ChatGPT

Designing an MSI-Coherent Single-Level Cache System in Logisim for a 32-bit RISC-V CPU

Cache Component Description and Interface

Each core in our system has a **single-level cache** acting as a fast local memory buffer between the CPU and main memory. The cache stores recently used 32-bit words (or blocks of words) so the CPU can access them with low latency. In a multi-core setup, each core has its **own cache** (private L1 cache), which necessitates a coherence protocol to keep data consistent across caches.

Interface and Signals: The cache component in Logisim will resemble a memory unit with added control signals for coherence. From the CPU's perspective, it provides a memory-like interface: - An **address input** (32-bit, for RV32I addressing). - A **data input** (for store operations) and **data output** (for load operations). - Control lines for **Read/Write** request and perhaps a **valid/ready** signal to handle stalls on misses. (For simplicity, one can assume the CPU waits if the cache is busy). - Optionally, a **hit/miss indicator** or **stall line** can be included to tell the CPU when a memory operation is delayed (miss) versus serviced immediately (hit).

Role in the System: The cache intercepts CPU memory requests. On a **cache hit**, it returns the data (for loads) or updates the cache (for stores) without going to main memory. On a **cache miss**, it initiates a fetch from main memory (and possibly notifies other caches via the coherence bus). Because each core may cache the same memory blocks, the cache also snoops on a shared bus to listen for other cores' memory operations. This way, if another core writes to a location that this cache has, it can update or invalidate its own copy to maintain coherence. In summary, the cache improves performance by caching data, while the coherence mechanism ensures all caches see a **consistent memory state** [1] [2].

Each cache is designed as a **write-back, write-invalidate cache** using the MSI coherence protocol (Modified, Shared, Invalid states). "Write-back" means when the CPU modifies data in the cache, we do not immediately write to main memory – instead, we mark the cache line as dirty (Modified) and write it back when it's evicted or another core needs it. "Write-invalidate" means if one core wants to write to a shared cache line, it will invalidate other cores' copies (via the bus) before writing [3] [2]. This fits the MSI protocol, where a core gains **exclusive access** to a cache line before writing.

Cache Implementation Details (Memory Layout, Tags, and State Bits)

Cache Memory Organization: For simplicity, we can design a **direct-mapped cache**, though the approach can be extended to associative caches. In a direct-mapped cache, each memory block maps to exactly one cache line. The cache consists of an array of entries (lines), each entry storing: - A **valid bit/state** (which in MSI will be part of the state bits). - **Coherence state bits** (2 bits to encode M, S, or I state). - A **tag** (the high-order address bits identifying which memory block is in this line). - The **data block** (one or more words).

Address breakdown: A 32-bit address is divided into fields for cache indexing: - **Offset bits:** Select the byte/word within a cache line (depending on block size). If the cache block size is B bytes, then `offset = log2(B)` bits. - **Index bits:** Select the cache line index. If the cache has N lines, `index = log2(N)` bits. - **Tag bits:** The remaining upper bits of the address serve as the tag, which identifies the memory block. Tag size = 32 − (index bits + offset bits).

For example, if we use a 16-line cache with 16 bytes per line, we have offset = 4 bits (16B line), index = 4 bits (16 lines), and tag = 32 − 8 = 24 bits for the tag. The cache would then have 16 entries, each storing a 24-bit tag, 2-bit state, and 16 bytes of data.

Tag and Data Storage: In Logisim, we can implement the cache storage using the built-in memory components: - Use a small **RAM** (or Register Array) for the **data blocks**. For the above example, this could be a memory with 16 addresses (index as address lines), each storing 16 bytes (128 bits) of data. The memory's data width can be set to the block size (e.g. 128 bits) to retrieve the whole block. - Use another small RAM (or combine with the above) for **tag and state**. One approach is to have a separate memory (16 addresses) where each entry stores the tag plus coherence state bits. For instance, we could allocate a 32-bit wide memory for tag+state: the upper 24 bits store the tag, and the lower 2 bits store the MSI state (that leaves some unused bits in a 32-bit word, which is fine). Alternatively, two distinct memories can be used: one for tags and one for states, each indexed by the cache line number.

