MMU-Based Cache Control Architecture for Multi-Core Processors

Student: Ovidiu-Alexandru Lates

Structure of Computers System

Technical University of Cluj-Napoca

Contents

1.Introduction

1.1Context

The aim of this project is to offer a Memory Management Unit (MMU) that can be used to manage cache memory in multi-core processors. The MMU manages data exchange between main memory and the cache and delivers data most likely to be required again for rapid access. It will fetch the data required from the main memory and replace the least recently used blocks of memory with the cache, optimizing system and memory performance.

Cache memory is accountable for the overall speed of a computer since it accommodates a much lower data access time compared to main memory. With efficient management of cache operations using a special-purpose MMU, the system is able to contribute better performance, reduced latency, and better sharing of resources among several processor cores.

1.2Objectives

The system features a Cache Controller designed in VHDL that implements the MESI (Modified, Exclusive, Shared, Invalid) coherence protocol. Its primary role is to act as a "Snooping Agent," monitoring the shared bus to preserve data consistency across multiple caches.

Key Functionalities:

- Bus Snooping & Transaction Management: The controller continuously monitors (snoops) the shared bus. When a core issues a command (Read or Read-Exclusive), the controller broadcasts the address to all other caches (o_snoop_check) to see if they hold a copy of the data.
- Intervention & Flow Control (The "Abort" Mechanism): The controller has the authority to stop (abort) a memory transaction if it detects a conflict.
    - Read Interception: If a core requests data that is currently held by another cache in the Modified (M) state, the controller asserts the o_bus_abort signal.

    - Purpose: This stops the RAM from reading stale data. Instead, it forces the owner to "Flush" (write back) the modified data to RAM first. Once the flush is complete, the controller releases the abort, allowing the original read to proceed with the correct data.
- State Transition Enforcement: The controller dynamically updates the MESI states of cache lines based on bus traffic:
    - Write Invalidation: If a core wants to modify a line (CMD_BUS_RDX), the controller signals all other caches to transition that line to Invalid (I), ensuring only one core owns the data.
    - Sharing: If a core reads a line held by another in the Exclusive (E) state, the controller signals the owner to downgrade its state to Shared (S).

2.Bibliographic Research

2.1 Associative cache memory

A fully associative cache in computer architecture is a cache organization containing a single set with B ways, where B is the number of blocks, allowing any memory address to map to any block in the cache. This structure is also described as a B-way set associative cache with one set, meaning all blocks reside in a single set and any block can store any memory address. Caches maintain copies of data stored elsewhere, typically in main memory. Since caches are smaller than main memory, only a subset of the data can be stored, and bookkeeping data is maintained to track the origin of the data. Cache associativity describes how cache blocks are organized and how memory addresses map to cache locations. The fully associative cache is the most flexible, as any address can be stored in any block, functionally equivalent to a set-associative cache with a single set containing all blocks. This flexibility results in the fewest conflict misses for a given cache capacity but requires more hardware for additional tag comparisons, making it best suited for relatively small caches due to the large number of comparators needed.

To determine whether a block is present in the cache, the tag is compared with the tags stored in all cache lines. If a match is found, it is a cache hit, and the data is retrieved from that cache line. If no match is found, it's a cache miss, and the required data is fetched from main memory.

2.2 Snoopy cache protocols

Snoopy cache protocols are very popular in shared bus multiprocessors due to their relative simplicity. They have both write-update and write-invalidate policy versions. Write-invalidate snoopy cache protocols resemble this protocol in many ways and therefore are also easy to understand after studying a write-update protocol.

The definition of transmission routes of commands can be omitted in snoopy cache protocols since the commands are uniformly broadcasted on the shared bus. The protocol applies both the write-back and the write-through update policies. The former is used for private blocks, the latter for shared blocks.

2.3 Write-back

The data is updated only in the cache and updated into the memory at a later time. Data is updated in the memory only when the cache line is ready to be replaced.

Dirty Bit: Each Block in the cache needs a bit to indicate if the data present in the cache was modified(Dirty) or not modified(Clean). If it is clean there is no need to write it into the memory. It is designed to reduce write operation to a memory.
If **Cache fails** or if the **System fails or power** outages the modified data will be lost. Because it's nearly impossible to restore data from cache if lost.

2.4 State of the blocks

MESI Protocol -
It is the most widely used cache coherence protocol. Every cache line is marked with one the following states:

- **Modified -**
  This indicates that the cache line is present in current cache only and is dirty i.e its value is different from the main memory. The cache is required to write the data back to main memory in future, before permitting any other read of invalid main memory state.
- **Exclusive -**
  This indicates that the cache line is present in current cache only and is clean i.e its value matches the main memory value.

- **Shared -**
  It indicates that this cache line may be stored in other caches of the machine.
- **Invalid -**
  It indicates that this cache line is invalid.

Modified	00
Exclusive	01
Shared	10
Invalid	11

2.5 Rules cache write-back

- A write can only be done if the cache line is in the Modified or Exclusive state. If it is in the Shared state, all other cached copies must be invalidated first. A write moves the line into the Modified State.
- A cache can discard a Shared line at any time, changing to the Invalid state. A Modified line is written back first.
- If a cache holds a line in the Modified state, reads from other caches in the system will get the updated data from the cache. Conventionally, this is done by first writing the data to main memory and then changing the cache line to the Shared state, before performing a read.
- A cache that has a line in the Exclusive state must move the line to the Shared state when another cache reads that line.
- The Shared state might not be precise. If one cache discards a Shared line, another cache might not be aware that it could now move the line to Exclusive status.

