Final Project Report

The goal is to design and simulate a ***split L1 cache*** for a 32-bit processor which is used with up to three other processors in a shared memory configuration. The system employs a ***MESI protocol*** to ensure cache coherence. The L1 cache is split into L1 instruction cache and L1 data cache. The L1 data cache is write-back using write allocate (except for the first write to a line which is a write-through). L1 data cache and L1 instruction cache both employ the ***LRU replacement policy*** and both the L1 data and instruction caches are backed by a shared L2 cache. The cache hierarchy employs inclusivity.

L1 cache: The L1 cache is also called the primary cache. It is used for temporary storage of instructions and data.

L1 instruction cache: The L1 instruction cache is two-way set associative and consists of 16K sets of 64-byte lines.
Number of lines in L1 instruction cache = 16K x 2 = 32K lines
L1 instruction cache size = 2MB

Address format:

Tag (12 bits)	Index (14 bits)	Byte-Offset (6 bits)

L1 data cache: The L1 data cache is a four-way set associative and consists of 16K sets of 64-byte lines.
Number of lines in L1 data cache = 16K x 4 = 64K lines
L1 data cache size = 4MB

Address format:

Tag (12 bits)	Index (14 bits)	Byte-Offset (6 bits)

Therefore, the total size of the cache is 6MB.

L2 cache: The L2 cache is also known as secondary cache.

LRU: The Least Recently Used replacement policy is used to replace the cache line that is least recently used.

MESI: The MESI protocol is a cache coherence protocol. It supports the write-back cache. The MESI protocol has four states, Modified, Exclusive, Shared and Invalid.

1. Modified - cache line has been modified, is different from main memory - is the only cached copy. (multiprocessor ‘dirty’)
2. Exclusive - cache line is the same as main memory and is the only cached copy
3. Shared - Same as main memory but copies may exist in other caches.
4. Invalid - Line data is not valid

Summary of the Test cases:

Operation – 0: Read Operation is verified for the following cases:
a. Read when Cache is empty.
b. Read to the least recently used line.
c. Read to the random cache line.
d. Read to a most recently used line.
e. Read to evict the least recently used line .
f. Read to a different cache line.

Operation- 1: Write operation is verified for the following test cases.
a. Write when the cache is empty.
b. Write to the cache line when it is in Exclusive state.
c. Write when the cache line is in modified state.
    i. If the cache line is not LRU then next state if modified state.
    ii. If the cache line is LRU then next state is exclusive state.

Operation- 2: Read from instruction cache: is verified for the following cases:

a. Cache hit.
b. Cache miss when the line is modified.
c. Cache miss when the line is not modified.

Operation – 3: Invalidate Command from L2 cache.

a. Invalidating the cache line with modified state.
b. Invalidating the cache line with exclusive state.

Operation – 4: Data request from L2 cache.

a. Data request from L2 cache when cache line for L1 cache is modified.
b. Data request from L2 cache when the cache line for L1 cache is in the shared state.
c. Data request from L2 cache when the cache line for L1 cache is in the exclusive state.

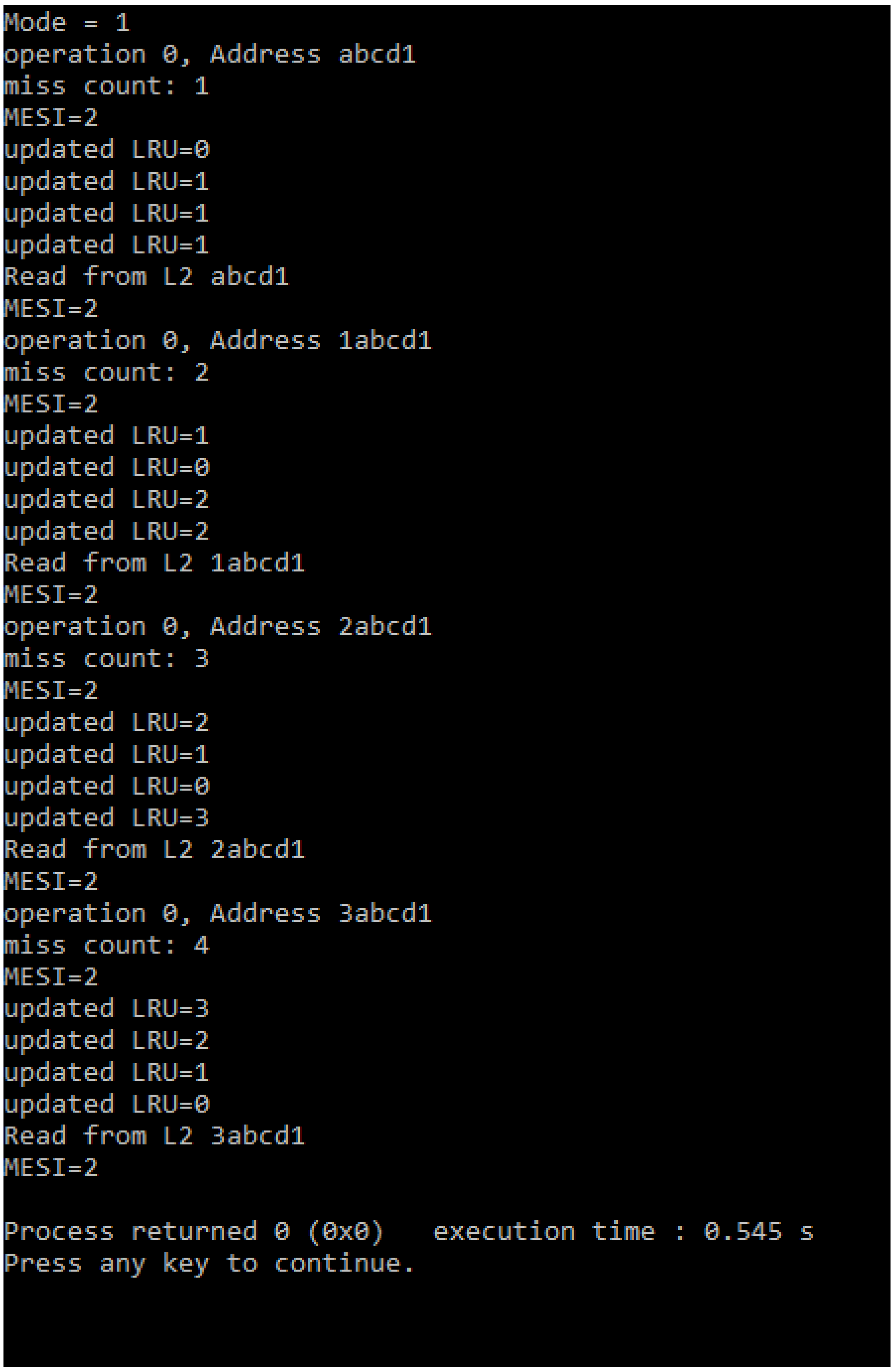
Operation – 8: Clear cache and reset all state.

Operation – 9: Print contents and state of the cache.

Test case 1 – Operation 0: Read data request to L1 cache
Filling an empty data cache
Observation: The LRU bits are updated
Expected output:

TAG	INDEX	LRU	MESI
000000000000	abcd1	3	SHARED
000000000001	1abcd1	2	SHARED
000000000010	2abcd1	1	SHARED
000000000011	3abcd1	0	SHARED

Output:


```
Mode = 1
operation 0, Address abcd1
miss count: 1
MESI=2
updated LRU=0
updated LRU=1
updated LRU=1
updated LRU=1
Read from L2 abcd1
MESI=2
operation 0, Address 1abcd1
miss count: 2
MESI=2
updated LRU=1
updated LRU=0
updated LRU=2
updated LRU=2
Read from L2 1abcd1
MESI=2
operation 0, Address 2abcd1
miss count: 3
MESI=2
updated LRU=2
updated LRU=1
updated LRU=0
updated LRU=3
Read from L2 2abcd1
MESI=2
operation 0, Address 3abcd1
miss count: 4
MESI=2
updated LRU=3
updated LRU=2
updated LRU=1
updated LRU=0
Read from L2 3abcd1
MESI=2

Process returned 0 (0x0)   execution time : 0.545 s
Press any key to continue.
```

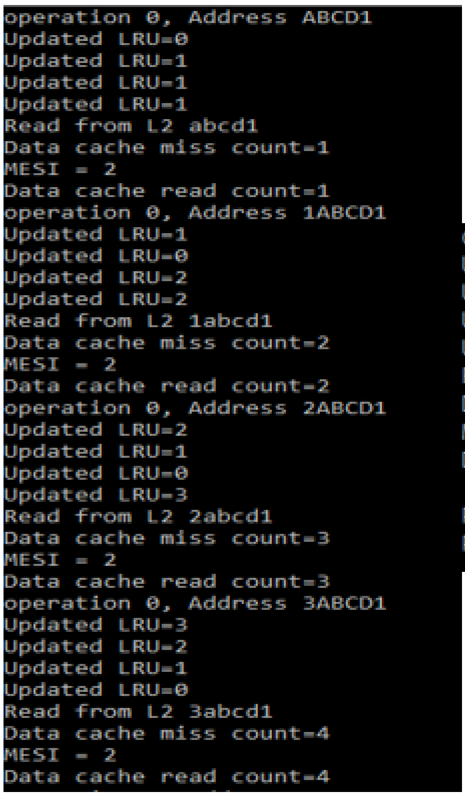
Miss count = 4
Hit count = 0

Test case 2 – Operation 0: – Read from the data cache.
Reading the Least recently used line.
<u>Observation: </u>The LRU bits are updated.
<u>Expected output:</u>

