ECE 585
Simulated Cache Test Plan
Team 4

Portland State University
Maseeh College of Engineering & Computer Science
Department of Electrical & Computer Engineering

Authors:
Kane-Pardy, Chris
Melli, Ameer
Monti, Caleb

November 19, 2024

ECE 585
Microprocessor System Design

Contents

1 Debug Mode (Verbose)

Objectives

- **Ensure Full Verbose Output with Internal State Tracking:** Verify that when Debug Mode (Verbose) is enabled, the simulation outputs detailed information on every internal operation, including cache actions, bus operations, snoop responses, internal state transitions, decisions (like evictions and misses), and any ‘fprintf‘ debug messages used throughout the code.
- **Verify Mode Switching Works Correctly Without Recompilation:** Ensure that the simulation can easily switch between modes (Normal and Debug) via command-line arguments, with Debug Mode providing the most granular information, including verbose cache operation logs and any ‘fprintf‘ output, and without needing to recompile the system.

Test Scenarios and Procedures

1. Ensure Full Verbose Output and Track Internal State Transitions

1.1 Run a Trace File in Debug Mode (Verbose)

Procedure:

1. Run a trace file that includes cache read and write operations, enabling Debug Mode (Verbose).
2. Verify that the output includes:
    - Detailed logs of every cache read/write operation, including address, action, and cache state changes.
    - All bus operations (e.g., READ, WRITE, INVALIDATE) with relevant address and data (if applicable).
    - Snoop results (e.g., HIT, HITM) and how the system handles these snoops.
    - Intermediate state transitions, such as evictions, misses, and replacements during cache operations.
    - Any ‘fprintf‘ debug messages inserted in the code (e.g., debug prints showing variable states, function calls, or cache-related actions).
    - Information about the application of the replacement policy (e.g., pseudo-LRU), showing how cache access patterns influence eviction and replacement decisions.
    - A final summary of statistics after the trace execution, including:
        - Total cache reads
        - Total cache writes
        - Cache hits
        - Cache misses
        - Cache hit ratio

Expected Outcome:
The verbose output should show comprehensive logs for every operation, cache state change, bus interaction, internal decision-making point (e.g., eviction or miss decisions), and all debug messages from ‘fprintf‘ statements. At the end of the trace, the output should include the accurate final summary statistics (reads, writes, hits, misses, hit ratio).

2. Verify Mode Switching Works Correctly Without Recompilation

2.1 Toggle Between Normal and Debug Mode (Verbose) Using Command-Line Arguments

Procedure:

1. Run a trace file in Normal Mode and verify the output includes only basic logs (e.g., cache read/write, hits/misses, bus operations).
2. Switch to Debug Mode (Verbose) via command-line arguments, without recompiling the system.
3. Re-run the same trace and verify that the output now includes detailed logs for cache operations, bus activity, snoop responses, and ‘fprintf‘ debug output.
4. Toggle back to Normal Mode and verify that the output returns to basic logs without ‘fprintf‘ debug messages.

Expected Outcome:
The mode can be toggled between Normal and Debug Mode (Verbose) seamlessly via command-line arguments, with Debug Mode displaying significantly more detailed output, including cache operations, state transitions, bus interactions, and ‘fprintf‘ debug messages.

Expected Results Summary

Each test scenario should confirm that:

- **Verbose Output with State Tracking:** Debug Mode should provide full and detailed logs for every operation, cache transaction, and snoop result, including internal state transitions, decision-making (e.g., evictions, misses, replacements), and any ‘fprintf‘ debug messages used in the code. Additionally, a final summary of statistics (reads, writes, hits, misses, hit ratio) should be displayed after the trace execution.
- **Mode Switching:** The simulation should allow toggling between Normal and Debug Mode (Verbose) via command-line arguments without recompilation. Debug Mode should output significantly more detailed information, including ‘fprintf‘ debug output.

2 Normal Mode Functionality

Objectives

- **Validate Detailed Output in Normal Mode:** Ensure that when running in Normal Mode, the simulation provides both real-time logs of cache operations, bus activities, snoop results, and internal state transitions, as well as a final summary of statistics (cache reads, writes, hits, misses, and hit ratio).
- **Ensure Mode Switching Works Correctly Without Recompilation:** Verify that Normal Mode and Silent Mode can be toggled seamlessly via command-line arguments, adjusting the output format accordingly, without requiring recompilation of the simulation.

Test Scenarios and Procedures

1. Validate Output in Normal Mode

1.1 Execute a Trace File in Normal Mode and Check Output

Procedure:

1. Run a trace file with a mix of cache read/write operations, snoop requests, and bus transactions, ensuring Normal Mode is enabled.
2. Observe the output during execution and verify it includes:
    - Per-operation logs for cache operations, such as hits, misses, writes, and reads.
    - Detailed bus operations (e.g., READ, WRITE, INVALIDATE).
    - Snoop results (e.g., HIT, HITM).
    - No information is omitted from the real-time logs during the trace.
3. After the trace completes, confirm that the following summary statistics are displayed:
    - Total cache reads
    - Total cache writes
    - Cache hits
    - Cache misses
    - Cache hit ratio
4. Verify that these statistics are accurate based on the trace operations.

Expected Outcome:
In Normal Mode, the simulation should display detailed logs during the trace, including cache operations, bus transactions, and snoop results, followed by an accurate final summary of statistics (reads, writes, hits, misses, hit ratio).

2. Ensure Mode Switching Works Correctly Without Recompilation

2.1 Toggle Between Normal and Silent Mode Using Command-Line Arguments

Procedure:

1. Run a trace file with Normal Mode enabled and verify that detailed logs and the final statistics are shown.
2. Without recompiling, switch to Silent Mode via the command-line arguments and re-run the same trace.
3. Confirm that Silent Mode only shows the summary statistics at the end, while Normal Mode shows both detailed logs during execution and the summary at the end.

Expected Outcome:
Normal and Silent Modes should be toggleable via command-line arguments without recompilation. In Normal Mode, the output should show detailed logs during the execution and the summary at the end. In Silent Mode, only the summary statistics should be shown.