All possible states of the cache are:

MESI Local Read Hit

  - Line must be in one of MES
  - This must be correct local value (if M it must have been modified locally)

- Simply return value
- No state change

MESI Local Read Miss

- No other copy in caches
  - Processor makes bus request to memory
  - Value read to local cache, marked E
- One cache has E copy
  - Processor makes bus request to memory
  - Snooping cache puts copy value on the bus
  - Memory access is abandoned
  - Local processor caches value
  - Both lines set to S
- Several caches have S copy
  - Processor makes bus request to memory
  - One cache puts copy value on the bus (arbitrated)
  - Memory access is abandoned
  - Local processor caches value
  - Local copy set to S
  - Other copies remain S
- One cache has M copy
  - Processor makes bus request to memory
  - Snooping cache puts copy value on the bus
  - Memory access is abandoned
  - Local processor caches value
  - Local copy tagged S
  - Source (M) value copied back to memory
  - Source value M -> S

MESI Local Write Hit

Line must be one of MES

- M
  - line is exclusive and already ‘dirty’
  - Update local cache value
  - no state change
- E
  - Update local cache value
  - State E -> M

- S
  - Processor broadcasts an invalidate on bus
  - Snooping processors with S copy change S->I
  - Local cache value is updated
  - Local state change S->M

MESI Local Write Miss

Detailed action depends on copies in other processors

- No other copies
  - Value read from memory to local cache
  - Value updated
  - Local copy state set to M
- Other copies, either one in state E or more in state S
  - Value read from memory to local cache -
    bus transaction marked RWITM (read with intent to modify)
  - Snooping processors see this and set their copy state to I
  - Local copy updated & state set to M
- Another copy in state M
  - Processor issues bus transaction marked RWITM
  - Snooping processor sees this
    - Blocks RWITM request
    - Takes control of bus
    - Writes back its copy to memory
    - Sets its copy state to I
- Another copy in state M (continued)
  - Original local processor re-issues RWITM request
  - Is now simple no-copy case
    - Value read from memory to local cache
    - Local copy value updated
    - Local copy state set to M

2.6 Replacement policy

When the cache hits its capacity limit, the Least Recently Used (LRU) cache eviction policy is designed to eliminate the item that has been accessed the least recently. Items that have not been accessed for a longer period of time are assumed to be less likely to be used in the near future. When the cache is full, LRU evicts

the item that hasn't been accessed in the longest time since it keeps track of the order in which items are accessed.

The basic idea behind implementing the LRU (Least Recently Used) cache in this hardware project is to manage element access and removal efficiently using a **shift-register array approach**. Instead of pointers or hash maps, the system uses a physical array of 16 lines where the physical index determines the age of the data.

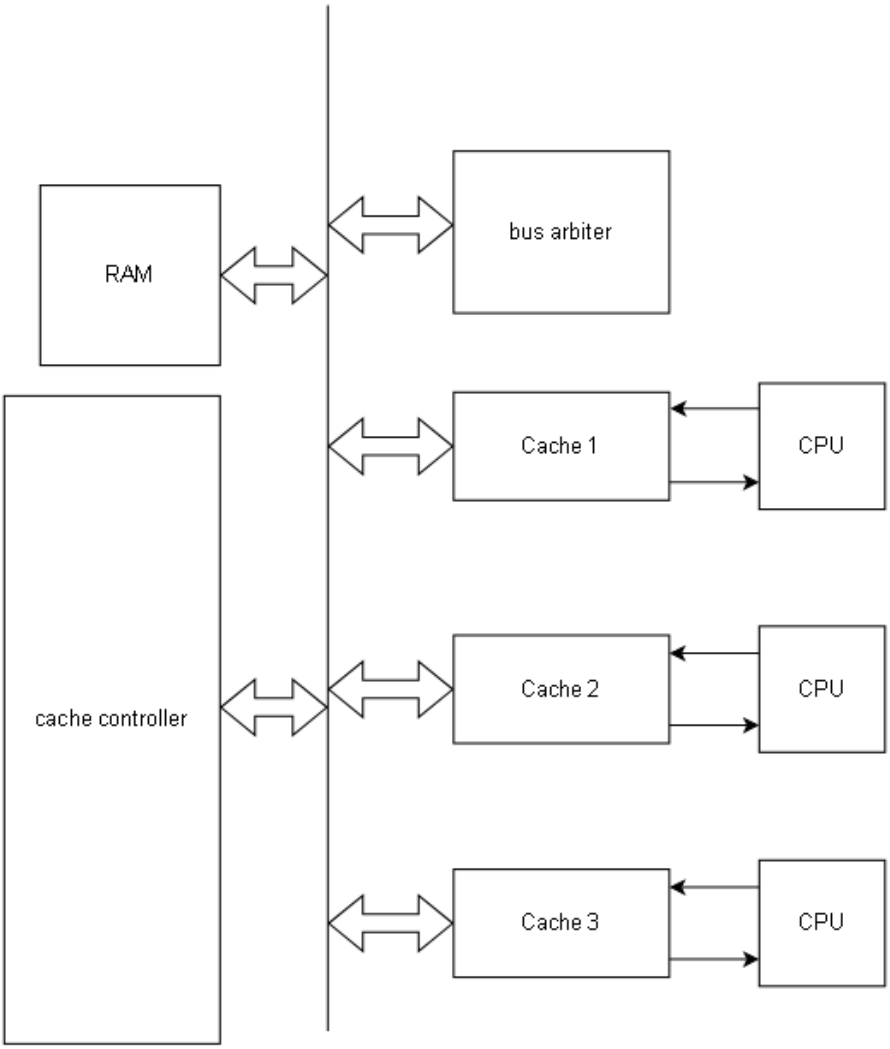
- **Adding a New Key-Value Pair (Cache Miss)** When adding a new cache line (on a Miss), the entire array is shifted downwards by one position (index j moves to j+1). The new line is then written directly to **Index 0**. This physical movement ensures that the newly added line is physically located at the "Most Recently Used" position.

