ChatGPT

Integrating Coherent Dual-Core System in Logisim

Integrating CPU0 and CPU1 Cache Modules

Yes – you will need to include both the **CPU0** and **CPU1** cache controller modules and hook them up to your dual-core pipeline in Logisim. The easiest approach is to **import the VHDL entities** for `cpu0` and `cpu1` as subcircuits (assuming you're using Logisim-Evolution, which supports VHDL components). Each of these will represent one core's cache+coherence controller. Once imported, you can place two instances (one for each core) into your main design. If Logisim cannot directly import them, you may need to manually construct equivalent logic using Logisim blocks, but importing is preferable to reuse the given code.

Connect the pipeline to the cache controllers: Encapsulate your existing single-core pipeline into a subcircuit (e.g. a subcircuit for **Core0**). Duplicate it for **Core1** so you have two identical processor pipelines. **Remove any direct connections from the pipeline to main memory**, and instead connect the pipeline's memory interface signals to the CPU0/CPU1 cache modules. For example:

- The pipeline's memory address output (for data access in MEM stage) should go into the cache module's address input (`wantedAddress` for CPU0, or `addr` for CPU1).
- The pipeline's read/write control signals should inform the cache controller whether it's a read or write operation. In the CPU0 VHDL, there isn't a direct `read_en` input from the pipeline (it generates `cpu_rd_req_out` internally), so you may need to modify the CPU0 FSM or provide a trigger. Ideally, adjust the CPU0 code to **remove its internal automatic FSM sequence** and let the pipeline control it: e.g. have an input signal for "pipeline read request" or "pipeline write request" that sets `cpuCacheOperation` (`read_cache` or `write_cache`) and the `wantedAddressLatch`. Similarly for CPU1, use its `read_en` and `write_en` inputs (these are already in CPU1's port) driven by the pipeline's control (the MEM stage's load/store signals and write-data).
- The data path for writes: pipeline's data to write should go into the cache module's data-in (`cpu0_data_in` or `data_in` for CPU1).
- The data path for reads: the cache module's data-out should feed back to the pipeline (so the pipeline's MEM/WB stage gets the data either from cache or memory). For CPU1, this is straightforward (`data_out` port gives the data read from cache/memory). For CPU0's VHDL, you might need to expose the `sendDataFromCacheToCpu` (which is the data retrieved from cache or memory) as an output so the pipeline can use it.

In summary, **each CPU core pipeline will connect to its cache controller** subcircuit instead of directly to a memory. The cache controller will manage whether data comes from its own cache or needs to be fetched, and when a write happens it will update cache and coordinate coherence. You'll effectively have two "CPU + Cache" pairs.

Implementing SDRAM with a Logisim Memory

You can implement the shared SDRAM using a standard **Logisim RAM component** configured as 32-bit width. The VHDL `sdram` module is basically a small memory array (8 words in the code) with two ports. In Logisim, a built-in RAM is typically single-ported, so to handle two cores you have a couple of options:

- **Single Memory + Arbitration:** Use one RAM component for the actual memory storage and ensure that only one core accesses it at a time. You will need an **arbiter** that decides which core's request goes to memory when both cores make requests simultaneously. The provided `FIFO` module can serve as an arbiter/queue for memory requests – you can import the FIFO VHDL as a subcircuit and use it to buffer one request if both cores hit memory at the same time. Connect `cpu0_read_in`, `cpu0_write_in`, etc. of the FIFO to the signals coming from CPU0's cache controller (`read_en`, `write_en`, address, data-out), and similarly for CPU1. The FIFO's outputs (`read_out`, `write_out`, `addr_out`, `data_out`) then connect to the single RAM's control pins:
- Tie `write_out` to the RAM's **WE** (write enable) pin, and use `read_out` to control the RAM's **OE** (output enable). When `read_out` is 1 (a read operation), enable the memory's output; when 0 (no read), you can disable output to free the data bus.
- Connect `addr_out` to the RAM's address input, and `data_out` to the RAM's data input (for writes).

- The memory's data **output** bus will carry the read data. This output should feed into the requesting cache controller's input. In practice, since the FIFO only issues one request at a time, you know which core is being served: e.g. if `read_out=1` and it came from CPU0's request, then the data appearing on RAM's output is meant for CPU0's cache. You may need a way to route the memory's output to the correct core (more on that in the bus section below).