Expected Results Summary

Each test scenario should confirm that:

- **Full Output in Normal Mode:** Normal Mode displays detailed logs during the trace, including cache operations, bus activities, and snoop results, followed by accurate summary statistics (reads, writes, hits, misses, hit ratio).
- **Mode Switching:** Normal and Silent Modes can be toggled without recompilation. Normal Mode outputs both detailed logs and the final statistics, while Silent Mode shows only the summary statistics.

3 Silent Mode Functionality

Objectives

- **Verify Suppression of Detailed Logs in Silent Mode:** Ensure that all detailed logs, including cache operations, bus activities, and snoop results, are suppressed in Silent Mode so that only summary statistics appear at the end of the trace.
- **Ensure Proper Display of Cache Usage Summary:** Confirm that Silent Mode correctly displays a concise summary of cache usage statistics at the end of each trace, including reads, writes, hits, misses, and hit ratio, without detailed intermediate output.
- **Confirm Mode Toggle Functionality Without Recompilation:** Verify that Silent Mode can be toggled on or off without recompilation, ideally via command-line arguments, to confirm flexible mode selection for testing and deployment.

Test Scenarios and Procedures

1. Verify Suppression of Detailed Logs and Display of Cache Usage Summary in Silent Mode

1.1 Run a Trace File in Silent Mode and Check Summary Statistics

Procedure:

1. Execute a trace file in Silent Mode that includes a variety of operations (reads, writes, snoop requests).
2. Observe the output during execution and confirm that:
    - No intermediate logs for individual cache operations, bus transactions, or snoop results are displayed.
    - At the end of the trace, only a concise summary with the following statistics is displayed:
        - Total cache reads
        - Total cache writes
        - Cache hits
        - Cache misses
        - Cache hit ratio

Expected Outcome:

Silent Mode suppresses all detailed logs and only displays the summary statistics at the end of the trace. The statistics accurately reflect the counts from the trace.

2. Confirm Mode Toggle Functionality Without Recompilation

2.1 Toggle Silent and Normal Mode with Command-Line Argument

Procedure:

1. Run a trace file first in Silent Mode and confirm that only the summary is displayed.
2. Without recompiling, run the same trace file in Normal Mode (by providing a different command-line argument) and confirm that detailed logs and summary statistics appear.

Expected Outcome:

Silent and Normal Modes can be toggled easily without requiring recompilation, demonstrating mode flexibility and allowing either mode to be selected as needed.

Expected Results Summary

After executing each test:

- **Suppression of Detailed Logs:** Detailed logs are correctly suppressed in Silent Mode, with no intermediate cache operation logs or snoop results displayed.
- **Summary Statistics:** Silent Mode displays only the summary statistics at the end of the trace, covering the key metrics: reads, writes, hits, misses, and hit ratio.
- **Mode Toggle Functionality:** Mode switching works without recompilation, enabling seamless toggling between Silent and Normal Modes.

4 Associativity Verification

Objectives

- **Verify 16-Way Associativity:** Ensure the cache can store up to 16 unique lines in a single set. In a 16-way associative cache, each set can hold 16 different cache lines, allowing multiple memory addresses mapping to the same set index to coexist.
- **Check Cache Set Overflow:** Confirm that when more than 16 unique lines are mapped to the same set, the cache correctly evicts lines according to the eviction policy, maintaining a maximum of 16 lines per set.
- **Validate Address Mapping and Set Distribution:** Ensure that lines with the same set index but different tags are stored in different ways within each set, confirming correct mapping and distribution.

Test Scenarios and Procedures

1. Verify 16-Way Associativity

1.1 Fill a Set with 16 Unique Lines

Procedure:

1. Select a single set index and generate 16 unique addresses that should map to this set.
2. Load each address into the cache sequentially and confirm that all 16 lines fit into the set without eviction.

Expected Outcome:
All 16 lines should be stored in the selected set, confirming that the set can hold up to 16 unique lines in a 16-way set associative cache.

1.2 Verify Proper Distribution of Lines Across 16 Ways

Procedure:

1. Load 16 unique addresses into a single set.
2. Verify that each address maps to a different way within the set, ensuring that the cache uses all 16 ways for the given set.

Expected Outcome:
Each address is correctly mapped to a separate way within the set, demonstrating proper distribution and usage of all 16 ways.

2. Check Cache Set Overflow

2.1 Exceed Capacity of a Set (Eviction on 17th Line)

Procedure:

1. Load 16 unique addresses into a single set.
2. Add a 17th unique address and confirm that it causes an eviction in the selected set, which already holds 16 lines. (This does not need to explicitly test pseudo-LRU, just that an "overflow" in our set causes an eviction.

Expected Outcome:
The 17th line triggers an eviction in the selected set, confirming that the cache correctly enforces the set's capacity limit of 16 lines.

3. Mapping and Set Distribution

3.1 Verify Cross-Set Mapping (Different Index & Tags)

Procedure:

1. Choose multiple addresses with different indexes and tags.
2. Load these addresses into the cache and confirm that they are mapped to different sets.

Expected Outcome:
Addresses with different set indexes should map to different sets, ensuring that the cache uses both the index and tag to determine set placement correctly.

3.2 Verify Mapping When Tags Are Identical but Indexes Differ

Procedure:

1. Choose addresses with identical tags but different indexes.
2. Load these addresses into the cache and verify that they are placed in different sets according to their indexes.

Expected Outcome:
Addresses with the same tag but different indexes should be placed in different sets, confirming that the index correctly determines set placement.

3.3 Hit Verification

Procedure:

1. Fill one set halfway (same index, but 8 different tags).
2. Reference one of the tags that already resides in the set again.

Expected Outcome:
The set should have 8 total entries, accessing the same element again should yield a hit and not a line fill.

Expected Results Summary

Each test scenario should confirm that:

- **16-way associativity is maintained:** Each set holds up to 16 unique lines, and no more, confirming that the cache behaves as a 16-way set associative cache.
- **Cache overflow behavior is correct:** When more than 16 lines are mapped to a set, the cache evicts one line to maintain the 16-line capacity per set.
- **Address mapping and distribution is accurate:** Addresses are correctly mapped to the appropriate sets based on their index, and the cache ensures that no two lines with the same index and tag (duplicates) reside in the cache.

