

**Table of Contents**

[Introduction. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2](#f3x7405bjynx)

[Project Specifications and Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3](#jk6fmng8x1cn)

[LRU Policy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6](#1fxsebj5ut9x)

[MESI Protocol . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7](#ddpf7xtoylzs)

[Cache Structure. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1](#vstxgnwf8au1)3

[Design Assumptions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1](#jy3atchlzj34)4

[Test Plan . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1](#kix.ag6s8frou9s)5

[Demo . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .2](#iqr8fjav6egx)5

[Key Points about the design. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .2](#gmmwsinx7dwt)7

[Member Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29](#r4fr4hpefd5r)

[Appendix . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30](#2b76jyui0ms0)

(Click on a topic to advance to its page)

Introduction

A computer cache is a type of high-speed memory that stores frequently accessed data and instructions for quick access by the processor. The primary purpose of a cache is to reduce the time it takes to access data from the main memory, which is much slower compared to the cache memory.

The cache hierarchy typically consists of multiple levels of cache memory, with each level having different sizes, access times, and replacement policies. The first level of cache memory, L1 cache, is the smallest and fastest cache memory, located on the same chip as the processor. It stores a small amount of data and instructions that the processor is currently using or likely to use in the near future.

The second level of cache memory, L2 cache, is larger and slower than the L1 cache, but still faster than the main memory. L2 cache acts as a buffer between L1 cache and the main memory and stores data and instructions that are less frequently used by the processor.

The third level of cache memory, L3 cache, is even larger and slower than L2 cache, but still faster than the main memory. L3 cache is typically shared between multiple processors in a system and is used to improve the overall system performance by reducing the time it takes to access data from the main memory.

Cache memory is crucial for improving the performance of modern computer systems. It enables faster access to data and instructions, which reduces the time it takes to perform various operations. Additionally, cache memory can reduce the number of memory accesses, which results in lower power consumption and improved performance. Overall, designing an efficient cache hierarchy is critical for achieving high-performance computing.

Our project involves the design and simulation of a split L1 cache for a new 32-bit processor. The cache will be used with up to three other processors in a shared memory configuration, and the system will employ a MESI protocol to ensure cache coherence. The L1 instruction cache will be four-way set associative, while the L1 data cache will be eight-way set associative, each consisting of 16K sets of 64-byte lines. The L1 data cache will use a write-back policy with write allocate, except for the first write to a line which will use a write-through policy. Both caches will use a Least Recently Used (LRU) replacement policy and will be backed by a shared L2 cache. The cache hierarchy will employ inclusivity. Our project aims to optimize cache performance and coherence to improve the overall system performance.

Project Specifications and Requirements

The split L1 cache is designed for a new 32-bit processor.

* The cache must be compatible with up to three other processors in a shared memory configuration.
* The MESI protocol is used to ensure cache coherence.
* The L1 instruction cache is four-way set associative, with 16K sets and 64-byte lines.
* The L1 data cache is eight-way set associative, with 16K sets of 64-byte lines.
* The L1 data cache uses a write-back policy with write allocate.
* The first write to a line in the L1 data cache is write-through, and subsequent writes are write-back.
* Both caches employ the Least Recently Used (LRU) replacement policy.
* The cache hierarchy employs inclusivity.
* Both caches are backed by a shared L2 cache.
* These design specifications and requirements aim to optimize the cache performance and coherence to improve the overall system performance.

Our design will be implemented in C++, and the code should maintain and report the following key statistics for each cache upon completion of execution of each trace file:

* Number of cache reads
* Number of cache writes
* Number of cache hits.
* Number of cache misses.
* Cache hit ratio.

By tracking and reporting these statistics, we can evaluate the effectiveness of our cache design and make any necessary improvements to optimize cache performance.

The L1 caches must communicate with the shared L2 cache to maintain inclusivity and implement the MESI protocol. To simulate this communication, the following messages (where <address> is a hexadecimal address) should be displayed:

* Return data to L2 <address>: When the cache receives a 4 in the trace file, it should signal that it's returning the data for that line (if present and modified).
* Write to L2 <address>: This operation is used to write back a modified line to L2 upon eviction from the L1 cache. It is also used for an initial write-through when a cache line is written for the first time so that the L2 knows it has been modified and has the correct data.
* Read from L2 <address>: This operation is used to obtain the data from L2 on an L1 cache miss.
* Read for Ownership from L2 <address>: This operation is used to obtain the data from L2 on an L1 cache write miss.

These specifications ensure that the L1 caches are properly communicating with the shared L2 cache, allowing for efficient cache coherence and improved system performance.

We will be using two modes:

* Mode 0: Display only the required summary of usage statistics and responses to 9s in the trace file.
* Mode 1: Display everything from Mode 0, as well as the communication messages to the L2 cache described above.

The simulation reads cache access caches/events from a trace text file, and it can support any trace file provided without recompiling the program so long as the provided file has the correct format:

3 333DE896

2 333DE890

4 269DE915

9 553DE938

1 363DE999

8 722DE323

n address, where n is:

| n | Semantics |
| --- | --- |
| 0 | Read data request to L1 data cache |
| 1 | Write data request to L1 data cache |
| 2 | Instruction fetch (a read request to L1 instruction cache) |
| 3 | Invalidate command from L2 |
| 4 | Data request from L2 (in response to snoop) |
| 8 | Clear the cache and reset all state (and statistics) |
| 9 | Print contents and state of the cache (allow subsequent trace activity) |

*Table 2.1: n address identifiers.*

LRU Policy

LRU (Least Recently Used) policy is a caching strategy where the least recently used item gets evicted first. This means each item needs to be identified/organized in terms of its usage.

In our case, we are assigning the LRU item to bit 000 and the most recently used (MRU) item to 111. Everything in between is also put in an orderly fashion (001, 010, 011, ...); each are 3-bits long.

For example, let’s say item E has just been accessed:

Item A B C D E F G H

LRU BITS 000 001 010 011 100 101 110 111

*Original*

Since E just became MRU, it will be assigned LRU bits 111. For every item that is valued higher than E, we will shift its corresponding LRU bits left once (decrement). For every item that is valued less than E, its LRU bits will not change.

Item A B C D E F G H

LRU BITS 000 001 010 011 111 100 101 110

*After Accessing E*

Items F, G, and H were valued higher than E and have been decremented. A, B, C, and D were valued less than E so they remain unchanged.

Another example, let’s say we need to evict the LRU in the original lineup to add Item Z:

Item Z B C D E F G H

LRU BITS 111 000 001 010 011 100 101 110

*After Adding Item Z*

All the other LRU bits will be decremented, and Item Z becomes 111 (MRU).

MESI Protocol

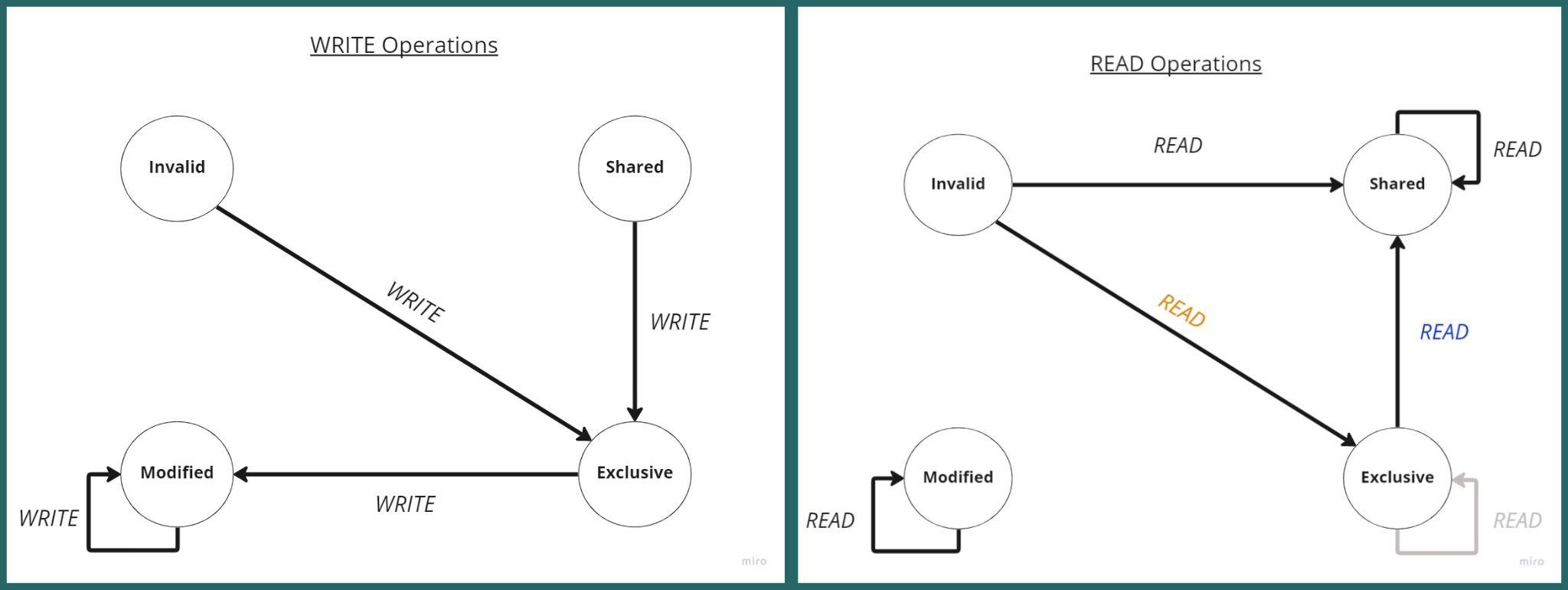
Within a write-back cache, the MESI Protocol is used to maintain cache coherency through snooping techniques. MESI is an acronym that stands for: Modified, Exclusive, Shared, Invalid. These are states in which cached data can be located, following a bus transaction.

If cached data is in a Modified state, the cached copy is the only valid copy of the line. No other processor cache contains the line, and the memory copy is out of date.

If cached data is in an Exclusive state, no other processor has a copy of the cached line. The processor’s cache and memory are identical.

If cached data is in a Shared state, at least one other processor has a copy of the line. The cached copies and the memory are identical.