- **Updating an Existing Key (Cache Hit)** If the key (Tag) is already present in the cache at a specific index, the system performs a shift operation. It saves the data at Index k, shifts all lines from Index 0 to k-1 down by one position, and then writes the saved data back into **Index 0**. This operation ensures that the accessed line is moved to the front of the array while preserving the relative order of the other lines

- **Priority and Aging** The priority of lines is determined strictly by their **physical array index**. Index 0 represents the Most Recently Used (MRU) line and has the highest priority. As new data enters or existing data is accessed, older lines "drift" towards higher indices. The last index (Index 15) represents the Least Recently Used (LRU) line.

- **Eviction Strategy** When the cache needs to store a new line and the array is full, the eviction is implicit. Because the entire array shifts down to make room at Index 0, the data currently at the **last index (Index 15)** is simply pushed out of the array. The system checks this last line before shifting; if it is marked as "Modified," it is sent to the Write-Back buffer before being overwritten.

3. Design and components



3.1 Main memory

I decided to use a main memory with 2^{16}addresses, and each block has a size of 1 byte.

Main memory

Data 8 bits

The read signal indicates that there is a read request on the data bus, while the write signal indicates that a write operation is requested.

address – 16 bits

read data – 32 bits

write data – 32 bits

read, write – 1 bit

The RAM operates synchronously. On the rising edge of the clock:

- **Write Operation:** If the i_write signal is asserted, the 32-bit value on the write data bus is stored at the specified address index.
- **Read Operation:** The memory continuously outputs the data located at the current address index. The system logic ensures this data is only sampled by the cache when the read operation is valid and the RAM latency counter has been satisfied (as managed by the top_module).

3.2 Cache

I have 3 cache memories, each cache line is 47 bits. I am using 16 lines for all 3 caches. Of the 48 bits, 14 are for the tag, 2 for the MESI protocol, and 32 for the data.

Tag 14 bits	MESI 2bits	Data 00 8 bits	Data 01 8 bits	Data 10 8 bits	Data 11 8 bits

3.2.1 Processes

1. CPU Hit Detection Process

This process purely monitors the current state of the cache to see if the CPU's requested address is already present.

- It loops through all 16 cache lines.
- It compares the Tag (upper 14 bits) of each line with the CPU's address.
- It checks if the line is valid (MESI state is not "11" (invalid)).
- Output: If a match is found, it sets the s_hit signal to '1' and records the s_hit_index so the main process knows where the data is.

2. Snoop Hit Detection & Broadcast Process

This process handles the "Snooping" logic required for cache coherence.

- It compares the Snoop Address (i_address_cc) sent by the controller against all internal cache tags.
- Broadcast: If it finds a match (and the line is valid), it *immediately* outputs the data (o_data_broadcast) and the MESI state (o_mesi_cache). This allows the controller to see if this cache has a modified copy of the data another core is asking for.
- Hit Flag: If i_data_request is active, it asserts o_snoop_hit to inform the controller that this cache holds the data.

3. Main Sequential Process

It executes on the rising edge of the clock. Its responsibilities are split into two parts:

A. Snoop Updates (Coherence Action)

- If the controller signals an update (i_snoop_update_en), this process modifies the MESI state of a specific line (e.g., changing "Modified" to "Shared" or "Invalid").

- Crucial Logic: It includes the wait for memory availability (i_mem_ready) before downgrading a "Modified" line, ensuring flushes complete correctly.

B. CPU State Machine

- S_IDLE:
  - Hit Logic: If a hit occurs, it updates the LRU (Least Recently Used) order by moving the accessed line to Index 0. For writes, it updates the data and marks the state as Modified ("00"). For reads, it outputs the data to the CPU.
  - Miss Logic: If a miss occurs, it determines if an eviction is needed (checking the last line at Index 15). It prepares the memory request (o_mem_request) and transitions to the appropriate wait state.
- S_MISS_WAIT_WRITE / S_MISS_WAIT_READ:
  - These states pause operation while waiting for the Main Memory (RAM) to provide data (i_mem_ready).
  - Once memory is ready, it shifts the array to make space, constructs the new cache line (Tag + State + Data)

3.2.2 Signals

1. Global Signals

- i_clk: System clock signal. All cache operations are synchronous to the rising edge of this clock.
- i_rst: Synchronous reset signal. When active ('1'), it invalidates all cache lines and resets the state machine to IDLE.

2. CPU Interface (Core <> Cache)

- i_address_cpu: The 16-bit memory address requested by the CPU. The cache uses the upper 14 bits as the Tag and the lower 2 bits for Byte Offset.

- i_write_data_cpu: The 8-bit (1-byte) data the CPU wants to write into the cache.
- i_read: Active-high control signal indicating the CPU wants to perform a Read operation.
- i_write: Active-high control signal indicating the CPU wants to perform a Write operation.
- o_read_data_cpu: The 8-bit data byte returned to the CPU during a read hit or after a miss refill.
- o_hit: Status signal indicating immediate success. '1' means the requested data was found in the cache (Hit); '0' indicates a Miss (CPU must wait).