State management per line: We use 2 bits per cache line to represent the MSI state: - `00 = Invalid (I)` - `01 = Shared (S)` - `10 = Modified (M)` - (`11` unused, but could be reserved for a potential future state like Exclusive in MESI).

At reset, all lines start in the **Invalid** state (meaning they contain no valid data). A **Valid bit** is effectively implied by state $\neq$ Invalid – if state is I, the line is not valid. (In MSI, the "Invalid" state means the line *is not in the cache* or not valid [4].) When a line is loaded from memory into the cache, we set its state to S (if shared with others) or M (if this core will modify it exclusively). If a line in M state is evicted (replaced), the cache must write back the data to main memory (because it's dirty) before marking it invalid [5]. In Logisim, this write-back can be triggered by the cache controller (more on this in the coherence logic section).

Tag comparison logic: The cache includes comparators to check, on each CPU memory access, whether the addressed block is present. The index field of the address selects a cache line; the stored tag in that line is compared with the address's tag bits: - If the tag matches **and** the state is not Invalid, it's a **cache hit**. - If the tag doesn't match, or the line's state is Invalid, it's a **miss** (or an invalid entry).

On a hit, the cache can proceed to service the request (with some coherence considerations if it's a write – see next section). On a miss, the cache will initiate a memory fetch via the shared bus.

MSI Coherence Protocol: State Transitions and Bus Snooping Logic

We implement the **MSI coherence protocol** to ensure consistency across caches. Each cache line transitions between **Modified (M)**, **Shared (S)**, and **Invalid (I)** states based on two kinds of events: 1. **Processor requests** from its own core (CPU reads or writes to that memory block). 2. **Bus snooping events** caused by other cores' actions on the shared bus (other cores' reads/writes that involve this cache's block).

MSI State Meanings Recap: - **Invalid (I):** The cache line is not valid. It contains no useful data (or data that is not coherent). On any access, it's a miss and the data must be fetched from memory (or another cache) [4]. - **Shared (S):** The cache line holds a valid copy of data, identical to main memory. It may be present in other caches as well (shared read-only). No cache has modified this data. Reads are allowed; writes require an upgrade to exclusive access [6]. - **Modified (M):** The cache line holds the only up-to-date copy of the data, which has been modified locally. Main memory is stale (not updated yet). The line is exclusively owned by this cache – no other cache has it (others would be in I state for this block). Before any other core can take this block, this cache must write it back to memory (or supply it to the requesting core) [5] [2].

Processor request transitions (within the cache controller):

- **CPU Read (PrRd):**
- If the line is in **Modified (M)** or **Shared (S)**, it's a hit. The cache supplies the data to the CPU. State stays the same (M stays M, S stays S) [7].
- If the line is **Invalid (I)** (miss): The cache issues a **BusRd** (bus read) request on the shared bus [8]. This bus transaction will cause:
    - All other caches to snoop this address. Any cache with the line in M must **flush** (write back) the data and downgrade to S (or I), and any cache with it in S will detect the read but can keep it shared [9] [10].
    - Memory (or a cache that had M) will provide the data. The requesting cache then stores the block, and because it was a read request, it will mark the line **Shared (S)**. (It's shared because other caches might also have it or can cache it now; if no other cache had it, we still conservatively mark S in MSI protocol). The state transitions from I to S on PrRd/BusRd [11].
- **CPU Write (PrWr):**
- If the line is in **Modified (M)**, it's already exclusively held by this cache. The CPU can write the data immediately in the cache (hit), update the byte/word in the cache, and the state remains M [12]. No other cache has this block (by invariant of M state), so no bus action is needed on a write hit to M.
- If the line is **Shared (S)** (the cache has a clean copy that may exist in others): To get exclusive ownership for writing, this cache must **invalidate other caches' copies**. It issues a **BusUpgr** (bus upgrade) request [13] [10]. BusUpgr is a coherence signal meaning "I have this block cached and intend to write to it, so *invalidate all other shared copies*." Other caches snoop this and any that have the block in S will transition to I (invalid). No data transfer is needed because the writer already has the up-to-date data. The local cache then transitions its state from S to **Modified (M)** (and writes the data). After this, it has exclusive, modified copy [10]. *(If the MSI design doesn't explicitly implement BusUpgr, one can achieve the same effect by issuing a BusRdX (read-for-exclusive) even on a write hit, but BusUpgr is an optimization for write hits.)*
- If the line is **Invalid (I)** (write miss): The cache must fetch the block with the intent to modify. It issues a **BusRdX** (read with intent to exclusive) on the bus [8]. BusRdX means "I want the block and will modify it, so supply me the data and invalidate others' copies." This triggers snooping:
    - If another cache has the block in **Modified**, it will flush the data to the bus (or memory) and transition to Invalid (since the requester will own it) [9].
    - If another cache has it in **Shared**, it will simply invalidate that line (go to I) [14].

