ECE 485 Final Project Report

Design and Simulation of an L1 data and instruction caches

Team 2

Dmitrii Fotin, dfotin@pdx.edu
John Michael Mertz, jmertz@pdx.edu
Bulou Tian, bulou@pdx.edu
Josh Varughese, jv23@pdx.edu

Table of Contents

Introduction

This proposal outlines the design process and results of an 8-way set associative inclusive data cache and a 4-way set associative inclusive instruction cache. Each cache contains 2^{14} sets with one cache line per way, and implements a counter LRU and MESI cache coherence protocol.

The caches are to be simulated in Python with external trace files containing cache commands to execute.

Project Specifications and Requirements

The goal is to design and simulate a split L1 cache for a 32-bit processor in a shared memory configuration (with up to 3 other processors) that includes an 8-way set associative L1 data cache with 2^{14} sets, 1 cache line per way and 64-byte lines and a 4-way set associative L1 instruction cache with 2^{14} sets, 1 cache line per way and 64-byte lines. The system uses the MESI protocol for cache coherence and a counter LRU.
The system implements the write-through policy on the initial write command and a write-back policy for all consecutive write commands.
The outputs include a table displaying the tag, LRU and MESI information for all non-empty sets as well as a table of cache statistics that includes data on number of cache reads, writes, hits, misses and hit ratio per cache.

LRU Policy

The 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. We chose to structure the LRU so that the MRU bits would be 111 (data cache) and 11 (instruction cache), and our LRU bits would be 000 (data cache) and 00 (instruction cache). When a tag gets updated, the LRU bits in that way get changed to the MRU (111 or 11) while every other way that was initially greater than it gets decremented. An example of this can be seen below:

Index/Set	Way	Tag	LRU	MESI
0x0788	1	0xA01	000	E
0x0788	2	0x412	001	E
0x0788	3	0x846	010	E
0x0788	4	0x746	011	E
0x0788	5	0x939	100	E
0x0788	6	0x812	101	E
0x0788	7	0x009	110	E
0x0788	8	0x113	111	E

Index/Set	Way	Tag	LRU	MESI
0x0788	1	0xA01	000	E
0x0788	2	0x412	001	E
0x0788	3	0x846	111	S
0x0788	4	0x746	010	E
0x0788	5	0x939	011	E
0x0788	6	0x812	100	E
0x0788	7	0x009	101	E
0x0788	8	0x113	110	E

Images 1 and 2: Examples of simulated cache outputs that display the tag, LRU and MESI information of non-empty sets.

In the first image, one particular set in the data cache has been populated with 8 different ways. The next instruction performs a read to a tag that already exists in the cache. This updates the MESI bit for this way (i.e. way 3), and causes it to become the MRU. Way 3's LRU bits were 010 before the read instruction, and then got updated to 111. All LRU bits that were greater than 010 then got decremented by 1.

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 in following a bus transaction. The finite state machine of MESI state transitions is shown below.

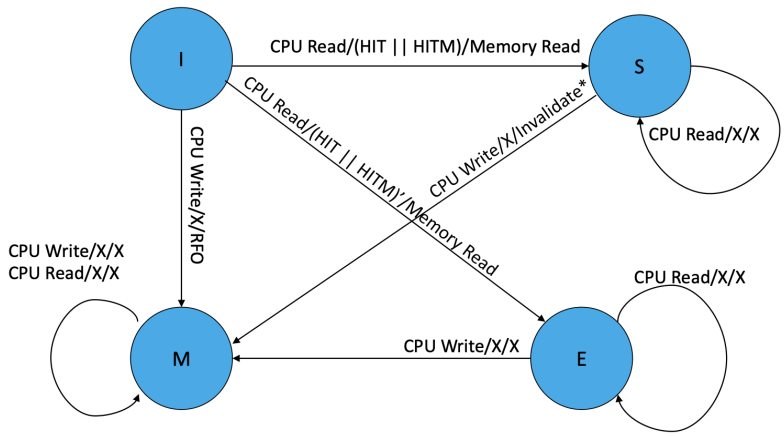


Image 3: MESI state transition diagram

In our program the MESI bits were represented by decimal numbers 0 through 3, where 0 represents invalid, 1 represents shared, 2 represents exclusive, and 3 represents modified. Any other number entry for the MESI bits would be an incorrect value.

Cache Structure

The cache is meant for a 32-bit processor which signifies how many bits need to be allocated for the entire address field. The index section identifies which set in the cache we are in. We can calculate the required bits for the index by knowing how many sets each cache has. Since we know it requires 16K sets, the required bits (x) for the index is simply $2^x = 16k$. $log_2(16k) = 14$, therefore 14 bits are needed to specify the index. We can also calculate the byte offset based on the caches 64-byte lines. Similarly to the index calculations, $2^x = 64$ can be solved as follows: $log_2(64) = 6$. We can specify the index with 14 bits and the byte offset with 6 bits.

