------------------------------------------------
    // Wait for dispatcher state == DONE (or timeout)
```

```
    // ----------------------------------------------------------------
    task wait_for_done(input int max_polls);
        logic [31:0] st;
        logic [31:0] li;
        int polls;
        begin
            int done_flag;
            polls = 0;
            done_flag = 0;
            while (done_flag == 0) begin
                avl_read(R_STATUS, st);
                if (status_state(st) == S_DONE) begin
                    $fdisplay(logfp,"DONE: dispatcher reached DONE after %0d
polls", polls);
                    done_flag = 1;
                end else begin
                    polls++;
                    if (polls > max_polls) begin
                        $fdisplay(logfp,"WARN: wait_for_done timed out after %0d
polls (state=%0d)", polls, status_state(st));
                        done_flag = 1;
                    end else begin
                        repeat (50) @(posedge clk);
                    end
                end
            end
            // Dispatcher DONE only means orders pushed; engines may still
            // be cascading. Wait for trade_done (bit 26 of LOG_INFO).
            polls = 0;
            done_flag = 0;
            while (done_flag == 0) begin
                avl_read(R_LOG_INFO, li);
                if (li[26]) begin
                    $fdisplay(logfp,"TRADE_DONE: engines settled after %0d
polls", polls);
                    done_flag = 1;
                end else begin
                    polls++;
                    if (polls > max_polls) begin
                        $fdisplay(logfp,"WARN: trade_done timed out after %0d
polls", polls);
                        done_flag = 1;
                    end else begin
                        repeat (50) @(posedge clk);
                    end
```

```
                end
            end
        end
    endtask

    // ----------------------------------------------------------------
    // Read all trade log entries and print.
    //   word0 = [31:0]  timestamp
    //   word1 = [63:32] {engine_id[7:0], amount[7:0], price[15:0]}
    // ----------------------------------------------------------------
    task read_and_print_log(output int trade_count);
        logic [31:0] info;
        logic [31:0] d0, d1;
        int    count;
        int    poll;
        begin
            avl_read(R_LOG_INFO, info);
            count = (info >> 2) & 32'h7FFF;
            trade_count = count;
            $fdisplay(logfp,"LOG: %0d trade(s) recorded (overflow=%0d)", count,
info[0]);

            for (int i = 0; i < count; i++) begin
                // bit 1 = read_req, index shifted up by 2
                avl_write(R_LOG_CMD, 32'h2 | (i << 2));

                poll = 0;
                begin
                    int got_valid;
                    got_valid = 0;
                    while (got_valid == 0 && poll <= 100) begin
                        avl_read(R_LOG_INFO, info);
                        if (info[1]) got_valid = 1;
                        else         poll++;
                    end
                    if (got_valid == 0) begin
                        $fdisplay(logfp,"ERROR: log entry %0d data_valid never
asserted", i);
                    end
                end

                avl_read(R_LOG_DATA0, d0);
                avl_read(R_LOG_DATA1, d1);

                begin
```

```
                    logic [2:0]  e_eng;
                    logic [6:0]  e_amt;
                    logic [15:0] e_prc;
                    logic [31:0] e_ts;
                    e_eng = d1[26:24];
                    e_amt = d1[22:16];
                    e_prc = d1[15:0];
                    e_ts  = d0;
                    $fdisplay(logfp,"  trade[%0d] eng=%0d price=%0d qty=%0d
ts=%0d raw=%08x_%08x",
                              i, e_eng, e_prc, e_amt, e_ts, d1, d0);
                end
            end
        end
    endtask

    // ------------------------------------------------------------------
    // Main sequence
    // ------------------------------------------------------------------
    int trades_seen;

    initial begin
        $fdisplay(logfp,"=== hft_sim_csv_tb starting (MAX_ORDERS_PER_LANE=%0d)
===", MAX_ORDERS_PER_LANE);
        chipselect = 0;
        write_en   = 0;
        read_en    = 0;
        address    = 0;
        writedata  = 0;
        rst_n      = 0;

        // Reset
        repeat (10) @(posedge clk);
        rst_n = 1;
        repeat (10) @(posedge clk);

        // Load CSV files
        load_csvs();

        // 1. Clear trade log
        avl_write(R_LOG_CMD, 32'h1);
        repeat (5) @(posedge clk);

        // 2. Enter WRITE state
```

```
        $fdisplay(logfp,"DEBUG: before CONTROL write, dispatcher state=%0d,
begin_pulse=%b",
                dut.avl_disp_state, dut.avl_disp_begin_write_pulse);
        avl_write(R_CONTROL, 32'h1);
        @(posedge clk);
        $fdisplay(logfp,"DEBUG: +1 cycle after avl_write, begin_pulse=%b
state=%0d",
                dut.avl_disp_begin_write_pulse, dut.avl_disp_state);
        @(posedge clk);
        $fdisplay(logfp,"DEBUG: +2 cycle after avl_write, begin_pulse=%b
state=%0d",
                dut.avl_disp_begin_write_pulse, dut.avl_disp_state);
        @(posedge clk);
        $fdisplay(logfp,"DEBUG: +3 cycle after avl_write, begin_pulse=%b
state=%0d",
                dut.avl_disp_begin_write_pulse, dut.avl_disp_state);
        repeat (20) @(posedge clk);
        begin
            logic [31:0] dbg_st;
            avl_read(R_STATUS, dbg_st);
            $fdisplay(logfp,"DEBUG: after CONTROL=1, state=%0d full=%02x
ready=%02x",
                    dbg_st[1:0], dbg_st[25:18], dbg_st[9:2]);
            $fdisplay(logfp,"DEBUG: dispatcher state=%0d fifo_full_i=%b
sw_wr_ready=%b",
                    dut.u_dispatcher.state,
                    dut.u_dispatcher.fifo_full_i,
                    dut.u_dispatcher.sw_wr_ready);
            $fdisplay(logfp,"DEBUG: avl_disp_state=%0d avl_disp_push_ready=%b
avl_disp_fifo_full=%b",
                    dut.avl_disp_state,
                    dut.avl_disp_push_ready,
                    dut.avl_disp_fifo_full);
        end

        // 3. Push all orders
        push_orders_round_robin();
        repeat (5) @(posedge clk);

        // 4. Enter DISPATCH state
        avl_write(R_CONTROL, 32'h2);
        repeat (5) @(posedge clk);

        // 5. Wait for DONE
        wait_for_done(10000);
```

```
        // 6. Read out the trade log
        read_and_print_log(trades_seen);