3. Controller & Snoop Interface (Cache <> Controller)

These signals allow the Cache Controller to maintain coherence (MESI) across cores.

- i_address_cc: The 16-bit "Snoop Address" received from the Controller. The cache checks its tags against this address to see if it holds a copy of the data being accessed by another core.
- i_data_request: A control signal from the Controller asking the cache to check for a Snoop Hit. If high, the cache compares i_address_cc with its tags.
- i_snoop_update_en: A control signal authorizing the cache to update the MESI state of a snoop-hit line (e.g., downgrading from Modified to Shared).
- i_snoop_new_state: The specific 2-bit MESI state (e.g., Shared 10) provided by the Controller that the cache must adopt during a snoop update.
- o_snoop_hit: Output signal telling the Controller that the i_address_cc was found in this cache (Tag Match).
- o_data_broadcast: The 32-bit data word sent to the system bus during a Snoop Hit. If this cache has Modified data, it places it here so it can be flushed to RAM or sent to another cache.
- o_mesi_cache: The current MESI state of the line found during a snoop check. Used by the Controller to determine if a Flush is needed (if state is M).

4. Memory Interface (Cache <> Bus/RAM)

- o_mem_request: Active-high signal requesting access to the main system bus (generated on a Cache Miss).
- o_mem_address: The 16-bit address sent to the Bus Arbiter/RAM to fetch the missing cache line.
- o_rwitm: "Read With Intent To Modify". A specialized request signal indicating the cache wants to write to a line it doesn't have (Write Miss). It tells the controller to fetch the data *and* invalidate other copies immediately.
- i_mem_ready: Handshake signal indicating that the RAM (or Top Module logic) has completed the read/write operation and valid data is available.
- i_mem_data: The 32-bit data word fetched from main memory to refill a cache line.
- i_mesi_cc: The MESI state assigned to the new line fetched from memory (usually Exclusive 01).
- o_writeback_en: Control signal indicating an Eviction. It tells the Top Module that the data on o_data_out is valid and must be written to RAM before the new read can proceed.
- o_data_out: The 48-bit evicted line (Tag + MESI + Data) removed from the cache to make room for new data. This is sent to the write-back buffer.

3.3 Cache controller

The Cache Controller operates using two distinct processes to manage bus traffic and enforce the MESI coherence protocol.

1. Flush Timer Process (Sequential)

This synchronous process manages the timing for Abort/Flush operations. When the controller detects that a requested memory block is held in the "Modified" state by another core, it asserts the o_bus_abort signal. This process implements a wait counter that delays the state update (downgrading M $\to$ S) until the flush to RAM is fully completed, preventing the race condition where RAM might read stale data.

2. Coherence Control Logic (Combinatorial)

This process acts as the decision engine for the system. It continuously "snoops" the bus address (i_bus_addr) and performs the following actions:

- Snoop Broadcast: It drives the o_snoop_check signals to query all caches (except the requesting core) for the target address.
- State Enforcement: Based on the snoop results (i_snoop_hit and i_snoop_mesi), it determines if a Flush is required (if the hit is on a 'Modified' line) or if an Invalidation is needed (for Write misses).
- Grantee Update: It calculates the correct initial state (o_response_mesi) for the requesting core—assigning Shared (S) if copies exist elsewhere, or Exclusive (E) if the data is unique to the requester.

3.3.1Process

Flush Timer Process (Sequential)

Handles the timing for the Abort/Flush mechanism.

- Logic:
    - When an Abort is generated (because a "Modified" snoop hit occurred), the RAM needs 2 clock cycles to write the dirty data.
    - This process implements a counter. It ensures the o_snoop_update_en signal (which downgrades the owner from M to S) is delayed until the flush completes.
    - Without this delay, the owner would become "Shared" too early, causing the Abort signal to drop before the RAM write was finished.

Control & Protocol Logic Process (Combinatorial)

Implements the MESI decision matrix.

- Logic Steps:
    1. Trigger Snoops: It asserts o_snoop_check for all cores *except* the i_bus_source_id .

2. Analyze Results: It checks i_snoop_hit and i_snoop_mesi for every snooping core.
3. Handle Read Request (CMD_BUS_RD):
    - Hit on Modified (M): Asserts o_bus_abort (Flush). Sets new state for owner to Shared (S).
    - Hit on Exclusive (E): No Flush needed. Sets new state for owner to Shared (S).
    - Hit on Shared (S): No action needed (already Shared).
4. Handle Write Request (CMD_BUS_RDX):
    - Hit on Any Valid State (M, E, S): Asserts o_snoop_update_en and sets new state to Invalid (I).
    - If the hit was on M, it *also* asserts o_bus_abort to flush the old data before invalidating.
5. Determine Grantee State:
    - If any other core reported a Hit (v_copy_exists = '1'), the requesting core receives the line in the Shared (S) state.
    - If no one else has it, the requesting core receives it in the Exclusive (E) state

3.3.2 Signals

1. Global Signals

- clk: System clock. The controller uses this for the sequential flush timer logic.
- rst: Asynchronous reset. Clears the flush counter and resets the controller logic.

2. Bus Interface (Input from Bus/Arbiter)

- i_bus_addr (16-bit): The address currently being accessed on the bus by the "Grantee" (the core that owns the bus).
-

- i_bus_cmd (2-bit): The command issued by the Grantee.
    - 01 (Read): Standard read request (Line Miss).
    - 10 (Read Exclusive): Read with intent to modify (Write Miss).
- i_bus_source_id (Integer 0-2): The ID of the core currently acting as the Grantee. The controller ignores snoops from this ID (a core cannot snoop itself).
- o_bus_abort: Critical Signal. Active-high output sent to the Top Module.
    - It indicates "Stop! Do not read from RAM yet.". It is asserted when a Snoop Hit finds a Modified (M) line in another cache. It forces the system to pause the Read and perform a Write-Back (Flush) of the dirty data to RAM first.