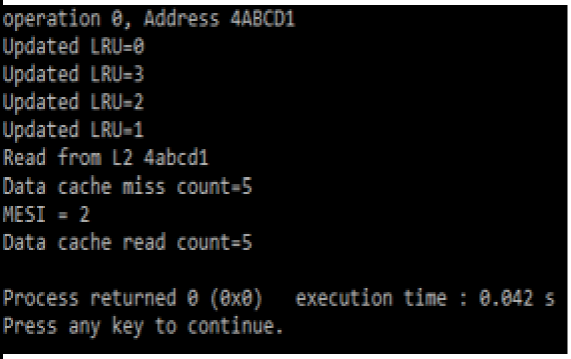
TAG	INDEX	LRU	MESI

000000000100	4abcd1	0	SHARED
000000000001	1abcd1	3	SHARED
000000000010	2abcd1	2	SHARED
000000000011	3abcd1	1	SHARED

Output:



```
operation 0, Address ABCD1
Updated LRU=0
Updated LRU=1
Updated LRU=1
Updated LRU=1
Read from L2 abcd1
Data cache miss count=1
MESI = 2
Data cache read count=1
operation 0, Address 1ABCD1
Updated LRU=1
Updated LRU=0
Updated LRU=2
Updated LRU=2
Read from L2 1abcd1
Data cache miss count=2
MESI = 2
Data cache read count=2
operation 0, Address 2ABCD1
Updated LRU=2
Updated LRU=1
Updated LRU=0
Updated LRU=3
Read from L2 2abcd1
Data cache miss count=3
MESI = 2
Data cache read count=3
operation 0, Address 3ABCD1
Updated LRU=3
Updated LRU=2
Updated LRU=1
Updated LRU=0
Read from L2 3abcd1
Data cache miss count=4
MESI = 2
Data cache read count=4
```



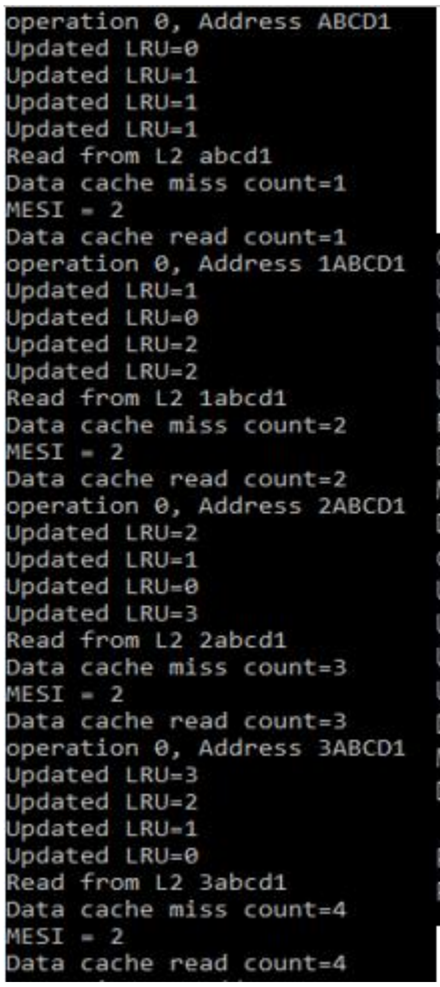
```
operation 0, Address 4ABCD1
Updated LRU=0
Updated LRU=3
Updated LRU=2
Updated LRU=1
Read from L2 4abcd1
Data cache miss count=5
MESI = 2
Data cache read count=5

Process returned 0 (0x0)   execution time : 0.042 s
Press any key to continue.
```

Miss count = 5

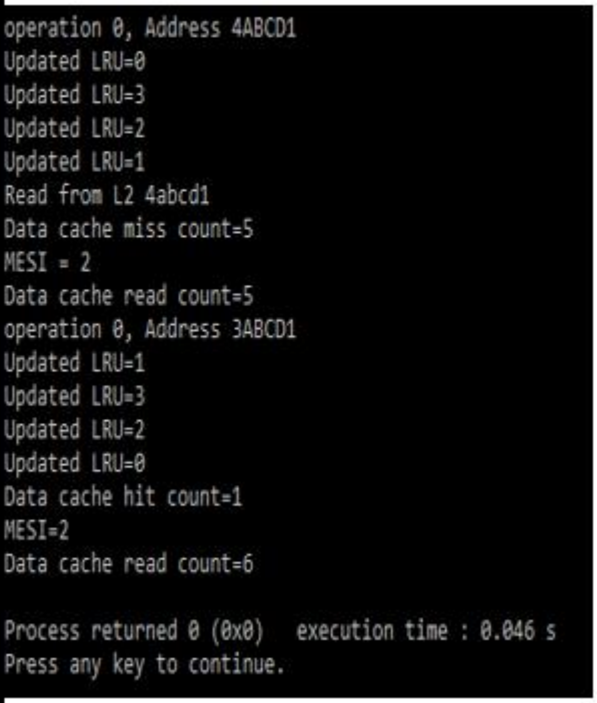
Test case 3– Operation 0: – Read from the data cache.
Accessing a cache line with random LRU
<u>Expected output:</u>

TAG	INDEX	LRU	MESI
000000000100	abcd1	1	SHARED
000000000001	abcd1	3	SHARED
000000000010	abcd1	2	SHARED
000000000011	abcd1	0	SHARED

Output:



```
operation 0, Address ABCD1
Updated LRU=0
Updated LRU=1
Updated LRU=1
Updated LRU=1
Read from L2 abcd1
Data cache miss count=1
MESI = 2
Data cache read count=1
operation 0, Address 1ABCD1
Updated LRU=1
Updated LRU=0
Updated LRU=2
Updated LRU=2
Read from L2 1abcd1
Data cache miss count=2
MESI = 2
Data cache read count=2
operation 0, Address 2ABCD1
Updated LRU=2
Updated LRU=1
Updated LRU=0
Updated LRU=3
Read from L2 2abcd1
Data cache miss count=3
MESI = 2
Data cache read count=3
operation 0, Address 3ABCD1
Updated LRU=3
Updated LRU=2
Updated LRU=1
Updated LRU=0
Read from L2 3abcd1
Data cache miss count=4
MESI = 2
Data cache read count=4
```



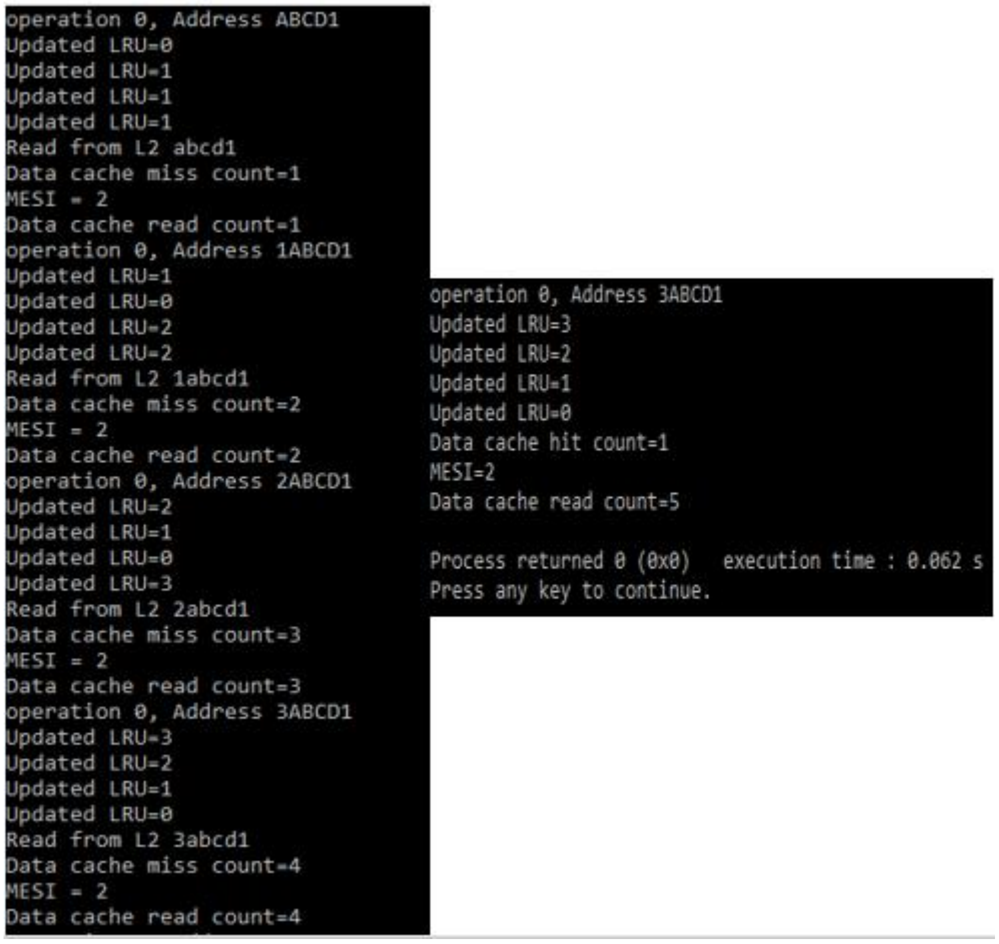
```
operation 0, Address 4ABCD1
Updated LRU=0
Updated LRU=3
Updated LRU=2
Updated LRU=1
Read from L2 4abcd1
Data cache miss count=5
MESI = 2
Data cache read count=5
operation 0, Address 3ABCD1
Updated LRU=1
Updated LRU=3
Updated LRU=2
Updated LRU=0
Data cache hit count=1
MESI=2
Data cache read count=6

Process returned 0 (0x0)   execution time : 0.046 s
Press any key to continue.
```