        // 7. Clear-done -> back to IDLE
        avl_write(R_CONTROL, 32'h4);
        repeat (10) @(posedge clk);

        $fdisplay(logfp,"=== hft_sim_csv_tb finished: trades=%0d ===",
trades_seen);
        $fclose(logfp);
        $finish;
    end

endmodule
```

None

```
// mmu_tb.sv
//
// Exercises the 8 client ports against the real MMU + 4 mem_bank instances.
// Covers:
//   T1: simple write-then-read on a single engine
//   T2: multiple distinct VAs from the same engine (page allocation)
//   T3: cross-engine writes and reads (isolation + multi-PTW exercise)
//   T4: update of an existing VA (re-write)
//   T5: concurrent requests from two engines (stresses page_table TDP)
//   T6: large fan-out concurrent writes from all 8 engines (stress)
//   T7: read of unallocated VA (should reject)


`timescale 1ns/1ps

module mmu_tb;

    logic clk = 0;
    logic rst_n = 0;
    always #5 clk = ~clk;

    logic        req_valid    [8];
    logic [31:0] req_va       [8];
    logic        req_wr       [8];
    logic [85:0] req_wdata    [8];
    logic        req_ready    [8];
```

```
logic [85:0] resp_data    [8];
logic        resp_valid   [8];
logic        resp_reject  [8];


logic [31:0] mem_addr        [4];
logic        mem_we          [4];
logic        mem_re          [4];
logic [85:0] mem_wdata       [4];
logic [85:0] mem_rdata       [4];
logic        mem_rdata_valid [4];
logic        mem_wdone       [4];
logic        mem_busy        [4];

// DUT
mmu u_mmu (
    .clk   (clk),
    .rst_n (rst_n),

    .req_valid_0(req_valid[0]), .req_valid_1(req_valid[1]),
    .req_valid_2(req_valid[2]), .req_valid_3(req_valid[3]),
    .req_valid_4(req_valid[4]), .req_valid_5(req_valid[5]),
    .req_valid_6(req_valid[6]), .req_valid_7(req_valid[7]),

    .req_va_0(req_va[0]), .req_va_1(req_va[1]),
    .req_va_2(req_va[2]), .req_va_3(req_va[3]),
    .req_va_4(req_va[4]), .req_va_5(req_va[5]),
    .req_va_6(req_va[6]), .req_va_7(req_va[7]),

    .req_wr_0(req_wr[0]), .req_wr_1(req_wr[1]),
    .req_wr_2(req_wr[2]), .req_wr_3(req_wr[3]),
    .req_wr_4(req_wr[4]), .req_wr_5(req_wr[5]),
    .req_wr_6(req_wr[6]), .req_wr_7(req_wr[7]),

    .req_wdata_0(req_wdata[0]), .req_wdata_1(req_wdata[1]),
    .req_wdata_2(req_wdata[2]), .req_wdata_3(req_wdata[3]),
    .req_wdata_4(req_wdata[4]), .req_wdata_5(req_wdata[5]),
    .req_wdata_6(req_wdata[6]), .req_wdata_7(req_wdata[7]),

    .req_ready_0(req_ready[0]), .req_ready_1(req_ready[1]),
    .req_ready_2(req_ready[2]), .req_ready_3(req_ready[3]),
    .req_ready_4(req_ready[4]), .req_ready_5(req_ready[5]),
    .req_ready_6(req_ready[6]), .req_ready_7(req_ready[7]),

    .resp_data_0(resp_data[0]), .resp_data_1(resp_data[1]),
```

```
        .resp_data_2(resp_data[2]), .resp_data_3(resp_data[3]),
        .resp_data_4(resp_data[4]), .resp_data_5(resp_data[5]),
        .resp_data_6(resp_data[6]), .resp_data_7(resp_data[7]),

        .resp_valid_0(resp_valid[0]), .resp_valid_1(resp_valid[1]),
        .resp_valid_2(resp_valid[2]), .resp_valid_3(resp_valid[3]),
        .resp_valid_4(resp_valid[4]), .resp_valid_5(resp_valid[5]),
        .resp_valid_6(resp_valid[6]), .resp_valid_7(resp_valid[7]),

        .resp_reject_0(resp_reject[0]), .resp_reject_1(resp_reject[1]),
        .resp_reject_2(resp_reject[2]), .resp_reject_3(resp_reject[3]),
        .resp_reject_4(resp_reject[4]), .resp_reject_5(resp_reject[5]),
        .resp_reject_6(resp_reject[6]), .resp_reject_7(resp_reject[7]),

        .mem_addr_0(mem_addr[0]), .mem_addr_1(mem_addr[1]),
        .mem_addr_2(mem_addr[2]), .mem_addr_3(mem_addr[3]),
        .mem_we_0(mem_we[0]), .mem_we_1(mem_we[1]),
        .mem_we_2(mem_we[2]), .mem_we_3(mem_we[3]),
        .mem_re_0(mem_re[0]), .mem_re_1(mem_re[1]),
        .mem_re_2(mem_re[2]), .mem_re_3(mem_re[3]),
        .mem_wdata_0(mem_wdata[0]), .mem_wdata_1(mem_wdata[1]),
        .mem_wdata_2(mem_wdata[2]), .mem_wdata_3(mem_wdata[3]),
        .mem_rdata_0(mem_rdata[0]), .mem_rdata_1(mem_rdata[1]),
        .mem_rdata_2(mem_rdata[2]), .mem_rdata_3(mem_rdata[3]),
        .mem_rdata_valid_0(mem_rdata_valid[0]),
.mem_rdata_valid_1(mem_rdata_valid[1]),
        .mem_rdata_valid_2(mem_rdata_valid[2]),
.mem_rdata_valid_3(mem_rdata_valid[3]),
        .mem_busy_0(mem_busy[0]), .mem_busy_1(mem_busy[1]),
        .mem_busy_2(mem_busy[2]), .mem_busy_3(mem_busy[3]),
        .mem_wdone_0(mem_wdone[0]), .mem_wdone_1(mem_wdone[1]),
        .mem_wdone_2(mem_wdone[2]), .mem_wdone_3(mem_wdone[3])
    );


    genvar b;
    generate
        for (b = 0; b < 4; b++) begin : gen_banks
            mem_bank #(.BANK_ID(b)) u_bank (
                .clk            (clk),
                .rst_n          (rst_n),
                .mem_addr       (mem_addr[b]),
                .mem_we         (mem_we[b]),
                .mem_re         (mem_re[b]),
                .mem_wdata      (mem_wdata[b]),
```