    - Memory (if no cache had a modified copy) supplies the data. The requesting cache receives the block and marks it **Modified (M)** (owner and writer). The state goes I → M on a write miss (BusRdX) [8]. The CPU's write is then performed in the cache.

Bus snooping and external requests: Each cache includes logic to **monitor the shared bus** for transactions from other cores. In MSI, caches snoop two main types of bus transactions (from other caches): - **BusRd (Read request)**: Another core is reading a block. If a cache snoops a BusRd for an address that it holds: - If its state for that address is **Modified (M)**, this means it has the most recent data while memory is stale. It must **flush** the data to the bus (so memory or the requester can take the updated data) and downgrade its state to **Shared (S)** [9]. (After flushing, the data is now consistent in memory and shared, not exclusive to that cache.) The cache remains in S (it still has a valid copy, now consistent with memory). - If its state is **Shared (S)**, it means it also has a clean copy. It can simply remain in S. (The data is already consistent with memory, so no action needed except perhaps noting that it's still shared) [10]. - If its state is **Invalid (I)**, it doesn't have the block, so it does nothing. (No copy to provide or invalidate.)

- **BusRdX (Read-for-Exclusive request)**: Another core wants to read and then modify a block (or just invalidate others for an upcoming write). When a cache snoops BusRdX for an address:
- If it has that address in **M** or **S**, it **must invalidate its copy** because another core is taking ownership to modify. That is, it will set its state to **Invalid (I)**. If it was in M, it also flushes the data to memory/bus before invalidating [9] [14]. If it was in S, it just invalidates (since memory has an up-to-date copy or the requester will provide the new data after write).

- If it is in **I**, it again does nothing (already invalid).

- **BusUpgr (Invalidate request)**: Another core, which already has a Shared copy, signals that it's going to write to that block. Snooping a BusUpgr is similar to BusRdX for other caches:

- If a cache has the block in **Shared (S)** and sees BusUpgr on the bus, it must **invalidate** its copy (S → I) [14]. (No data transfer is needed because the upgrader already has the data.)

- If a cache has it in M (which shouldn't happen, as only one cache can have M and it wouldn't issue BusUpgr), or I, no action (M case not applicable in MSI as explained [15]).

- **Flush/Write-back:** This is not a request but an action – when a cache in M state responds to a BusRd or BusRdX by writing the data back, it places the data on the bus (and/or updates main memory). We may use a control line or bus signal to indicate a **flush** in Logisim. During a flush, the memory will be updated with the new data. After flushing, the cache can downgrade state (M→S or M→I as above). Main memory now has the latest data, which can be supplied to the requesting core if needed [9].

The MSI protocol thus ensures **coherence invariants**: at most one cache can have a block in Modified state (exclusive ownership) and if any cache has it Modified, no others have a valid copy. Shared copies are always consistent with memory. Any write invalidates other copies [16] [1]. These rules guarantee that a read returns the latest written value and that writes are seen in the correct order system-wide [17] [1].