If cached data is in a Invalid state, the cached copy is invalid. It doesn't hold a copy of the line in memory.



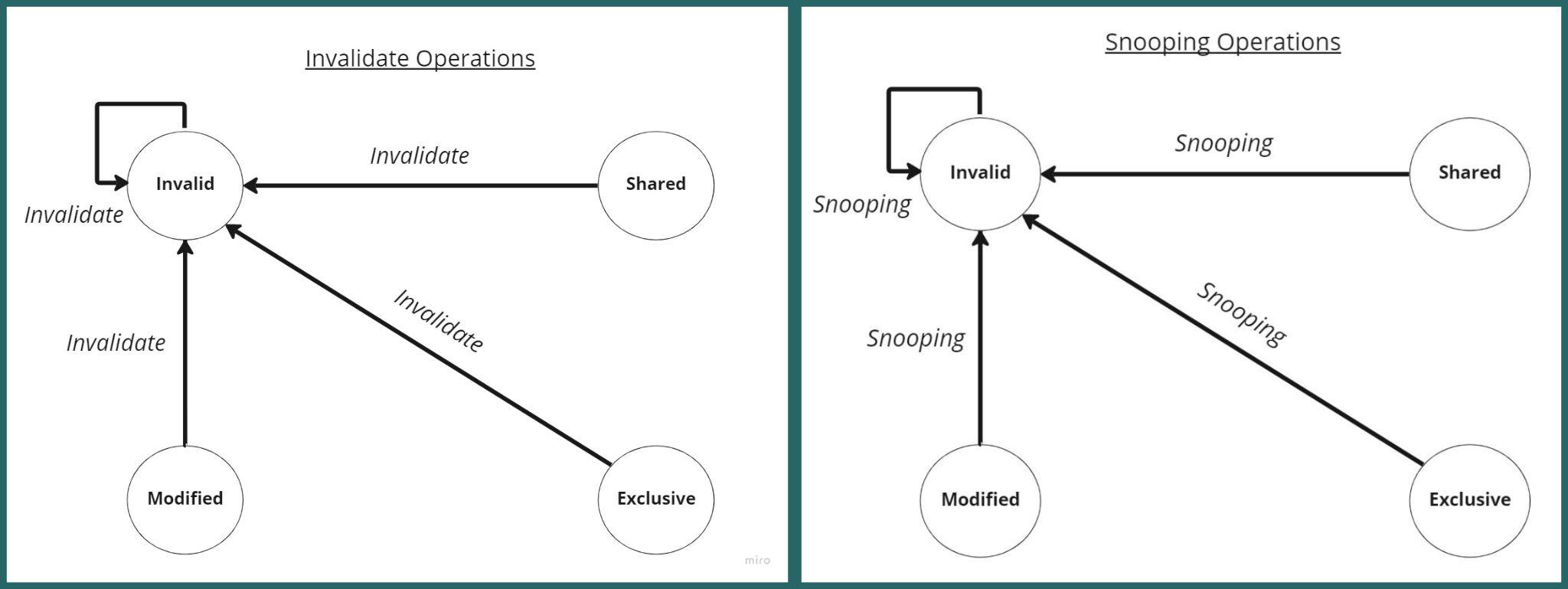
*Figure 5.1: State machines for Write and Read operations in the Data Cache.*

**WRITE**

* I to E: Snoop finds the data is not in any other cache, so the processor reads data from memory.
* S to E: Processor writing to the data in its own cache while the data is in a shared state - the written data will become exclusive.
* E to M: Processor doing CPU write to exclusive cache line in same cache. It is modified from the original data from memory; it is outdated.
* M to M: Processor doing a CPU write to a modified line.

**READ**

* I to E: Processor doing first read of data and goes to E state, since no other processor has any shared data.
* E to E: Processor performing CPU read to data that is in E state. It is in the same cache, no other cache has it. Does not need anything with other cache lines. (This path will never be taken: we’re assuming it’ll always be read by a different processor. Hence this path is grayed out in our diagram.)
* E to S: Processor has a hit on its cache from another processor, so it changes its status from Exclusive to Shared.
* S to S: Processor doing a CPU read to data from a cache that is a S state and the data to be read is in the same cache.
* I to S: Processor doing a CPU read to data that is in I state, data is not in the same cache. This read stays the same to represent all subsequent reads.
* M to M: Processor doing a CPU read to a modified line.

*Figure 5.2: State machines for Invalidate and Snooping operations in the Data Cache.*

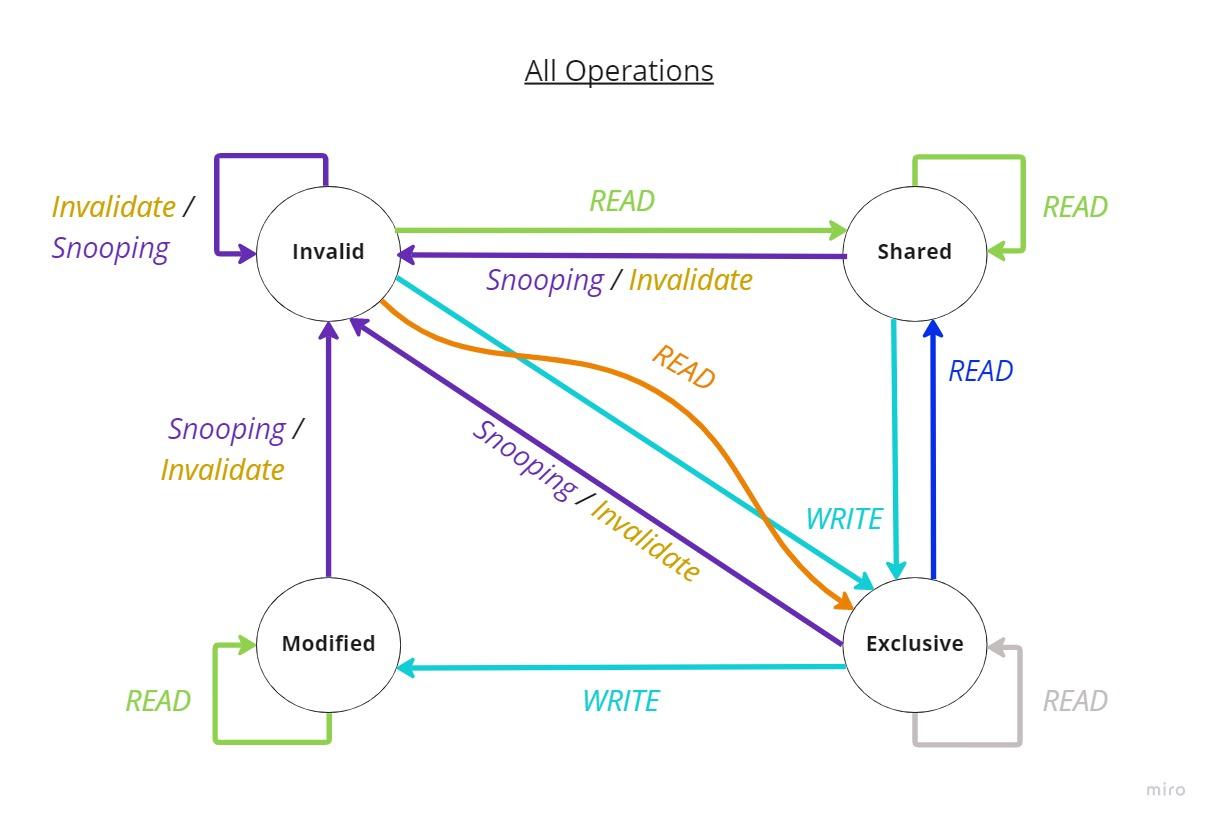
**Invalidate**

* I to I: Processor does not find the data in its own cache.
* S to I: When a processor performs Read For Ownership on shared data in another processor's cache, the other processor invalidates its data.
* E to I: When a processor performs Read For Ownership on exclusive data in another processor's cache, the other processor invalidates its data.
* M to I: When a processor performs Read For Ownership on modified data in another processor's cache, the other processor invalidates its data.

**Snooping**

* I to I: There will be no cache HIT operation, rather a cache miss operation.
* S to I: There is a RFO activity on the bus by another processor to the snooping processors same shared cache line.
* E to I: There is a RFO by another processor and it will write to the memory location so this snooping processor's cache line will be outdated.
* M to I: There is a RFO observed, resulting in this snooping.

RFO (read for ownership) is the action of a memory read but is an indicator to the snooping processor on FSB (front-side bus) the intent to write that memory location. The idea of snooping is to observe other processor’s state and activity. All processors snoop on a memory transaction.