5 Alignment Verification

Objectives

- **Ensure Proper Handling of Address Alignment:** Verify that the cache correctly handles memory addresses, ensuring that they align with the cache line size (64 bytes) as required by the system.
- **Verify Cache Line Addressing Does Not Span Across Multiple Lines:** Confirm that memory references landing on cache line boundaries result in a single cache line fill, without spanning multiple lines.
- **Check for Correct Operation when Address is Misaligned:** Ensure the system correctly fetches the appropriate 64-byte chunk based on the misaligned address, aligning it with the correct cache line boundaries.

Test Scenarios and Procedures

1. Ensure Proper Handling of Address Alignment

1.1 Test Aligned Address Access

Procedure:

1. Run a trace file where the first address reference is aligned to the cache line size (64 bytes).
2. Make subsequent accesses that differ in address (Byte Select) but correspond to the same 64-byte chunk of memory.

Expected Outcome:
All requests after the initial fill should result in a hit to the line cached with the first access, confirming proper handling of aligned addresses.

1.2 Test for Cache Line Boundary Integrity (Accesses Near Boundaries)

Procedure:

1. Run a trace file that includes references near cache line boundaries (e.g., 0x3F, 0x40, 0x7F, 0x80, etc.).
2. Ensure that each access:
   - Results in a separate cache miss for each cache line.
   - Does not span across multiple cache lines, ensuring that data from different cache lines is fetched correctly without overlap.
3. Test byte-level accesses near the boundary (e.g., 0x3F, 0x40) to verify that no data crosses the boundary between two cache lines.

Expected Outcome:
Each address (including byte-level accesses) should be confined to its respective 64-byte cache line. No access should span across cache lines, and each access near a

boundary (whether byte-level or word-level) should result in a miss and a cache line fill for the appropriate line.

2. Check for Correct Operation when Address is Misaligned

2.1 Test for Misaligned Access

Procedure:

1. Run a trace that includes misaligned address references (e.g., addresses not divisible by 64).
2. Verify that the correct 64-byte cache line containing the requested address is fetched from DRAM.

Expected Outcome:
The fetched cache line should contain the selected byte, with the address correctly offset from the aligned 64-byte line fetched from memory. For example, if the address ends with 0x4F, the line fetched should span from 0x40 to 0x7F, including the requested address.

2.2 Test for Handling of Multiple Misaligned Addresses

Procedure:

1. Run a trace with multiple misaligned addresses, each within a different 64-byte cache line (e.g., 0x00, 0x4F, 0x80, 0xC7).
2. Verify that each misaligned address results in a correct cache line fetch from DRAM.

Expected Outcome:
The system should correctly handle each misaligned address and fetch the appropriate 64-byte chunk from memory, ensuring that no byte accesses spill over into adjacent cache lines.

Expected Results Summary

Each test scenario should confirm that:

- **Aligned Addresses:** All memory addresses should be aligned with the cache line size (64 bytes), with no errors in cache operations.
- **Cache Line Boundary Integrity:** Memory accesses should never span across multiple cache lines, as per the project assumption. Accesses near cache line boundaries (including byte-level) should correctly map to the appropriate cache line.
- **Misaligned Access Handling:** The system should correctly handle misaligned addresses by fetching the correct 64-byte chunk, ensuring proper alignment or adjusting the access as required.

6 Pseudo-LRU Replacement Policy Verification

Objectives

- **Verify Pseudo-LRU on Set Overflow:** Ensure that when a 17th line is added to a set already containing 16 lines, the cache correctly evicts the least recently used line according to the pseudo-LRU policy.
- **Validate Pseudo-LRU Tracking on Access:** Verify that each cache access (read/write) properly updates the pseudo-LRU tracking bits to reflect the most recently used cache line.
- **Test Edge Cases for Eviction Order:** Test eviction in scenarios where only a subset of lines have been accessed, ensuring the least recently used line is evicted even if it has not been accessed for some time.

Test Scenarios and Procedures

1. Verify Pseudo-LRU on Set Overflow

1.1 Insert 17 Unique Lines into a Set and Test Eviction

Procedure:

1. Fill a set with 16 unique lines.
2. Access some lines in a specific order (e.g., lines 1, 2, 3) to establish a pseudo-LRU state.
3. Add a 17th line and verify that the least recently used line (according to the pseudo-LRU policy) is evicted.

Expected Outcome:
The cache evicts the pseudo least recently used line, confirming that the pseudo-LRU policy works correctly on overflow and that the eviction respects the access history.

2. Validate Pseudo-LRU Tracking on Access

2.1 Update Pseudo-LRU Tracking on Access

Procedure:

1. Fill a set with 16 lines to establish the initial pseudo-LRU state.
2. Access lines in various orders (e.g., access lines 5, 3, 8, etc.).
3. After each access, check that the pseudo-LRU tracking bits are updated, marking the accessed line as the most recently used.

Expected Outcome:
Each access correctly updates the pseudo-LRU state, with the most recently accessed line marked as "most recent" and the rest in their correct relative order.

3. Test Edge Cases for Eviction Order

3.1 Evict the Least Recently Used Line After Partial Access

Procedure:

1. Fill a set with 16 lines.
2. Access only a subset of lines (e.g., lines 1, 3, 5), leaving others untouched.
3. Add a 17th line and verify that the least recently used untouched line is evicted.

Expected Outcome:
The cache evicts the least recently used line that has not been accessed, even if it has not been accessed since its initial load.

3.2 Consistent Eviction with Mixed Access Patterns

Procedure:

1. Load 16 lines into the cache.
2. Access the lines in a non-sequential pattern (e.g., 1, 16, 2, 15, etc.).
3. Add a 17th line and confirm that the eviction follows the pseudo-LRU order, evicting the least recently used line.

Expected Outcome:
The least recently used line is evicted, regardless of the access pattern, ensuring consistent eviction behavior.

Expected Results Summary

Each test scenario should confirm that:

- **Pseudo-LRU Replacement on Set Overflow:** The cache correctly evicts the least recently used line when a 17th line is added to a fully populated set, according to the pseudo-LRU policy.
- **Accurate Tracking of Cache Accesses:** The pseudo-LRU bits are correctly updated after each cache access, reflecting the most recently used state.
- **Edge Case Handling:** Even with partial or non-sequential accesses, the cache correctly evicts the least recently used line, demonstrating robust eviction behavior.