3. Snoop Interface (To/From Caches)

- o_snoop_addr (16-bit): A copy of the bus address sent to *all* caches (except the source) to check their tags.
- o_snoop_check (3-bit): Signal telling specific caches to check their tags against o_snoop_addr.
- i_snoop_hit (3-bit): Input vector where each bit corresponds to a core (0, 1, 2). A '1' indicates that the core has a copy of the requested address.
- i_snoop_mesi (Array): The current MESI state of the line found in each snooping core. The controller uses this to decide if a Flush (M state) is required.
- o_snoop_update_en (3-bit): Control signal telling a specific cache to update the state of its snoop-hit line.
- o_snoop_new_state (Array): The *new* MESI state the cache must adopt (e.g., downgrading M to S).

4. Response Interface

- o_response_mesi (Array): The MESI state assigned to the *Requesting Core* (Grantee) for its new line.
    - If valid copies exist elsewhere Returns Shared (S).
    - If no copies exist Returns Exclusive (E).

3.4 Cores

The Core (CPU) is implemented not as a full instruction processor, but as a deterministic traffic generator controlled by a Finite State Machine (FSM). Its primary purpose is to issue Read/Write requests to the L1 Cache and validate the response latency.

The process operates through four sequential states:

- S_IDLE: The core waits for a start command (cmd_enable) from the testbench. Once triggered, it latches the target address and data.
- S_ACCESS: The core drives the address (o_address_cpu) and asserts the Read (o_read) or Write (o_write) strobe to the L1 Cache interface.
- S_WAIT_HIT: The core enters a wait state, holding the request active until the L1 Cache asserts the Hit (i_hit) signal. This handles both immediate cache hits and the variable latency of cache misses (waiting for RAM).
- S_DONE: Upon receiving the Hit signal, the core captures any read data, de-asserts the control strobes, and outputs a status_done signal to handshake with the testbench.

3.4.1 Process

The CPU logic is implemented in a single Sequential Finite State Machine (FSM)

State Machine: r_state

1. S_IDLE
    - Purpose: Waits for a command from the testbench.
    - Behavior: Checks cmd_enable. If high, it latches the address, data, and R/W mode into internal registers (r_addr, r_data, r_rw) and transitions to S_ACCESS .
2. S_ACCESS
    - Purpose: Applies the request to the Cache interface.
    - Behavior:
        - Drives o_address_cpu with the latched address.
        - If writing, drives o_write_data_cpu and asserts o_write .

        - If reading, asserts o_read.
        - Transitions immediately to S_WAIT_HIT.

3. S_WAIT_HIT
    - Purpose: Blocks execution until the cache responds.
    - Behavior:
        - It continuously checks i_hit.
        - If i_hit = '0' (Miss/Wait): It keeps o_read or o_write asserted to maintain the request active .
        - If i_hit = '1' (Hit/Done):
            - If it was a Read operation, it captures i_read_data_cpu into status_data_out.
            - It de-asserts o_read and o_write.
            - Transitions to S_DONE.
4. S_DONE
    - Purpose: Signals completion to the testbench.
    - Behavior: Sets status_done <= '1' for one clock cycle and returns to S_IDLE to await the next command .

3.4.2 Signals

1. Global Signals

- clk: System clock. The CPU state machine advances on the rising edge.
- rst: Synchronous reset. Resets the state machine to S_IDLE and clears all outputs.

2. Command Interface (Input from Testbench)

- cmd_enable: Active-high start signal. When '1', the CPU begins processing the command present on the inputs.
- cmd_rw: Read/Write selector.

    - 0: Read Request.
    - 1: Write Request.
- cmd_addr (16-bit): The address the testbench wants to access.
- cmd_data_in (8-bit): The data to be written (only used if cmd_rw = '1').

3. Status Interface (Output to Testbench)

- status_done: Active-high handshake signal. Indicates that the requested memory transaction (Read or Write) has completed successfully.
- status_data_out (8-bit): The data retrieved from the cache (only valid after a Read operation completes).

4. Cache Interface (To/From Cache)

- o_address_cpu (16-bit): The address sent to the L1 Cache.
- o_write_data_cpu (8-bit): The data byte sent to the L1 Cache for writing.
- o_read: Active-high Read strobe sent to the cache.
- o_write: Active-high Write strobe sent to the cache.
- i_read_data_cpu (8-bit): Input data received from the L1 Cache during a read.
- i_hit: Input signal from the cache indicating the request was served (Hit). The CPU waits for this signal to go high before finishing the transaction.

3.5 Bus Arbiter

The Bus Arbiter acts as the central gatekeeper for the shared system bus, ensuring that only one cache accesses the main memory or broadcasts signals at a time. It prevents data collisions by resolving contention between simultaneous requests from multiple cores.

3.5.1 Processes

The arbiter is implemented using a single synchronous process.

Arbitration Policy: Fixed Priority The current implementation uses a Fixed Priority Scheme where lower ID numbers have higher priority. This is defined by the if-elsif structure inside the process:

1. Priority 1 (Highest): Core 0. If i_req(0) is high, Core 0 is granted the bus immediately, regardless of other requests .
2. Priority 2: Core 1. If Core 0 is not requesting but i_req(1) is high, Core 1 is granted access .
3. Priority 3 (Lowest): Core 2. Granted only if neither Core 0 nor Core 1 is requesting access .