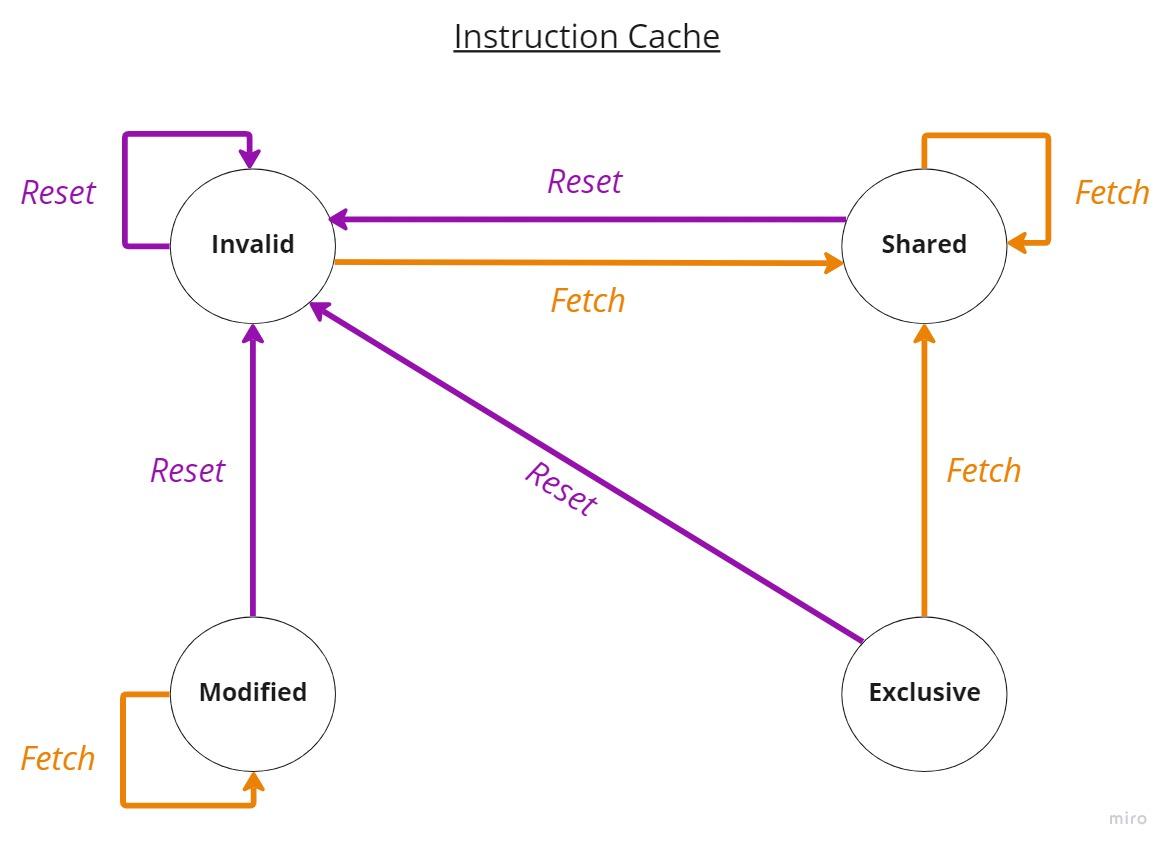
**

*Figure 5.3: State machine exhibiting all operations for the Data Cache.*

*Figure 5.3* shows each Read, Write, Invalidate, and Snooping operation in one FSM. These operations take place within the Data Cache.

The read from exclusive to exclusive path will never be taken based on our assumptions) we’re assuming it’ll always be read by a different processor. Hence this path is grayed out in our diagram. We only included it here in our diagram for completeness as it helped us understand the project and debug our design.

The first read when the program runs will go from the invalid to the exclusive state, based on our assumptions. Subsequent reads will go from the invalid to the shared state.

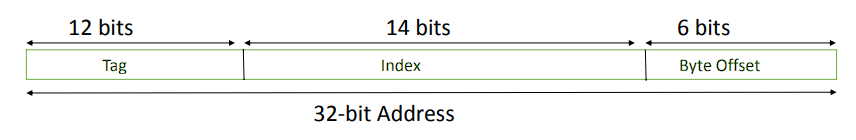


*Figure 5.4: State machine exhibiting Reset and Fetch operations for the Instruction Cache.*

*Figure 5.4* shows Reset and Fetch operations in one FSM. These operations take place within the Instruction Cache. A reset takes everything to Invalid, essentially the initial state.

Cache Structure

We will be designing and simulating a split L1 cache for a new 32-bit processor. The L1 cache is split into two different caches: data cache and instruction cache. The data cache is 8-way Set Associative, consisting of 16K set and 64-byte lines. The instruction cache is 4-way Set Associative, consisting of 16K set and 64-byte lines as well. We will know which cache we are accessing through the trace file.



*Figure 6.1: Cache Structure.*

Since this is a 32-bit processor, there will be 32 address bits. With 64-byte lines, there are 6 address bits (26) for Byte Select bits. 16K sets cache is reflected in the 14 address bits (214) for index. We then know there are 12 tag bits by looking at the difference of byte offset bits and index bits from address bits (32-6-14 = 12).

The 12 tag bits serve as identifiers for a line of data. The 14 index bits indicate a “set” from the cache. The 6 byte offset bits represent the location displacement. This is important to note for managing testcases.

Design Assumptions

* When the system is initially started, all cache lines are in the invalid state. The first write operation performed on a cache line transitions it to the exclusive state.
* In addition, when a processor reads data for the first time, it enters the exclusive state since no other processor has shared the data yet. In the event of a read miss, the processor first checks the L1 caches of other processors for the data. If the data is not found in any L1 cache, a copy is retrieved from the main memory and written to the cache. If all cache lines are occupied, the least recently used cache line is selected for eviction to accommodate the new data. If the evicted line is in the modified state, it must be written back to memory before eviction.
* If a cache line is in the Exclusive state and is read by another processor during a read operation, the MESI bits transition from Exclusive to Shared. This is assumed to cover all MESI states. If the read is from the same processor, the cache line will remain in the Exclusive state and will never transition to the Shared state. Therefore, for the purpose of this simulation, assume the read is from another processor.
* During a cache read or write miss, if a set contains only one invalid cache line, the new data will be written to that line with an invalid MESI bit, and its LRU value will be updated to 111 to make it the most recently used. If there are multiple invalid cache lines in the set, the new data will be written to the invalid line with the least LRU value to minimize eviction of useful data.

**Test Plan**

**PROGRAM START / RESET**

| **Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 0 | **Reads** | 0 |
| **Index** | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | I | I | I | I | I | I | I | I |  | **Misses** | 0 | **Misses** | 0 |
|  |  |  |  |  |  |  |  |  |  | **Hit Ratio** | 0 | **Hit Ratio** | 0 |

*MESI values are I (Invalidated) by default.*

**-------------------- Scenario 1 --------------------**

Loading all data cache lines of a particular set.

TRACE\_FILE\_01:

0 847DE198

0 722DE323

0 401DE268

0 386DE170

0 269DE972

0 241DE661

0 992DE271

0 434DE709

Final results of *TRACE\_FILE\_01*:

| **Scenario 1: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 8 | **Reads** | 0 |
| **Index** | 847 | 722 | 401 | 368 | 269 | 241 | 992 | 434 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | E | E | E | E | E | E | E | E |  | **Misses** | 8 | **Misses** | 0 |
|  |  |  |  |  |  |  |  |  |  | **Hit Ratio** | 0 : 8 | **Hit Ratio** | 0 |

**Step 1:** 0 847DE198 *Binary* - 1000 0100 0111 1101 1110 0001 1001 1000

*Byte offset* - 01 1000 *Index* - 1101 1110 0001 10 *Tag* - 1000 0100 0111

| **Step 1: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 1 | **Reads** | 0 |
| **Index** | 847 | 000 | 000 | 000 | 000 | 000 | 000 | 000 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | E | - | - | - | - | - | - | - |  | **Misses** | 1 | **Misses** | 0 |

**Step 2:** 0 722DE323 *Binary* - 0111 0010 0010 1101 1110 0011 0010 0011

*Byte offset* - 10 0011 *Index* - 1101 1110 0011 00 *Tag* - 0111 0010 0010

| **Step 2: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 1 | **Reads** | 0 |
| **Index** | 847 | 722 | 000 | 000 | 000 | 000 | 000 | 000 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 000 | 001 | 000 | 000 | 000 | 000 | 000 | 000 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | E | E | - | - | - | - | - | - |  | **Misses** | 1 | **Misses** | 0 |

**Step 3:** 0 401DE268 *Binary* - 0100 0000 0001 1101 1110 0010 0110 1000

*Byte offset* - 10 1000 *Index* - 1101 1110 0010 01 *Tag* - 0100 0000 0001

| **Step 3: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 1 | **Reads** | 0 |
| **Index** | 847 | 722 | 401 | 000 | 000 | 000 | 000 | 000 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 000 | 001 | 010 | 000 | 000 | 000 | 000 | 000 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | E | E | E | - | - | - | - | - |  | **Misses** | 1 | **Misses** | 0 |

**Step 4:** 0 386DE170 *Binary* - 0011 1000 0110 1101 1110 0001 0111 0000

*Byte offset* - 11 0000 *Index* - 1101 1110 0001 01 *Tag* - 0011 1000 0110

| **Step 4: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 1 | **Reads** | 0 |
| **Index** | 847 | 722 | 401 | 386 | 000 | 000 | 000 | 000 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 000 | 001 | 010 | 011 | 000 | 000 | 000 | 000 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | E | E | E | E | - | - | - | - |  | **Misses** | 1 | **Misses** | 0 |

**Step 5:** 0 269DE972 *Binary* - 0010 0110 1001 1101 1110 1001 0111 0010

*Byte offset* - 11 0010 *Index* - 1101 1110 1001 01 *Tag* - 0010 0110 1001

| **Step 5: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 1 | **Reads** | 0 |
| **Index** | 847 | 722 | 401 | 386 | 269 | 000 | 000 | 000 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 000 | 001 | 010 | 011 | 100 | 000 | 000 | 000 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | E | E | E | E | E | - | - | - |  | **Misses** | 1 | **Misses** | 0 |

**Step 6:** 0 241DE661 *Binary* - 0010 0100 0001 1101 1110 0110 0110 0001

*Byte offset* - 10 0001 *Index* - 1101 1110 0110 01 *Tag* - 0010 0100 0001

| **Step 6: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 1 | **Reads** | 0 |
| **Index** | 847 | 722 | 401 | 368 | 269 | 241 | 000 | 000 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 000 | 001 | 010 | 011 | 100 | 101 | 000 | 000 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | E | E | E | E | E | E | - | - |  | **Misses** | 1 | **Misses** | 0 |

**Step 7:** 0 992DE271 *Binary* - 1001 1001 0010 1101 1110 0010 0111 0001

*Byte offset* - 11 0001 *Index* - 1101 1110 0010 01 *Tag* - 1001 1001 0010

| **Step 7: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 1 | **Reads** | 0 |
| **Index** | 847 | 722 | 401 | 368 | 269 | 241 | 992 | 000 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 000 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | E | E | E | E | E | E | E | - |  | **Misses** | 1 | **Misses** | 0 |

**Step 8:** 0 434DE709 *Binary* - 0100 0011 0100 1101 1110 0111 0000 1001

*Byte offset* - 00 1001 *Index* - 1101 1110 0111 00 *Tag* - 0100 0011 0100

| **Step 8: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 1 | **Reads** | 0 |
| **Index** | 847 | 722 | 401 | 368 | 269 | 241 | 992 | 434 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | E | E | E | E | E | E | E | E |  | **Misses** | 1 | **Misses** | 0 |

**-------------------- Scenario 2 --------------------**

R/W on cache after Scenario 1.