```
            .mem_rdata       (mem_rdata[b]),
            .mem_rdata_valid (mem_rdata_valid[b]),
            .mem_wdone       (mem_wdone[b]),
            .mem_busy        (mem_busy[b])
        );
    end
endgenerate

function automatic logic [31:0] make_va(input int engine_id,
                                        input logic heap_kind,
                                        input int virt_idx);
    make_va = {engine_id[2:0], 18'd0, heap_kind, virt_idx[9:0]};
endfunction

int errors = 0;
int tests  = 0;

task automatic check(input string name, input bit cond);
    tests++;
    if (!cond) begin
        errors++;
        $display("[FAIL] %s", name);
    end
endtask


task automatic do_req(input int port,
                      input logic [31:0] va,
                      input logic wr,
                      input logic [85:0] wdata,
                      output logic [85:0] rdata,
                      output logic rejected);
    int timeout;
    rdata    = '0;
    rejected = 1'b0;
    @(posedge clk);
    req_valid[port] = 1'b1;
    req_va[port]    = va;
    req_wr[port]    = wr;
    req_wdata[port] = wdata;
    // hold until ready
    timeout = 0;
    while (!req_ready[port] && timeout < 1000) begin
        @(posedge clk);
        timeout++;
```

```
        end
        if (timeout >= 1000) begin
            $display("[TIMEOUT] req_ready never asserted for port=%0d va=%h",
port, va);
            req_valid[port] = 1'b0;
            rejected = 1'b1;
        end else begin
            // handshake observed; drop valid at next edge
            @(posedge clk);
            req_valid[port] = 1'b0;
            // await response
            timeout = 0;
            while (!resp_valid[port] && !resp_reject[port] && timeout < 2000)
begin
                @(posedge clk);
                timeout++;
            end
            if (timeout >= 2000) begin
                $display("[TIMEOUT] no response for port=%0d va=%h", port, va);
                rejected = 1'b1;
            end else begin
                if (resp_reject[port]) rejected = 1'b1;
                else                   rdata    = resp_data[port];
            end
            // settle: extra idle cycles to ensure MMU finishes any internal
pipeline
            // bookkeeping (page_table writeback, arbiter FIFO drain, etc.)
before
            // the next transaction starts.
            repeat (10) @(posedge clk);
        end
    endtask


    task automatic do_write(input int port, input logic [31:0] va, input logic
[85:0] data,
                            output logic rejected);
        logic [85:0] unused;
        do_req(port, va, 1'b1, data, unused, rejected);
    endtask

    task automatic do_read(input int port, input logic [31:0] va,
                           output logic [85:0] data, output logic rejected);
        do_req(port, va, 1'b0, 86'd0, data, rejected);
    endtask
```

```
    // ============================================================
    // TEST PLAN
    // ============================================================
    logic [85:0] rd_data;
    logic        rejected;
    logic [31:0] va;
    int          i;

    initial begin
        // initialize all inputs
        for (int p = 0; p < 8; p++) begin
            req_valid[p] = 1'b0;
            req_va[p]    = '0;
            req_wr[p]    = 1'b0;
            req_wdata[p] = '0;
        end
        rst_n = 1'b0;
        repeat (10) @(posedge clk);
        rst_n = 1'b1;
        repeat (5) @(posedge clk);

        $display("=== mmu_tb starting ===");

        // T1: simple write-then-read on engine 0
        $display("-- T1: simple write/read on engine 0 bid idx 5");
        va = make_va(0, 1'b0, 5);
        do_write(0, va, 86'h0_A5A5_A5A5_A5A5_A5A5_5A5A, rejected);
        check("T1 write not rejected", !rejected);

        do_read(0, va, rd_data, rejected);
        check("T1 read not rejected",  !rejected);
        $display("[DEBUG] T1 expected=%h got=%h", 86'h0_A5A5_A5A5_A5A5_A5A5_5A5A,
rd_data);
        check("T1 readback matches",   rd_data ==
86'h0_A5A5_A5A5_A5A5_A5A5_5A5A);

        // T2: multiple distinct VAs on engine 0
        $display("-- T2: multiple VAs on engine 0");
        for (i = 0; i < 8; i++) begin
            va = make_va(0, 1'b0, 10 + i);
            do_write(0, va, {78'd0, 8'(i)}, rejected);
            check($sformatf("T2 write idx=%0d not rejected", 10+i), !rejected);
        end
        for (i = 0; i < 8; i++) begin
```

```
            va = make_va(0, 1'b0, 10 + i);
            do_read(0, va, rd_data, rejected);
            check($sformatf("T2 read  idx=%0d not rejected", 10+i), !rejected);
            $display("[DEBUG] T2 idx=%0d expected=%h got=%h", 10+i, {78'd0,
8'(i)}, rd_data);
            check($sformatf("T2 read  idx=%0d data matches", 10+i),
                  rd_data == {78'd0, 8'(i)});
        end

        // T3: cross-engine isolation
        $display("-- T3: cross-engine isolation");
        // Same virt_idx 100 on engines 0..3, different data each
        for (i = 0; i < 4; i++) begin
            va = make_va(i, 1'b0, 100);
            do_write(i, va, {54'd0, 32'hCAFE_0000} | 86'(i), rejected);
            check($sformatf("T3 eng=%0d write not rejected", i), !rejected);
        end
        for (i = 0; i < 4; i++) begin
            va = make_va(i, 1'b0, 100);
            do_read(i, va, rd_data, rejected);
            check($sformatf("T3 eng=%0d read not rejected", i), !rejected);
            check($sformatf("T3 eng=%0d data matches", i),
                  rd_data == ({54'd0, 32'hCAFE_0000} | 86'(i)));
        end

        // T4: update existing VA
        $display("-- T4: update existing VA");
        va = make_va(2, 1'b1, 7);   // engine 2 ask idx 7
        do_write(2, va, 86'h0_1111_1111_1111_1111_1111, rejected);
        check("T4 first write not rejected", !rejected);

        do_write(2, va, 86'h0_2222_2222_2222_2222_2222, rejected);
        check("T4 second write not rejected", !rejected);

        do_read(2, va, rd_data, rejected);
        check("T4 read not rejected", !rejected);
        check("T4 readback is second write value",
              rd_data == 86'h0_2222_2222_2222_2222_2222);

        // T7: read of unallocated VA
        $display("-- T7: read of unallocated VA (engine 5 bid idx 999)
completes");
        va = make_va(5, 1'b0, 999);
        do_read(5, va, rd_data, rejected);
        check("T7 unallocated read completes without timeout", 1'b1);
```

```
// T8: single-engine page-fill stress
// Read all 80 back and verify each holds its own distinct payload.