Idle State If no bits in i_req are active, the arbiter de-asserts o_bus_active, signaling to the system that the bus is free and no transactions should occur.

3.5.2 Signals

1. Global Signals

- clk: System clock. The arbitration logic is synchronous and evaluates requests on the rising edge.
- rst: Asynchronous reset. When asserted, it resets the grant ID to 0 and marks the bus as inactive.

2. Request Interface (Input from Caches)

- i_req (3-bit vector):
    - The request lines from the three Caches.
    - Bit 0: Request from Cache 0.
    - Bit 1: Request from Cache 1.
    - Bit 2: Request from Cache 2.
    - These signals are connected to the o_mem_request output of each cache.

3. Grant Interface (Output to System)

- o_grant_id (Integer 0 to 2):
    - The ID of the core currently granted access to the bus.

    - This signal is used by the Top Module to control the multiplexers that route address and data signals to the RAM and Cache Controller.
- o_bus_active:
    - Active-high Status signal.
    - '1': The bus is currently in use (a request is being served).
    - '0': The bus is idle (no requests pending).

3.6 Top module

The Top Module is the structural backbone of the project. It instantiates all sub-components (CPUs, Caches, Controller, Arbiter, RAM) and implements the Shared Bus Logic. It handles the multiplexing of signals between the active bus master and the system, and it generates the critical timing signals required for RAM operations.

3.6.1 Processes

RAM Latency Sequencer (Sequential Process)

Manages the timing for memory transactions to support the "Single-Port RAM" constraint.

- Logic:
    - It uses a counter (s_mem_counter) that increments when the bus is active.
    - Scenario A (Normal Read): Counts to 1, then asserts s_ram_ready_delayed.
    - Scenario B (Write-back / Flush): Counts to 2.
        - Cycle 0: Allows RAM to write the dirty data (Eviction or Flush).
        - Cycle 1: Allows RAM to read the new requested data.

        - Cycle 2: Asserts s_ram_ready_delayed to tell the cache the data is valid.

2. Bus Multiplexer (Combinatorial Process)

Routes the request from the "Granted" cache to the system bus.

- Logic:
    - If s_bus_active is '0', the bus is floating (Zero/NOP).
    - Otherwise, it assigns s_sys_addr = s_mem_addr(s_grant_id).
    - It translates the cache's o_rwitm signal into the bus command (CMD_BUS_RD or CMD_BUS_RDX).

3. Snoop Data Mux (Combinatorial Logic)

Selects the data source during a Flush operation.

- Logic:
    - If s_snoop_hit(0) is high, it selects data from Cache 0.
    - If s_snoop_hit(1) is high, it selects data from Cache 1, etc.
    - This ensures that when the Controller aborts a transaction to flush Core X, the RAM receives the data from Core X, not the requesting core.

4. RAM Address Mux (Combinatorial Logic)

Handles the "Address Switching" required for atomic evictions.

- Logic:
    - If s_cache_wb_en (Writeback Enable) is high, it checks the Latency Counter.
    - Time < 1: It sends s_evict_addr to the RAM (to write the old line).
    - Time >= 1: It switches to s_sys_addr (to read the new line).
    - This logic fixes the race condition where the RAM would otherwise read from the wrong address during an eviction.

3.6.2 Signals

1. Testbench Interface (Global IO)

- clk, rst: Global system clock and reset.
- cmdX_... / stsX_...: Individual signals for the 3 CPUs, connected directly to the testbench traffic generators. These are packed into arrays (arr_cmd_...) internally to simplify the instantiation loop.

2. Bus Control Signals (Internal)

- s_grant_id (Integer): Output from the Arbiter. Indicates which cache currently owns the bus. This signal controls all the system multiplexers (address mux, data mux, command mux).
- s_bus_active: Status signal from Arbiter. Used to enable/disable RAM operations (RAM is only accessed when the bus is active).
- s_bus_abort: The "Flush" signal from the Controller. When high, it overrides the normal flow to force a RAM write of snoop data.

3. System Bus Lines (Shared Resources)

- s_sys_addr (16-bit): The address currently on the bus.
    - During a normal Read/Write, this comes from the Granted Cache.
    - During a Write-back (Eviction), this switches to the Eviction Address.
- s_sys_cmd (2-bit): The command currently on the bus (Read 01 or Read Exclusive 10).
- s_sys_wdata (32-bit): The data being written to RAM. Muxed from either:
    - The Granted Cache's eviction output (during capacity miss).
    - The Snooping Cache's broadcast output (during a Flush/Abort).

4. RAM Interface Signals

- s_ram_ready_delayed: The "Ready" handshake sent to the caches.
    - Logic: It goes high 1 or 2 cycles after a request starts, depending on whether a write-back is occurring.
- s_ram_write / s_ram_read: Combinatorial control signals for the RAM module, derived from the bus state and abort signals.

4.Testing and verification

The cache was tested using the Vivado simulator and verified on the Basys 3 board.

4.1 Simulation Testing Scenarios

The simulation is controlled by the btn_next_test signal. The testbench pauses after each scenario, allowing the user to inspect the waveform before triggering the next test.

Test 1: Local Read Miss (Cold Start)

- Action: CPU 0 reads address 0x1000.
- Expected Behavior: Since the cache is empty, this triggers a Read Miss. The Controller grants the bus to Core 0, fetches data from RAM, and loads it into Cache 0.
- Verification: Cache 0 state transitions to Exclusive (E). The read operation completes successfully.