TRACE\_FILE\_02:

0 434DE738

1 992DE240

2 269DE955

1 570DE383

1 553DE321

2 333DE896

Final results of *TRACE\_FILE\_02*:

Scenario 1 + 2 Total:

| **Scenario 2: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 9 | **Reads** | 2 |
| **Index** | 570 | 553 | 333 | 368 | 269 | 241 | 992 | 434 |  | **Writes** | 3 |  |  |
| **LRU Bits** | 100 | 110 | 111 | 000 | 011 | 001 | 101 | 010 |  | **Hits** | 2 | **Hits** | 0 |
| **MESI** | M | M | S | E | S | E | M | S |  | **Misses** | 10 | **Misses** | 2 |
|  |  |  |  |  |  |  |  |  |  | **Hit Ratio** | 2 : 10 | **Hit Ratio** | 0 : 2 |

0 = Read

1 = Write

2 = Instruction Fetch

R/W/H/M in Scenario 2 alone after Scenario 1:

| **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- |
| **Reads** | 1 | **Reads** | 2 |
| **Writes** | 3 | **Writes** | 0 |
| **Hits** | 2 | **Hits** | 0 |
| **Misses** | 2 | **Misses** | 2 |
| **Hit Ratio** | 2 : 2 | **Hit Ratio** | 0 : 2 |

**Step 1:** 0 434DE738 *Binary* - 0100 0011 0100 1101 1110 0111 0011 1000

*Byte offset* - 00 1001 *Index* - 1101 1110 0111 00 *Tag* - 0100 0011 0100

| **Step 1: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | **1** | **Reads** | 0 |
| **Index** | 847 | 722 | 401 | 368 | 269 | 241 | 992 | 434 |  | **Writes** | **0** |  |  |
| **LRU Bits** | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |  | **Hits** | **1** | **Hits** | 0 |
| **MESI** | E | E | E | E | E | E | E | S |  | **Misses** | **0** | **Misses** | 0 |

\*tag and index needs to match to be the same for a hit

**Step 2:** 1 992DE240 *Binary* - 1001 1001 0010 1101 1110 0010 0100 0000

*Byte offset* - 00 0000 *Index* - 1101 1110 0010 01 *Tag* - 1001 1001 0010

| **Step 2: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | **0** | **Reads** | 0 |
| **Index** | 847 | 722 | 401 | 368 | 269 | 241 | 992 | 434 |  | **Writes** | **1** |  |  |
| **LRU Bits** | 000 | 001 | 010 | 011 | 100 | 101 | 111 | 110 |  | **Hits** | **1** | **Hits** | 0 |
| **MESI** | E | E | E | E | E | E | M | S |  | **Misses** | **0** | **Misses** | 0 |

**Step 3:** 2 269DE955 *Binary* - 0010 0110 1001 1101 1110 1001 0101 0101

*Byte offset* - 01 0101 *Index* - 1101 1110 1001 01 *Tag* - 0010 0110 1001

| **Step 3: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | **0** | **Reads** | 1 |
| **Index** | 847 | 722 | 401 | 368 | 269 | 241 | 992 | 434 |  | **Writes** | **0** |  |  |
| **LRU Bits** | 000 | 001 | 010 | 011 | 111 | 100 | 110 | 101 |  | **Hits** | **0** | **Hits** | 0 |
| **MESI** | E | E | E | E | S | E | M | S |  | **Misses** | **0** | **Misses** | 1 |

\*instruction fetch for something written into data cache will be a miss

**Step 4:** 1 570DE383 *Binary* - 0101 0111 0000 1101 1110 0011 1000 0011

*Byte offset* - 00 0011 *Index* - 1101 1110 0011 10 *Tag* - 0101 0111 0000

| **Step 4: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | **0** | **Reads** | 0 |
| **Index** | 570 | 722 | 401 | 368 | 269 | 241 | 992 | 434 |  | **Writes** | **1** |  |  |
| **LRU Bits** | 111 | 000 | 001 | 010 | 110 | 011 | 101 | 100 |  | **Hits** | **0** | **Hits** | 0 |
| **MESI** | M | E | E | E | S | E | M | S |  | **Misses** | **1** | **Misses** | 0 |

**Step 5:** 1 553DE321 *Binary* - 0101 0101 0011 1101 1110 0011 0010 0001

*Byte offset* - 10 0001 *Index* - 1101 1110 0011 00 *Tag* - 0101 0101 0011

| **Step 5: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | **0** | **Reads** | 0 |
| **Index** | 570 | 553 | 401 | 368 | 269 | 241 | 992 | 434 |  | **Writes** | **1** |  |  |
| **LRU Bits** | 101 | 111 | 000 | 001 | 100 | 010 | 110 | 011 |  | **Hits** | **0** | **Hits** | 0 |
| **MESI** | M | M | E | E | S | E | M | S |  | **Misses** | **1** | **Misses** | 0 |

**Step 6:** 2 333DE896 *Binary* - 0011 0011 0011 1101 1110 1000 1001 0110

*Byte offset* - 01 0110 *Index* - 1101 1110 1000 10 *Tag* - 0011 0011 0011

| **Step 6: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | **0** | **Reads** | 1 |
| **Index** | 570 | 553 | 333 | 368 | 269 | 241 | 992 | 434 |  | **Writes** | **0** |  |  |
| **LRU Bits** | 100 | 110 | 111 | 000 | 011 | 001 | 101 | 010 |  | **Hits** | **0** | **Hits** | 0 |
| **MESI** | M | M | S | E | S | E | M | S |  | **Misses** | **0** | **Misses** | 1 |

**-------------------- Scenario 3 --------------------**

Other instructions on cache after Scenario 2.

TRACE\_FILE\_03:

3 333DE896

2 333DE890

4 269DE915

9 553DE938

1 363DE999

8 722DE323

Final results of *TRACE\_FILE\_03*:

Scenario 1 + 2 + 3 Total:

| **Scenario 3: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 0 | **Reads** | 0 |
| **Index** | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | I | I | I | I | I | I | I | I |  | **Misses** | 0 | **Misses** | 0 |
|  |  |  |  |  |  |  |  |  |  | **Hit Ratio** | 0 | **Hit Ratio** | 0 |

2 = Instruction Fetch

3 = Invalidate

4 = Data request (snoop)

8 = Clear

9 = Print (cache summary)

*In Scenario 3, we clear the cache and summary statistics, so the final stats are 0.*

Total stats in Scenario 3 before clearing the cache (before step 6):

| **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- |
| **Reads** | 9 | **Reads** | 3 |
| **Writes** | 4 |  |  |
| **Hits** | 2 | **Hits** | 1 |
| **Misses** | 11 | **Misses** | 2 |
| **Hit Ratio** | 2 : 11 | **Hit Ratio** | 1 : 2 |

**Step 1:** 3 333DE896 *Binary* - 0011 0011 0011 1101 1110 1000 1001 0110

*Byte offset* - 01 0110 *Index* - 1101 1110 1000 10 *Tag* - 0011 0011 0011

| **Step 1: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 0 | **Reads** | 0 |
| **Index** | 570 | 553 | 333 | 368 | 269 | 241 | 992 | 434 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 100 | 110 | 111 | 000 | 011 | 001 | 101 | 010 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | M | M | I | E | S | E | M | S |  | **Misses** | 0 | **Misses** | 0 |

No changes to stats - only updates communication msgs

**Step 2:** 2 333DE890 *Binary* - 0011 0011 0011 1101 1110 1000 1001 0000

*Byte offset* - 01 0000 *Index* - 1101 1110 1000 10 *Tag* - 0011 0011 0011

| **Step 2: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 0 | **Reads** | 1 |
| **Index** | 570 | 553 | 333 | 368 | 269 | 241 | 992 | 434 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 100 | 110 | 111 | 000 | 011 | 001 | 101 | 010 |  | **Hits** | 0 | **Hits** | 1 |
| **MESI** | M | M | S | E | S | E | M | S |  | **Misses** | 0 | **Misses** | 0 |

**Step 3:** 4 269DE915 *Binary* - 0010 0110 1001 1101 1110 1001 0001 0101

*Byte offset* - 01 0101 *Index* - 1101 1110 1001 00 *Tag* - 0010 0110 1001

| **Step 3: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** |  | **Reads** |  |
| **Index** | 570 | 553 | 333 | 368 | 269 | 241 | 992 | 434 |  | **Writes** |  |  |  |
| **LRU Bits** | 011 | 101 | 110 | 000 | 111 | 001 | 100 | 010 |  | **Hits** |  | **Hits** |  |
| **MESI** | M | M | S | E | I | E | M | S |  | **Misses** |  | **Misses** |  |

LRU 4 + → decrement

No changes to stats - only updates communication msgs

**Step 4:** 9 553DE938 *Binary* - 0011 0011 0011 1101 1110 1000 1001 0110

*Byte offset* - 01 0110 *Index* - 1101 1110 1000 10 *Tag* - 0011 0011 0011

| **Step 4: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 9 | **Reads** | 3 |
| **Index** | 570 | 553 | 333 | 368 | 269 | 241 | 992 | 434 |  | **Writes** | 3 |  |  |
| **LRU Bits** | 011 | 101 | 110 | 000 | 111 | 001 | 100 | 010 |  | **Hits** | 2 | **Hits** | 1 |
| **MESI** | M | M | S | E | I | E | M | S |  | **Misses** | 10 | **Misses** | 2 |
| \*This cache summary is the total from previous scenarios up until here | | | | | | | | |  | **Hit Ratio** | 2 : 10 | **Hit Ratio** | 1 : 2 |

**Step 5:** 1 363DE999 *Binary* - 0011 0011 0011 1101 1110 1000 1001 0110

*Byte offset* - 01 0110 *Index* - 1101 1110 1000 10 *Tag* - 0011 0011 0011

| **Step 5: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 0 | **Reads** | 0 |
| **Index** | 570 | 553 | 333 | 363 | 269 | 241 | 992 | 434 |  | **Writes** | 1 |  |  |
| **LRU Bits** | 010 | 100 | 101 | 111 | 110 | 000 | 011 | 001 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | M | M | E | M | I | E | M | S |  | **Misses** | 1 | **Misses** | 0 |

**Step 6:** 8 722DE323 *Binary* - 0011 0011 0011 1101 1110 1000 1001 0110

*Byte offset* - 01 0110 *Index* - 1101 1110 1000 10 *Tag* - 0011 0011 0011

| **Step 6: Cache Contents Table** | | | | | | | | |  | **Data Cache** | | **Instr. Cache** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Way** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |  | **Reads** | 0 | **Reads** | 0 |
| **Index** | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 |  | **Writes** | 0 |  |  |
| **LRU Bits** | 000 | 000 | 000 | 000 | 000 | 000 | 000 | 000 |  | **Hits** | 0 | **Hits** | 0 |
| **MESI** | I | I | I | I | I | I | I | I |  | **Misses** | 0 | **Misses** | 0 |

Demo

Our team had a demo scheduled with our TA Nithyakalyani Sampath on March 13, 2023, at 11:45 am. All five team members were present during the demo. We began the demo by giving Nithyakalyani a brief overview of our code structure and the parser function. As per her instructions, we ran the program in mode 0, which is the statistics-only mode, on the provided trace file. After running the program, we obtained the output from the trace file which included the following statistics.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Simulation of an L1 split cache of a 32-bit Processor

ECE 585 Final Project Winter 2023 - Team 1

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

Mode 0: Display only the required summary of usage statistics and responses to 9s in the trace file.