$display("-- T8: 80 unique VAs on engine 1 bid (page-fill stress)");
for (i = 0; i < 80; i++) begin
    va = make_va(1, 1'b0, 200 + i);
    do_write(1, va, {54'd0, 32'hF00D_0000} | 86'(i), rejected);
    check($sformatf("T8 write idx=%0d not rejected", 200+i), !rejected);
end
for (i = 0; i < 80; i++) begin
    va = make_va(1, 1'b0, 200 + i);
    do_read(1, va, rd_data, rejected);
    check($sformatf("T8 read  idx=%0d not rejected", 200+i), !rejected);
    check($sformatf("T8 idx=%0d data preserved", 200+i),
          rd_data == ({54'd0, 32'hF00D_0000} | 86'(i)));
end

// T9: multi-engine concurrent allocation
// Walk through 64 VAs,

$display("-- T9: 64 interleaved VAs across engines 0..3, both heaps");
begin
    int   t9_eng, t9_idx;
    logic t9_kind;
    for (i = 0; i < 64; i++) begin
        t9_eng  = i % 4;
        t9_kind = (i / 4) % 2;
        t9_idx  = 400 + (i / 8);
        va = make_va(t9_eng, t9_kind, t9_idx);
        do_write(t9_eng, va, {54'd0, 32'hABCD_0000} | 86'(i), rejected);
        check($sformatf("T9 i=%0d eng=%0d kind=%0b idx=%0d write ok",
                        i, t9_eng, t9_kind, t9_idx), !rejected);
    end
    for (i = 0; i < 64; i++) begin
        t9_eng  = i % 4;
        t9_kind = (i / 4) % 2;
        t9_idx  = 400 + (i / 8);
        va = make_va(t9_eng, t9_kind, t9_idx);
        do_read(t9_eng, va, rd_data, rejected);
        check($sformatf("T9 i=%0d read ok", i), !rejected);
        check($sformatf("T9 i=%0d data preserved", i),
              rd_data == ({54'd0, 32'hABCD_0000} | 86'(i)));
    end
end
```

```
        // T10: repeated update on same VAs

        $display("T10: 4 updates per VA on T2's 8 engine-0 slots");
        for (int round = 1; round <= 3; round++) begin
            for (i = 0; i < 8; i++) begin
                va = make_va(0, 1'b0, 10 + i);
                do_write(0, va, {54'd0, 32'(round * 256), 8'(i)}, rejected);
                check($sformatf("T10 round=%0d idx=%0d write ok", round, 10+i),
                      !rejected);
            end
        end
        for (i = 0; i < 8; i++) begin
            va = make_va(0, 1'b0, 10 + i);
            do_read(0, va, rd_data, rejected);
            check($sformatf("T10 read idx=%0d ok", 10+i), !rejected);
            check($sformatf("T10 idx=%0d final value matches", 10+i),
                  rd_data == {54'd0, 32'(3 * 256), 8'(i)});
        end

        $display("=== mmu_tb finished: %0d / %0d failed ===",
                 errors, tests);
        if (errors == 0) $display(">>> ALL TESTS PASSED <<<");
        else             $display(">>> %0d TESTS FAILED <<<", errors);
        $finish;
    end

    // safety timeout
    initial begin
        #20_000_000;
        $display("[FATAL] global timeout");
        $finish;
    end

endmodule
```

None

```
// symbol_engine_tb.sv: Testbench for symbol_engine
//
// Drives orders into one symbol_engine (ENGINE_ID=0) and verifies the
// trade-controller behavior: no trade on non-overlapping prices, exact
// matches, partial fills in both directions, and that the heap sizes
// reflect what the trade controller pops.
```

```
`timescale 1ns/1ps

module symbol_engine_tb;

    parameter int NODE_WIDTH = 86;
    parameter int RD_LATENCY = 4;

    // Clock and reset

    logic clk = 1'b0;
    logic rst_n;
    always #5 clk = ~clk;   // 100 MHz

    // DUT-facing signals

    logic                  order_in_valid;
    logic [31:0]           order_in_data;   // DISPATCH_ORDER: {type[1],
price[16], quantity[15]}
    logic                  order_in_ready;
    logic [31:0]           now_ts;          // free-running timestamp counter

    logic                  trade_out_valid;
    logic [NODE_WIDTH-1:0] trade_out_data;
    logic                  trade_out_ready;

    logic                  mmu_req_valid;
    logic [31:0]           mmu_req_va;
    logic                  mmu_req_wr;
    logic [NODE_WIDTH-1:0] mmu_req_wdata;
    logic                  mmu_req_ready;
    logic [NODE_WIDTH-1:0] mmu_resp_data;
    logic                  mmu_resp_valid;
    logic                  mmu_resp_reject;

    logic [13:0]           bid_size_o;
    logic [13:0]           ask_size_o;

    // DUT

    symbol_engine #(
        .ENGINE_ID  (0),
        .NODE_WIDTH (NODE_WIDTH),
        .SYMBOL     (21'd0)
    ) dut (
```

```
    .clk             (clk),
    .rst_n           (rst_n),
    .now_ts          (now_ts),
    .order_in_valid  (order_in_valid),
    .order_in_data   (order_in_data),
    .order_in_ready  (order_in_ready),
    .trade_out_valid (trade_out_valid),
    .trade_out_data  (trade_out_data),
    .trade_out_ready (trade_out_ready),
    .mmu_req_valid   (mmu_req_valid),
    .mmu_req_va      (mmu_req_va),
    .mmu_req_wr      (mmu_req_wr),
    .mmu_req_wdata   (mmu_req_wdata),
    .mmu_req_ready   (mmu_req_ready),
    .mmu_resp_data   (mmu_resp_data),
    .mmu_resp_valid  (mmu_resp_valid),
    .mmu_resp_reject (mmu_resp_reject),
    .bid_size_o      (bid_size_o),
    .ask_size_o      (ask_size_o)
);