- **Dual-Port Memory approach:** If you want to mimic the VHDL's two-port memory (allowing both cores to read simultaneously), Logisim doesn't have a built-in true dual-port RAM by default. However, you could simulate it by using **two identical RAM components** wired in parallel (ensuring they stay in sync on writes). One RAM output goes to CPU0, the other to CPU1. For writes, you'd write to both memories so both stay consistent. This approach is a bit clunky and not truly necessary for coherence demonstration – it might be safer to stick with a single memory and arbitrate access, because coherence protocol typically assumes a **shared bus** (one memory access at a time). Using arbitration (like the FIFO logic or a simple priority scheme) will ensure only one core's request goes to memory at once, which aligns with a bus-based MSI protocol assumption.

Initialize and size the RAM: You can set the Logisim RAM component's depth to 8 (or 32K if you plan a larger memory) and width to 32 bits. Load it with initial values if needed (the VHDL code shows addresses 0–7 initialized to 0x0, 0x1, 0x2, … 0x7). This will act as main memory. In Logisim's simulation, the memory will output the data at the addressed location when OE is enabled and will update on the rising clock edge if WE is asserted (assuming the clock is connected to the RAM if you use synchronous memory). For simplicity, you might use **asynchronous read** (Logisim's default RAM reads outputs combinatorially) – then controlling OE is important to avoid bus conflicts.

Establishing a Shared Bus and Tri-State Buffers

In a multi-core coherent system, a **shared bus** is typically used for memory and cache-to-cache communications. You will need to create a bus structure in Logisim so that the two caches and the main memory can all exchange address and data signals. The key is to ensure only one device drives a given bus at a time – this is where **tri-state buffers** come in.

Address Bus: Only the **requesting core** should drive the address lines to memory. You can use a **multiplexer** or a controlled buffer on the address lines: - Use a 2-to-1 MUX to select either CPU0's address or CPU1's address to send to the RAM's address input. The selection can be driven by the arbitration logic (e.g. the FIFO's logic or a priority signal). For example, if CPU0's request is being served (FIFO chooses CPU0 or a priority flag is set), select CPU0's address; if CPU1's turn, select CPU1's address. - Alternatively, put a tri-state buffer on each core's address output: CPU0's address buffer enabled when CPU0 has the bus, CPU1's enabled when CPU1 has the bus (the arbiter can provide an enable signal). The outputs of these buffers tie together to a common address bus wire leading to memory. Only one buffer will be active at once to drive the address [1].

Data Bus: The data lines need to allow multiple sources and destinations: - **Memory to CPU**: When a memory read is performed, the RAM outputs data that must travel to the requesting CPU's cache/controller. - **CPU to Memory**: On a memory write, the CPU's data must go to RAM. - **Cache-to-cache (CPU to CPU)**: In the MSI protocol, if one cache has a modified copy of a line that the other core needs, it will supply that data directly to the other core (instead of going out to main memory). This means one CPU's cache can drive data onto the bus for the other to read.

To implement this, use a **shared data bus** (matching the data width, 32 bits) connecting all three components (CPU0 cache, CPU1 cache, and Memory). Connect each component's data output to this bus through a tri-state buffer: - **Memory's data output**: Put a controlled buffer on the RAM's data output. Enable this buffer only when the memory is responding to a read. For instance, tie the buffer's enable to the arbiter's `read_out` signal (or an OE control) – when a read is active, memory drives the bus with the data; otherwise the memory's output is high-impedance. - **CPU0 cache data-out**: CPU0's cache controller will sometimes need to put data on the bus (specifically, to supply data to CPU1 on a cache-to-cache transfer). In the CPU0 VHDL, this occurs when it has a modified line that CPU1 requests – CPU0 sets `cache_to_cache_resp_out = '1'` and drives `cache_to_cache_resp_out_data`. You should route `cache_to_cache_resp_out_data` through a tri-state buffer onto the shared data bus. Enable that buffer when CPU0 is responding to a request (perhaps when `cache_to_cache_resp_out_ready = '1'` **and** `cache_to_cache_resp_out = '1'`). That combination indicates "CPU0 has data ready for CPU1," so CPU0 drives the bus with the cache line data at that time. - **CPU1 cache data-out**: Similarly, CPU1's cache might supply data to CPU0. In the CPU1 VHDL, if `request_cpu1` input is `1` (meaning CPU0 is requesting data from CPU1's cache), then CPU1 sets `grant_cpu1_data` to the cache data. Use a tri-state buffer on `grant_cpu1_data` connected to the shared data bus. Enable it when CPU1 should grant data – essentially when `request_cpu1='1'` (i.e., CPU0's request is active) and CPU1 indeed has the data. You may also incorporate a condition that CPU1 had a hit (perhaps indicated by `hit` signal or the MSI state being modified). In practice, since CPU1's logic doesn't explicitly output a "has data" flag, you might assume that if `request_cpu1` is asserted, CPU1 will drive the bus with its data immediately (provided its MSI state is Modified or Shared).