Test 2: Remote Read Miss (Snoop Hit on Exclusive)

- Action: CPU 1 reads address 0x1000.
- Expected Behavior: Core 0 (holding the line in E) snoops the request. It acknowledges the hit, preventing a RAM read. Both caches update their state.
- Verification: Both Cache 0 and Cache 1 transition the line 0x1000 to Shared (S).

Test 3: Local Read Hit (Shared)

- Action: CPU 0 reads address 0x1000 again.
- Expected Behavior: The data is already present in the Shared (S) state. The cache returns the data immediately without activating the bus.
- Verification: sts0_done asserts quickly. The bus remains idle. The state remains Shared (S).

Test 4: Write Invalidation (Shared to Modified)

- Action: CPU 1 writes 0xAA to address 0x1000.

- Expected Behavior: Since the state is Shared, CPU 1 cannot write locally. It broadcasts an "Invalidate" command on the bus. CPU 0 detects this and invalidates its copy.
- Verification: Cache 1 upgrades to Modified (M). Cache 0 transitions to Invalid (I).

Test 5: Remote Read Miss (Snoop Flush on Modified)

- Action: CPU 0 reads address 0x1000 (which is now Invalid for Core 0).
- Expected Behavior: CPU 1 holds the modified data (0xAA). The Controller asserts o_bus_abort to pause the read. CPU 1 flushes 0xAA to RAM. After the flush, CPU 0 reads the new data.
- Verification: RAM is updated with 0xAA. Both Cache 0 and Cache 1 transition to Shared (S). The data read by CPU 0 is 0xAA.

Test 6: Silent Upgrade (Exclusive to Modified)

- Step A: CPU 2 reads a new address 0x2000 (transitions to Exclusive).
- Step B: CPU 2 writes 0xBB to 0x2000.
- Expected Behavior: Since CPU 2 already has exclusive ownership, it updates the data locally without issuing a bus command (Silent Upgrade).
- Verification: Cache 2 transitions from Exclusive (E) to Modified (M) with zero bus overhead.

Test 7: Read With Intent To Modify (RWITM)

- Action: CPU 0 writes 0xCC to address 0x2000 (which CPU 2 holds in M).
- Expected Behavior: This is a Write Miss. CPU 0 issues a RWITM command. CPU 2 detects the conflict, flushes its data (0xBB) to RAM, and invalidates itself. CPU 0 then gains ownership.
- Verification: Cache 0 transitions to Modified (M) containing 0xCC. Cache 2 transitions to Invalid (I).

Test 8: Data Verification

- Action: CPU 1 reads address 0x2000.
- Expected Behavior: CPU 0 (owner) flushes the latest data (0xCC) to RAM. CPU 1 reads it.

- Verification: The data returned to CPU 1 must be 0xCC. This confirms the entire history of modifications was preserved correctly.

Test 9: Capacity & Eviction

- Step A: CPU 0 sequentially writes to 16 unique addresses (0x3000 to 0x303C) to fill the cache completely with Modified lines.
- Step B (Eviction): CPU 0 writes to a 17th address (0x0050).
- Expected Behavior: The cache detects it is full. It selects the Least Recently Used (LRU) line (Index 0, address 0x3000, data 0x10) for eviction. The Top Module writes this data to RAM before loading the new line.
- Verification: CPU 1 reads the *evicted* address 0x3000 from RAM. The data returned must be 0x10, proving the write-back occurred successfully.

4.2 Basys 3 Hardware Testing Guide

This section outlines the step-by-step procedure to verify the multi-core cache system on the Basys 3 FPGA board. The test logic is driven by an internal ROM that executes specific memory operations (Reads/Writes) sequentially each time you press the center button.

User Interface (LEDs & Buttons)

- Button C (btnC): STEP. Press to execute the current test step.
- Button U (btnU): RESET. Press to reset the system to Step 0.
- LED 15 (LD15): ERROR. If this turns ON, the test failed (Data Mismatch).
- LEDs 13-6 (LD13-LD6): DATA. Displays the 8-bit Data Read (r_last_read) from the operation just completed.
- LEDs 5-0 (LD5-LD0): STEP #. Binary counter showing the Next Step to be executed.

Perform the following actions in order. Watch the LEDs to verify the result of each step.

1. System Reset

- Action: Press btnU (Reset).
- Verify: All LEDs should turn OFF (Step 0, No Error).

2. Execution Sequence

Step 0: Cold Read Miss

- Action: Press btnC.
- Description: CPU 0 reads address 0x1000. RAM is empty (0x00).
- Expected Result:
    - Step LEDs (LD5-0): 000001 (1)
    - Data LEDs (LD13-6): 00000000 (0x00)
    - Error LED (LD15): OFF

Step 1: Snoop Read (Share)

- Action: Press btnC.
- Description: CPU 1 reads 0x1000. It snoops CPU 0, and both become Shared.
- Expected Result:
    - Step LEDs (LD5-0): 000010 (2)
    - Data LEDs (LD13-6): 00000000 (0x00)
    - Error LED (LD15): OFF

Step 2: Write Upgrade (Modified)

- Action: Press btnC.
- Description: CPU 1 writes 0xAA to 0x1000. CPU 0 Invalidates.
- Expected Result:
    - Step LEDs (LD5-0): 000011 (3)
    - Data LEDs (LD13-6): *(Ignored on Write)*
    - Error LED (LD15): OFF