// MMU stub needed encase future test touch virtual memory

assign mmu_req_ready   = 1'b1;
assign mmu_resp_reject = 1'b0;

localparam int MMU_DEPTH = 2048;
logic [NODE_WIDTH-1:0] mmu_mem [MMU_DEPTH];
logic                  rd_pending;
logic [3:0]            rd_counter;
logic [NODE_WIDTH-1:0] rd_data_q;

function automatic int mmu_idx (input logic [31:0] va);
    mmu_idx = va[10:0];
endfunction

always_ff @(posedge clk or negedge rst_n) begin
    if (!rst_n) begin
        rd_pending     <= 1'b0;
        rd_counter     <= '0;
        rd_data_q      <= '0;
        mmu_resp_valid <= 1'b0;
        mmu_resp_data  <= '0;
        for (int i = 0; i < MMU_DEPTH; i++) mmu_mem[i] <= '0;
    end else begin
```

```
            mmu_resp_valid <= 1'b0;

            if (mmu_req_valid && mmu_req_wr)
                mmu_mem[mmu_idx(mmu_req_va)] <= mmu_req_wdata;

            if (mmu_req_valid && !mmu_req_wr && !rd_pending) begin
                rd_pending <= 1'b1;
                rd_counter <= RD_LATENCY[3:0];
                rd_data_q  <= mmu_mem[mmu_idx(mmu_req_va)];
            end

            if (rd_pending) begin
                if (rd_counter > 0) begin
                    rd_counter <= rd_counter - 4'd1;
                end else begin
                    mmu_resp_valid <= 1'b1;
                    mmu_resp_data  <= rd_data_q;
                    rd_pending     <= 1'b0;
                end
            end
        end
    end

    // Trade-event capture

    int trade_count;
    logic [NODE_WIDTH-1:0] last_trade;

    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) begin
            trade_count <= 0;
            last_trade  <= '0;
        end else if (trade_out_valid && trade_out_ready) begin
            trade_count <= trade_count + 1;
            last_trade  <= trade_out_data;
            $display("  [trade] price=%0d amount=%0d (count now %0d)",
                     trade_out_data[84:69], trade_out_data[68:53], trade_count +
1);
        end
    end

    // DISPATCH_ORDER builder: {type[1], price[16], quantity[15]} MSB-first
    function automatic logic [31:0] build_dispatch (
        input logic [15:0] price,
        input logic [14:0] amount,
```

```
        input logic        type_bit
    );
        build_dispatch = {type_bit, price, amount};
    endfunction

    function automatic logic [15:0] np (input logic [NODE_WIDTH-1:0] n);
        np = n[84:69];
    endfunction
    function automatic logic [15:0] na (input logic [NODE_WIDTH-1:0] n);
        na = n[68:53];
    endfunction



    int errors = 0;

    task automatic submit_order (
        input logic [15:0] price,
        input logic [15:0] amount,    // upper bit dropped; quantity is 15 bits
        input logic        is_ask
    );
        @(posedge clk);
        while (!order_in_ready) @(posedge clk);
        order_in_valid <= 1'b1;
        order_in_data  <= build_dispatch(price, amount[14:0], !is_ask);
        @(posedge clk);
        order_in_valid <= 1'b0;
        order_in_data  <= '0;
        // Block until the trade controller is back to idle so the next
        // submit_order doesn't race the cascade trade loop.
        @(posedge clk);
        while (!order_in_ready) @(posedge clk);
    endtask

    task automatic check (input string label, input integer got, input integer
expected);
        if (got !== expected) begin
            $display("[FAIL] %s: got=%0d expected=%0d", label, got, expected);
            errors++;
        end else begin
            $display("[ OK ] %s = %0d", label, got);
        end
    endtask
```

```
    // Test sequence

    // free-running timestamp counter
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) now_ts <= 32'd0;
        else        now_ts <= now_ts + 32'd1;
    end

    initial begin
        $display("=== symbol_engine_tb starting ===");

        order_in_valid  = 1'b0;
        order_in_data   = '0;
        trade_out_ready = 1'b1;

        rst_n = 1'b0;
        repeat (4) @(posedge clk);
        rst_n = 1'b1;
        @(posedge clk);

        // Test 1: lone bid does not trade
        $display("-- T1: lone bid 100 x 5");
        submit_order(16'd100, 16'd5, 1'b0);
        check("trades after lone bid", trade_count, 0);
        check("bid size after lone bid", bid_size_o, 1);
        check("ask size after lone bid", ask_size_o, 0);

        // Test 2: ask above best bid does not trade
        $display("-- T2: ask 250 x 5 (does not match bid 100)");
        submit_order(16'd250, 16'd5, 1'b1);
        check("trades after non-matching ask", trade_count, 0);
        check("ask size after non-matching ask", ask_size_o, 1);

        // Test 3: bid that matches the best ask exactly
        $display("-- T3: bid 250 x 5 (exact match with ask 250 x 5)");
        submit_order(16'd250, 16'd5, 1'b0);
        check("trades after exact match", trade_count, 1);
        check("trade price",  np(last_trade), 250);
        check("trade amount", na(last_trade), 5);
        check("bid size after exact match",  bid_size_o, 1);   // bid 100 still
in book
        check("ask size after exact match",  ask_size_o, 0);

        // Test 4: ask above best bid - still no match
        $display("-- T4: ask 300 x 8 (above best bid 100)");
```

```
    submit_order(16'd300, 16'd8, 1'b1);
    check("trades after high ask", trade_count, 1);
    check("ask size", ask_size_o, 1);

    // Test 5: bid larger than the ask amount - partial fill, bid remains
    $display("-- T5: bid 300 x 12 (matches ask 300 x 8, bid remains x 4)");
    submit_order(16'd300, 16'd12, 1'b0);
    check("trades after partial bid", trade_count, 2);
    check("partial trade price",  np(last_trade), 300);
    check("partial trade amount", na(last_trade), 8);
    check("bid size after partial",  bid_size_o, 2);   // bid 100 + bid 300x4
    check("ask size after partial",  ask_size_o, 0);

    // Test 6: ask smaller than the leftover bid - exact-fill cleanup
    $display("-- T6: ask 300 x 4 (matches leftover bid 300 x 4 exactly)");
    submit_order(16'd300, 16'd4, 1'b1);
    check("trades after cleanup", trade_count, 3);
    check("cleanup trade amount", na(last_trade), 4);
    check("bid size after cleanup", bid_size_o, 1);   // only bid 100 left
    check("ask size after cleanup", ask_size_o, 0);

    // Test 7: ask smaller than bid amount - partial fill, ask remains
    $display("-- T7: bid 200 x 3, then ask 200 x 5 (ask remains x 2)");
    submit_order(16'd200, 16'd3, 1'b0);   // bid 200 x 3, no ask, no trade
    check("trades after lone bid 200", trade_count, 3);
    submit_order(16'd200, 16'd5, 1'b1);   // ask 200 x 5, partial fill
    check("trades after partial ask", trade_count, 4);
    check("partial-ask trade amount", na(last_trade), 3);
    check("ask size after partial ask", ask_size_o, 1);