Bus Architecture and Multi-Core Interconnection

To support coherence, all caches are connected via a **common shared bus** for memory transactions and snooping. The bus is essentially a set of wires for address, data, and control signals that all caches (and the main memory) can observe. Only one device at a time may drive the bus (to avoid electrical conflict), but all may listen. We use a **snoop-based bus protocol**: when a cache needs to perform a memory operation that could affect others, it broadcasts a request on the bus, and all other caches snoop (monitor) these bus signals to react accordingly [18] [19].

Illustration: Multi-core system with each CPU having a private L1 cache, all connected to a shared bus and a common main memory. Coherence is maintained by having caches snoop on the bus for other cores' memory operations (e.g., invalidating their copies when another core writes).

Bus Signals: We will design the bus with the following main signals: - **Bus Address (ADDR):** (32 bits) carries the address of the memory operation. When a cache initiates a transaction (BusRd/BusRdX), it places the target address on this bus. All caches latch this address to check if it matches a line they hold. - **Bus Command (CMD):** indicates the type of operation. We can encode this as a few bits: - For example: `00 = No request (idle)`, `01 = BusRd (read)`, `10 = BusRdX (read for exclusive)`, `11 = BusUpgr (invalidate)`. We could also include a code for `Flush` or write-back, or use a separate control line for flush actions. - **Bus Data:** used to transfer data during read and write-back operations. The width might be 32 bits (a word) or the full line width. A simple approach is to use 32 bits and transfer one word at a time (multiple cycles for a whole line), or use a wider bus equal to the cache line size for single-cycle block transfer (Logisim allows wide wires). - **Control/Handshake signals:** We might include a **Memory Ready** or **Acknowledgment** signal to coordinate data transfer. But in a simple Logisim simulation, it may be enough to assume the memory responds in fixed time and the caches "wait" accordingly.

Bus Wiring in Logisim: All caches and the main memory are connected to the same shared Bus lines. In Logisim, this can be accomplished by using **tri-state buffers (controlled buffers)** or a multiplexer to connect multiple outputs to a single wire without conflict. Each cache's bus interface will drive the bus lines only when it has been granted control; otherwise, it outputs high-impedance (Z). For example, we put a tri-state buffer on each cache's address output line leading to the shared address bus. Only the cache whose turn it is will enable its buffer (thus placing an address on the bus); all others will have their buffers off (outputting nothing, i.e., Z) [20]. The same applies for the data bus: caches only drive it when supplying a flush (write-back) – otherwise they release it. The memory unit can also be tied to these lines via tri-state: it drives data onto the bus when responding to a read, and it reads from the bus when a cache flushes data.

Because only one device drives the bus at a time, we need **bus arbitration**. A simple scheme is to have a round-robin or fixed priority arbiter that grants the bus to a requesting core. In a Logisim simulation, arbitration can be abstracted or even manual (for example, step one core at a time). For instructional purposes, you might incorporate a small arbiter circuit: - Caches raise a **Bus Request** line when they have a pending miss or write that needs bus access. - An arbiter (could be a priority encoder or a simple fixed-priority selector) grants one cache the bus by enabling that cache's tri-state buffers and issuing a **Bus Grant** signal. Only that cache then drives the address and command lines. - After the bus transaction completes (data transferred and snooped), the grant is released or the cache lowers its request.

Alternatively, if you prefer not to build an arbiter, you can control the bus in the simulation by ensuring only one cache attempts a transaction at a time (e.g., use separate clock phases or manually trigger one core's memory operation at a time).