Each destination then reads from the bus when relevant: - CPU0's cache controller will listen on the bus when it has sent a request to CPU1. CPU0 VHDL uses `cache_to_cache_resp_in_data` as input for data from the other cache – connect the shared data bus to that input (through maybe a splitter, since it's already a bus). - CPU1's controller will read from the bus when memory responds or when CPU0 supplies data. In CPU1's VHDL, `data_out` is actually driven internally (either from cache or after getting memory data or CPU0's data). But since you've connected the bus to CPU1's memory input (`sdram_data` in CPU1 ports) or `cpu0_data` input, ensure the bus feeds those correctly. Concretely, you can connect the shared data bus to CPU1's `cpu0_data` input (the port intended to carry data from CPU0's cache). CPU1's code, on an invalidate, does `cache(...).data <= cpu0_data` and outputs `data_out <= cpu0_data`, meaning it's using that bus data.

Using tri-state buffers ensures the bus behaves properly like a multi-driver bus: **only one component's output is actively driving the data lines at a time** [1]. When not enabled, each buffer outputs a high-impedance (floating) so it doesn't interfere. This way, for example, during a memory read, the memory drives the bus and the caches' buffers are off; during a cache-to-cache transfer, the memory buffer is off and only the supplying cache's buffer is on.

Tip: In Logisim, controlled buffers (found under Gates as "Tristate Buffer") can be set to 32-bit width for the data bus. Connect the bus wire to all buffer outputs. Use clear labels or colors for the shared bus wires to avoid confusion. Ensure your control logic never enables two drivers at once (or you'll get bus contention errors).

Connecting the Coherence Signals (MSI Protocol)

With the physical connections in place (two cores and a shared memory bus), you must tie together the **coherence signals** so that the MSI protocol functions. There are two ways to coordinate coherence here, based on the provided code:

1. Direct Snooping Signals (CPU0 ↔ CPU1): The CPU0 VHDL is designed to send and receive snoop signals directly to the other core. If you use this approach, you will **wire CPU0 and CPU1 modules directly to each other's coherence ports:**

- **Write intent (invalidate others):** Connect `cpu0.cpu_wr_req_out` to the other core's input that triggers invalidation. CPU0 sets `cpu_wr_req_out='1'` when it is about to perform a write to an address. In CPU0's own logic, the other core is expected to see this and invalidate that cache line. CPU1 does not have `cpu_wr_req_in` in its port list, but it *does* have an `invalidate` input (driven by the MSI module in the other approach). You can use this directly: tie `CPU0.cpu_wr_req_out` into `CPU1.invalidate`. This means whenever CPU0 writes a line (Modified state), CPU1 will receive an invalidate signal. Conversely, connect `CPU1`'s equivalent signal to `CPU0`. CPU1's VHDL doesn't output a `cpu_wr_req_out` explicitly, but it marks its state as Modified on a write. For symmetry, you might generate a signal to invalidate CPU0 when CPU1 writes. One way is to use the **MSI protocol module** for this part (see below), or modify CPU1's code to raise an output when it writes. A simpler method: tie CPU1's `write_en` (the input that signals a write) as an indicator – when CPU1 performs a memory write, that could alert CPU0. However, just a raw write signal is not enough (it doesn't carry the address). **Better approach:** use the MSI protocol component as a

monitor to handle invalidation signals, or add a custom output from CPU1 (similar to cpu_wr_req_out) that goes high when CPU1 writes a cache line.