    $display("=== symbol_engine_tb finished: %0d error(s) ===", errors);
    if (errors == 0) $display(">>> ALL TESTS PASSED <<<");
    else             $display(">>> TESTS FAILED <<<");

    $finish;
end

// Watchdog

initial begin
    #500000;
    $display("[FAIL] symbol_engine_tb timed out");
    $finish;
end
```

```
endmodule
```

None

```
// system_tb.sv
//
// Wires together everything top.sv wires except the order_dispatcher and
// the real MMU. Drives orders directly into each of the 8 symbol_engine
// instances and uses a per-port MMU stub
//


`timescale 1ns/1ps

module system_tb;

    localparam int N               = 8;
    localparam int NODE_WIDTH      = 86;
    localparam int RD_LATENCY      = 4;
    localparam int TRADE_LOG_DEPTH = 16;
    localparam int LOG_AW          = $clog2(TRADE_LOG_DEPTH);
    localparam int LOG_CW          = $clog2(TRADE_LOG_DEPTH + 1);

    logic clk = 1'b0;
    logic rst_n;
    always #5 clk = ~clk;

    // Free-running timestamp shared by all engines

    logic [31:0] now_ts;
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) now_ts <= 32'd0;
        else        now_ts <= now_ts + 32'd1;
    end

    // Per-engine bus

    logic                  order_in_valid [N];
    logic [31:0]           order_in_data  [N];
    logic                  order_in_ready [N];

    logic [N-1:0]          eng_trade_valid;
    logic [NODE_WIDTH-1:0] eng_trade_data  [N];
    logic [N-1:0]          eng_trade_ready;
```

```
logic                  mmu_req_valid   [N];
logic [31:0]           mmu_req_va      [N];
logic                  mmu_req_wr      [N];
logic [NODE_WIDTH-1:0] mmu_req_wdata   [N];
logic                  mmu_req_ready   [N];
logic [NODE_WIDTH-1:0] mmu_resp_data   [N];
logic                  mmu_resp_valid  [N];
logic                  mmu_resp_reject [N];

logic [13:0]           bid_size [N];
logic [13:0]           ask_size [N];

// 8 symbol engines

genvar e;
generate
    for (e = 0; e < N; e++) begin : g_engines
        symbol_engine #(.ENGINE_ID(e)) u_engine (
            .clk             (clk),
            .rst_n           (rst_n),
            .now_ts          (now_ts),
            .order_in_valid  (order_in_valid[e]),
            .order_in_data   (order_in_data[e]),
            .order_in_ready  (order_in_ready[e]),
            .trade_out_valid (eng_trade_valid[e]),
            .trade_out_data  (eng_trade_data[e]),
            .trade_out_ready (eng_trade_ready[e]),
            .mmu_req_valid   (mmu_req_valid[e]),
            .mmu_req_va      (mmu_req_va[e]),
            .mmu_req_wr      (mmu_req_wr[e]),
            .mmu_req_wdata   (mmu_req_wdata[e]),
            .mmu_req_ready   (mmu_req_ready[e]),
            .mmu_resp_data   (mmu_resp_data[e]),
            .mmu_resp_valid  (mmu_resp_valid[e]),
            .mmu_resp_reject (mmu_resp_reject[e]),
            .bid_size_o      (bid_size[e]),
            .ask_size_o      (ask_size[e])
        );
    end
endgenerate

// Trade aggregator + log

logic                  agg_trade_valid;
```

```
logic [NODE_WIDTH-1:0] agg_trade_data;
logic                  agg_trade_ready;

trade_aggregator #(.N(N), .NODE_WIDTH(NODE_WIDTH)) u_agg (
    .clk             (clk),
    .rst_n           (rst_n),
    .eng_trade_valid (eng_trade_valid),
    .eng_trade_data  (eng_trade_data),
    .eng_trade_ready (eng_trade_ready),
    .trade_out_valid (agg_trade_valid),
    .trade_out_data  (agg_trade_data),
    .trade_out_ready (agg_trade_ready)
);

logic                  sw_re;
logic [LOG_AW-1:0]     sw_addr;
logic [NODE_WIDTH-1:0] sw_rdata;
logic [LOG_CW-1:0]     sw_count;
logic                  sw_overflow;
logic                  sw_clear;

trade_log #(.NODE_WIDTH(NODE_WIDTH), .LOG_DEPTH(TRADE_LOG_DEPTH)) u_log (
    .clk            (clk),
    .rst_n          (rst_n),
    .trade_in_valid (agg_trade_valid),
    .trade_in_data  (agg_trade_data),
    .trade_in_ready (agg_trade_ready),
    .sw_re          (sw_re),
    .sw_addr        (sw_addr),
    .sw_rdata       (sw_rdata),
    .sw_count       (sw_count),
    .sw_overflow    (sw_overflow),
    .sw_clear       (sw_clear)
);

// Per-port MMU stub. One memory per engine, hashed by va[10:0];

localparam int MMU_DEPTH = 2048;

logic [NODE_WIDTH-1:0] mmu_mem [N][MMU_DEPTH];
logic                  rd_pending [N];
logic [3:0]            rd_counter [N];
logic [NODE_WIDTH-1:0] rd_data_q  [N];

function automatic int mmu_idx (input logic [31:0] va);
```

```
        mmu_idx = va[10:0];
    endfunction

    genvar p;
    generate
        for (p = 0; p < N; p++) begin : g_mmu_stub
            assign mmu_req_ready[p]   = 1'b1;
            assign mmu_resp_reject[p] = 1'b0;

            always_ff @(posedge clk or negedge rst_n) begin
                if (!rst_n) begin
                    rd_pending[p]      <= 1'b0;
                    rd_counter[p]      <= '0;
                    rd_data_q[p]       <= '0;
                    mmu_resp_valid[p]  <= 1'b0;
                    mmu_resp_data[p]   <= '0;
                    for (int i = 0; i < MMU_DEPTH; i++) mmu_mem[p][i] <= '0;
                end else begin
                    mmu_resp_valid[p] <= 1'b0;

                    if (mmu_req_valid[p] && mmu_req_wr[p]) begin
                        mmu_mem[p][mmu_idx(mmu_req_va[p])] <= mmu_req_wdata[p];
                        mmu_resp_valid[p] <= 1'b1;
                        mmu_resp_data[p]  <= '0;
                    end

                    if (mmu_req_valid[p] && !mmu_req_wr[p] && !rd_pending[p])
begin
                        rd_pending[p] <= 1'b1;
                        rd_counter[p] <= RD_LATENCY[3:0];
                        rd_data_q[p]  <= mmu_mem[p][mmu_idx(mmu_req_va[p])];
                    end

                    if (rd_pending[p]) begin
                        if (rd_counter[p] > 0) begin
                            rd_counter[p] <= rd_counter[p] - 4'd1;
                        end else begin
                            mmu_resp_valid[p] <= 1'b1;
                            mmu_resp_data[p]  <= rd_data_q[p];
                            rd_pending[p]     <= 1'b0;
                        end
                    end
                end
            end
        end
```