7 Write-Allocate Validation

Write-Allocate Test Plan

Objective: Verify that the cache correctly implements the Write-Allocate policy for write misses. Ensure proper cache line allocation, loading, and state transitions, while also verifying that cache statistics are updated accordingly.

- **Verify Write-Allocate Behavior on Write Misses:** Confirm that a cache miss on a write results in the allocation of a new cache line and writing the data into the cache instead of bypassing it.
- **Ensure Cache Line is Properly Allocated and Loaded:** Ensure that when a new line is allocated due to a write miss, the entire cache line (typically 64 bytes) is fetched from the next level of memory and placed into the cache.
- **Validate Subsequent Writes to the Allocated Line:** Ensure that after a write miss triggers a write-allocate, subsequent writes to the same address hit the cache.
- **Confirm Consistency with MESI Protocol:** Verify that the cache line's state transitions according to the MESI protocol after a write-allocate (e.g., transitioning to Modified).
- **Verify Statistics Update for Write-Allocate Events:** Ensure that cache statistics, such as write misses and cache allocations, are correctly updated during write-allocate operations.

Test Scenarios and Procedures

1. Verify Write-Allocate Behavior on Write Misses

1.1 Perform Write Miss on an Empty Cache

Procedure:

1. Use a trace file where the first operation is a write to an address not currently in the cache.
2. Observe the cache behavior: confirm that a write miss occurs and that the cache allocates a new cache line for the missed address.

Expected Outcome:
The cache reports a write miss, and a new cache line is allocated for the missed address.

2. Ensure Cache Line is Properly Allocated and Loaded During Write-Allocate

2.1 Check Cache Line Loading After Write Miss

Procedure:

1. Use a trace file with a write miss to an address in a full set.
2. Verify that the cache evicts the proper line via Pseudo-LRU and loads the full cache line (e.g., 64 bytes) from the next level of memory during the write-allocate process.

Expected Outcome:
The Pseudo-Least-Recently-Used cache line is evicted, and the line corresponding to the missed address is fetched and loaded into the cache.

3. Validate Subsequent Writes to the Allocated Line

3.1 Perform Multiple Writes After Write-Allocate

Procedure:

1. Use a trace where a write miss occurs, followed by additional writes to the same address.
2. Verify that the subsequent writes result in cache hits.

Expected Outcome:
The first write causes a miss and triggers write-allocate. Subsequent writes to the same address should hit the cache.

4. Confirm Consistency with MESI Protocol

4.1 Verify State Transition After Write-Allocate

Procedure:

1. Use a trace file that triggers a write miss, resulting in a write-allocate.
2. Observe the MESI state of the allocated line to confirm it transitions appropriately to Modified.

Expected Outcome:
The cache line should transition to the correct MESI (Modified) state after the write-allocate operation.

5. Verify Statistics Update for Write-Allocate Events

5.1 Check Cache Statistics After Write-Allocate

Procedure:

1. Use a trace file with a known number of write misses.
2. Verify that the number of write misses and cache allocations reported in the statistics matches the expected values.

Expected Outcome:
Cache statistics should accurately reflect the number of write misses and allocations caused by write-allocate operations.

Expected Results Summary

After executing each test:

- **Write Miss Behavior:** Write misses should trigger the allocation of a new cache line (write-allocate).
- **Line Loading:** The entire cache line should be loaded into the cache from the next level during the write-allocate process.
- **Subsequent Writes:** Subsequent writes to the allocated line should result in cache hits.
- **MESI Protocol Compliance:** Cache line state transitions should follow the MESI protocol (e.g., to Modified) after a write-allocate.
- **Statistics:** Cache statistics should correctly track write misses and allocations triggered by the write-allocate policy.

This test plan ensures that the write-allocate policy is correctly implemented, with proper cache line allocation, loading, subsequent hits, MESI state transitions, and accurate statistics tracking.

8 MESI State Accuracy

Objective

To thoroughly test the correct implementation of the MESI (Modified, Exclusive, Shared, Invalid) protocol in cache memory systems. This includes verifying correct state transitions during cache read, write, snoop, and eviction operations, ensuring cache coherence in a multi-processor environment, and tracking MESI-related statistics.

MESI States Overview

- **Modified (M)**: The cache has a copy of the data that has been modified, and it is the only cache with this data. The cache is responsible for writing back the data when evicted.
- **Exclusive (E)**: The cache has the only copy of the data, but the data has not been modified. This means no other cache has the line, and it is clean in this cache.
- **Shared (S)**: The cache line is present in multiple caches, and it is not modified (it is consistent with memory).
- **Invalid (I)**: The cache does not have a valid copy of the data.

Test Scenarios and Procedures

1. State Transitions on Cache Reads

A cache read can result in different state transitions based on the current state of the cache line.

1.1 Read to a Shared Line (S)

Procedure:

1. Simulate a read request to a line in the **Shared** state, which is cached by multiple processors.
2. Confirm that no state change occurs because the cache line is shared.

Expected Outcome:
The cache should remain in the **Shared** state. The line is shared among multiple caches, so no transition occurs.

1.2 Read to an Invalid Line (I)

Procedure:

1. Simulate a read request to a line in the **Invalid** state.
2. Verify the state transition based on the cache configuration (i.e., whether the data is fetched from memory or another cache).

Case 1: Fetching from a Cache
Procedure:

1. The data is cached by another processor, and the cache line transitions to **Shared** in the sending processor (if previously Exclusive).
2. The line is marked **Shared** in the requesting cache.

Expected Outcome:
The cache line transitions to **Shared** (S) in both caches since it is being shared between them.

Case 2: Fetching from Memory
Procedure:

1. The cache needs to fetch the line from memory.
2. The cache line enters **Exclusive** (E) since this is the only processor accessing the data at that point.

Expected Outcome:
The line enters the **Exclusive** state, and it is clean.

1.3 Read to a Line in the Exclusive State (E)

Procedure:

1. Simulate a read request to a line that is in the **Exclusive** state, where only one cache has it.
2. Confirm that no transition occurs if the requesting processor is the same as the owner.