Miss count: 5
Hit count: 1

Test case 4– Operation 0: – Read from the data cache: Accessing the most recently used line

TAG	INDEX	LRU	MESI
000000000000	abcd1	3	SHARED
000000000001	abcd1	2	SHARED
000000000011	abcd1	1	SHARED
000000000011	abcd1	0	SHARED

Output:



```
operation 0, Address ABCD1
Updated LRU=0
Updated LRU=1
Updated LRU=1
Updated LRU=1
Read from L2 abcd1
Data cache miss count=1
MESI = 2
Data cache read count=1
operation 0, Address 1ABCD1
Updated LRU=1
Updated LRU=0
Updated LRU=2
Updated LRU=2
Read from L2 1abcd1
Data cache miss count=2
MESI = 2
Data cache read count=2
operation 0, Address 2ABCD1
Updated LRU=2
Updated LRU=1
Updated LRU=0
Updated LRU=3
Read from L2 2abcd1
Data cache miss count=3
MESI = 2
Data cache read count=3
operation 0, Address 3ABCD1
Updated LRU=3
Updated LRU=2
Updated LRU=1
Updated LRU=0
Read from L2 3abcd1
Data cache miss count=4
MESI = 2
Data cache read count=4
```

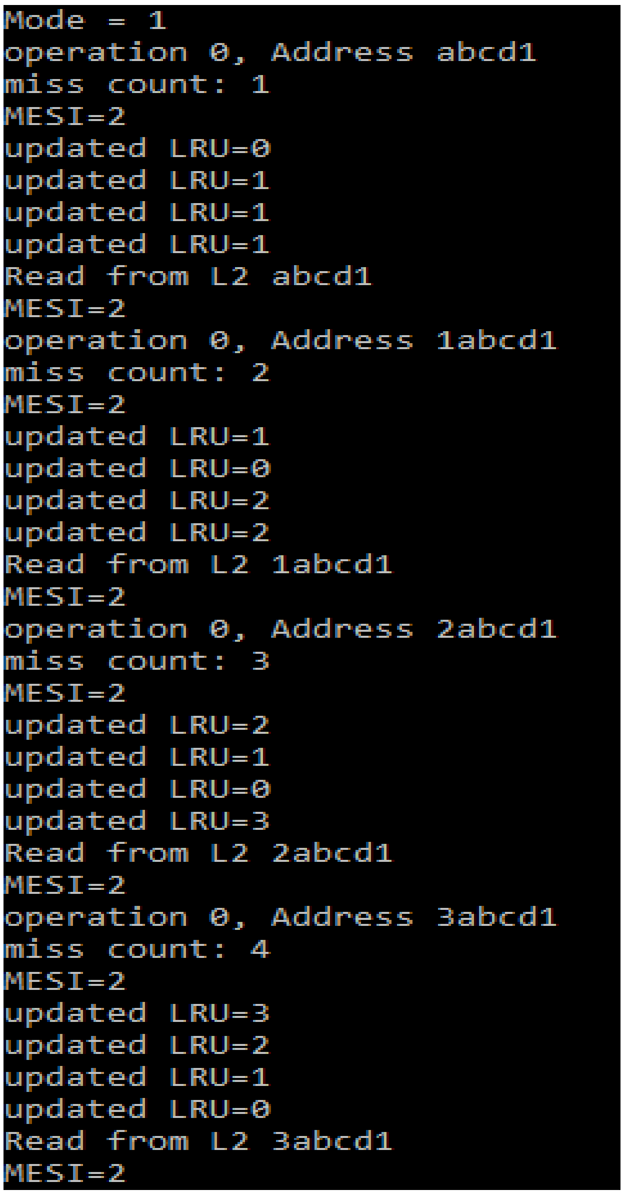
```
operation 0, Address 3ABCD1
Updated LRU=3
Updated LRU=2
Updated LRU=1
Updated LRU=0
Data cache hit count=1
MESI=2
Data cache read count=5

Process returned 0 (0x0)   execution time : 0.062 s
Press any key to continue.
```

Miss count: 4
Hit count: 1

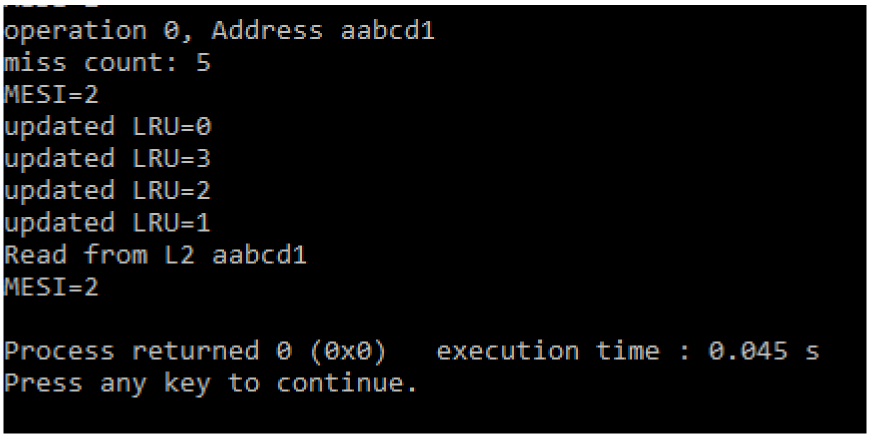
Test case 5– Operation 0: – Read from the data cache:
Evicting the LRU cache line and update the LRU
Observation: The least recently used cache line is replaced
Expected output:

TAG	INDEX	LRU	MESI
000000001010	abcd1	0	SHARED
000000000010	2abcd1	3	SHARED
000000000011	3abcd1	2	SHARED
000000000010	2abcd1	1	SHARED

Output:



```
Mode = 1
operation 0, Address abcd1
miss count: 1
MESI=2
updated LRU=0
updated LRU=1
updated LRU=1
updated LRU=1
Read from L2 abcd1
MESI=2
operation 0, Address 1abcd1
miss count: 2
MESI=2
updated LRU=1
updated LRU=0
updated LRU=2
updated LRU=2
Read from L2 1abcd1
MESI=2
operation 0, Address 2abcd1
miss count: 3
MESI=2
updated LRU=2
updated LRU=1
updated LRU=0
updated LRU=3
Read from L2 2abcd1
MESI=2
operation 0, Address 3abcd1
miss count: 4
MESI=2
updated LRU=3
updated LRU=2
updated LRU=1
updated LRU=0
Read from L2 3abcd1
MESI=2
```



```
operation 0, Address aabcd1
miss count: 5
MESI=2
updated LRU=0
updated LRU=3
updated LRU=2
updated LRU=1
Read from L2 aabcd1
MESI=2

Process returned 0 (0x0)   execution time : 0.045 s
Press any key to continue.
```

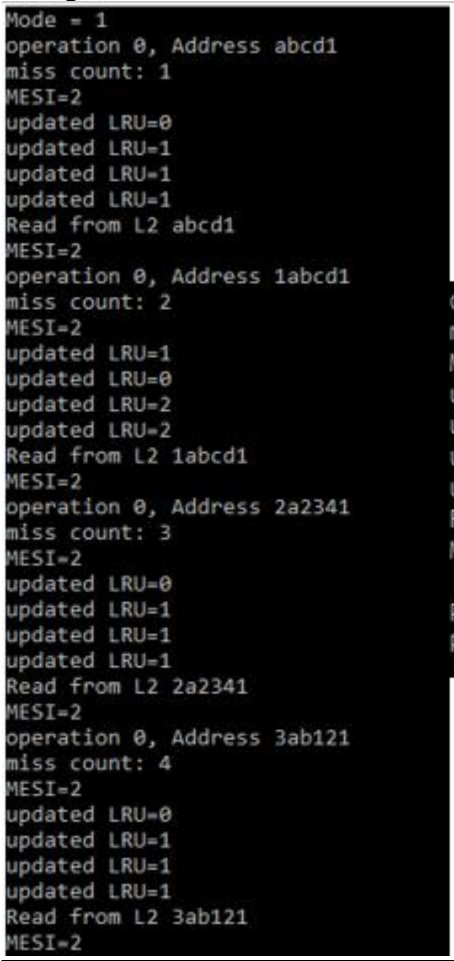
Miss count = 5

Test case 6– Operation 0: – Read from the data cache

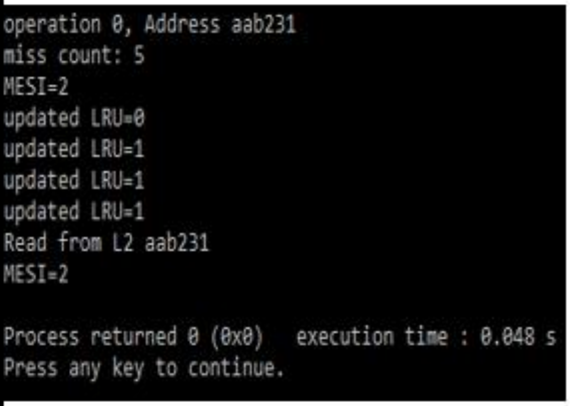
Read to a different cache set.
Observation: LRU bits are updated for each cache line
Since, the cache set is different. The LRU for each set is shown below.
Expected output:

TAG	INDEX	LRU	MESI
000000000000	ABCD1	0111	SHARED
000000000001	ABCD1	1022	SHARED
000000000010	A2341	1011	SHARED
000000000011	AB121	0111	SHARED
000000001010	AB231	0111	SHARED

Output:



```
Mode = 1
operation 0, Address abcd1
miss count: 1
MESI=2
updated LRU=0
updated LRU=1
updated LRU=1
updated LRU=1
Read from L2 abcd1
MESI=2
operation 0, Address 1abcd1
miss count: 2
MESI=2
updated LRU=1
updated LRU=0
updated LRU=2
updated LRU=2
Read from L2 1abcd1
MESI=2
operation 0, Address 2a2341
miss count: 3
MESI=2
updated LRU=0
updated LRU=1
updated LRU=1
updated LRU=1
Read from L2 2a2341
MESI=2
operation 0, Address 3ab121
miss count: 4
MESI=2
updated LRU=0
updated LRU=1
updated LRU=1
updated LRU=1
Read from L2 3ab121
MESI=2
```



```
operation 0, Address aab231
miss count: 5
MESI=2
updated LRU=0
updated LRU=1
updated LRU=1
updated LRU=1
Read from L2 aab231
MESI=2

Process returned 0 (0x0)   execution time : 0.048 s
Press any key to continue.
```

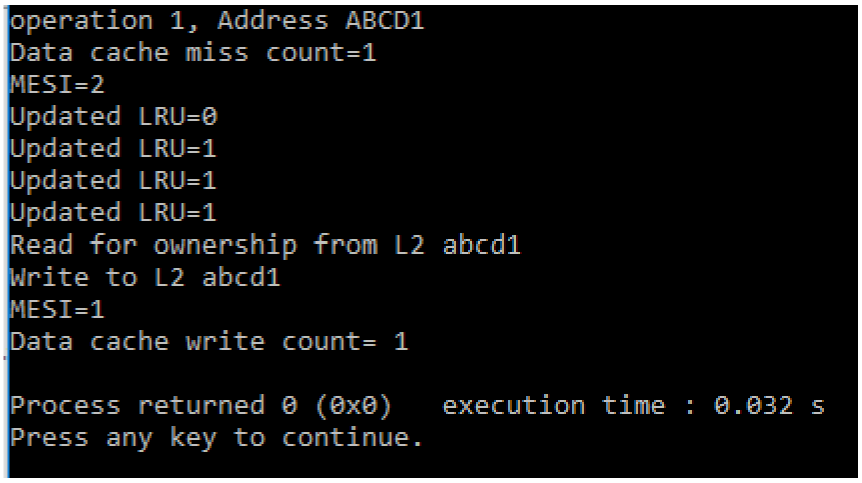
Miss count = 5

Test case 7– Operation 1: - Write when the data cache is empty
Observation: When the cache is empty the initial state is in shared state.

1. The first write is always miss and after first write the cache line state changes from shared to exclusive state.
2. The LRU bits have changed.
3. The write counter is incremented
4. Also, the miss counts are updated.

Expected Output:
Address:ABCD1
State changes from shared to exclusive state.

OUTPUT:



```
operation 1, Address ABCD1
Data cache miss count=1
MESI=2
Updated LRU=0
Updated LRU=1
Updated LRU=1
Updated LRU=1
Read for ownership from L2 abcd1
Write to L2 abcd1
MESI=1
Data cache write count= 1

Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

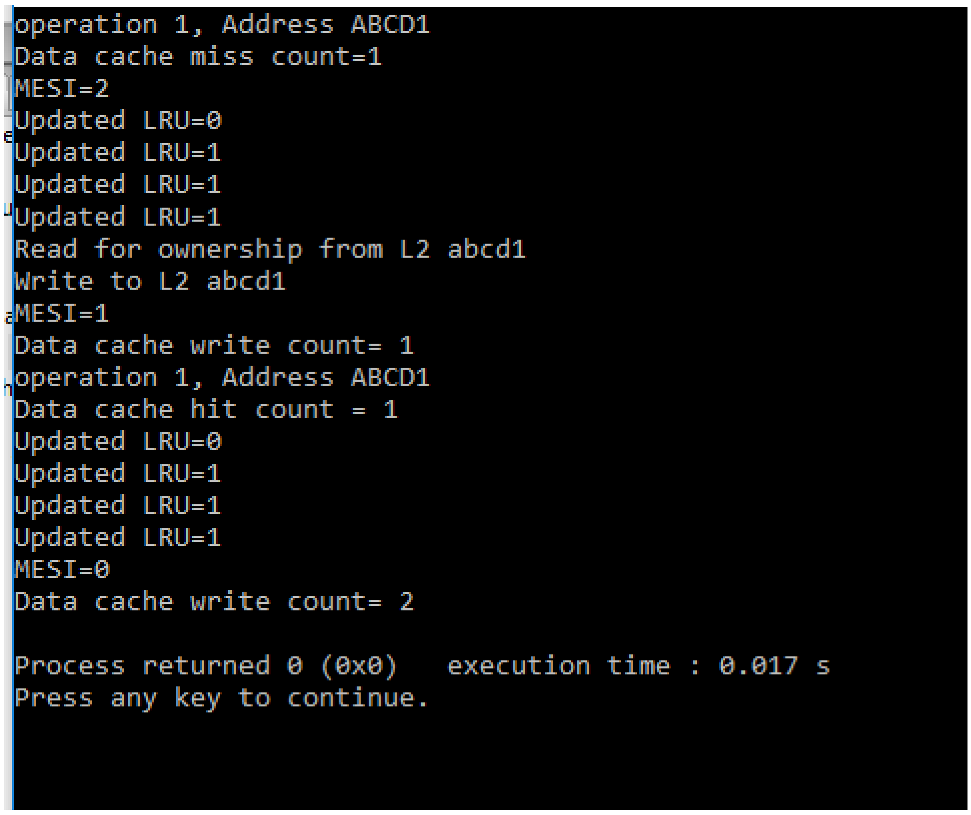
Cache miss = 1

Test case 8– Operation 1: - Write to the cache line when it is in Exclusive state.
Observations:

1. When the cache is in exclusive state and write operation is performed on the same cache line, the cache line state changes from exclusive state to modified state.
2. Since the initial state, in this case is exclusive state, the hit counter is updated.
3. The LRU bits are updated.

<u>Expected output:</u> For Address: ABCD1 the data cache changes from Shared MESI state (2) to Exclusive MESI state (1).

<u>OUTPUT:</u>



```
operation 1, Address ABCD1
Data cache miss count=1
MESI=2
Updated LRU=0
Updated LRU=1
Updated LRU=1
Updated LRU=1
Read for ownership from L2 abcd1
Write to L2 abcd1
MESI=1
Data cache write count= 1
operation 1, Address ABCD1
Data cache hit count = 1
Updated LRU=0
Updated LRU=1
Updated LRU=1
Updated LRU=1
MESI=0
Data cache write count= 2

Process returned 0 (0x0)   execution time : 0.017 s
Press any key to continue.
```

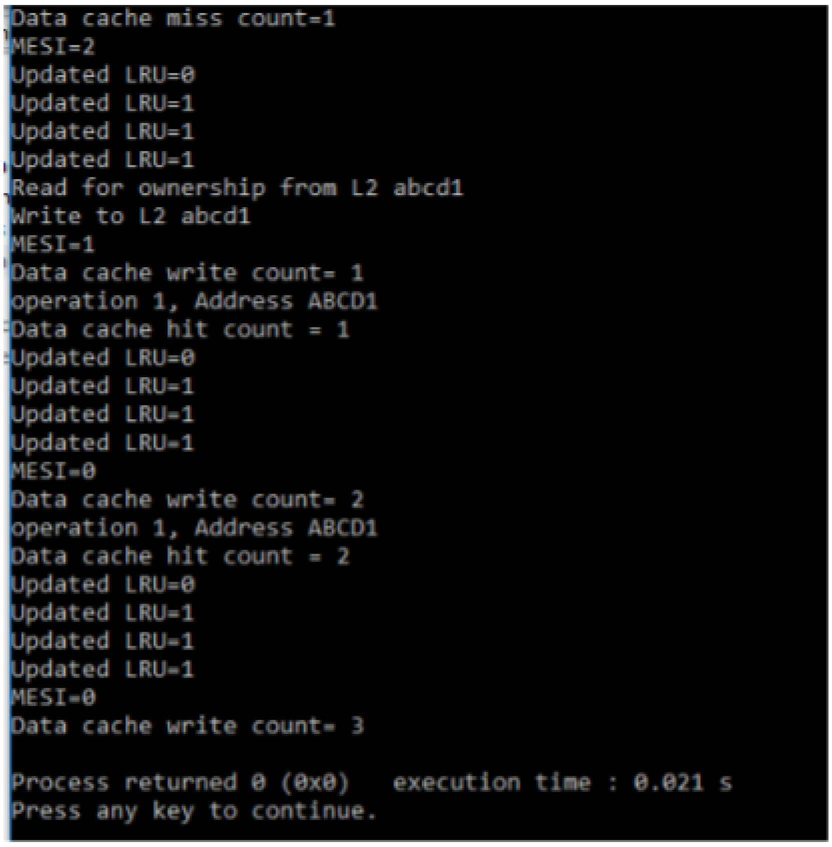
Cache hits = 1
Cache Miss = 1

T<u>est case 9– Operation 1</u>: Write when the cache line Is in modified state.
i. If the cache line is not LRU then next state if modified state.
<u>Observations:</u>

1. When the cache is in modified state and if the cache line is not LRU then the next state is modified state.

<u>Expected output:</u>
Address: ABCD1
Initial state: Shared → Exclusive → Modified → Modified.

<u>Output:</u>