- **Address lines for snooping:** Connect `cpu0.cpu_req_addr_out` to CPU1 so that when CPU0 broadcasts a write or read intent, the address is known to CPU1. In CPU0 VHDL, `cpu_req_addr_out` carries the address of its current memory operation (e.g., the address it is writing or requesting). CPU1's VHDL doesn't have a dedicated `cpu_req_addr_in` for snooping, but the **MSI protocol** entity expects both CPU addresses. If using direct wiring, you can tie `CPU0.cpu_req_addr_out` to an input of CPU1's module that represents "other core's operation address." CPU1 does have `addr` (its own operation address) and its internal cache tags, but to properly invalidate, it should compare addresses. You might consider adding a new input to CPU1 (if you can modify VHDL) for "invalidate_address" and use that in the invalidation process. If that's not feasible, the MSI module will handle it (see next approach).

- **Read (cache-to-cache) requests:** Connect `CPU0.cache_to_cache_req_out` to `CPU1` so that CPU0 can ask CPU1 for data on a cache miss. In CPU0's logic, when it has a read miss, it sets `cache_to_cache_req_out = '1'` along with `cache_to_cache_req_address_out` (the address needed). On CPU1's side, the equivalent is that it should receive this request. CPU1's VHDL uses `request_cpu0` **output** as "CPU1 requests data from CPU0," and `request_cpu1` **input** as "CPU0 requests from CPU1." So connect:

- `CPU0.cache_to_cache_req_out` → `CPU1.request_cpu1`. Now when CPU0 needs data, CPU1's logic sees `request_cpu1='1'` and will prepare to output its data.

- Also connect `CPU0.cache_to_cache_req_address_out` → (somehow to CPU1, though CPU1 code doesn't have an explicit port for the other's address). Here's where there is a gap: CPU1's code doesn't take an external address for the request; it assumes if `request_cpu1=1`, the address in question might be its current `addr`. In practice, to be correct, you should ensure that at the moment of CPU0's request, CPU1's `addr` lines are set to the address being requested. One way: you could drive CPU1's `addr` input from CPU0's request address whenever `request_cpu1` is high (perhaps via a multiplexer: normally CPU1.addr comes from its pipeline, but if another core's request comes in, feed that address into CPU1's cache for lookup). This would allow CPU1 to check the proper index/tag. This is an advanced detail – if not easily done, you may rely on the assumption that both CPUs might be requesting the same address in a test scenario.

- Conversely, connect `CPU1.request_cpu0` → `CPU0.cache_to_cache_req_in`. That handles the case if CPU1 ever needs data from CPU0's cache. CPU0 will receive `cache_to_cache_req_in='1'` along with `cache_to_cache_req_address_in` (which CPU1 doesn't output explicitly, but if you modify CPU1 to output a request address, connect it similarly). CPU0's logic will then snoop its cache and respond.

- **Cache-to-cache response (data & ready flags):** Connect the response signals so that when one cache supplies data, the other knows. CPU0 VHDL produces:

- `cache_to_cache_resp_out` (a flag indicating "I have the data") and `cache_to_cache_resp_out_data` (the actual data).

- `cache_to_cache_resp_out_ready` (a strobe to say the data is on the bus now). Connect `CPU0.cache_to_cache_resp_out` to CPU1 if needed (CPU1's design doesn't explicitly use a flag; it simply uses `invalidate` or directly takes data). If using MSI module, this flag might not be needed. For direct wiring, you could ignore the flag and use the presence of data on the bus as implicit (or use it to gate tri-state enable as we did).

For data: as mentioned above, connect `CPU0.cache_to_cache_resp_out_data` to the shared data bus (and thus into `CPU1.cpu0_data` input). CPU1 will capture this via `cpu0_data` input and place it in its cache (the CPU1 code does `cache(...).data <= cpu0_data` when `invalidate=1`, treating it like a transfer).