```
    endgenerate

    // Trade-event capture (counts every accepted trade out of the aggregator)

    int agg_count;
    always_ff @(posedge clk or negedge rst_n) begin
        if (!rst_n) agg_count <= 0;
        else if (agg_trade_valid && agg_trade_ready) agg_count <= agg_count + 1;
    end

    // Symbols matching symbol_engine's defaults for ENGINE_ID 0..7

    function automatic logic [20:0] expected_symbol (input int eng);
        unique case (eng)
            0: expected_symbol = {7'd65, 7'd80, 7'd76};   // APL
            1: expected_symbol = {7'd66, 7'd83, 7'd88};   // BSX
            2: expected_symbol = {7'd66, 7'd85, 7'd83};   // BUS
            3: expected_symbol = {7'd77, 7'd77, 7'd77};   // MMM
            4: expected_symbol = {7'd83, 7'd70, 7'd84};   // SFT
            5: expected_symbol = {7'd66, 7'd85, 7'd88};   // BUX
            6: expected_symbol = {7'd84, 7'd85, 7'd83};   // TUS
            7: expected_symbol = {7'd87, 7'd77, 7'd84};   // WMT
            default: expected_symbol = 21'd0;
        endcase
    endfunction

    // Stimulus tasks

    int errors = 0;

    task automatic check (input string label, input integer got, input integer
expected);
        if (got !== expected) begin
            $display("[FAIL] %s: got=%0d expected=%0d", label, got, expected);
            errors++;
        end else begin
            $display("[ OK ] %s = %0d", label, got);
        end
    endtask

    task automatic submit_order (
        input int          eng,
        input logic [15:0] price,
        input logic [15:0] amount,
        input logic        is_ask
```

```
    );
        @(posedge clk);
        while (!order_in_ready[eng]) @(posedge clk);
        order_in_valid[eng] <= 1'b1;
        order_in_data[eng]  <= {is_ask, price, amount[14:0]};
        @(posedge clk);
        order_in_valid[eng] <= 1'b0;
        order_in_data[eng]  <= '0;
        @(posedge clk);
        while (!order_in_ready[eng]) @(posedge clk);
    endtask

    // Submit but don't wait for the engine to return to idle. Used for the
overflow test
    task automatic submit_order_no_wait (
        input int          eng,
        input logic [15:0] price,
        input logic [15:0] amount,
        input logic        is_ask
    );
        @(posedge clk);
        while (!order_in_ready[eng]) @(posedge clk);
        order_in_valid[eng] <= 1'b1;
        order_in_data[eng]  <= {is_ask, price, amount[14:0]};
        @(posedge clk);
        order_in_valid[eng] <= 1'b0;
        order_in_data[eng]  <= '0;
    endtask

    task automatic sw_pop (input int addr, output logic [NODE_WIDTH-1:0] data);
        @(posedge clk);
        sw_re   <= 1'b1;
        sw_addr <= addr[LOG_AW-1:0];
        @(posedge clk);
        sw_re   <= 1'b0;
        @(posedge clk);
        data = sw_rdata;
    endtask

    task automatic sw_clear_log;
        @(posedge clk);
        sw_clear <= 1'b1;
        @(posedge clk);
        sw_clear <= 1'b0;
        @(posedge clk);
```

```
endtask

// Test sequence

initial begin
    logic [NODE_WIDTH-1:0] trade;
    logic [20:0]           sym;
    int                    captured_count;

    $display("=== system_tb starting (N=%0d, LOG_DEPTH=%0d) ===",
             N, TRADE_LOG_DEPTH);

    for (int i = 0; i < N; i++) begin
        order_in_valid[i] = 1'b0;
        order_in_data[i]  = '0;
    end
    sw_re    = 1'b0;
    sw_addr  = '0;
    sw_clear = 1'b0;

    rst_n = 1'b0;
    repeat (4) @(posedge clk);
    rst_n = 1'b1;
    @(posedge clk);

    check("agg count after reset", agg_count, 0);
    check("log count after reset", sw_count, 0);

    // T1: lone bid into engine 0, then matching ask. One trade total.
    $display("-- T1: engine 0 bid 100x5 then ask 100x5");
    submit_order(0, 16'd100, 16'd5, 1'b0);
    submit_order(0, 16'd100, 16'd5, 1'b1);
    repeat (20) @(posedge clk);
    check("trades after T1", agg_count, 1);
    check("log count after T1", sw_count, 1);

    sw_pop(0, trade);
    sym = trade[52:32];
    check("T1 trade symbol",  sym,           expected_symbol(0));
    check("T1 trade price",   trade[84:69],  100);
    check("T1 trade amount",  trade[68:53],  5);

    sw_clear_log;
    check("log count after clear", sw_count, 0);
```

```
        // T2: drive a matching pair into each of the 8 engines back-to-back.

        $display("-- T2: matched pair to all 8 engines");
        for (int i = 0; i < N; i++) begin
            submit_order(i, 16'(200 + i*10), 16'(i + 1), 1'b0); // bid
        end
        for (int i = 0; i < N; i++) begin
            submit_order(i, 16'(200 + i*10), 16'(i + 1), 1'b1); // ask
        end

        repeat (200) @(posedge clk);
        check("trades after T2", agg_count, N + 1); // 1 from T1 wasn't cleared
from agg
        check("log count after T2", sw_count, N);

        captured_count = 0;
        for (int i = 0; i < N; i++) begin
            sw_pop(i, trade);
            sym = trade[52:32];
            if (trade[84:69] != 0) captured_count++;
        end
        check("captured trades from log", captured_count, N);

        sw_clear_log;

        // T3: fill the trade_log via engine 0, then trigger overflow.