```
Data cache miss count=1
MESI=2
Updated LRU=0
Updated LRU=1
Updated LRU=1
Updated LRU=1
Read for ownership from L2 abcd1
Write to L2 abcd1
MESI=1
Data cache write count= 1
operation 1, Address ABCD1
Data cache hit count = 1
Updated LRU=0
Updated LRU=1
Updated LRU=1
Updated LRU=1
MESI=0
Data cache write count= 2
operation 1, Address ABCD1
Data cache hit count = 2
Updated LRU=0
Updated LRU=1
Updated LRU=1
Updated LRU=1
MESI=0
Data cache write count= 3

Process returned 0 (0x0)   execution time : 0.021 s
Press any key to continue.
```

Hit count= 2
Miss count= 1

Test case 10- Operation 2: Instruction fetch (a read request to L1 instruction cache).
Observations:

1. When a read request for from CPU occurs and if it not there, then the read request at L2 occurs, it is in shared state. The miss count and the instruction read count is 1 now.
2. Again, when a read from L2 for 1ABCD1 occurs. Therefore, the miss count and read count will be 2.

3. When a read for ABCD1 occurs again. Since, this address is already available, it’s a hit and the hit count is 1 and the instruction read count will now be 3.
4. When a read for 2ABCD1 occurs, it is in shared state. The miss count is 3 and the instruction read count is 4.

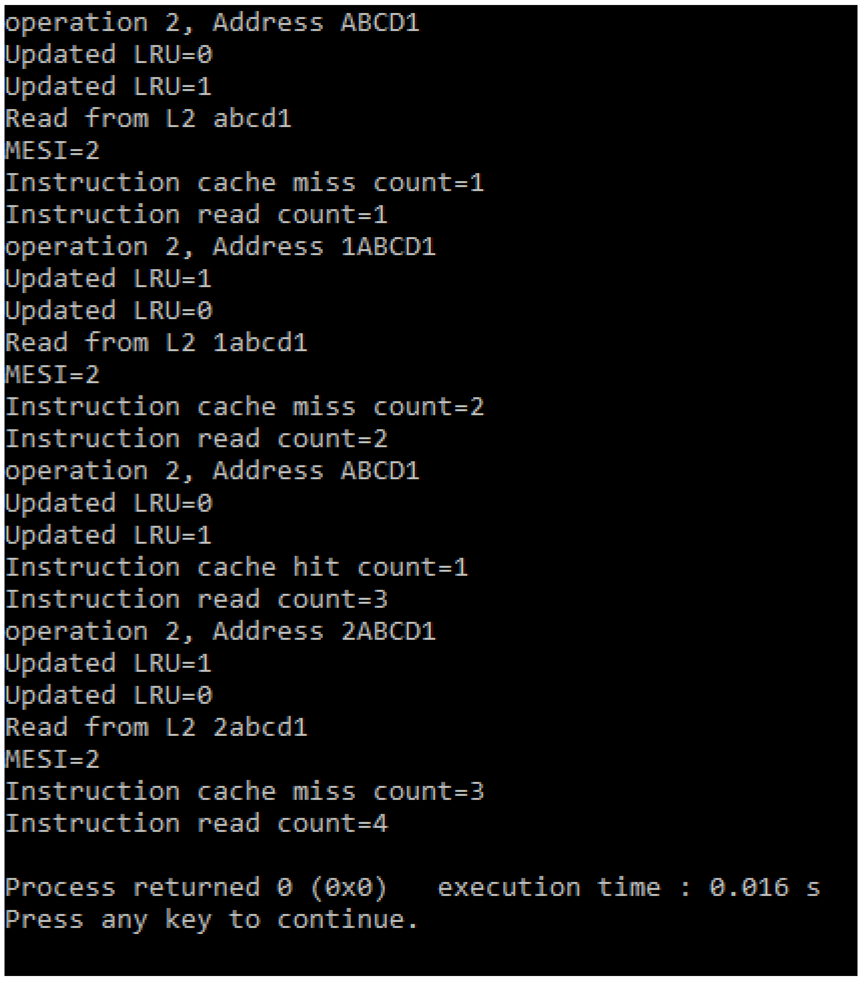
Expected output:

Address: 1ABCD1 → Miss

Address:ABCD1 → Hit

Address:2ABCD1 → Miss

Output:



```
operation 2, Address ABCD1
Updated LRU=0
Updated LRU=1
Read from L2 abcd1
MESI=2
Instruction cache miss count=1
Instruction read count=1
operation 2, Address 1ABCD1
Updated LRU=1
Updated LRU=0
Read from L2 1abcd1
MESI=2
Instruction cache miss count=2
Instruction read count=2
operation 2, Address ABCD1
Updated LRU=0
Updated LRU=1
Instruction cache hit count=1
Instruction read count=3
operation 2, Address 2ABCD1
Updated LRU=1
Updated LRU=0
Read from L2 2abcd1
MESI=2
Instruction cache miss count=3
Instruction read count=4

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

Instruction miss count = 3

Instruction read count = 4

Test case 11- Operation 3: Invalidate command from L2

Conditions:

1. Invalidating the data cache line
2. Invalidating the instruction cache line

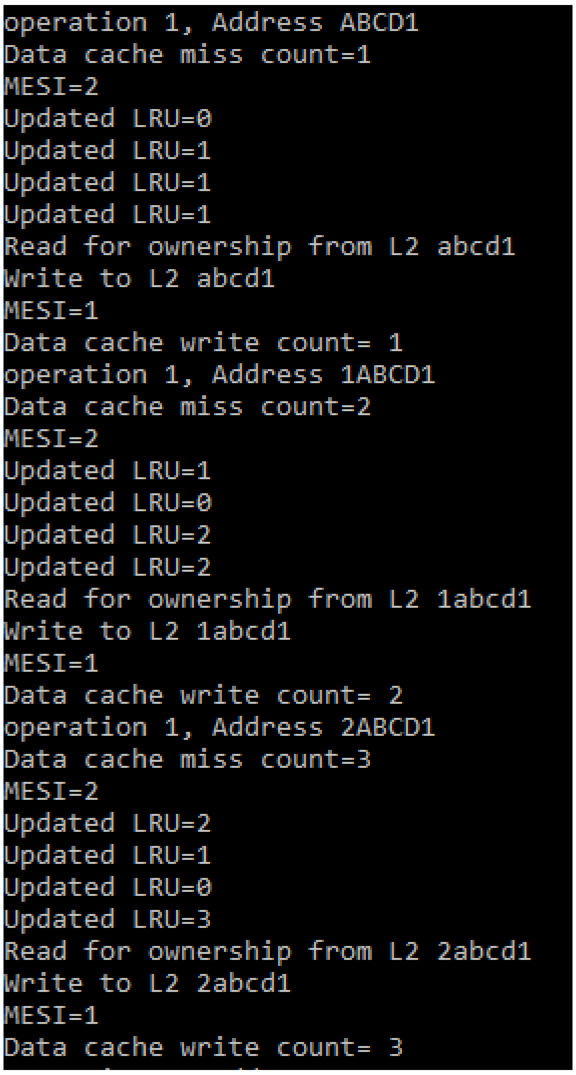
Observations for invalidating the data cache line:

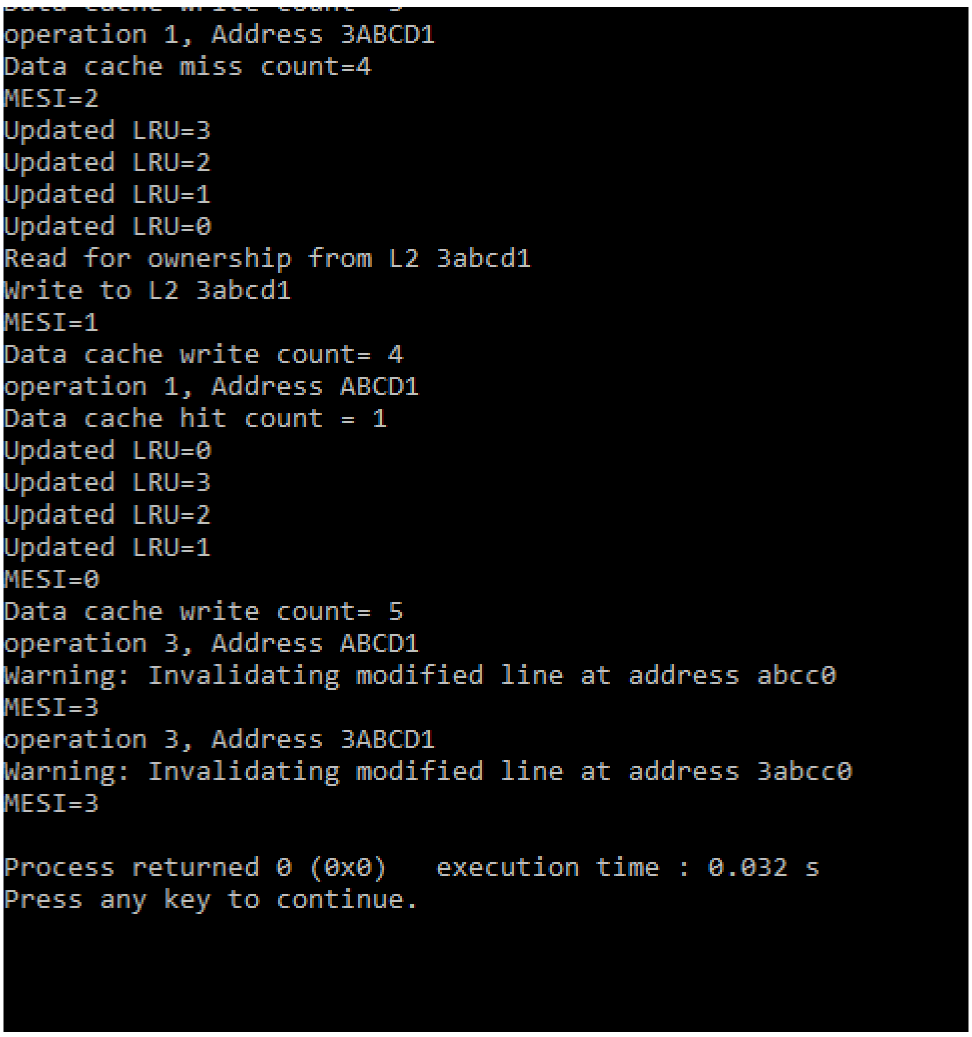
1. The cache line with address ABCD1 moves from shared state to exclusive and from exclusive to modified state by subsequent writes.
2. This modified line is then invalidated and the state is changed from modified state to invalidate state.
3. The cache line with address 3ABCD1 moves from shared state to exclusive state by performing write operation.
4. This line is then invalidated and the state changes from exclusive to invalidate state.
5. The LRU bits are also changed for the subsequent operations.

Expected output:

Address: ABCD1 → Shared Mesi state → Exclusive Mesi state → Modified Mesi state-->Invalidate Mesi state.
Address: 3ABCD1 → Shared Mesi state → Exclusive Mesi state → Invalidate Mesi state.

Output:


```
operation 1, Address ABCD1
Data cache miss count=1
MESI=2
Updated LRU=0
Updated LRU=1
Updated LRU=1
Updated LRU=1
Read for ownership from L2 abcd1
Write to L2 abcd1
MESI=1
Data cache write count= 1
operation 1, Address 1ABCD1
Data cache miss count=2
MESI=2
Updated LRU=1
Updated LRU=0
Updated LRU=2
Updated LRU=2
Read for ownership from L2 1abcd1
Write to L2 1abcd1
MESI=1
Data cache write count= 2
operation 1, Address 2ABCD1
Data cache miss count=3
MESI=2
Updated LRU=2
Updated LRU=1
Updated LRU=0
Updated LRU=3
Read for ownership from L2 2abcd1
Write to L2 2abcd1
MESI=1
Data cache write count= 3
```

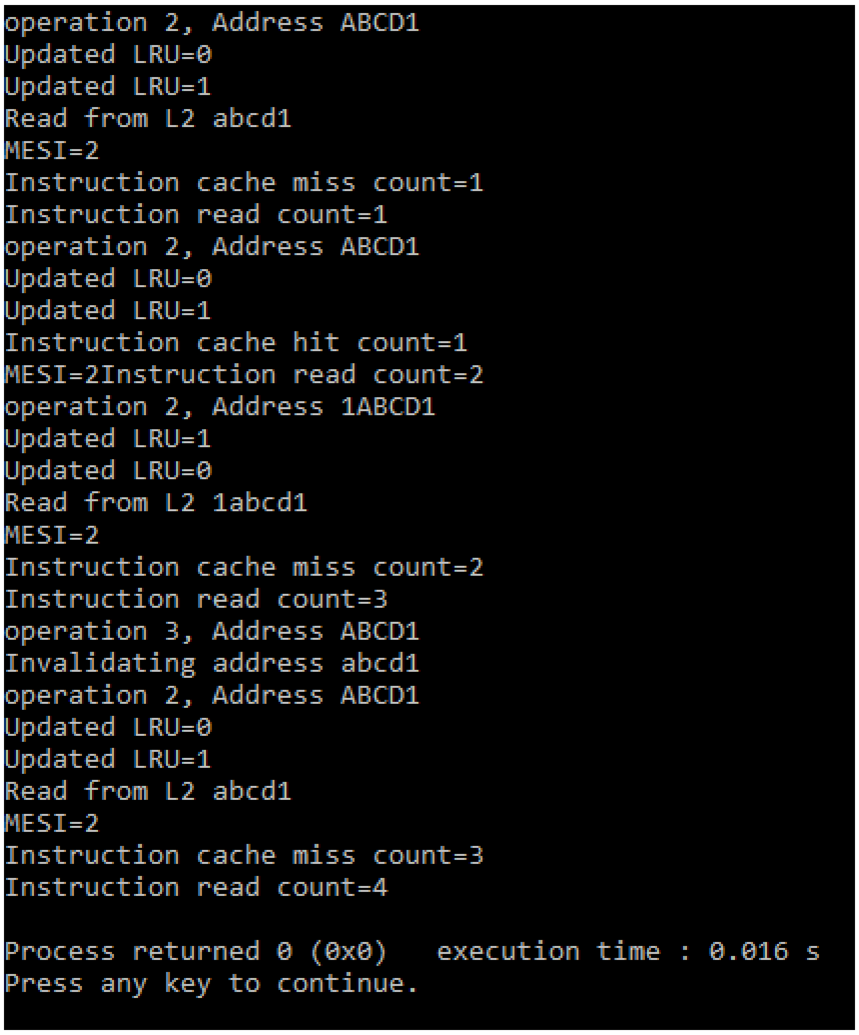


```
operation 1, Address 3ABCD1
Data cache miss count=4
MESI=2
Updated LRU=3
Updated LRU=2
Updated LRU=1
Updated LRU=0
Read for ownership from L2 3abcd1
Write to L2 3abcd1
MESI=1
Data cache write count= 4
operation 1, Address ABCD1
Data cache hit count = 1
Updated LRU=0
Updated LRU=3
Updated LRU=2
Updated LRU=1
MESI=0
Data cache write count= 5
operation 3, Address ABCD1
Warning: Invalidating modified line at address abcc0
MESI=3
operation 3, Address 3ABCD1
Warning: Invalidating modified line at address 3abcc0
MESI=3

Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

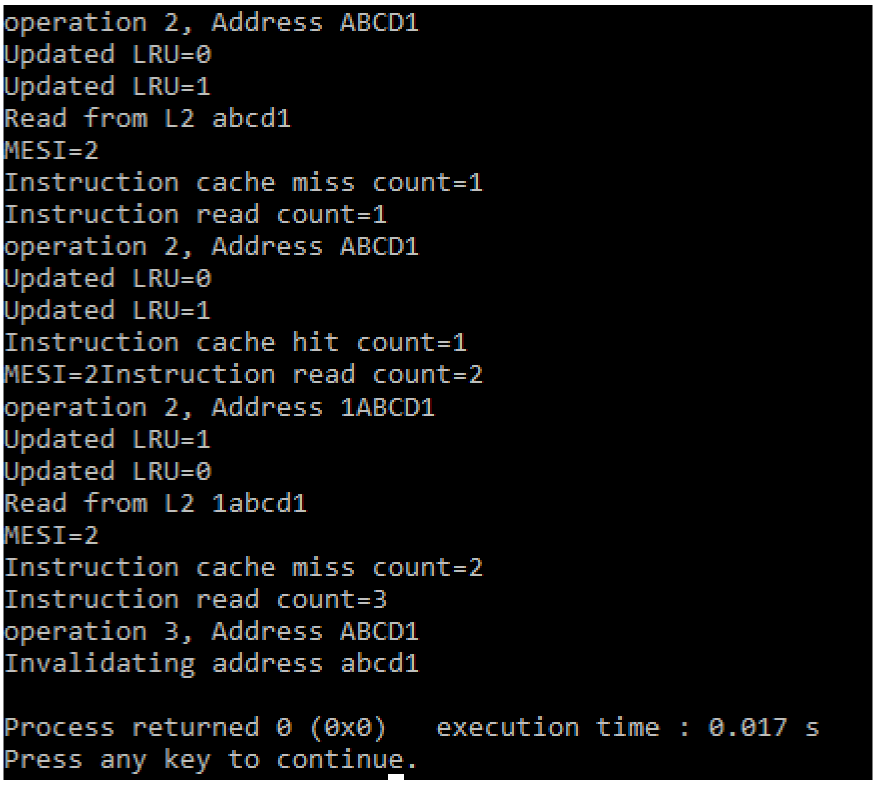
<u>ii) Observations and Expected Output for Invalidating the Instruction cache line:</u>

1. For the instruction cache below the state changes from shared to invalidate when the invalidate operation is performed.
2. When the read is performed for the invalidated cache line, the instruction is read from L2 and the state of the instruction cache line changes from invalidate to shared state.

<u>Output:</u>