Expected Outcome:
No state change occurs, and the line remains in the **Exclusive** state if the cache has the only copy of the data.

2. State Transitions on Cache Writes

Writing to a cache line triggers state changes in both the requesting cache and other caches, if they have the same line.

2.1 Write to an Invalid Line (I)

Procedure:

1. Simulate a write request to a line in the **Invalid** state.
2. Verify that the cache line transitions to the **Modified** state after fetching the data.

Expected Outcome:
The cache transitions the line to the **Modified** state (M). The cache is the only one that has a copy of the data, and it is modified.

2.2 Write to a Shared Line (S)

Procedure:

1. Simulate a write request to a line in the **Shared** state.
2. The cache should broadcast an invalidation to other caches that may have a copy of the line.
3. Confirm that the line transitions to the **Modified** state in the requesting cache.

Expected Outcome:
The requesting cache moves the line to **Modified** (M), and all other caches invalidate the line (I).

2.3 Write to an Exclusive Line (E)

Procedure:

1. Simulate a write request to a line in the **Exclusive** state.

2. The cache should transition the line to the **Modified** state.

Expected Outcome:
The cache moves the line to **Modified** (M), indicating that the cache has the only copy of the line and has modified it.

3. Snooping Behavior

Snooping is the process by which caches observe the bus to detect and respond to the actions of other processors.

3.1 Handle Bus Read Operation to a Modified Line (M)

Procedure:

1. Simulate a bus read operation requesting data from a line in the **Modified** state.
2. The sending and requesting cache should transition the line to **Shared** and provide the data.

Expected Outcome:
The line transitions to **Shared** (S), and the cache supplies the data on the bus to be snarfed while the updated data is written back to the DRAM.

3.2 Handle Bus Read with Intent to Modify (RWIM)

Procedure:

1. Simulate a bus read with intent to modify (RWIM) for a line in the **Modified** state.
2. The cache should invalidate the line and provide the data to the requesting processor.

Expected Outcome:
The line transitions to **Invalid** (I), and the cache provides the data to the bus.

3.3 Handle Bus Write Operation to a Shared Line (S)

Procedure:

1. Simulate a bus write operation that writes to a line in the **Shared** state.
2. The requesting cache should invalidate the line in other caches.

Expected Outcome:
The cache transitions the line to **Modified** (M), and all other caches invalidate their copy of the line.

4. Shared State (S) Management

Shared state management refers to situations where the cache line is present in multiple caches and must remain consistent.

4.1 Verify Multiple Reads by Different Processors

Procedure:

1. Simulate multiple processors reading the same line.
2. Confirm that the line remains in the **Shared** state across all caches.

Expected Outcome:
The line remains in the **Shared** state, as it is being accessed by multiple processors.

4.2 Handle Read/Write Conflict

Procedure:

1. One processor performs a read, and another performs a write to the same line.
2. Verify that the line transitions to the **Modified** state for the writing cache, and other caches are invalidated.

Expected Outcome:
The line in the writing cache transitions to **Modified** (M), while the read cache remains in **Shared** (S) until the write operation invalidates the shared copies.

5. Exclusive State (E) Management

The Exclusive state represents when a cache has the only copy of the data and is clean.

5.1 Verify Exclusive State Entry

Procedure:

1. Simulate a read to a cache line that no other cache has accessed.
2. Confirm that the line enters the **Exclusive** state.

Expected Outcome:
The cache line transitions to the **Exclusive** state since this is the only processor accessing the data and no modification has been made.

6. Modified State (M) Management

The Modified state represents when a cache holds the only copy of the modified data.

6.1 Test Eviction of Modified Line

Procedure:

1. Simulate a cache replacement for a line in the **Modified** state.
2. Verify that the data is written back to memory before eviction.

Expected Outcome:
The line is written back to memory before being evicted to maintain consistency.

7. Cache Coherence Across Multiple Processors

In multi-processor systems, the MESI protocol ensures that all caches are kept coherent.

7.1 Simulate Multi-Processor Access

Procedure:

1. Simulate multiple processors accessing and modifying the same cache line.
2. Verify that all caches maintain coherence, including the proper invalidations and state transitions.

Expected Outcome:
Cache coherence is maintained across all processors, with appropriate invalidations, state transitions, and data updates.

8. Statistics Update During MESI Operations

8.1 Check Statistics for State Transitions

Procedure:

1. Use a trace file designed to exercise all MESI state transitions.
2. Verify that state transition counts and related statistics are correctly recorded.

Expected Outcome:
Statistics should match the expected number of transitions and operations, reflecting correct MESI protocol behavior.

Expected Results Summary

After executing each test:

- **State Transitions:** All MESI state transitions are correct for reads, writes, and snooping.
- **Shared, Exclusive, and Modified States:** Each state is entered and managed correctly based on access patterns.
- **Snooping Behavior:** Cache responds correctly to bus operations from other processors.
- **Cache Coherence:** MESI protocol maintains consistency across all caches.
- **Statistics:** Cache statistics accurately reflect MESI operations.

9 Cache Coherence

Objectives

- **Verify Execution of Bus Read and Write Operations:** Ensure the LLC initiates, processes, and correctly handles all bus operations, including read, write, invalidate, and RWIM, while maintaining consistency with memory and other caches.
- **Validate MESI Protocol Compliance:** Confirm the proper implementation of all MESI state transitions during cache reads, writes, snooping, and evictions.
- **Test Multi-Processor Shared Memory Operations:** Verify cache coherence and conflict resolution when multiple processors read, write, and modify shared cache lines.
- **Evaluate Snooping Behavior:** Ensure the LLC observes and responds accurately to bus operations initiated by other caches or processors.
- **Monitor Reporting to Higher-Level Caches:** Confirm that the LLC communicates evictions, state changes, and cache statistics accurately and efficiently.
- **Measure and Report Cache Statistics:** Ensure precise tracking of MESI transitions, snooping responses, invalidations, and bus operations, with statistics matching expected outcomes.

Test Scenarios and Procedures

1. Verify Execution of Bus Read Operations

1.1 Cache Miss Triggers Bus Read

Procedure:

1. Simulate a cache miss for a read request.
2. Verify that the cache initiates a bus read operation to fetch the required data.