CPU0's `cache_to_cache_resp_out_ready` can be used to tell CPU1 when to latch the data (if we had a latch). In Logisim, you might simply connect it to a control or to CPU1's internal logic if modifiable. Similarly, for the other direction: - CPU1 has `grant_cpu1_data` (data for CPU0) and no explicit ready flag. CPU0 expects `cache_to_cache_resp_in_ready` and `cache_to_cache_resp_in` signals. We can simulate those: for instance, when CPU1 drives `grant_cpu1_data` on the bus, assert a signal to CPU0 to indicate "data ready." You could derive this by simply using `CPU0.cache_to_cache_resp_in_ready = CPU1.request_cpu1` (since if CPU0 asked, and CPU1 is responding immediately, assume ready in next cycle). And `CPU0.cache_to_cache_resp_in` (the flag for hit/miss) could be tied to logic: if CPU1 had the data, set it to '1'; if not, '0'. Perhaps use CPU1's `msi_state` or `hit` to decide this. For example, if CPU1's cache was valid for that address and in Modified/Shared state, then `cache_to_cache_resp_in='1'`. This part may require a little custom logic outside the VHDL modules, because CPU1's code doesn't output a "have it" flag explicitly.

2. Using the `msi_protocol` Module: The alternative (and perhaps cleaner) approach is to use the provided `msi_protocol` entity as a centralized snoop controller for invalidations, while still using cache-to-cache data transfers for modified lines. In this scheme: - Each cache controller should output its current MSI **state** and the **address** it is operating on to the `msi_protocol`. For example, whenever a core is about to perform a memory operation, present the state of that cache line (Invalid/Shared/Modified) and the address to the msi_protocol inputs `cpu0_state`, `cpu0_addr` (and similarly for CPU1). In CPU1's VHDL, `msi_state` is an inout that always reflects the state of the current cache line; you can wire that to `cpu1_state`. For CPU0, you may need to derive a 2-bit state code: e.g., when CPU0's cache FSM is in a read or write operation, map its internal `cache(...).MSI` for the relevant line to a 2-bit code (`00` invalid, `01` shared, `10` modified) and output that. - The msi_protocol will compare addresses. If, for instance, CPU0 is writing to a line (CPU0_state = "10" for Modified) and CPU1 has that same address in Shared state (`cpu1_state = "01"` and addresses match), the msi_protocol raises `invalidate_cpu1 = '1'` [2]. Connect `invalidate_cpu1` output to CPU1's `invalidate` input. This will cause CPU1's cache to invalidate that line (as CPU1's process checks `if invalidate='1' then ... cache.valid <= '0'; msi_state <= "00"`). - Similarly, if CPU1 writes to a line and CPU0 had it shared, `invalidate_cpu0` will go high. Now, CPU0's VHDL doesn't have an `invalidate` port, but it does react to an external write via `cpu_wr_req_in`. You can tie `invalidate_cpu0` into `CPU0.cpu_wr_req_in`, **and** ensure the address from CPU1 is connected to `CPU0.cpu_req_addr_in`. This way, when msi_protocol detects CPU1's write, it drives CPU0's cpu_wr_req_in=1 with the matching address on cpu_req_addr_in, and CPU0's process will invalidate its line (the CPU0 code does `if (cpu_wr_req_in='1') and addresses match then cache(i).MSI <= invalid` for that address). - Note that using msi_protocol means the caches don't directly send "invalidate" signals to each other; instead they report their actions to the protocol, and the protocol sends out invalidation signals. **You**

should still connect the cache-to-cache data request lines as above for data transfer on read misses (msi_protocol covers the *invalidation* on writes, but it doesn't directly handle data sharing on reads). In other words, you'd use msi_protocol for the "Snoop invalidate" part of MSI, and still use `cache_to_cache_req` / `resp` lines for the "supply modified data" part of the protocol.

- If using msi_protocol, you might not need CPU0's `cpu_wr_req_out` and `cpu_rd_req_out` signals at all – those intents are implicitly handled by the protocol's monitoring of state changes. It's okay to leave them unconnected or just used internally for debug. Focus on wiring the state & address lines and the invalidate outputs.

In summary, **choose one coherence signaling approach** and wire accordingly. A consistent approach is: use msi_protocol for all invalidation signaling (simpler, since it checks addresses and ensures only the correct line is invalidated), and use the direct cache-to-cache request/response lines for data transfer on read-miss. This hybrid would look like: - CPU0 & CPU1 both connected to msi_protocol (state & addr in, invalidate out). - CPU0 & CPU1 caches also directly connected via `cache_to_cache_req` and `resp` lines (for data sharing). - When a core writes, it updates its state to Modified; the protocol sees that and sends invalidate to the other core. When a core reads and misses, it sends `cache_to_cache_req` to other; if other has it Modified, it responds with data (and likely updates state to Shared in both caches).