Cache–Bus interactions: Every cache connects to the bus with both **outputs** (to initiate its own transactions) and **inputs** (to snoop others' transactions): - **Outputs to Bus:** Each cache outputs an address and command when it needs to. These go through tri-state buffers controlled by the arbitration logic. The cache also outputs data to the bus when performing a write-back (flush) – this could be controlled by a separate line indicating "this data is valid on the bus now." - **Inputs from Bus (Snooping):** The shared bus address and command lines are wired as inputs into every cache's controller. In Logisim, you can wire the bus lines into each cache subcircuit (perhaps via input pins on the cache module). This way, even when a cache isn't driving the bus, it can see the current **BusAddr** and **BusCmd** signals. The cache's snoop control logic will compare BusAddr with its own tags: - If BusCmd is BusRd or BusRdX and the BusAddr matches a line in M or S state, the cache will perform the flush/invalidations as described earlier (update its state, and if needed, put data on bus for flush). - If BusCmd is BusUpgr and BusAddr matches a line in S state, the cache invalidates that line (S → I). - **Memory on Bus:** The main memory is also attached. It watches BusCmd and BusAddr: - On a **BusRd/BusRdX**, memory will prepare to supply the data (unless a cache intervenes with a flush). If a flush occurs (a cache had M), memory will be updated with that data; the requesting cache can then take that data. In a simple design, memory can always put the data on the bus after a fixed delay, and we assume if a cache had M it has written the data to memory first (so memory's copy is up-to-date). - On a **BusRdX/BusUpgr**, memory doesn't need to do anything for invalidation (other than possibly supply data for BusRdX). On a **Flush** operation, memory will write the data from the bus into the addressed location (memory's write enable controlled by the flush signal).

In Logisim, the **shared memory** can be a single RAM module. Connect the RAM's address input to the BusAddr, its data input/output to the BusData (with appropriate tri-state or multiplexing logic if needed), and have the cache controllers manage the RAM's chip select or output enable: - For a read: when BusCmd indicates a read and no cache blocks it, enable memory's output so it drives BusData. - For a write-back: when a flush occurs, enable memory's write with BusAddr and take data from BusData lines.

The key point is that **all caches see the same bus traffic**. So, when one cache initiates a transaction, every other cache's snoop logic runs in the same cycle to update or invalidate their copies as required. This bus-based snooping approach is suitable for a small number of cores (e.g., 2-4 cores) and is exactly how classic SMPs maintained coherence [18].

Step-by-Step Guide: Implementing the Design in Logisim

Implementing an MSI-coherent cache system in Logisim can be complex, but breaking it into steps helps:

1. Build the Cache Subcircuit: - **Data Store:** Add a Logisim memory for cache data. In the component's properties, set the number of addresses to the number of cache lines, and the data width to the size of each line (in bits). For example, for a cache with 16 lines of 4 bytes each, use 16 addresses and 32-bit data width (if simulating one word per line), or adjust for actual block size. - **Tag/State Store:** Add another small memory (or use D flip-flops/registers for a smaller cache) to store the tag and state bits for each line. This memory's address count equals the number of cache lines; data width should fit the tag bits plus 2 state bits. For instance, 16 addresses with 26-bit tag + 2-bit state = 28 bits (you might round up to a convenient width like 32 bits) per entry. - **Address Splitter:** Use a **Splitter** component to divide the 32-bit CPU address into *tag*, *index*, and *offset* sub-buses. Connect the index to the address input of the data memory and tag memory. (Ensure both memories use the same index so that they correspond to the same line.) - **Tag Compare:** Use a **Comparator** to check if the address tag (from the CPU) equals the stored tag from the tag memory output for that index. Also check the state bits from that memory. - **Hit/Miss Logic:** Implement combinational logic that determines if a cache **hit** occurs: `(TagMatch == 1) AND (State != Invalid)`. If a hit, the cache can use the data memory's output directly (for reads). If a miss, the cache will activate a bus transaction. - **Data output path:** For a read hit, output the word at [index, offset] from the data memory to the CPU. If the cache line size is more than one word, you'll use the offset bits to select the correct word or byte from the data memory output (e.g., use a splitter to select the right 32-bit portion of a wider line, or configure memory with byte-enable if supported). - **CPU write (store) path:** For writes, have a way to input data from CPU to the cache. If it's a hit in M or S state: * If state is M (already exclusive), write the CPU's data into the data memory at [index, offset] (and set a dirty flag if not already). State remains M. * If state is S (shared), treat it as a miss/upgrade: you will trigger a BusUpgr transaction (see below) before actually modifying the data. One way is to stall the write, issue BusUpgr, update state to M, then perform the write. In Logisim, this may require a simple FSM or a two-step process (first cycle do bus op, second cycle do the write). - **State update logic:** Decide how to update the state bits on various events. For example: * On a **read miss** (cache line loaded): if BusRd returns data, set state = S (by writing the tag memory at [index] with new tag and state S). Unless you implement an Exclusive state optimization (MESI), in MSI protocol a newly loaded line from memory is typically marked Shared [21] . * On a **write miss**: after BusRdX, set state = M for that line. * On a **write hit** (in S): after BusUpgr, set state = M. * On a **BusRd snoop**: if this cache's tag matches BusAddr: - If state was M, write back data (flush) then set state = S. - If state was S, stay in S. * On a **BusRdX snoop**: if tag matches: - If state was M, flush data then set state = I. - If state was S, set state = I. * On a **BusUpgr snoop**: if tag matches and state was S, set state = I. * On an **eviction** (replacement): if state is M, we must flush the data to memory (write-back) and then invalidate the line (state = I). If state is S, we can evict without write-back (since memory is up-to-date) – just mark I. In practice, you can integrate eviction into the miss handling: when a new block is fetched into a full cache, evict the LRU or index (direct-mapped: it's just the index). - You may implement the above state changes via a small finite state machine or combinational logic with the bus signals. In Logisim, a simple approach is to use **registers** or the tag memory itself to store updated state, and use control signals triggered by bus events to write to them (e.g., use a controlled **"write enable"** on the tag/state memory when updating the state or tag).