Expected Outcome:
The cache sends a bus read request, and the data is fetched.

1.2 Snoop Result HIT on Bus Read

Procedure:

1. Simulate a bus read operation where another cache has the line in an unmodified valid state.
2. Verify that the cache that already contains the line sends a HIT signal along with its MESI state on the bus.

Expected Outcome:
The cache receives the data, HIT is sent on the bus, and the line transitions correctly.

1.3 Cache Miss for Line in Exclusive State in Another Cache

Procedure:

1. Simulate a read request for a line in an Exclusive state in another cache.
2. Ensure the other cache transitions to the Shared state.

Expected Outcome:
The requesting cache transitions the line to Shared. The original Exclusive cache also transitions the line to Shared.

2. Verify Execution of Bus Write Operations

2.1 Writeback on Modified Line Eviction

Procedure:

1. Simulate the eviction of a line in the Modified state.
2. Confirm that the cache initiates a bus write operation to write back the modified data to memory.

Expected Outcome:
Data is written back to memory via a bus write operation.

2.2 Eviction of a Shared Line in 2 L1's

Procedure:

1. Simulate the eviction of a line in a Shared state from one L1, that remains in one other L1.

Expected Outcome:
No write-back to memory occurs. The line transitions to Invalid, and the other cache transitions to exclusive.

3. Verify Bus Read with Intent to Modify (RWIM)

3.1 RWIM for Line in Shared State

Procedure:

1. Simulate a write request to a line in the Shared state.
2. Verify that the cache initiates a RWIM operation on the bus to gain exclusive ownership of the line.

Expected Outcome:
The CPU sends an RWIM request, the requesting cache transitions the line to the Modified state, and the line becomes invalid in the other shared caches.

3.2 RWIM for Line in Invalid State

Procedure:

1. Simulate a write request to a line in the Invalid state.
2. Verify that the cache initiates a RWIM operation and fetches the data from memory before modification.

Expected Outcome:
The line transitions to Modified, and the data is fetched before modification.

4. Verify Bus Invalidate Operations

4.1 Invalidate Shared Lines on Write

Procedure:

1. Simulate a write request to a line in the Shared state.
2. Verify that the cache broadcasts an invalidate operation on the bus to the other caches.

Expected Outcome:
All other caches invalidate their copies of the line.

4.2 Invalidate Other Lines During RWIM

Procedure:

1. Simulate a RWIM operation for a line shared across multiple caches.
2. Confirm that other caches invalidate their copies in response to the RWIM.

Expected Outcome:
All other caches receive an invalidate bus operation and their copies transition to the invalid state.

5. Validate Responses to Bus Snooping

5.1 Respond with HIT

Procedure:

1. Simulate a snooped bus read operation for a line in the Shared or Exclusive state.
2. Confirm the cache reports a HIT response.

Expected Outcome:
The cache reports HIT.

5.2 Respond with HITM

Procedure:

1. Simulate a snooped read bus operation for a line in the Modified state.
2. Confirm the cache reports a HITM response and writes back the data to DRAM.

Expected Outcome:
The cache reports HITM and writes back the line.

5.3 No Response

Procedure:

1. Simulate a snooped bus read operation for a line in the invalid state in all caches.
2. Verify there is no response for the other caches observed on the bus.

Expected Outcome:
The caches do not reply with HIT or HITM.

6. Verify Cache Statistics for Bus Operations

6.1 Count Bus Reads and Writes

Procedure:

1. Use a trace file that triggers multiple read and write operations.
2. Verify that the cache correctly counts the number of bus reads and writes.

Expected Outcome:
Statistics for bus reads and writes match the expected count.

6.2 Count Invalidate and RWIM Operations

Procedure:

1. Use a trace file with scenarios requiring invalidation and RWIM operations.
2. Confirm that the cache accurately tracks the count of these operations.

Expected Outcome:
Statistics for invalidations and RWIM match expectations.

7. Reporting Results to Higher-Level Caches

7.1 Eviction of Modified Line

Procedure:

1. Simulate the eviction of a line in the Modified state.
2. Confirm that the L1 sends the modified data to the LLC for write-back.

Expected Outcome:

- Modified data is written back to memory. The MESI state transitions to invalid.

7.2 Reporting Line State Changes

Procedure:

1. Simulate a snooping event that causes a transition of a cache line from Modified to Shared or Exclusive.
2. Verify that the LLC updates the higher-level cache with the new state.

Expected Outcome:

- Higher-level cache is informed of state transitions.
- No inconsistencies occur between the LLC and higher-level cache.

7.3 Reporting Cache Statistics

Procedure:

1. Run a workload trace that triggers multiple state transitions and bus operations.
2. Verify that the LLC aggregates and reports accurate statistics, including hits, misses, invalidations, and RWIM counts.

Expected Outcome:

- Reported statistics match expected values based on the trace.
- No loss of accuracy or data during reporting.

8. Multi-Processor Shared Memory Configuration

8.1 Simultaneous Reads by Multiple Processors

Procedure:

1. Simulate multiple processors issuing read requests for the same cache line.
2. Verify that the cache line remains in the Shared state across all processors.

Expected Outcome:

Line remains in the Shared state. Data consistency is maintained across all processors.

8.2 Simultaneous Write Conflicts

Procedure:

1. Simulate multiple processors attempting to write to the same cache line.
2. Monitor how the LLC prioritizes the operations and resolves conflicts.

Expected Outcome:

One processor must wait until the data has been written to and read back to the DRAM before completing its write. The first processor to write to the line must also transition into the invalid state.

8.3 Mixed Read/Write Operations

Procedure:

1. Simulate a scenario where one processor reads a line while another attempts to write to the same line.
2. Verify proper handling of invalidations and MESI transitions.

Expected Outcome:

The reading processor receives valid data before the writing one. When the writing processor receives the line, it transitions the line to Modified, and all other copies move to invalid.