Mode 1: Display everything from mode 1, as well as the communication messages to the L2 cache.

Please enter the mode number you'd like to select (0,1): 0

\*\* KEY CACHE USAGE STATISTICS \*\*

-- DATA CACHE STATISTICS --

number of Cache Reads: 159631

number of Cache Writes: 83030

number of Cache Hits: 241528

number of Cache Misses: 1133

Cache Hit Ratio: 0.995331

Cache Hit Percentage: 99.5331 %

-- INSTRUCTION CACHE STATISTICS --

number of Cache Reads: 757341

number of Cache Hits: 754368

number of Cache Misses: 2973

Cache Hit Ratio: 0.996074

Cache Hit Percentage: 99.6074 %

--- L1 Split Cache Analysis and Simulation Completed! —

**Note:**

During the Demo, our program was run in mode 0, which only outputs the statistics and not the communication messages with L2. However, if the program is run in mode 1, the communication messages with L2 would be outputted. Please note that the statistics will only be displayed if there is a 9 in the trace file.

Maintaining Key Points During Our Project Development

- Quality of implementation:

We ensured a high-quality implementation of our design by utilizing classes and creating an object-oriented structure. The project includes a head with declarations, installations, and instantiations, as well as a combined .cpp file that includes the main function and the 11 other functions used in our design.

- Structure and clarity of design:

To maintain a modular, easy-to-implement, and understandable structure, we used classes and named functions clearly. We defined key constants in the header file to aid readers in understanding what abstract numbers are used for. We also commented thoroughly on our code, providing a comment for each function used. The function comments clearly explain what the function does, how it does it, and what inputs it takes or outputs it returns, if any.

- Readability:

We prioritized readability throughout the project (code and report) by explaining why we chose each value (constants, variables, flags, etc.) used in our design based on the design specification and assumptions. Our code is designed so that someone who has not worked on it can easily understand what we are doing and how the code works.

- Maintainability:

By using classes and creating an object-oriented structure, implementing numerous functions specific to each task of our design, thoroughly commenting on our code, and documenting our big structure and design decisions in our final report, we were able to keep our project easily maintainable and configurable.

- Testing:

We created various trace files to test the overall design and various corner cases, as elaborated in the testing section above. We tailor-designed trace files to test the instruction cache, data cache, eviction policy, LRU algorithm, MESI protocol transitions, parser function, and other parts of our design. Our design's modularity and function-based approach allowed us to individually test specific tasks before testing the entire system. Based on our testing results, we improved our design to achieve the expected outputs from our hand calculations.

- Presentation of results:

We made sure our output was clear and well-presented, with well-spaced and well-aligned text. We presented an introduction to the program's purpose, provided two modes for the user to choose from, and separated instruction and data cache statistics to make the results more understandable. We concluded the output with a message indicating that the analysis was complete. Overall, we aimed to make the program and its results understandable to someone who had not worked on the project or read the final report.

Member Contributions

* Mohamed Ghonim
  + Developed and coded the LRU bits finding and updating algorithm. Integrated the MESI protocol into the design functions. I also worked on the implementation and code for the read, write and fetch functions. Additionally, I integrated all functions and code together to complete the project. I also collaborated with the team to structure the overall design and participated in testing and final report documentation.
* Ahliah Nordstrom
  + Developed the methodology for the cache structure and MESI state machine states/cases. I helped develop this into pseudocode and C++. Then, I helped to create testing trace files and scenarios by hand. I also contributed to all report documentation.
* Brandon Bieschke
  + Developed the import code for inputting data from a file. Developed the logic to convert input formatting in order to route instructions. Organized the case statements to route instructions. Assisted with code integration, debugging, and testing.
* Assefa Setgen
  + Worked on the snooping, invalidate, and clear functions during the algorithm building process. Conducted multiple tests to the program and debugged the code for the evaluation and verification of the code. Worked on the scenarios crafting process. Participated in the finite state machine diagram for MESI.
* Celina Wong
  + Contributed to the introduction in the main() function where the user input is prompted, the combination of the statistics when a 9 is encountered in print\_stats() function, developed documentation of various test cases and hand-solved them to compare with the stat summary/results of our program, and tested each step in the scenarios of the test cases with the program by setting a 9 after each line of the trace file to ensure sequential accuracy.

Appendix

In this appendix, we have included the complete implementation of our design in C++. The code is divided into two files: header.h and combined.cpp, which contains the main function along with the other 11 functions used in our design. We have taken care to provide detailed comments throughout our code and to ensure that lines and comments are properly aligned for optimal readability. However, please note that the formatting may be lost in this appendix due to Word formatting. To ensure the best possible code readability, we recommend opening the .cpp and .h files in an official C++ IDE. Additionally, our submission includes the .h, .cpp, and .exe files of our design.

Header.h file:

#pragma once

/\*

ECE 585 Microprocessor System Design Final Project

Winter 2023 - Team 1

Mohamed Ghonim - Ahliah Nordstrom - Brandon Bieschke - Celina Wong - Assefa Setgen

\*/

// This header file contains the directives, declarations, classes, and instantiations to be used in our project //

#include <iostream>

#include <fstream>

#include <vector>

#include <string>

#include <stdlib.h>

#include <stdio.h>

#include <bitset>

#include <sstream>

#include <string.h>

using namespace std; // To indicate that we can use object and variable names from the standard library

// Define some constants we will be using in the design, this is optional, but will help keep outdesign neat

// Good Coding practice is to make those all upper case

constexpr auto BYTE\_OFFSET = 6; // the byte offset is 6 bits

constexpr auto CACHE\_INDEX = 14; // the set bits/cache index is 14 bits

constexpr auto TAG\_BITS = 12; // the tag bits are (32 - 14 - 6 = 12 bits)

constexpr auto MASK\_BYTE\_SELECT = 0x0000003F; // mask the least significant 6 bits for the byte offset

constexpr auto MASK\_CACHE\_INDEX = 0x000FFFFF; // mask the least significant 20 bits for the (byte offset + cache index/set)

constexpr auto MASK\_TAG\_BITS = 0xFFF0000; // mask the most significant 12 bits for the byte offset

constexpr auto EMPTY\_TAG = 4096; //We are using the value 4096 to indicate an empty tag(since 0 - 4095 are used)

//Class/container for the cache variables (Properties)

// We are using unsigned integers here, but we're

class cache {

public:

char MESI\_char = 'I'; // MESI bits initialized in the Invalid state.

/\*I - Invalid, S - Shared, M - Modified, E - Exclusive \*/

unsigned int LRU\_bits; // LRU bits (3 bits for L1\_data and 2 for L1\_inst)

unsigned int address; // Address bits

unsigned int set\_bits; // Set bits

unsigned int tag\_bits; // tag bits

};

// instantiate data and instruction caches from the cache class

cache L1\_data[8][16384]; // 8-way 16k sets data cache [8-way][16k]

cache L1\_inst[4][16384]; // 4-way 16k sets instruction cache [4-way][16k]

// This is a class/container for the cache statistics with parameters that counts number of reads, writes, hits, misses, and ratios

class usage\_statistics {

public:

// Data cache

unsigned int data\_read; // Data cache read count

unsigned int data\_write; // Data cache write count

unsigned int data\_hit; // Data cache hit count

unsigned int data\_miss; // Data cache miss count

float data\_hit\_ratio; // Data hit/miss ratio

// Instruction cache

unsigned int inst\_read; // Instruction cache read count

unsigned int inst\_miss; // Instruction cache miss count

unsigned int inst\_hit; // Instruction cache hit count

float inst\_hit\_ratio; // Instruction hit/miss ratio

};

// Instantiation of the usage\_statistics class

usage\_statistics statistics;

// Output mode, mode is either 0 or 1. We might add an additional mode 2 for debug information.