Finding the required bits for the tag is done by taking the address field (32) and subtracting the 2 sections that we know (index and byte offset). The tag is then 32 - 14 - 6 = 12, and the address field can be seen in the table below:

Table 1. Address Field with specific lengths for tag, index, and offset

Tag	Index	Offset
12 Bits	14 Bits	6 Bits

The algorithm of the simulated cache is shown below.

```
Set up 3-dimensional arrays for caches and variables to store temporary data
3D array structure:
Number of sets by number of ways by 4
The four elements in each way contain a tag in binary, an LRU state in decimal, a MESI state in decimal and a full
address in hexadecimal.
Prompt for input of trace file name
Error checking for empty/non-existent file
Parse out the file contents line by line into respective variables:
         Command (0, 1, 2, 3, 4, 8, 9)
         Tag (convert to binary)
         Set index (convert to decimal)
Store full address in hex for reference when communicating with L2 cache
Enter switch case for command (0, 1, 2, 3, 4, 8, 9)
0 (Read data request to L1 data cache)
         Check if tag already in any way of data set at specified index
         YES
                  Set MESI to S (01)
                  Decrement LRU for all ways above
                  SET LRU for this way to 111
         NO
                  Check for empty ways
                  YES
                           Save tag to empty way in array
                  NO
                           Check for invalid sets
                           YES
                                    Evict LRU Invalid set
                           NO
                                    Evict LRU
                           Save tag to evicted set
                  Set MESI to E (10)
                  Decrement LRU for all ways above
```

```
                SET LRU for this way to 111

1 (Write data request to L1 data cache)
        Check if tag already in any way of data set at specified index
                YES
                        Check MESI bits
                        If invalid - RFO from L2
                        Set MESI to M (11)
                        Decrement LRU for all ways above
                        Set LRU for this way to 111
                NO
                        Check for empty ways
                        YES
                                Save tag to empty way in array
                        NO
                                Check for invalid sets
                                YES
                                        Evict LRU Invalid set
                                NO
                                        Pass latest data for specified set/tag to L2
                                        Evict LRU
                                Save tag to evicted set
                        Set MESI to M (11)
                        Decrement LRU for all ways above
                        Set LRU for this way to 111
                        Save tag to LRU way

2 (Read to L1 instruction cache)
        Check if tag already in any way of instruction set at specified index
        YES
                Set MESI to S (01)
                Decrement LRU for all ways above
                SET LRU for this way to 111
        NO
                Check for empty ways
                YES
                        Save tag to empty way in array
                NO
                        Check for invalid sets
                        YES
                                Evict LRU Invalid set
                        NO
                                Evict LRU
                        Save tag to evicted set
                Decrement LRU for all ways above
                SET LRU for this way to 11

3 (Invalidate command from L2)
        Set MESI to I (00) for specified set/tag
        Decrement LRU for all ways above
        Set LRU to 111

4 (Data request from L2 in response to snoop)
        If specified set/tag in modified state - pass latest data for specified set/tag to L2
        Set MESI to I (00) for specified set/tag
        Decrement LRU for all ways above
        Set LRU to 111

8 (Clear the cache and reset all state and statistics)
        Set MESI bits for all sets/ways to I (00)

9 (Print contents and state of the cache)
        For each non-empty set print out tag, LRU and MESI contents of each way

If not EOF
        Go to parse next line
Else
```

```
        Request input for next trace file name
```

Design Assumptions

	M	E	S	I
Bin	11	10	01	00
Dec	3	2	1	0

LRU: 000
MRU: 111

LRU→ MRU								
Bin	000	001	010	011	100	101	110	111
Dec	0	1	2	3	4	5	6	7

* LRU = 111 in Code means it's our Most Recently Used.

MESI decimal (1 decimal digit)
LRU decimal (1 decimal digit)
Tag Hexadecimal (8 string characters)

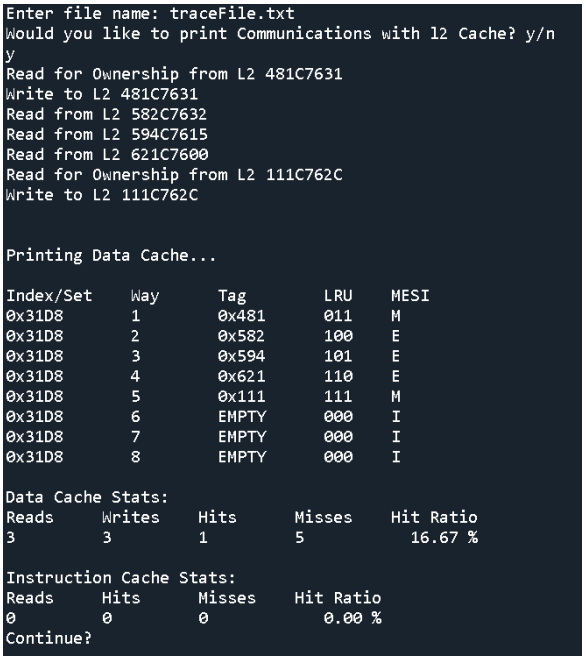
The team used decimal values to track set index, way, MESI and LRU within the code but converted them to expected formats for output tables.
Tag was processed as a string of binary digits.