        $display("-- T3: fill + overflow trade_log via engine 0");
        for (int i = 0; i < TRADE_LOG_DEPTH; i++) begin
            submit_order(0, 16'(300 + i), 16'd1, 1'b0);
            submit_order(0, 16'(300 + i), 16'd1, 1'b1);
        end
        repeat (50) @(posedge clk);
        check("log count when full",   sw_count,    TRADE_LOG_DEPTH);
        check("overflow not yet",      sw_overflow, 0);

        submit_order_no_wait(0, 16'd400, 16'd1, 1'b0);
        submit_order_no_wait(0, 16'd400, 16'd1, 1'b1);
        repeat (300) @(posedge clk);
        check("overflow latched",      sw_overflow, 1);

        sw_clear_log;
        repeat (50) @(posedge clk);
        check("overflow after clear",  sw_overflow, 0);
```

```
        $display("=== system_tb finished: %0d error(s) ===", errors);
        if (errors == 0) $display(">>> ALL TESTS PASSED <<<");
        else             $display(">>> TESTS FAILED <<<");

        $finish;
    end

    // Watchdog

    initial begin
        #2000000;
        $display("[FAIL] system_tb timed out");
        $finish;
    end


endmodule
```

None

```
// trade_log_tb.sv :

// Pushes a small number of trade events, reads them back through the
// SW port, verifies count, exercises the full -> overflow -> clear path.

`timescale 1ns/1ps

module trade_log_tb;

    parameter int NODE_WIDTH = 86;
    parameter int LOG_DEPTH  = 8;       // shrunk for fast overflow
    localparam int CNT_WIDTH  = $clog2(LOG_DEPTH + 1);
    localparam int ADDR_WIDTH = $clog2(LOG_DEPTH);

    logic clk = 1'b0;
    logic rst_n;
    always #5 clk = ~clk;

    logic                  trade_in_valid;
    logic [NODE_WIDTH-1:0] trade_in_data;
    logic                  trade_in_ready;

    logic                  sw_re;
    logic [ADDR_WIDTH-1:0] sw_addr;
```

```
    logic [NODE_WIDTH-1:0] sw_rdata;
    logic [CNT_WIDTH-1:0]  sw_count;
    logic                  sw_overflow;
    logic                  sw_clear;

    trade_log #(
        .NODE_WIDTH (NODE_WIDTH),
        .LOG_DEPTH  (LOG_DEPTH)
    ) dut (
        .clk            (clk),
        .rst_n          (rst_n),
        .trade_in_valid (trade_in_valid),
        .trade_in_data  (trade_in_data),
        .trade_in_ready (trade_in_ready),
        .sw_re          (sw_re),
        .sw_addr        (sw_addr),
        .sw_rdata       (sw_rdata),
        .sw_count       (sw_count),
        .sw_overflow    (sw_overflow),
        .sw_clear       (sw_clear)
    );

    int errors = 0;

    task automatic check (input string label, input integer got, input integer
expected);
        if (got !== expected) begin
            $display("[FAIL] %s: got=%0d expected=%0d", label, got, expected);
            errors++;
        end else begin
            $display("[ OK ] %s = %0d", label, got);
        end
    endtask

    task automatic push_trade (input logic [NODE_WIDTH-1:0] data);
        @(posedge clk);
        trade_in_valid <= 1'b1;
        trade_in_data  <= data;
        @(posedge clk);
        trade_in_valid <= 1'b0;
    endtask

    task automatic read_trade (input int addr, output logic [NODE_WIDTH-1:0]
data);
        @(posedge clk);
```

```
    sw_re   <= 1'b1;
    sw_addr <= addr[ADDR_WIDTH-1:0];
    @(posedge clk);
    sw_re   <= 1'b0;
    @(posedge clk);
    data = sw_rdata;
endtask

initial begin
    logic [NODE_WIDTH-1:0] d;

    $display("=== trade_log_tb starting (LOG_DEPTH=%0d) ===", LOG_DEPTH);

    trade_in_valid = 1'b0;
    trade_in_data  = '0;
    sw_re          = 1'b0;
    sw_addr        = '0;
    sw_clear       = 1'b0;

    rst_n = 1'b0;
    repeat (4) @(posedge clk);
    rst_n = 1'b1;
    @(posedge clk);

    check("ready after reset",    trade_in_ready, 1);
    check("count after reset",    sw_count,       0);
    check("overflow after reset", sw_overflow,    0);

    // Push 3 trades, verify count, read them back
    push_trade(86'h11111111111111111111);
    push_trade(86'h22222222222222222222);
    push_trade(86'h33333333333333333333);
    @(posedge clk);
    check("count after 3 pushes", sw_count, 3);

    read_trade(0, d);
    check("read[0] low bits",  d[31:0], 32'h11111111);
    read_trade(1, d);
    check("read[1] low bits",  d[31:0], 32'h22222222);
    read_trade(2, d);
    check("read[2] low bits",  d[31:0], 32'h33333333);

    // Fill the remaining slots
    push_trade(86'h44444444444444444444);
    push_trade(86'h55555555555555555555);
```

```
        push_trade(86'h66666666666666666666);
        push_trade(86'h77777777777777777777);
        push_trade(86'h88888888888888888888);
        @(posedge clk);
        check("count when full",      sw_count,       LOG_DEPTH);
        check("ready when full",      trade_in_ready, 0);
        check("overflow before drop", sw_overflow,    0);

        // Try to push when full
        push_trade(86'h99999999999999999999);
        @(posedge clk);
        check("count after dropped",  sw_count,    LOG_DEPTH);
        check("overflow after drop",  sw_overflow, 1);

        // Clear
        @(posedge clk);
        sw_clear <= 1'b1;
        @(posedge clk);
        sw_clear <= 1'b0;
        @(posedge clk);
        check("count after clear",    sw_count,    0);
        check("overflow after clear", sw_overflow, 0);
        check("ready after clear",    trade_in_ready, 1);

        // Push fresh data after clear, verify it lands at slot 0
        push_trade(86'hAAAAAAAAAAAAAAAAAAAA);
        @(posedge clk);
        check("count after post-clear push", sw_count, 1);
        read_trade(0, d);
        check("post-clear read[0]", d[31:0], 32'hAAAAAAAA);

        $display("=== trade_log_tb finished: %0d error(s) ===", errors);
        if (errors == 0) $display(">>> ALL TESTS PASSED <<<");
        else             $display(">>> TESTS FAILED <<<");

        $finish;
    end

    initial begin
        #50000;
        $display("[FAIL] trade_log_tb timed out");
        $finish;
    end

endmodule
```