```
operation 2, Address ABCD1
Updated LRU=0
Updated LRU=1
Read from L2 abcd1
MESI=2
Instruction cache miss count=1
Instruction read count=1
operation 2, Address ABCD1
Updated LRU=0
Updated LRU=1
Instruction cache hit count=1
MESI=2Instruction read count=2
operation 2, Address 1ABCD1
Updated LRU=1
Updated LRU=0
Read from L2 1abcd1
MESI=2
Instruction cache miss count=2
Instruction read count=3
operation 3, Address ABCD1
Invalidating address abcd1
operation 2, Address ABCD1
Updated LRU=0
Updated LRU=1
Read from L2 abcd1
MESI=2
Instruction cache miss count=3
Instruction read count=4

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```



```
operation 2, Address ABCD1
Updated LRU=0
Updated LRU=1
Read from L2 abcd1
MESI=2
Instruction cache miss count=1
Instruction read count=1
operation 2, Address ABCD1
Updated LRU=0
Updated LRU=1
Instruction cache hit count=1
MESI=2Instruction read count=2
operation 2, Address 1ABCD1
Updated LRU=1
Updated LRU=0
Read from L2 1abcd1
MESI=2
Instruction cache miss count=2
Instruction read count=3
operation 3, Address ABCD1
Invalidating address abcd1

Process returned 0 (0x0)   execution time : 0.017 s
Press any key to continue.
```

Test case 12 - Data request from L2 (in response to snoop)

1. Data request from L2 cache when cache line for L1 cache is modified.
2. Data request from L2 cache when the cache line for L1 cache is in the shared state.
3. Data request from L2 cache when the cache line for L1 cache is in the exclusive state.

Observation and Expected Result :

1. Data request from L2 cache when cache line for L1 cache is modified

a. When the data request is for a modified line, the L2 cache has the stale copy of data, by data request from L2 the data is updated in L2 cache and the L1 cache line state changes from modified to shared state.
b. This is observed for address: ABCD1
c. The LRU bits are updated.

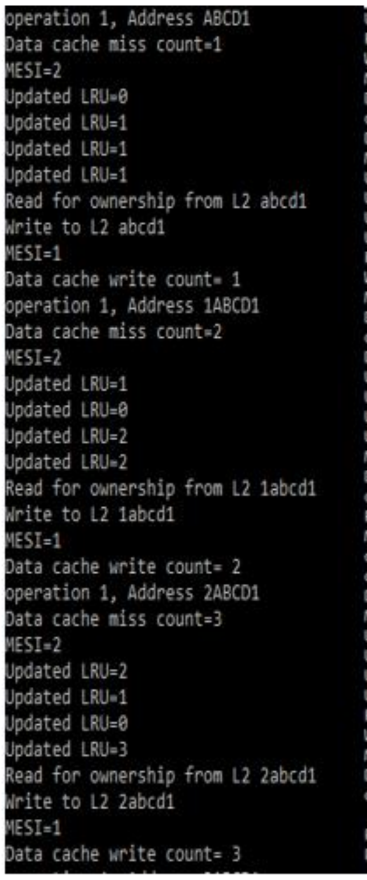
2. Data request from L2 cache when the cache line for L1 cache is in the shared state.

a. When the data request is made from L2 to the shared cache line the next state after this operation is exclusive state, it would perform read for ownership from L2 cache to indicate the snooping processor on FSB our intent to write to the specified memory location.
b. This is shown by address 2abcd1
c. The LRU bits are updated.

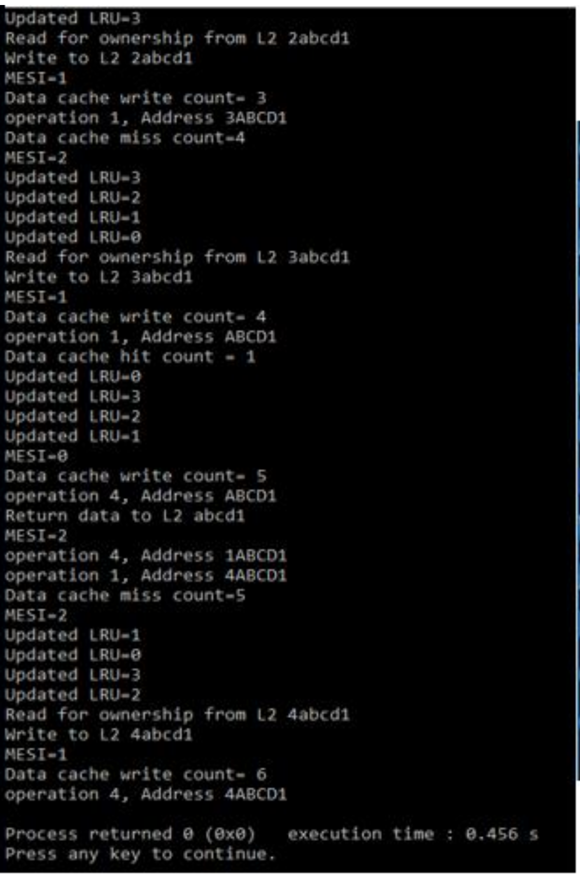
3. Data request from L2 cache when the cache line for L1 cache is in the shared state.

a. When the data is requested from L2 cache to a shared L1 cache line, the L1 cache line state remains the same as shared state and the data is not read back to L2 cache since the data is already present in L2 cache.
b. This operation is shown for address 4ABCD1
c. The LRU bits are updated.

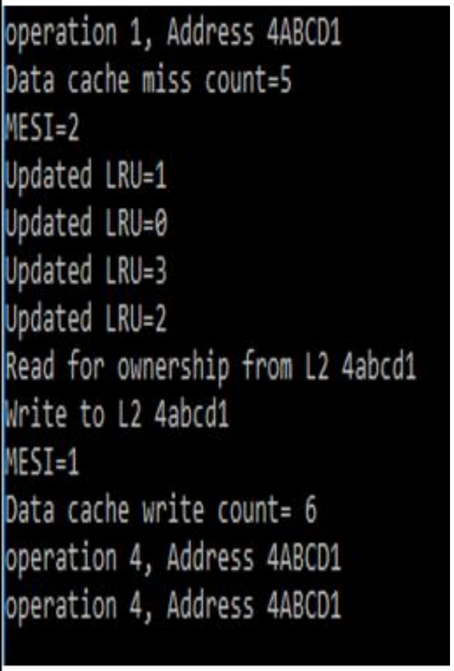
Output:



```
operation 1, Address ABCD1
Data cache miss count=1
MESI=2
Updated LRU=0
Updated LRU=1
Updated LRU=1
Updated LRU=1
Read for ownership from L2 abcd1
Write to L2 abcd1
MESI=1
Data cache write count= 1
operation 1, Address 1ABCD1
Data cache miss count=2
MESI=2
Updated LRU=1
Updated LRU=0
Updated LRU=2
Updated LRU=2
Read for ownership from L2 1abcd1
Write to L2 1abcd1
MESI=1
Data cache write count= 2
operation 1, Address 2ABCD1
Data cache miss count=3
MESI=2
Updated LRU=2
Updated LRU=1
Updated LRU=0
Updated LRU=3
Read for ownership from L2 2abcd1
Write to L2 2abcd1
MESI=1
Data cache write count= 3
```



```
Updated LRU=3
Read for ownership from L2 2abcd1
Write to L2 2abcd1
MESI=1
Data cache write count= 3
operation 1, Address 3ABCD1
Data cache miss count=4
MESI=2
Updated LRU=3
Updated LRU=2
Updated LRU=1
Updated LRU=0
Read for ownership from L2 3abcd1
Write to L2 3abcd1
MESI=1
Data cache write count= 4
operation 1, Address ABCD1
Data cache hit count = 1
Updated LRU=0
Updated LRU=3
Updated LRU=2
Updated LRU=1
MESI=0
Data cache write count= 5
operation 4, Address ABCD1
Return data to L2 abcd1
MESI=2
operation 4, Address 1ABCD1
operation 1, Address 4ABCD1
Data cache miss count=5
MESI=2
Updated LRU=1
Updated LRU=0
Updated LRU=3
Updated LRU=2
Read for ownership from L2 4abcd1
Write to L2 4abcd1
MESI=1
Data cache write count= 6
operation 4, Address 4ABCD1

Process returned 0 (0x0)   execution time : 0.456 s
Press any key to continue.
```



```
operation 1, Address 4ABCD1
Data cache miss count=5
MESI=2
Updated LRU=1
Updated LRU=0
Updated LRU=3
Updated LRU=2
Read for ownership from L2 4abcd1
Write to L2 4abcd1
MESI=1
Data cache write count= 6
operation 4, Address 4ABCD1
operation 4, Address 4ABCD1
```