Step-by-Step Integration Plan

Bringing it all together, here's a recommended sequence to integrate everything:

1. **Encapsulate Each CPU Core**: Turn your single pipeline design into a subcircuit (e.g., "Core0") so you can instantiate it twice. Inside this subcircuit, include the pipeline stages IF/ID/EX/MEM/WB as you have, but remove the direct connection to data memory in the MEM stage. Instead, bring the memory interface signals out as pins (address, data out, data in, read enable, write enable). This subcircuit represents a CPU core **without L1 cache**.

2. **Attach Cache Controllers**: For each core, attach the imported cache controller VHDL:

3. Place the `cpu0` component alongside Core0 pipeline, and connect Core0's memory interface to it. For example, Core0's MEM stage address → `cpu0.wantedAddress`; Core0's write data → `dataToSendToCache` (you may need to modify cpu0's VHDL to expose that signal or use `Sdram_data_out` for writes); Core0's read strobe triggers the cpu0 to perform a read operation. This may require adding a control input to the cpu0 module's FSM (or manipulating its state machine as described earlier). Essentially, when Core0 pipeline issues a load or store, signal the cpu0 controller to execute a `read_cache` or `write_cache` operation for that address.
4. Connect the outputs of cpu0 back to the Core0 pipeline: e.g., the data that comes from cache/memory should go into Core0's MEM stage data input (the value that will be written back to a register). Also, the cache controller should indicate if it's ready or if the pipeline must wait – you might use the `hit` signal or similar to stall the pipeline until data is available. (If a miss triggers a longer memory fetch, you will need to hold the pipeline's progress. In Logisim, you can stall the pipeline by not advancing IF/PC and holding the pipeline registers when the cache controller is busy. This is a complex control mechanism – if not implemented, you may still observe coherence behavior by single-stepping the clock).

5. Do the same for Core1: connect the `cpu1` cache component to the Core1 pipeline. Core1's pipeline memory address → `cpu1.addr`, etc. CPU1's `data_out` will go to Core1's pipeline as the read result. CPU1's module already has `read_en` and `write_en` inputs which you tie to Core1's control signals (load/store).

6. **Integrate the Coherence Signals**: Now wire the coherence connections between `cpu0` and `cpu1` modules:

7. **Using MSI module:** Connect `cpu0_state` / `cpu0_addr` from the cpu0 module (you may need to generate these from internal signals as discussed) and `cpu1_state` / `cpu1_addr` (e.g., use `cpu1.msi_state` and the `addr` input) into the `msi_protocol`. Then connect `msi_protocol.invalidate_cpu0` to an appropriate invalidation input on cpu0 (one way is to OR this signal into cpu0's internal logic that invalidates on external write – effectively treat it as `cpu_wr_req_in`). Connect `invalidate_cpu1` to `cpu1.invalidate`. This ensures that when one core writes a shared line, the other core's cache line is invalidated.
8. **Direct cache-to-cache signals:** Connect `cpu0.cache_to_cache_req_out` to `cpu1.request_cpu1` (CPU0 requests data from CPU1), and `cpu1.request_cpu0` to `cpu0.cache_to_cache_req_in` (CPU1 requests from CPU0). Also join `cpu0.cache_to_cache_req_address_out` with `cpu0.cpu_req_addr_out` (they're likely the same in the code) and make sure CPU1 somehow gets that address (as noted, possibly feed it into `cpu1.addr` when servicing the request). Connect data and response lines: `cpu0.cache_to_cache_resp_out_data` to CPU1's side (either directly to `cpu0_data` input or onto the bus), and `cpu1.grant_cpu1_data` to CPU0's side (`cache_to_cache_resp_in_data`). If `cache_to_cache_resp_out` flags are used, connect those as well or use them for controlling tri-states.