// This is declared in the header/globally because we'll check it throughout different function

unsigned int mode;

/\*........................................\*/

// Declaring the functions used in the design

// All functions used in the main .cpp file need to be declared here.

void clear\_cache();

int matching\_tag(unsigned int tag, unsigned int set, char which\_cache);

void L1\_LRU(unsigned int way, unsigned int set, bool empty\_flag, char which\_cache);

int find\_LRU(unsigned int set, char which\_cache);

void invalidate(unsigned int addr);

void fetch\_inst(unsigned int addr);

void snoop(unsigned int addr);

void read(unsigned int addr);

void write(unsigned int addr);

void print\_stats();

void parser(int argc, char\*\* argv);

Combined.cpp file:

#include "header.h"

/\*

ECE 585 Microprocessor System Design Final Project

Winter 2023 - Team 1

Mohamed Ghonim - Ahliah Nordstrom - Brandon Bieschke - Celina Wong - Assefa Setgen

\*/

/\* This is the main function of our project

\* It prompts the user to enter the mode (0 or 1) of operation,

\* takes in the file name of the trace txt file

\* and calls the import function

\*/

int main(int argc, char\*\* argv) {

// Initialize the cache at the beginning

clear\_cache();

cout << "\n - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n" << endl;

cout << "\t Simulation of an L1 split cache of a 32-bit Processor\n" << endl;

cout << "\t ECE 585 Final Project Winter 2023 - Team 1" << endl;

cout << "\n - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n" << endl;

cout << " Mode 0: Display only the required summary of usage statistics and responses to 9s in the trace file." << endl;

cout << " Mode 1: Display everything from mode 1, as well as the communication messages to the L2 cache." << endl;

do {

cout << "\n Please enter the mode number you'd like to select (0,1): ", cin >> mode;

if (mode > 1) {

cout << "\n\tInvalid mode value." << endl;

}

} while (mode > 1);

cout << "\n"; //spacing to make it look neater

// call the parser function, and give it the command line file names

parser(argc, argv);

cout << "\n\t --- L1 Split Cache Analysis and Simulation Completed! --- " << endl;

return 0;

}

/\* This is the file parser function

\* It reads the trace file and gets the operation and the address from each line in the file

\* The function doesn't return anything, but it takes the program to the respective n functions

\* Depending on the operation n \*/

void parser(int argc, char\*\* argv) {

// Reading the trace file from the command line

char\* trace\_file = argv[1]; //argv is an array of the strings

// Check for command line test file

if (argc != 2) { // if the count of arguments is not 2

cout << "\n Not able to read the trace file!" << endl;

exit(1); // exit (0) is exit after successful run, exit(1) exit after failure

}

char trace\_line[1024]; // the address is 8 char (hex) and one black space (1) and one op char = 10 char = 2^10 = 1024

char trace\_operation[1]; // n is one character long

unsigned int operation; // Operation parsed from input

unsigned int address; // Address parsed from input

FILE\* fp; // .txt test file pointer (fp)

fp = fopen(trace\_file, "r"); // "r" switch of the fopen function opens a file for reading

while (fgets(trace\_line, 1024, fp)) {

// sscanf reads the data from trace\_line and stores it in trace\_operation and address

sscanf(trace\_line, "%c %x", trace\_operation, &address);

// %c is a switch for a single character, %x is for hexadecimal in lowercase letters

if (!strcmp(trace\_line, "\n") || !strcmp(trace\_line, " ")) {

//strcmp compares two strings

} else {

operation = atoi(trace\_operation); //Parses the C-string str interpreting its content as an integral number, which is returned as a value of type int.

switch (operation) {

case 0: read(address); break;

case 1: write(address); break;

case 2: fetch\_inst(address); break;

case 3: invalidate(address); break;

case 4: snoop(address); break;

case 8: clear\_cache(); break;

case 9: print\_stats(); break;

default:

cout << "\n the value of n (the operation) is not valid \n" << endl;

break;

}

}

}

fclose(fp);

return;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*

\*\*\*\*

\* Functions

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Data read Function

\* in response to n = 0 in the trace file

\* This function attempts to read a an address/line from the data cache

\* If we have a miss, we place the line in an empty line

\* if no empty lines, we check for an a line in the invalid state, evict it and place the line there

\* if not invalid states, we evict the LRU and place the line in that way

\* This function takes in an address, and doesn't need to return anything

\* For the data read MESI protocol we have:

\* If we're in 'E' we go to 'S', if we're in 'S' stay in 'S', if in 'M' stay in 'M', if in 'I' go to 'E'

\*/

void read(unsigned int addr) {

unsigned int tag = addr >> (BYTE\_OFFSET + CACHE\_INDEX); // Shift the address right by (6+14=20) to get the tag

unsigned int set = (addr & MASK\_CACHE\_INDEX) >> BYTE\_OFFSET;

// Mask the address with 0x000FFFFF and shift by 6 to get the cache index/set

bool empty\_flag = 0; // boolean flag for empty way in the cache, if 1, we have an empty way

int way = -1; // way in the cache set. Initialized with an invalid way value

statistics.data\_read++; // Increment the number of reads for the data cache

way = matching\_tag(tag, set, 'D');

// Look for a matching tag already in the data cache

if (way >= 0) {

// if we have a matching tag, then we have a data cache hit! (unless invalid MESI state)

switch (L1\_data[way][set].MESI\_char) {

case 'M': // in we're in the modified state

statistics.data\_hit++; // Increment the data hit counter L1\_data[way][set].MESI\_char = 'M'; // stay in the modified state

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

// Update the data cache LRU count

L1\_LRU(way, set, empty\_flag, 'D'); break;

case 'E': // if we're in the exclusive state, assume a different processor is reading

statistics.data\_hit++; // Increment the data hit counter L1\_data[way][set].MESI\_char = 'S';

// Move the Shared state, another processor is doing the read operation

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

L1\_LRU(way, set, empty\_flag, 'D');

// Update the data cache LRU count

break;

case 'S':

// if we are in the shared state

statistics.data\_hit++; // Increment the data hit counter

L1\_data[way][set].MESI\_char = 'S'; // remain in the shared state

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

// Update the data cache LRU count

L1\_LRU(way, set, empty\_flag, 'D'); break;

// if we are in the invalid state, we'll have a miss

case 'I':

statistics.data\_miss++; // Increment the data miss counter

L1\_data[way][set].MESI\_char = 'S'; // and we move to the Shared state.

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

// Update the data cache LRU count

L1\_LRU(way, set, empty\_flag, 'D');

if (mode == 1) {

// Simulating a cache read communication with L2

cout << " [Data-Operation] Read from L2: L1 cache miss, obtain data form L2 " << hex << addr << endl;

}

break;

}

}

else { // if we don't have a matching tag, we have a data cache miss

statistics.data\_miss++; // Increment the data cache miss counter

// First, check if we have any empty lines

for (int i = 0; (way < 0 && i < 8); ++i) {

if (L1\_data[i][set].tag\_bits == EMPTY\_TAG) {

way = i; // return the way that has an empty line

empty\_flag = 1; // if we have an empty line, toggle the empty\_flag to high

}

}

if (way >= 0) { // if we have an empty line, place the read data in it

L1\_data[way][set].MESI\_char = 'E';

// and mark it exclusive, it's the only line with this data

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

// Update the data cache LRU order/count

L1\_LRU(way, set, empty\_flag, 'D');

if (mode == 1) { // Simulating a cache read communication with L2

cout << " [Data-Operation] Read from L2: L1 cache miss, obtain data form L2 " << hex << addr << endl;

}

}

else { // if we don't have any empty lines, we need to evict the LRU line.

if (mode == 1) {

cout << " [Data-Operation] Read from L2: L1 cache miss, obtain data from L2 " << hex << addr << endl;

}

for (int n = 0; n < 8; ++n) {

if (L1\_data[n][set].MESI\_char == 'I') {

// Since we don't have empty lines, we check for a way with an invalid lines to evict

way = n;

}

else {

way = -1;

// if we don't have any invalid data cache lines, flag the way with "-1"

}

}

if (way < 0) { //If we don't have any invalid lines,

way = find\_LRU(set, 'D'); // Find the LRU way in the data cache

if (way >= 0) { // if we have a way that's 0, (LRU in the data cache, we use it)

L1\_data[way][set].MESI\_char = 'E'; // the data here will be new, and exclusive to this cache

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

L1\_LRU(way, set, empty\_flag, 'D'); // update the L1 data cache LRU bits

}

}

else { // if we have an invalid way, we evict it and

L1\_data[way][set].MESI\_char = 'E';

// the data here will be new, and exclusive to this cache

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

L1\_LRU(way, set, empty\_flag, 'D');

}

}

}

return;

}

/\* Data write Function

\* in response to n = 1 in the trace file

\* This function attempts to write a an address/line from the data cache

\* If we have a miss, we place the line in an empty line

\* if no empty lines, we check for an a line in the invalid state, evict it and place the line there

\* if not invalid states, we evict the LRU and place the line in that way

\* This function takes in an address, and doesn't need to return anything

\* For the data read MESI protocol we have:

\* If we're in 'M' we stay in 'M', if we're in 'E' we go to 'M', if in 'S' we go to 'E', if in 'I' go to 'E'

\*/

void write(unsigned int addr) {

unsigned int tag = addr >> (BYTE\_OFFSET + CACHE\_INDEX); // Shift the address right by (6+14=20) to get the tag

unsigned int set = (addr & MASK\_CACHE\_INDEX) >> BYTE\_OFFSET;

// Mask the address with 0x000FFFFF and shift by 6 to get the cache index/set

bool empty\_flag = 0; // boolean flag for empty way in the cache, if 1, we have an empty way

int way = -1; // way in the cache set. Initialized with an invalid way value

statistics.data\_write++; // Increment the number of write for the data cache

way = matching\_tag(tag, set, 'D'); // Look for a matching tag already in the data cache

if (way >= 0) { // if we have a matching tag, then we have a data cache hit! (unless invalid MESI state)

switch (L1\_data[way][set].MESI\_char) {

case 'M': // in we're in the modified state

statistics.data\_hit++; // Increment the data hit counter (We have a new hit!)

L1\_data[way][set].MESI\_char = 'M'; // stay in the modified state

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

L1\_LRU(way, set, empty\_flag, 'D'); // Update the data cache LRU count

break;

case 'E': // if we're writing in an exclusive state, the data will be modified

statistics.data\_hit++; // Increment the data hit counter (We have a new hit!)

L1\_data[way][set].MESI\_char = 'M'; // go to the modified state

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

L1\_LRU(way, set, empty\_flag, 'D'); // Update the data cache LRU count

break;

case 'S': // if we're writing in a shared state, the written data will be exclusive

statistics.data\_hit++; // Increment the data hit counter (We have a new hit!)