Expected Results Summary

Upon executing all tests:

- **Bus Operations:** All bus reads, writes, invalidations and RWIM operations execute correctly, maintaining consistency with memory and other caches.
- **MESI Protocol:** State transitions comply fully with MESI rules, ensuring correct handling of reads, writes, snooping, and evictions.
- **Multi-Processor Coherence:** Shared memory consistency is maintained across all processors during simultaneous reads, writes, and conflicts.
- **Snooping Behavior:** The LLC responds accurately to all snooping operations, including HIT, HITM, and NOHIT, while maintaining coherence.
- **Statistics Reporting:** Cache statistics, including hits, misses, invalidations, and RWIM operations, match expected values with no inaccuracies.
- **Higher-Level Cache Communication:** The LLC correctly reports state changes, evictions, and aggregated statistics to the higher-level cache.

10 Inclusivity Verification

Objectives:

Ensure that the Last-Level Cache (LLC) is inclusive of all lines present in the higher-level cache (L1), and verify correct eviction handling, modification propagation, and cache consistency across multiple processors.

- **Verify Line Inclusion:** Every line in the higher-level cache (L1) must also be present in the LLC.
- **Eviction Handling:** When a line is evicted from the LLC, it must be invalidated in the higher-level cache (L1) as well.
- **Modification Propagation (MESI Protocol):** Modifications in the higher-level cache (L1) should propagate to the LLC, ensuring consistency.
- **Concurrent Access by Multiple Processors:** LLC should maintain inclusivity and consistency when multiple processors access shared data.
- **Replacement Policy (Pseudo-LRU):** Ensure the LLC eviction follows Pseudo-LRU, and evicted lines are also removed from L1.
- **Forced Invalidation via Snooping:** Invalidation commands from the bus must propagate to both LLC and L1 caches.
- **Cache Clearing Operations:** Clearing the LLC should invalidate corresponding lines in the higher-level cache (L1).

Test Scenarios and Procedures

1. Verify Line Inclusion in LLC

1.1 Basic Operations for Line Inclusion

Procedure:

1. Perform a read operation for a line not present in LLC.
2. The line should be loaded into LLC and also propagated to L1 if not already present there.

Expected Outcome:
The cache line should appear in both LLC and L1, confirming line inclusion.

2. Modification Propagation via MESI Protocol

2.1 Reflect Modifications from L1 to LLC

Procedure:

1. Load a line into both LLC and L1.
2. Modify the line in L1 (e.g., write operation).
3. Verify that the modification in L1 is propagated to LLC, and LLC reflects the same state according to the MESI protocol.

Expected Outcome:
LLC should reflect the modification from L1, ensuring data consistency across caches.

3. Concurrent Access by Multiple Processors

3.1 Inclusivity with Shared Data

Procedure:

1. Simulate two processors loading the same line into their L1 caches.
2. Evict the line from LLC due to access by a third processor.
3. Verify that eviction from LLC invalidates the line in both L1 caches.

Expected Outcome:
The evicted line in LLC should be invalidated in all L1 caches that had a copy of it, ensuring consistency in a multi-processor setup.

4. Replacement Policy Verification (Pseudo-LRU)

4.1 Eviction and Replacement in LLC and L1

Procedure:

1. Populate LLC with unique addresses until it reaches capacity.
2. Access a new address, triggering a Pseudo-LRU eviction in LLC.
3. Verify that any evicted line in LLC is also evicted in L1 if it was present.

Expected Outcome:
Evicted lines from LLC should be removed from L1 as well, consistent with the pseudo-LRU replacement policy.

5. Forced Invalidation via External Snooping

5.1 Handle External Invalidation Requests

Procedure:

1. Load a line into both LLC and L1.
2. Issue an external invalidate command for this address via bus snooping that is seen by the LLC.
3. Verify that the line is invalidated in both LLC and L1.

Expected Outcome:
The line should be invalidated in both LLC and L1, ensuring consistency in response to external invalidation.

6. Cache Clearing Operations

6.1 Clear All Cache Entries in LLC and L1

Procedure:

1. Populate both LLC and L1 caches with data.
2. Issue a clear command for LLC, effectively resetting the cache.
3. Verify that all entries are removed from both LLC and L1.

Expected Outcome:
Both LLC and L1 should be cleared of all entries, confirming that cache clearing operates across the entire hierarchy.

Expected Results Summary

Each test scenario should confirm that:

- **Line Inclusion in LLC:** Every line in L1 should be mirrored in the LLC.
- **Eviction Handling:** Evictions in LLC must be reflected in L1 by invalidating or removing the evicted lines.
- **Modification Propagation:** Modifications in L1 should propagate to LLC, ensuring data consistency.
- **Concurrent Access:** Evictions in LLC due to multiple processors must invalidate the corresponding lines in L1.
- **Replacement Policy (Pseudo-LRU):** Evicted lines from LLC should be removed from L1 in accordance with the Pseudo-LRU policy.
- **Forced Invalidation via Snooping:** External snooping commands must invalidate lines in both LLC and L1.
- **Cache Clearing:** Clearing LLC should also clear L1, resetting the cache system.

This test plan verifies the LLC's inclusivity and ensures the correctness of eviction, modification propagation, concurrent access, and cache clearing operations across the entire cache hierarchy.

11 Write-Once Policy

Objectives

Ensure that the L1 cache correctly implements the write-once policy where the first write to a line is write-through to the LLC, and subsequent writes to the same line are write-back. The test plan should verify the behavior of the L1 cache and its interactions with the LLC while maintaining inclusivity.

- **First Write Verification:** Confirm that the first write to a line in the L1 cache results in a write-through to the LLC, updating both caches.
- **Subsequent Write Verification:** Validate that subsequent writes to the same line in the L1 cache are handled as write-back, updating only the L1 cache.
- **MESI State Transitions:** Verify that the L1 cache transitions through the correct MESI states to reflect the write-once policy and ensure data consistency between L1 and LLC.
- **Inclusivity Maintenance:** Ensure that despite the write-back policy for subsequent writes, the LLC maintains inclusivity by retaining a copy of the line.

Test Scenarios and Procedures

1. First Write to a Cache Line

1.1 Write to an Empty Line

Procedure:

1. Initialize the L1 cache as empty.
2. Simulate a write request from the processor to a new address that maps to the L1 cache.