9. In this step, also connect any **CPU handshake signals** that coordinate memory operations. For example, the TransactionsBetweenCPU code used `ready1_sig = r_en1 or wr_en1` as a handshake for SDRAM. In your design, the arbiter (FIFO) will handle readiness, but make sure the CPU caches know when memory responded. CPU0 VHDL uses `read_en` / `write_en` outputs to actually perform memory operations and expects data to return via `Sdram_data_in`. CPU1 uses `sdram_read`, `sdram_write` signals similarly with `sdram_data` lines. If you use the single memory + FIFO method, you won't have two separate ready lines – instead, the FIFO grants one at a time. You might need to generate a "memory response ready" signal back to the requesting cache: for instance, when the memory read data is on the bus, assert `cache_to_cache_resp_in_ready` to the core that was waiting (if using CPU0 signals), or simply have the cache's state machine detect the data presence (CPU0 code moves to `readResponseSDRAM` state when it set `read_en` and now sees data). Essentially, ensure the cache controllers know when the memory operation completed. This might be done by a combination of clock cycles and the design of the VHDL (which likely latches memory data on the falling edge after read_en was set). Be prepared to tweak timing if needed.

10. **Insert the Memory Arbiter (FIFO)**: Place the imported `FIFO` subcircuit between the caches and the physical RAM:

11. Connect `FIFO.cpu0_read_in` to `cpu0.read_en` output, and `FIFO.cpu0_write_in` to `cpu0.write_en`. Also connect `FIFO.cpu0_addr` to the address lines from CPU0 (likely

`cpu0.Sdram_addr` output or the `wantedAddressLatch` for memory access) – basically the address that CPU0 wants to access in main memory. Connect `FIFO.cpu0_data_in` to `cpu0.Sdram_data_out` (the data CPU0 wants to write to memory).

12. Do the same for CPU1: `FIFO.cpu1_read_in` ← `cpu1.sdram_read`, `FIFO.cpu1_write_in` ← `cpu1.sdram_write`; `FIFO.cpu1_addr` ← (the lower bits of) `cpu1.sdram_addr` (note: CPU1's `sdram_addr` is 32-bit, but your memory is small, likely you only use the lower 3 bits as in the VHDL); `FIFO.cpu1_data_in` ← `cpu1.data_in` (or the appropriate signal CPU1 uses for data to memory).
13. Connect the FIFO's output side to the RAM: `FIFO.read_out` to RAM OE, `FIFO.write_out` to RAM WE, `FIFO.addr_out` to RAM address, `FIFO.data_out` to RAM data input. The FIFO will ensure that if both cores request, one gets served now and the other is queued (it effectively creates a one-deep buffer for the second request). **Important:** Also connect the memory's output back to the correct CPU:
    - If the FIFO is empty (no queue), it serves a request immediately and outputs the address and control. When a read is served (read_out=1), data will be available from RAM at its output. You need to route that to the **cache that requested it**. Since the FIFO doesn't explicitly label which core was served in the immediate case, you have to infer it. One simple method is to use the `previousRequester` logic from FIFO: the FIFO code updates `previousRequester` to '0' or '1' whenever it issues a request (like setting it to '0' for CPU0 or '1' for CPU1). You could use this signal to drive a selector that sends the RAM output to the corresponding CPU's `Sdram_data_in`. For example, if `previousRequester='0'` after a read, route RAM's data to `cpu0.Sdram_data_in`; if '1', route to `cpu1.sdram_data` (CPU1's input). In practice, you can implement this with a 2-1 multiplexer on the bus going into the caches. Alternatively, since you already have a shared data bus and tri-states, you might simply put the RAM's output on the bus (enabled), and let both CPU caches listen, but only the one that was expecting a response will actually use it. (E.g., CPU0's FSM is in a state waiting for memory data, CPU1's is not, so CPU0 will capture it). This can work if carefully timed, but using a directed path via a MUX or separate lines might be clearer.
    - For queued requests, the FIFO moves the buffered request into slot 0 on the next cycle after the first is served. It will then assert read_out/write_out for the second request. At that time, `previousRequester` will update accordingly. So the selection mechanism above should also handle that case.

14. In summary, the FIFO plus either multiplexers or bus control will ensure the right core receives memory data. Don't forget to also handle memory **write** responses: a write doesn't return data, but if a core wrote a line in Modified state that wasn't in memory (write-back), the FIFO will issue a write to memory. That doesn't affect the other core except that it was an update of main memory. The MSI protocol already handled the coherence by invalidating the other cache, so no further action needed except performing the write.