L1\_data[way][set].MESI\_char = 'E'; // go to the exclusive state

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

L1\_LRU(way, set, empty\_flag, 'D'); // Update the data cache LRU count

break;

case 'I' // if we are in the invalid state, we'll have a miss

// Increment the data miss counter

statistics.data\_miss++; L1\_data[way][set].MESI\_char = 'E'; // go to the exclusive state

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

// Update the data cache LRU count

L1\_LRU(way, set, empty\_flag, 'D');

break;

}

}

else { // Data Cache Miss

statistics.data\_miss++;

// Increment the data miss counter

if (mode == 1) {

// Simulating a cache Read For Ownership communication from L2

cout << " [Data-Operation] Read for Ownership from L2 " << hex << addr << endl;

}

for (int i = 0; way < 0 && i < 8; ++i) {

// First, check if we have any empty lines

if (L1\_data[i][set].tag\_bits == EMPTY\_TAG) {

way = i;

// return the way that has an empty line

empty\_flag = 1;

// if we have an empty line, toggle the empty\_flag to high

}

}

if (way >= 0) {

// if we have an empty line, place the read data in it

L1\_data[way][set].MESI\_char = 'M'; // go to the modified state

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

L1\_LRU(way, set, empty\_flag, 'D'); // Update the data cache LRU order/count

if (mode == 1) { // Simulating an iniial write through communication from L2

cout << " [Data Write\_Through] Write to L2: we have a data Cache Miss " << hex << addr << endl;

}

}

else { // if we don't have any empty lines, we need to evict the LRU line.

if (mode == 1) { // Simulating an write back communication from L2

cout << " [Data Write\_Back] Write to L2: We have a data Cache Miss " << hex << addr << endl;

}

for (int n = 0; n < 8; ++n) {

if (L1\_data[n][set].MESI\_char == 'I') { // if we have an invalid way, we evict it

way = n; // return the way that has an Invalid line

}

else {

way = -1; // set the way to -1, meaning no invalid lines

}

}

if (way < 0) { //If we don't have any invalid lines,

way = find\_LRU(set, 'D'); // Find the LRU way in the data cache

if (way >= 0) { // if we have a way that's 0, (LRU in the data cache, we use it)

L1\_data[way][set].MESI\_char = 'M'; // go to the modified state

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

L1\_LRU(way, set, empty\_flag, 'D'); // update the L1 data cache LRU bits

}

}

else { // if we have an invalid way, we evict it

L1\_data[way][set].MESI\_char = 'M'; // go to the modified state

L1\_data[way][set].tag\_bits = tag;

L1\_data[way][set].set\_bits = set;

L1\_data[way][set].address = addr;

L1\_LRU(way, set, empty\_flag, 'D'); // update the L1 data cache LRU bits

}

}

}

return;

}

/\* Instruction Fetch Function

\* in response to n = 2 in the trace file

\* This function attempts to read/fetch a an address/line from the instruction cache

\* If we have a miss, we place the instruction in an empty line

\* if no empty lines, we check for an a line in the invalid state, evict it and place the line there

\* if not invalid states, we evict the LRU and place the line in that way

\* This function takes in an address, and doesn't need to return anything

\* For the instruction fetch MESI protocol we have:

\* If we're in 'E' we go to 'S', if we're in 'S' stay in 'S', if in 'M' stay in 'M', if in 'I' go to 'S'

\*/

void fetch\_inst(unsigned int addr) {

unsigned int tag = addr >> (BYTE\_OFFSET + CACHE\_INDEX); // Shift the address right by (6+14=20) to get the tag

unsigned int set = (addr & MASK\_CACHE\_INDEX) >> BYTE\_OFFSET; // Mask the address with 0x000FFFFF and shift by 6 to get the cache index/set

bool empty\_flag = 0; // boolean flag for empty way in the cache, if 1, we have an empty way

int way = -1; // way in the cache set. Initialized with an invalid way value

// Increment the number of reads for the instructions cache

statistics.inst\_read++;

way = matching\_tag(tag, set, 'I'); // Matching tag already in the instruction cache

if (way >= 0) {

// if we have a matching tag, then we have an instruction cache hit! (unless invalid MESI state)

switch (L1\_inst[way][set].MESI\_char) { // FSM implementing the MESI protocol

case 'M': // If we are in the modified state

statistics.inst\_hit++; // We have a valid hit, increment the instruction cache hit counter

L1\_inst[way][set].MESI\_char = 'M'; // We remain in the modified MESI state

L1\_inst[way][set].tag\_bits = tag;

L1\_inst[way][set].set\_bits = set;

L1\_inst[way][set].address = addr;

// Update the instruction cache LRU count

L1\_LRU(way, set, empty\_flag, 'I');

break;

case 'E': // If we are in the modified state, another processor is fetching

statistics.inst\_hit++;

// We have a valid hit, increment the instruction cache hit counter

L1\_inst[way][set].MESI\_char = 'S'; // Move the Shared state

L1\_inst[way][set].tag\_bits = tag;

L1\_inst[way][set].set\_bits = set;

L1\_inst[way][set].address = addr;

L1\_LRU(way, set, empty\_flag, 'I');

// Update the instruction cache LRU count

break;

case 'S':

// If we are in the shared state

statistics.inst\_hit++;

// We have a valid hit, increment the instruction cache hit counter

L1\_inst[way][set].MESI\_char = 'S';

// We remain in the modified MESI state

L1\_inst[way][set].tag\_bits = tag;

L1\_inst[way][set].set\_bits = set;

L1\_inst[way][set].address = addr;

// Update the instruction cache LRU count

L1\_LRU(way, set, empty\_flag, 'I');

break;

case 'I':

// If we are in the invalid state, we have a miss even though the tag exists

// Increment the instruction cache miss counter

statistics.inst\_miss++;

// Move the Shared state, another processor is doing the fetch

L1\_inst[way][set].MESI\_char = 'S';

L1\_inst[way][set].tag\_bits = tag;

L1\_inst[way][set].set\_bits = set;

L1\_inst[way][set].address = addr;

L1\_LRU(way, set, empty\_flag, 'I'); // Update the instruction cache LRU count

if (mode == 1) { // Simulating the cache fetch instruction communication with L2

cout << " [Instruction-Operation] Read from L2: L1 cache miss, obtain instruction form L2 " << hex << addr << endl;

}

break;

}

}

else { // if we don't have a matching tag, we have an instruction cache miss

statistics.inst\_miss++; // Increment the instruction cache miss counter

for (int i = 0; way < 0 && i < 4; ++i) { // First, check if we have any empty lines

if (L1\_inst[i][set].tag\_bits == EMPTY\_TAG) {

way = i; // return the way that has an empty line

empty\_flag = 1; // if we have an empty line, toggle the empty\_flag to high

}

}

if (way >= 0) {

// if we have an empty line, place the fetched instruction in it

// and mark it exclusive, it's the only line with this instruction

L1\_inst[way][set].MESI\_char = 'E'; L1\_inst[way][set].tag\_bits = tag;

L1\_inst[way][set].set\_bits = set;

L1\_inst[way][set].address = addr;

// Update the instruction cache LRU order/count

L1\_LRU(way, set, empty\_flag, 'I');

if (mode == 1) { // Simulating the cache fetch instruction communication with L2

cout << " [Instruction-Operation] Read from L2: L1 cache miss, obtain instruction form L2 " << hex << addr << endl;

}

}

else { // if we don't have any empty lines, we need to evict the LRU line.

if (mode == 1) {

cout << " [Instruction-Operation] Read from L2: L1 cache miss, obtain instruction from L2 " << hex << endl;

}

for (int n = 0; n < 4; ++n) {

if (L1\_inst[n][set].MESI\_char == 'I') { // Since we don't have empty lines, we check for a way with an invalid lines to evict

way = n;

}

else {

way = -1; // if we don't have any invalid instruction cache lines, flag the way with "-1"

}

}

if (way < 0) { //If we don't have any invalid lines,

way = find\_LRU(set, 'I'); // Find the LRU way in the instruction cache

if (way >= 0) {

// if we have a way that's 0, (LRU in the instruction cache, we use it)

L1\_inst[way][set].MESI\_char = 'E';

// the instruction here will be new, and exclusive to this cache

L1\_inst[way][set].tag\_bits = tag;

L1\_inst[way][set].set\_bits = set;

L1\_inst[way][set].address = addr;

L1\_LRU(way, set, empty\_flag, 'I'); // update the L1 instruction cache LRU bits

}

}

else { // if we have an invalid way, we evict it and

// the instruction here will be new, and exclusive to this cache

L1\_inst[way][set].MESI\_char = 'E'; L1\_inst[way][set].tag\_bits = tag;

L1\_inst[way][set].set\_bits = set;

L1\_inst[way][set].address = addr;

L1\_LRU(way, set, empty\_flag, 'I');

}

}

}

return;

}

/\* Snooping Function

\* in response to n = 4 in the trace file

\* This function is a simulation of L2 snooping on L1

\* During this snooping operation, L2 invalidates all the MESI bits.

\* For MESI protocol of the snooping operation, we have:

\* If we're in 'E' we go to 'I', if we're in 'S' go to 'I', if in 'I' stay in 'I', if in 'M' go to 'I'

\*/

void snoop(unsigned int addr) {

unsigned int tag = addr >> (BYTE\_OFFSET + CACHE\_INDEX); // Shift the address right by (6+14=20) to get the tag

unsigned int set = (addr & MASK\_CACHE\_INDEX) >> BYTE\_OFFSET; // Mask the address with 0x000FFFFF and shift by 6 to get the cache index/set

for (int i = 0; i < 8; ++i) { // Look for a matching tag in the data cache

if (L1\_data[i][set].tag\_bits == tag) {

L1\_data[i][set].MESI\_char = 'I'; // Change MESI bit to Invalid

L1\_data[i][set].tag\_bits = tag;

L1\_data[i][set].set\_bits = set;

L1\_data[i][set].address = addr;

if (mode == 1) { // Simulating a snoop return data communication with L2

cout << "[Snoop-operation] Return data to L2 " << hex << addr << endl;

}

}

}

return;

}

/\*

\* When this function is called it clears the cache to reset it to the initial empty state

\* and clears all the statistics.