Step 3: Snoop Flush Test

- Action: Press btnC.
- Description: CPU 0 reads 0x1000. CPU 1 must flush 0xAA to RAM.
- Expected Result:
    - Step LEDs (LD5-0): 000100 (4)
    - Data LEDs (LD13-6): 10101010 (0xAA)
    - Error LED (LD15): OFF

Step 4: Write Invalidate

- Action: Press btnC.
- Description: CPU 2 writes 0xBB to 0x1000. It invalidates other copies.
- Expected Result:
    - Step LEDs (LD5-0): 000101 (5)
    - Data LEDs (LD13-6): *(Ignored on Write)*
    - Error LED (LD15): OFF

Step 5: Read Verification

- Action: Press btnC.
- Description: CPU 1 reads 0x1000. Should see the new data 0xBB.
- Expected Result:
    - Step LEDs (LD5-0): 000110 (6)
    - Data LEDs (LD13-6): 10111011 (0xBB)
    - Error LED (LD15): OFF

Step 6 to 21: Cache Fill Burst (Automatic)

- Action: Press btnC ONCE.
- Description: CPU 0 sequentially writes to 16 addresses (0x3000 to 0x303C).
- Behavior: The system detects the fill loop and executes steps 7 through 21 automatically without waiting for button presses.

- Expected Result:
    - The Step LEDs will count up rapidly from 000111 to 010110.
    - Final Step LEDs (LD5-0): 010110 (22)
    - Error LED (LD15): OFF

Step 22: Force Eviction

- Action: Press btnC.
- Description: CPU 0 writes to 0x4000. Since the cache is full, this forces the oldest line (0x3000) to be written back to RAM.
- Expected Result:
    - Step LEDs (LD5-0): 010111 (23)
    - Data LEDs (LD13-6): *(Ignored on Write)*
    - Error LED (LD15): OFF

Step 23: Verify Eviction

- Action: Press btnC.
- Description: CPU 2 reads 0x3000 from RAM. It should find the data 0x10 that was evicted in Step 22.
- Expected Result:
    - Step LEDs (LD5-0): 011000 (24)
    - Data LEDs (LD13-6): 00010000 (0x10)
    - Error LED (LD15): OFF

Troubleshooting

- LED 15 Turns ON: This indicates a Data Mismatch. The system read a value different from the expected one. Press Reset (btnU) and retry.
- Burst Mode didn't stop: Ensure s_rom_step logic in top_m.vhd correctly checks for >= 7 and <= 21.

5.Conclusion

This project successfully designed, implemented, and verified a multi-core cache memory system capable of maintaining data coherence across three processors. By utilizing the MESI (Modified, Exclusive, Shared, Invalid) protocol on a shared bus architecture, the system demonstrated robust data consistency without relying on complex directory-based schemes.

Key Technical Achievements:

- Coherence Logic: The design effectively implemented a "Snooping" Cache Controller that monitors bus traffic to enforce protocol rules. The system correctly handles complex state transitions, including Bus Snarfing (updating Shared copies), Write Invalidation (broadcasting intent to modify), and Snoop Flushes (aborting reads to write back modified data).
- Memory Management: The implementation of a Least Recently Used (LRU) replacement policy ensured efficient cache utilization. The logic successfully manages capacity misses by automatically evicting old "Modified" data to RAM before accepting new entries.
- Synchronization: A critical achievement was the resolution of timing race conditions between the Cache Controller and Main Memory. The introduction of a "Flush Timer" and the o_bus_abort signal ensured that "dirty" data was fully written to RAM before any other core could read it, preventing stale data access.

Verification Results: The system was validated through a two-stage process:

1. Simulation (Vivado): Waveform analysis confirmed that all MESI state transitions occur cycle-accurately and that the arbiter correctly manages bus contention.
2. Hardware Emulation (Basys 3): The design was successfully synthesized and deployed on an FPGA. The automated test suite verified the end-to-end data path—from cache misses to snoop hits and evictions—proving that the logic operates correctly in a real-world hardware environment.

In conclusion, this project demonstrates a fully functional, synthesizable solution for cache coherence in embedded multiprocessor systems. The design serves as a reliable foundation that can be extended with larger cache sizes, multi-level hierarchies (L2), or more advanced arbitration policies.

6. Bibliography

[1] ScienceDirect, "Fully Associative Cache," *ScienceDirect Topics*. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/fully-associative-cache.

[2] Ginni, "What are snoopy cache protocols in computer architecture?," *Tutorialspoint*, Jul. 23, 2021. [Online]. Available: https://www.tutorialspoint.com/what-are-snoopy-cache-protocols-in-computer-architecture.

[3] GeeksforGeeks, "Cache Eviction Policies - System Design," *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/system-design/cache-eviction-policies-system-design/.

[4] GeeksforGeeks, "LRU Cache Implementation," *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/system-design/lru-cache-implementation/.

[5] GeeksforGeeks, "Cache Coherence Protocols in Multiprocessor System," *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/computer-organization-architecture/cache-coherence-protocols-in-multiprocessor-system/.

[6] University of Crete (CSD), "Lecture 15: Snoopy Coherence Protocols," *HY425 Computer Architecture Course Notes*. [Online]. Available: https://www.csd.uoc.gr/~hy425/2016f/lectures/Lec15_snoop_coherence.pdf.

[7] University of Michigan EECS, "Bus Basics and Bus Transactions," *EECS 373 Lecture Notes*, 2004. [Online]. Available: https://www.eecs.umich.edu/courses/eecs373.w04/Lectures/stever_old_lectures/lec3.pdf.