Test Plan

The team used the set of commands provided in the course materials as well as developed two sets of their own test cases to exhaustively test the outputs of the simulated cache.
Test results can be found in a separate file **Test_results.pdf** provided along with this report.

Demo

During the demo with the TA, the team obtained correct outputs for the test trace files provided by the TA. The screenshots of the outputs can be found below.



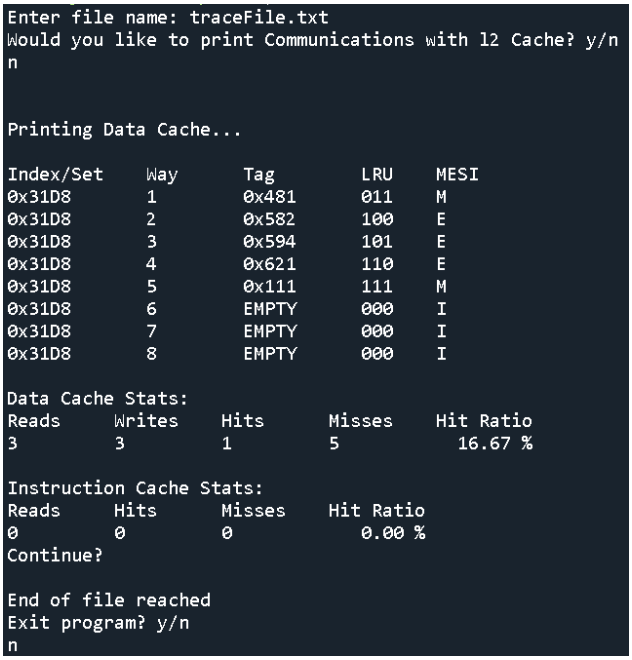
```
Enter file name: traceFile.txt
Would you like to print Communications with l2 Cache? y/n
y
Read for Ownership from L2 481C7631
Write to L2 481C7631
Read from L2 582C7632
Read from L2 594C7615
Read from L2 621C7600
Read for Ownership from L2 111C762C
Write to L2 111C762C


Printing Data Cache...

Index/Set    Way      Tag        LRU    MESI
0x31D8       1        0x481      011    M
0x31D8       2        0x582      100    E
0x31D8       3        0x594      101    E
0x31D8       4        0x621      110    E
0x31D8       5        0x111      111    M
0x31D8       6        EMPTY      000    I
0x31D8       7        EMPTY      000    I
0x31D8       8        EMPTY      000    I

Data Cache Stats:
Reads     Writes    Hits      Misses    Hit Ratio
3         3         1         5           16.67 %

Instruction Cache Stats:
Reads     Hits      Misses    Hit Ratio
0         0         0            0.00 %
Continue?
```



```
Enter file name: traceFile.txt
Would you like to print Communications with l2 Cache? y/n
n


Printing Data Cache...

Index/Set    Way      Tag        LRU    MESI
0x31D8       1        0x481      011    M
0x31D8       2        0x582      100    E
0x31D8       3        0x594      101    E
0x31D8       4        0x621      110    E
0x31D8       5        0x111      111    M
0x31D8       6        EMPTY      000    I
0x31D8       7        EMPTY      000    I
0x31D8       8        EMPTY      000    I

Data Cache Stats:
Reads     Writes    Hits      Misses    Hit Ratio
3         3         1         5           16.67 %

Instruction Cache Stats:
Reads     Hits      Misses    Hit Ratio
0         0         0            0.00 %
Continue?

End of file reached
Exit program? y/n
n
```

Images 4 and 5: Outputs with and without L2 communications for test trace file during the demo

Member Contribution

Dmitrii: Code for file input, input parsing, commands 1 and 4
John Michael: Code for output prints, commands 0 and 2
Bulou: Code for commands 3 and 8, creation of test cases, testing
Josh: Code for command 9, creation of test cases, testing

Conclusion

The simulated cache provided the expected outputs as tested exhaustively against the cases developed by the team. Team members gained valuable experience in designing actual caches with modern applications and protocols such as write-back vs. write-through, inclusive cache architecture, counter LRU and MESI.