2. Establish the Shared Bus: - Create a bus for **Address, Data, and Command**. In Logisim, you can represent the bus by grouping wires (for multi-bit) and using labeled wiring or a subcircuit for the bus. - **Tri-state buffers:** Place controlled buffers on each cache's outputs to the bus. For example: * For the address: each cache's address-out goes through a tri-state buffer onto the main Address bus line. The enable for this tri-state is the bus grant for that cache (only enabled when this cache is driving the bus). * Likewise for the data-out (used during flush/write-back): route each cache's data-out through a tri-state buffer to the Data bus, enabled only when that cache needs to output data. * For the BusCmd lines: if using a multi-bit command bus, you can also tri-state or encode such that only one cache drives it. Alternatively, a one-hot approach could be used (e.g., separate "Rd request" and "RdX request" lines that caches assert, and if one asserts, that's the bus command – but this needs arbitration to avoid two at once). A simpler method is to treat BusCmd as a bus similar to address, with tri-state buffers or a mux selecting which cache's command is put on the bus. - **Arbiter (optional):** Design a small arbiter that listens to each cache's request. This can be a separate subcircuit. It can output a one-hot grant signal to each cache. If you want to simplify, you can assign a manual control: e.g., a toggle to choose which cache drives the bus at a time if multiple request (simulating a very simple arbitration). - **Bus monitoring:** Connect the shared Bus lines as inputs back to each cache: * The main Address bus goes into each cache's **BusAddrIn** (for snooping). You might directly wire the bus address to the cache's comparator logic. * The BusCmd lines go into each cache's snoop control logic (to know if a BusRd, BusRdX, etc. is happening). * The Data bus could be input to caches if you want them to see data (not usually needed except if implementing cache-to-cache transfer, which we can skip for simplicity – data is taken from memory on miss). - Ensure the **Main Memory** is also connected: * Memory's address input connected to the Bus Address. * Memory's data input and output connected to the Bus Data (through tri-state or direct wiring depending on Logisim memory configuration): - If using Logisim's RAM, you have separate DIN and DOUT pins. Connect the DOUT to the Data bus via a tri-state buffer that enables on memory read. Connect the DIN to the Data bus for memory writes (flushes). * Memory control: You can use BusCmd signals to control memory: - On BusRd or BusRdX, after a brief delay or next cycle, memory should output data. You might tie the BusCmd to the memory's **OE** (output enable) such that when BusCmd indicates a read request and if no cache is driving data, memory drives the bus data. - On a flush (a cache writing back), you can enable memory's **WE** (write enable) when a cache asserts a flush. The address is on bus, data on bus, so memory will write that data. * In practice, coordinating flush and memory read can be tricky – one approach: when a cache in M snoops a BusRd, have it write the data to memory (set WE for memory, address = BusAddr, data = cache's data) *before* memory outputs data to the requester. This ensures memory has the latest copy to forward. Alternatively, memory could snoop a "Flush" signal and latch the incoming data.