\*/

void clear\_cache() {

// Clear and reset the data cache

for (int i = 0; i < 8; ++i) {

for (int j = 0; j < 16384; ++j) {

L1\_data[i][j].MESI\_char = 'I'; // Reset the MESI protocol to the Invalid state

L1\_data[i][j].LRU\_bits = 0; // reset the LRU bits to 0

L1\_data[i][j].tag\_bits = EMPTY\_TAG; // We are using the value 4096 to indicate an empty tag (since 0 - 4095 are used)

L1\_data[i][j].set\_bits = 0; // reset the set bits to 0

L1\_data[i][j].address = 0; // reset the address to 0

}

}

// Clearing the instruction cache

for (int n = 0; n < 4; ++n) {

for (int m = 0; m < 16384; ++m) {

// Reset the MESI protocol to the Invalid state

L1\_inst[n][m].MESI\_char = 'I'; L1\_inst[n][m].LRU\_bits = 0; // reset the LRU bits to 0

L1\_inst[n][m].tag\_bits = EMPTY\_TAG; // We are using the value 4096 to indicate an empty tag (since 0 - 4095 are used)

L1\_inst[n][m].set\_bits = 0; // reset the set bits to 0

L1\_inst[n][m].address = 0; // reset the address to 0

}

}

// Clear the instruction cache statistics and reset their values to 0

statistics.inst\_read = 0;

statistics.inst\_hit = 0;

statistics.inst\_miss = 0;

statistics.inst\_hit\_ratio = 0.0;

// Clear the data cache statistics and reset their values to 0

statistics.data\_read = 0;

statistics.data\_write = 0;

statistics.data\_hit = 0;

statistics.data\_miss = 0;

statistics.data\_hit\_ratio = 0.0;

return;

}

/\* Invalidate Operation

\* This function will change the MESI state to 'I' Invalid

\* Regardless of what it was before

\* This function is a simulation of an invalidate command from L2

\* The function takes in the address needed to be invalidated, doesn't output anything

\*/

void invalidate(unsigned int addr) {

unsigned int tag = addr >> (BYTE\_OFFSET + CACHE\_INDEX);

// Shift the address right by (6+14=20) to get the tag

unsigned int set = (addr & MASK\_CACHE\_INDEX) >> BYTE\_OFFSET;

// Mask the address with 0x000FFFFF and shift by 6 to get the cache index/set

// Search the L1\_data cache for the matching tag

for (int i = 0; i < 8; ++i) {

if (L1\_data[i][set].tag\_bits == tag) {

// change the MESI Protocol to the Invalid state

L1\_data[i][set].MESI\_char = 'I';

L1\_data[i][set].tag\_bits = tag;

L1\_data[i][set].set\_bits = set;

L1\_data[i][set].address = addr;

}

// There's no else. If we don't have a matching tag, do nothing

}

return;

}

/\*

This function updates the LRU bits for the Data Cache and the instruction caches

It takes 4 inputs. the cache way, the set, an empty cache flag, and a which\_way char flag

if the which\_way char flag is 'D' or 'd', we're operating on the data cache

if the which\_way char flag is 'I' or 'i', we're operating on the instruction cache

First, we check if a way is empty, if it's empty we put the data or instruction in it

and make it the MRU, my setting its LRU bits to 111 (0x7) if data, or 11 (0x3) if instruction

If a way is NOT empty, we compare LRU bits of the current set with the LRU bits of the other sets

this function updates the LRU, and doesn't retun anything. (no output)

\*/

void L1\_LRU(unsigned int way, unsigned int set, bool empty\_flag, char which\_cache) {

switch (which\_cache) {

case ('d'):

case ('D'): // We're in the L1\_data cache if which\_chache is 'D' or 'd'

// If the way is empty, we use it and decrement every way less than this way.

if (empty\_flag) {

for (int i = 0; i < way; ++i) {

L1\_data[i][set].LRU\_bits = --L1\_data[i][set].LRU\_bits; //pre-decrement the LRU bits

}

}

// If a way is NOT empty, we compare LRU bits of the current set with the LRU bits of the other sets

else {

for (int i = 0; i < 8; ++i) {

if (L1\_data[way][set].LRU\_bits > L1\_data[i][set].LRU\_bits) {

//no need to do anything, keep the same order

L1\_data[i][set].LRU\_bits = L1\_data[i][set].LRU\_bits; }

else

{ //pre-decrement the LRU bits

L1\_data[i][set].LRU\_bits = --L1\_data[i][set].LRU\_bits;

}

}

}

L1\_data[way][set].LRU\_bits = 0x7; // Set the current set to MRU 111 (0x7 in Hex)

break;

case ('i'):

case ('I'): // We're in the L1\_instruction cache if which\_chache is 'I' or 'i'

// If a way is empty, we use it and decrement every way less than this way

if (empty\_flag) {

for (int i = 0; i < way; ++i) {

L1\_inst[i][set].LRU\_bits = --L1\_inst[i][set].LRU\_bits;

}

}

// If a way is NOT empty, we compare LRU bits of the current set with the LRU bits of the other sets

else {

for (int i = 0; i < 4; ++i) {

if (L1\_inst[way][set].LRU\_bits > L1\_inst[i][set].LRU\_bits) {

//no need to do anything, keep the same order

L1\_inst[i][set].LRU\_bits = L1\_inst[i][set].LRU\_bits; }

else

{ //pre-decrement the LRU bits

--L1\_inst[i][set].LRU\_bits; }

}

}

L1\_inst[way][set].LRU\_bits = 0x3; // Set the current set to MRU 11 (0x3 in Hex)

break;

}

}

/\* Find the Least Recently Used Way function

\* This funcion finds the way which includes this set and has an LRU of 0

\* This function takes in the set, and a which\_cache flag

\* If which cache is 'D' or 'd' we want to find the Data cache LRU

\* If which cache is 'I' or 'i' we want to find the instruction cache LRU

\* The function returns the way least recently used (to be evicted).

\* If we can't find a LRU way, something is wrong, and we return -1

\*/

int find\_LRU(unsigned int set, char which\_cache) {

switch (which\_cache) {

case ('d'):

case ('D'): // We're in the L1\_data cache if which\_chache is 'D' or 'd'

for (int i = 0; i < 8; ++i) {

// find the set with the LRU bits of 0 (Least Recently Used)

if (L1\_data[i][set].LRU\_bits == 0) { return i; // return the way/index in which the LRU is

}

}

break;

case ('i'):

case ('I'): // We're in the L1\_instruction cache if which\_chache is 'I' or 'i'

for (int i = 0; i < 4; ++i) {

if (L1\_inst[i][set].LRU\_bits == 0) {

// find the set with the LRU bits of 0 (Least Recently Used)

return i; // return the way/index in which the LRU is

}

}

break;

}

return -1; // This means that something is wrong! We expected an LRU (0) but we couldn't find any

}

/\* matching\_tag function

\* This function looks for a matching tag in the cache.

\* If it finds a matching tag, it returns the way in which we have this matching tag.

\* If we have matching tag, we have a hit, except if the MESI char is "I" invalid.

\* This function takes in the tag, the set, and a char flag to indicate whether we're

\* looking in the data or the instructions cache.

\* Inputs: tag, set, and which\_chace flag.

\* Outputs: cache way, or -1 if we don't have a match

\*/

int matching\_tag(unsigned int tag, unsigned int set, char which\_cache) {

int i = 0;

switch (which\_cache) {

case ('d'):

case ('D'): // if flag is 'D' We're searching for a matching tag in the L1 data cache

while (L1\_data[i][set].tag\_bits != tag) { // search the data cache for matching tags

i++;

if (i > 7) { // We have 8 ways in the data cache 0 through 7

return -1;

// return -1 to imply that we don't have a matching tag in the data cache.

}

}

return i;

break;

case ('i'):

case ('I'): // if flag is 'I' We're searching for a matching tag in the L1 instruction cache

int i = 0;

// search the instruction cache for matching tags

while (L1\_inst[i][set].tag\_bits != tag) { i++;

if (i > 3) { // We have 4 ways in the instruction cache 0 through 3

return -1;

// return -1 to imply that we don't have a matching tag in the instruction cache.

}

}

}

return i;

// If we have a matching tag, return the way in which we have the matching tag. f

}

/\* Print cache content and state

\* in response to n = 9 in the trace file

\* This function doesn't take any inputs, and doesn't return any outputs

\* when called, it outputs the number of:

\* cache reads, writes, hits, misses, hit ratio for the data cache, and

\* cache reads/fetches, hits, misses, hit ratio for the instruction cache

\*/

void print\_stats() {

//Calculate the hit ratio for the data and the instruction caches

float data\_hit\_r = float(statistics.data\_hit) / (float(statistics.data\_miss) + float(statistics.data\_hit));

float inst\_hit\_r = float(statistics.inst\_hit) / (float(statistics.inst\_miss) + float(statistics.inst\_hit));

statistics.data\_hit\_ratio = data\_hit\_r;

statistics.inst\_hit\_ratio = inst\_hit\_r;

cout << "\n \t \*\* KEY CACHE USAGE STATISTICS \*\* " << endl;

if (statistics.data\_miss == 0) { // If we don't have any misses, then no operations took place!

cout << "no operations took place on the Data cache" << endl;

}

else {

cout << "\n\t -- DATA CACHE STATISTICS -- " << endl;

cout << " number of Cache Reads: \t" << dec << statistics.data\_read << endl;

cout << " number of Cache Writes: \t" << dec << statistics.data\_write << endl;

cout << " number of Cache Hits: \t \t" << dec << statistics.data\_hit << endl;

cout << " number of Cache Misses: \t" << dec << statistics.data\_miss << endl;

cout << " Cache Hit Ratio: \t\t" << dec << statistics.data\_hit\_ratio << endl;

cout << " Cache Hit Percentage: \t" << dec << (statistics.data\_hit\_ratio \* 100) << " %" << endl;

}

// If we don't have any misses, then no operations took place!

if (statistics.inst\_miss == 0) {

cout << "\n The cache instruction was not used/not operated on" << endl;

}

else {

cout << "\n\t -- INSTRUCTION CACHE STATISTICS -- " << endl;

cout << " number of Cache Reads: \t" << dec << statistics.inst\_read << endl;

cout << " number of Cache Hits: \t\t" << dec << statistics.inst\_hit << endl;

cout << " number of Cache Misses: \t" << dec << statistics.inst\_miss << endl;

cout << " Cache Hit Ratio: \t\t" << dec << statistics.inst\_hit\_ratio << endl;

cout << " Cache Hit Percentage: \t" << dec << (statistics.inst\_hit\_ratio) \* 100 << " %" << endl;

}

return;

}