15. **Connect Clocks and Resets**: Make sure all these components share the same clock (the global `clk` goes into both CPU pipelines, both cache controllers, the FIFO, and the memory if needed). Also connect the reset signal to all modules (`reset=1` should clear caches, flush FIFO, reset memory contents if you want, and reset pipeline state). The CPU0/CPU1 VHDL code is mostly synchronous; CPU0 uses falling edge for cache updates and rising for FSM, CPU1 uses rising edge. Ensure the clock is wired consistently (you might need to invert for CPU0 if you want its process to align

differently, but probably not – you can treat the falling-edge part as just internal detail, still driven by the same clock signal).

16. **Testing and Troubleshooting**: With everything wired, test the system. For an initial simple test, you might not run a full program, but rather manually stimulate a scenario:

17. Preload the instruction memory such that each core will perform some loads/stores to the same address (to see the coherence in action). For example, Core0 could execute a store to address X, and Core1 later tries to load from address X. Or vice versa.
18. Single-step the clock and watch the signals: when Core0 writes, you should see CPU0's cache go to Modified for that line, and an invalidate signal go to CPU1, causing CPU1's cache line to become Invalid. Then when Core1 tries to read X, it will miss in its cache, send a request to CPU0 (because CPU0 had the modified data), and CPU0 should output the data for CPU1. CPU1 will then update its cache with that data in Shared state (and CPU0 likely transitions its state to Shared as well after supplying it).
19. Verify the memory updates: if a line was Modified in CPU0 and then CPU1 requested it, CPU0 should also write it back to memory (either at eviction time or immediately depending on protocol; in this MSI, typically CPU0 would **not** write back on just sharing – MSI would normally write back only on eviction, but CPU0's code actually does write back if it evicts a modified line or if it shares? The CPU0 code sets memory write (`write_en`) when evicting a modified line; it doesn't write back on giving data, it just turns state to Shared).
20. Also test both cores writing different addresses at same time to see FIFO arbitration (one should queue). You can observe `previousRequester` toggling and the second write happening after the first.

Throughout this process, expect to iterate and possibly tweak the VHDL or the Logisim wiring. Coherent dual-core integration is complex, so it's normal to adjust details: - You might find the CPU0 state machine's built-in sequence (initial/requestData/writeData loop) conflicts with actual instruction-driven behavior. It's advisable to **modify CPU0's VHDL** to remove that forced sequence and instead let external inputs drive the cache operations. For example, remove the hard-coded transitions from `initial -> writeData -> requestData` and make it so that whenever `cpu_rd_req_out` is triggered by the pipeline, the cache goes into a read service routine, etc. Essentially, convert it from a test FSM into a pure cache controller that responds to requests. - Ensure that CPU1's cache properly checks tags on invalidate. The given CPU1 code invalidates whatever line is indexed by `addr(3:2)` whenever `invalidate` is high, assuming that's the line to invalidate. This works if the address on CPU1's `addr` bus at that moment is the one to invalidate. In a more robust design, you'd search the cache for the matching tag. If you feed the invalidate address into CPU1's `addr` input as suggested, it will invalidate the correct line. Keep an eye on this in simulation to be sure the correct line gets invalidated. - If the cores need to stall during cache misses (to wait for memory), implement a simple stall mechanism: e.g., if a cache miss is detected (no hit and waiting for either other cache or memory), you can freeze the pipeline's IF and ID stage registers (not allowing new instructions) until the data is ready. You might introduce a "wait" output from the cache controller to the pipeline. This is an advanced feature; if it's too much, you can manually step the clock slowly such that one memory operation completes before the next instruction enters MEM stage.

By following these steps – integrating the caches, establishing the shared bus with tri-state buffers, using the MSI protocol or direct signals for coherence, and arbitrating memory access – you will have a dual-core Logisim model with coherent caches. It's a complex endeavor, so build it incrementally and test as you go.

Start with basic communication (e.g., one core can write and the other invalidates), then add cache-to-cache transfers, and finally integrate with the full pipeline operation. Good luck with your multi-core Logisim implementation!

---

1 Tristate-Puffer/Inverter
https://cburch.com/logisim/docs/2.7/de/html/libs/gates/controlled.html

2 How to connect a bunch of outputs to the same BUS ? : r/logisim
https://www.reddit.com/r/logisim/comments/11cfwpn/how_to_connect_a_bunch_of_outputs_to_the_same_bus/