3. Multi-Core CPU Integration: - Instantiate multiple **CPU cores** (if you have a Logisim CPU design, e.g., a RV32I CPU subcircuit). If you don't have a full CPU, you can simulate cores with simple circuits that generate memory requests (a program counter plus an instruction sequence that performs loads/stores). - Attach each CPU's memory interface to its cache: the CPU's address output goes to the cache's address input, CPU data out to cache (for stores), and CPU data in from cache (for loads). Also hook up CPU control signals to the cache (e.g., a line indicating read vs write, and a handshake if needed). - All caches are connected to the single shared bus as described. Ensure the bus and coherence logic is in the **top-level circuit** so that all caches see the same bus. - **Shared Memory:** Use one main memory module (RAM) as the backing store, connected to the bus. Initialize this memory with some test data or program as needed. - **Clocking:** Use a coordinated clock for the CPUs and caches. In Logisim, you might have one global clock that drives the CPUs and some parts of the cache. The coherence actions might need a few cycles: * For example, on a miss, you might design the cache FSM to assert a bus request in one cycle, wait for data, and fill the cache in a subsequent cycle. So the cache may introduce wait states to the CPU. You can handle this by having the cache not acknowledge the CPU request until the operation is done. * You might slow down the clock or single-step the simulation to clearly observe the coherence events.

4. Testing Coherence Behavior: - Write or load a simple program that will run on the CPUs (or manually force some memory operations) to test the coherence: * Example: Core0 writes to address X, then Core1 reads X – Core1 should see the updated value, which tests that Core0's write-back or intervention worked. * Example: Both Core0 and Core1 read address Y (they end up Shared), then Core1 writes Y – Core0's cache should see the BusUpgr and invalidate Y. * Try sequences that cause different state transitions: e.g., Core0 reads (I→S), Core0 writes (S→M via upgrade), Core1 reads (other cache has M, so it flushes and goes to S, both end up S), etc. - Use Logisim's **poke tool** and **tick** to step through cycles. You can observe the bus lines with probes or the "Trace" feature. Watching the state bits in each cache (perhaps via an output pin or an LED that indicates M/S/I) is very useful. For instance, you could output the state of a particular cache line to a red/green LED: green for Shared, red for Modified, off for Invalid – or use a hex display for the 2-bit state. - Check that at all times the coherence invariant holds: if one cache has a line in M, no other cache should have it in S (they should be I). If two caches have a line, they must be in S state in both. By observing your state outputs, you can verify this property.

5. Practical Tips: - **Synchronization:** Realize that bus transactions are not instantaneous. In Logisim, you may need to introduce a delay (even a single tick) for the data to be transferred. For example, after a BusRd, give one cycle for memory to output data before latching it into the requesting cache. - **Tri-state usage:** Remember that in Logisim, all outputs driving a wire should be high-impedance except one. If you see contention (Logisim will warn of error on the bus), check your enable logic. A common implementation is to use a **demultiplexer or decoder to generate exclusive bus grants**, so only one cache's buffers are enabled at once [20]. - **Modularity:** Test one cache alone with a dummy memory to ensure it fetches on misses and writes back on eviction. Then introduce a second cache and the bus. - **Debugging:** Use the "tick once" button liberally to advance the simulation step by step. Place probes on the bus lines and on internal signals like the comparator output, hit/miss signal, and state bits. This will help you see, for instance, when a miss causes a BusRd, or when a snoop causes a state to flip to Invalid. - **Avoiding race conditions:** In a real hardware design, bus arbitration and transactions have well-defined phases. In Logisim, it might be possible that two caches "think" at the same time. Using a single shared clock and arbiter will serialize events. If two cores do happen to request at the exact same cycle, the arbiter should grant one and delay the other.