Expected Outcome:

- The L1 cache reports a write miss because the line is not present.
- The write is performed as write-through, updating both the L1 cache and the LLC.
- The L1 cache line transitions to the Modified state, reflecting that it holds the only up-to-date copy of the data.
- The LLC receives the write data through a bus write operation and updates its corresponding line.

2. Subsequent Writes to the Same Cache Line

2.1 Write to a Modified Line

Procedure:

1. Simulate a write request from the processor to an address that already resides in the L1 cache and is in the Modified or Exclusive state.

Expected Outcome:

- The L1 cache reports a write hit as the line is already present.
- The write is handled as write-back, meaning only the L1 cache is updated.
- The L1 cache line remains in the Modified state, indicating the data has been modified.
- The LLC is not involved in this write operation.

3. Statistics and State Tracking

3.1 Verify Write Counts and State Transitions

Procedure:

1. Execute a trace file containing a sequence of write requests, including both first writes and subsequent writes to the same cache lines.
2. Monitor the L1 cache to track the number of write-through and write-back operations.
3. Observe the MESI state transitions of the L1 cache lines to ensure they correctly reflect the write-once policy.

Expected Outcome:

- The L1 cache statistics accurately report the number of write-through operations, matching the count of first writes to the lines.
- The L1 cache statistics accurately report the number of write-back operations, corresponding to the subsequent writes to the same lines.
- The MESI states of the L1 cache lines transition as expected:
  - First write: transitions to Modified.
  - Subsequent writes: remains in the Modified state.

4. Inclusivity

4.1 Check Inclusivity After Write-Back

Procedure:

1. Perform a write to a line in the L1 cache (Write-Through).
2. Perform subsequent writes to the line (so that the LLC has an outdated copy of the data).
3. Initiate an eviction of the corresponding line from the LLC.

Expected Outcome:

- The only up-to-date data should be found in the L1, so it should first be written back to the LLC.
- After writing to the LLC the L1 should invalidate its line.
- The LLC should write the data back to DRAM, also invalidating its copy.

Expected Results Summary

Each test scenario should confirm:

- **First Write Handling:** The first write to a line in the L1 cache is treated as write-through, updating both the L1 cache and the LLC.
- **Subsequent Write Handling:** Subsequent writes to the same line in the L1 cache are performed as write-back, updating only the L1 cache.
- **MESI State Compliance:** The L1 cache correctly transitions MESI states to ensure data consistency between itself and the LLC according to the write-once policy.
- **Inclusivity:** Inclusivity is maintained between the L1 cache and the LLC, even with the write-back policy used for subsequent writes. The LLC retains a copy of the line even if it is not updated with every write.

12 Cache Statistics Reporting

Objectives

- **Verify Tracking of Cache Reads and Writes:** Ensure that each read and write request to the cache is correctly counted, providing an accurate record of the operations for each trace.
- **Confirm Cache Hit and Miss Count Accuracy:** Validate that cache hits and misses are tracked accurately, with each cache access correctly updating the hit or miss count based on the presence or absence of the requested line.
- **Calculate and Report Cache Hit Ratio:** Ensure that the cache hit ratio is calculated and reported accurately after each trace file is executed, based on the recorded hits and misses.
- **Output Statistics in Silent and Normal Modes:** Confirm that the statistics are correctly displayed in both silent and normal modes, ensuring that silent mode only displays a summary, while normal mode provides detailed logging.

Test Scenarios and Procedures

1. Verify Tracking of Cache Reads and Writes

1.1 Process Read and Write Requests

Procedure:

1. Execute a trace file that includes a mix of read and write requests to the cache.
2. Track the number of read and write operations performed on the cache.
3. At the end of the trace, compare the tracked counts with the expected values based on the trace file content.

Expected Outcome:

The counts for cache reads and writes should match the number of read and write requests in the trace file, confirming accurate operation tracking.

2. Confirm Cache Hit and Miss Count Accuracy

2.1 Track Hits and Misses During Cache Access

Procedure:

1. Execute a trace file containing multiple requests, with some resulting in cache hits and others in misses.
2. For each request, log whether it results in a cache hit or miss.
3. At the end of the trace, verify that the hit and miss counts match the expected outcomes based on the cache's initial state and the sequence of accesses.

Expected Outcome:

The cache hit and miss counts should be accurately reported, matching the expected results based on the trace file and cache behavior.

3. Calculate and Report Cache Hit Ratio

3.1 Compute and Display Cache Hit Ratio

Procedure:

1. Run a trace file and record the total number of cache hits and misses.
2. Calculate the hit ratio as:

$$\text{hit ratio} = \frac{\text{number of hits}}{\text{number of hits} + \text{number of misses}}$$

3. Confirm that the displayed hit ratio at the end of the trace is accurate, based on the recorded hits and misses.

Expected Outcome:

The displayed hit ratio should match the calculated value, ensuring that the system correctly tracks and reports cache performance.

4. Output Statistics in Silent and Normal Modes

4.1 Verify Output in Silent Mode

Procedure:

1. Run a trace file in silent mode.
2. Confirm that the only output displayed is the summary of cache statistics: total reads, writes, hits, misses, and hit ratio.

Expected Outcome:

Silent mode should display a concise summary of the cache statistics without any detailed operation logs, ensuring that it only provides the necessary summary information.

4.2 Verify Output in Normal Mode

Procedure:

1. Run the same trace file in normal mode.
2. Confirm that, in addition to the summary statistics, the output includes detailed logs of each cache operation, including bus operations, snoop results, and messages to the higher-level cache.

Expected Outcome:

Normal mode should provide detailed operation logs alongside the summary statistics, allowing for deeper analysis of cache behavior during execution.

Expected Results Summary

Each test scenario should confirm that:

- **Read and Write Tracking:** The cache correctly tracks the total number of read and write operations performed during the execution of the trace file.
- **Hit and Miss Count Accuracy:** The cache hit and miss counts are correctly reported and reflect the actual number of hits and misses based on the access patterns in the trace file.
- **Correct Hit Ratio Calculation:** The cache correctly calculates and reports the hit ratio based on the total number of hits and misses.
- **Correct Output in Silent and Normal Modes:** Silent mode outputs only a summary of statistics, while normal mode provides a detailed log of all cache operations.