By following these steps, you will have a simulated multi-core system where each core's memory operations go through its cache and the caches communicate over a shared bus to maintain coherence. The MSI protocol will enforce that all caches eventually agree on the values of memory.

Extensions and Enhancements

Once the basic MSI cache coherence is working in Logisim, there are several ways to extend or analyze the system for educational purposes:

- **MESI Protocol:** Add the **Exclusive (E)** state to create a MESI protocol (Modified, Exclusive, Shared, Invalid) [22] [23]. The Exclusive state would mean a cache has the only copy of a block but it is still clean (same as memory). This optimization would allow a cache to transition from E to M on a write *without* a bus transaction, since no other cache has the data. Implementing MESI involves adding one more state bit and modifying the state machine: on a BusRd, if no other cache had the data, the responding cache can give it in Exclusive state to the requester. This cuts down on BusUpgr events for data that isn't truly shared.

- **MOESI or Other Protocols:** For advanced exploration, add states like **Owned (O)** – which is in MOESI protocol [24] [25] – meaning a cache can supply data to others (it has the only dirty copy, memory is stale, but it will provide on read snoop instead of writing back immediately). This allows cache-to-cache transfers without immediate write-back. These extensions increase complexity but expose students to real-world optimizations in coherence protocols.

- **Performance Counters:** Incorporate counters to measure cache performance:

- Count the number of **cache hits** versus **misses**. For example, increment a hit counter on each CPU read/write that is serviced without bus traffic, and increment a miss counter on each BusRd/BusRdX.

You could use simple up-counters (registers that increment) and display their values on a 7-seg display or console.

- Count coherence events: how many BusRdX or BusUpgr were issued (write coherence traffic), how many flushes occurred (write-backs due to coherence), etc. This gives insight into the overhead of coherence.

- **Visualization Aids:** Since Logisim allows custom text labels, you can set up dynamic labels that show, for instance, the current BusCmd or the state of a particular line. Another idea is using colored LEDs or output pins to indicate each cache's status (e.g., an LED that lights when a cache is driving the bus, or separate LEDs that show "M state present in Cache A", etc.). This can make demos more intuitive.

- **Scalability experiments:** Try increasing the number of cores/caches on the bus (if Logisim can handle it) and see how the coherence still works but bus contention might increase. This can segue into discussing bus bandwidth limitations and the need for more scalable interconnects (e.g., point-to-point networks or directories for larger systems).

- **Timing Refinements:** The current design is likely an abstraction where bus transactions happen in a few ticks. One could enhance it by modeling separate phases (arbitration, address broadcast, data transfer) more explicitly with a state machine, and maybe adding *transient states* to MSI (to handle a line that is in the middle of being fetched or invalidated). This is quite advanced and mirrors real implementations that avoid race conditions in non-atomic actions [26].

By implementing this project, you gain insight into how cache coherence is maintained in hardware. The Logisim simulation, while simplified, enforces the same coherence rules used in real multiprocessors. Caches snoop on a common bus [19], using signals like BusRd, BusRdX, and BusUpgr to indicate their intents [27]. The MSI protocol ensures that when one core modifies data, all other cores will see that update or invalidate their stale copies [2] [1]. This hands-on approach in Logisim solidifies understanding of both cache design and the intricacies of multiprocessor coherence.

---

[1] [3] [17] [18] [19] [22] [24] [26] Cache Coherence I – Computer Architecture
https://www.cs.umd.edu/~meesh/411/CA-online/chapter/cache-coherence-i/index.html

[2] [4] [5] [7] [8] [9] [10] [11] [12] [13] [14] [15] [21] [27] MSI protocol - Wikipedia
https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15418-s20/www/exams/resources-exam2/wikipedia-msi-restricted.html

[6] [16] [23] [25] Cache Coherence Protocols in Multiprocessor System - GeeksforGeeks
https://www.geeksforgeeks.org/cache-coherence-protocols-in-multiprocessor-system/

[20] Three-State Bus Buffers - GeeksforGeeks
https://www.geeksforgeeks.org/three-state-bus-buffers/