Source code of Cache:

```
/*
 * main.c
 *
 * ECE 585 Final Project
 * Authors:
 * Manisha Balakrishna Garade
 * Amruta Kalambkar
 * Rashmi Rajendra Kulkarni
 * Daniel Christiansen
 *
 * Version 1.0
 */

#include <string.h>
```

```
#include "cache.h"

// Main cache operation functions
int read_data(int address);
int read_instruction(int address);
int write_data(int address);
void clear_cache();
void print_contents();
void invalidate_line(int address);
void return_data(int address);

// global variables
enum MODE_STATE mode; // Modes: silent, verbose, debug
struct cache data_cache, instruction_cache;

// index into cache line like so:
// data_cache.set[i].line[j].LRU, etc

int main(int argc, char* argv[])
{
        unsigned int operation;
        unsigned int address;
        FILE *fp;
        char ch;
        int s_ret;

        // Allocates caches
        if(allocate_cache(&data_cache, DCA, DCS, D_TAG_BITS,
D_INDEX_BITS, D_OFFSET_BITS))
                return -1;
        if(allocate_cache(&instruction_cache, ICA, ICS, I_TAG_BITS,
I_INDEX_BITS, I_OFFSET_BITS))
                return -1;

    // Clear caches & reset statistic so that all lines are invalid
    clear_cache();
```

```
        // set mode and filename globals
        if(parse_input(argc, argv))
                return -1;

        fp = fopen(argv[2],"r");
        if(NULL == fp)
        {
                fprintf(stderr, "Unable to open file: %s\n", argv[2]);
                return -1;
        }

        while(fscanf(fp,"%d",&operation)!=EOF)
    {
                if(operation==0)
                {
                        fscanf(fp,"%X\n",&address);
                        if(mode == DEBUG)
                                printf("operation     %d,     Address     %X\n",
operation, address);
                        read_data(address);
                }
                else if(operation==1)
       {
                        fscanf(fp,"%X\n",&address);
                        if(mode == DEBUG)
                                printf("operation     %d,     Address     %X\n",
operation, address);
                        write_data(address);
                }
                else if (operation == 2)
       {
                        fscanf(fp,"%X\n",&address);
                        if(mode == DEBUG)
                                printf("operation     %d,     Address     %X\n",
operation, address);
                        read_instruction(address);
                }
```

```
            else if(operation== 3)
        {
                    fscanf(fp,"%X\n",&address);
                    if(mode == DEBUG)
                            printf("operation      %d,      Address      %X\n",
operation, address);
                    invalidate_line(address);
            }
            else if (operation == 4)
        {
                    fscanf(fp,"%X\n",&address);
                    if(mode == DEBUG)
                            printf("operation      %d,      Address      %X\n",
operation, address);
                    return_data(address);
            }
            else if(operation == 8)
            {   printf("operation=%d\n",operation);
                    clear_cache();
                    do {
                            s_ret = fscanf(fp, "%c", &ch);
                    } while (((int)ch != 10) && ((int)ch !=13) && (s_ret
!= EOF));
        }
            else if(operation == 9)
            {   printf("operation=%d\n",operation);
                    print_contents();
          do {
                            s_ret = fscanf(fp, "%c", &ch);
                    } while (((int)ch != 10) && ((int)ch !=13) && (s_ret
!= EOF));
            }
            else
        {
                    fprintf(stderr, "Invalid operation\n");
                    do {
                            s_ret = fscanf(fp, "%c", &ch);
        } while (((int)ch != 10) && ((int)ch !=13) && (s_ret !=
EOF));
            }
      }

      return 0;
}

// Parses command line input
int parse_input(int argc, char* argv[])
{
      if(argc < 2 || argc > 3)
      {
         fprintf(stderr, "Incorrect number of arguments.\n");
            return -1;
   }

   if(0 == strcmp(argv[1], "-s"))
      mode = SILENT;
   else if(0 == strcmp(argv[1], "-v"))
      mode = VERBOSE;
      else if(0 == strcmp(argv[1], "-d"))
            mode = DEBUG;
   else if(0 == strcmp(argv[1], "--help"))
   {
      printf("\nProgram use:");
      printf("\n\tcachesim -[flag] [filename]");
      printf("\n\t-v\tverbose\t- print read/write operations");
      printf("\n\t-s\tsilent\t- only output information on command\n");
            return -1;
   }
   else
   {
      fprintf(stderr, "Invalid argument '%s'.\n", argv[1]);
      return -1;
   }
```

```
	return 0;
}

// Reads data cache
int read_data(int address)
{
	// mask tag & index
	// call cache_check
	// if hit, call update_LRU
	// if miss
	//   send read from L2 signal
	//   find line to evict, replace line, update LRU
	//   new line should be in shared state

	int new_index, new_tag, i;
	// mask tag & index
	new_index = d_index(address);
	new_tag = d_tag(address);

	// call cache_check
	i = cache_check(&data_cache, new_index, new_tag);
	if (i != -1)
	{
		// if hit, update LRU & statistics
		update_LRU(&data_cache, new_index, new_tag, i);
		data_cache.hits++;
		hits++;
		if (mode==DEBUG)
	       {
	          printf("Data cache hit count=%d\n",data_cache.hits);

printf("MESI=%d\n",data_cache.set[new_index].line[i].MESI);
	}
	}
	else
	{
		data_cache.misses++;
```

```
                misses++;
                // if miss, find line to evict, replace line, update LRU
                i = find_victim(&data_cache, new_index);
                // Write back only when line is modified
                if((data_cache.set[new_index].line[i].MESI                  ==
MODIFIED)
                        && ((mode == VERBOSE) || (mode == DEBUG)))
                        printf("Write to L2 %x\n",

        d_address(data_cache.set[new_index].line[i].tag,
                                new_index));
                data_cache.set[new_index].line[i].tag = new_tag;
                data_cache.set[new_index].line[i].MESI = SHARED;
                update_LRU(&data_cache, new_index, new_tag, i);
                if((mode == VERBOSE) || (mode == DEBUG))
                printf("Read from L2 %x\n", address);
                if (mode== DEBUG)
            {
              printf("Data cache miss count=%d\n",data_cache.misses);
              printf("MESI                         =                         %d
\n",data_cache.set[new_index].line[i].MESI);
            }
        }
        data_cache.reads++;
        reads++;
        if(mode==DEBUG)
       printf("Data cache read count=%d\n",data_cache.reads);
}


int write_data(int address)
{
        // mask index & tag bits
        // call cache_check function
        // if hit:
        //   if in shared state, write through, update MESI & LRU
        //   if not, update LRU
        // if miss:
        //   read for ownership from L2
        //   put in modified state

        int new_index, new_tag, i;

        // mask tag & index
        new_index = d_index(address);
        new_tag = d_tag(address);

        // call cache_check
        i = cache_check(&data_cache, new_index, new_tag);
        if (i != -1) // if hit
        {
                data_cache.hits++;
                hits++;
                if(mode==DEBUG)
                {
           printf("Data cache hit count = %d\n",data_cache.hits);
                }
                //if in shared state, write through
                if (data_cache.set[new_index].line[i].MESI == SHARED)
                {
                        if((mode == VERBOSE) || (mode == DEBUG))
                                printf("Write  to  L2  %x\n",  address);//  write
through
                        update_LRU(&data_cache, new_index, new_tag, i);
                         //update MESI & LRU
                        data_cache.set[new_index].line[i].MESI              =
EXCLUSIVE;
                        if (mode==DEBUG)

printf("MESI=%d\n",data_cache.set[new_index].line[i].MESI);
                }
                else // if in E or M, write back
                {
                        //update LRU
```

```
                    if      (data_cache.set[new_index].line[i].MESI      ==
EXCLUSIVE)
                            data_cache.set[new_index].line[i].MESI        =
MODIFIED;
                    update_LRU(&data_cache, new_index, new_tag, i);
                    if (mode==DEBUG)

printf("MESI=%d\n",data_cache.set[new_index].line[i].MESI);
                }
        }
        else  // if miss:
        {
                data_cache.misses++;
                misses++;
                i = find_victim(&data_cache, new_index);
                if(mode==DEBUG)
           printf("Data cache miss count=%d\n",data_cache.misses);
                //update_LRU(&data_cache, new_index, new_tag, i);
                // Write back only when line is modified
                if((data_cache.set[new_index].line[i].MESI                 ==
MODIFIED)
                    && ((mode == VERBOSE) || (mode == DEBUG)))
                    printf("Write to L2 %x\n",

        d_address(data_cache.set[new_index].line[i].tag,
                            new_index));
                data_cache.set[new_index].line[i].tag = new_tag;
                data_cache.set[new_index].line[i].MESI = SHARED;
                if(mode==DEBUG)

printf("MESI=%d\n",data_cache.set[new_index].line[i].MESI);
                update_LRU(&data_cache, new_index, new_tag, i);
                if((mode == VERBOSE) || (mode == DEBUG))
       {
                    printf("Read for ownership from L2 %x\n", address);
                    printf("Write to L2 %x\n", address);
                }
```

```
            data_cache.set[new_index].line[i].MESI = EXCLUSIVE;
        if(mode==DEBUG)
        printf("MESI=%d\n",data_cache.set[new_index].line[i].MESI);
         }
         data_cache.writes++;
         writes++;
         if(mode==DEBUG)
        printf("Data cache write count= %d\n",data_cache.writes);
}

int read_instruction(int address)
{
        // mask tag & index
        // call cache_check
        // if hit, call update_LRU
        // if miss
        //   send read from L2 signal
        //   find line to evict, replace line, update LRU
        //   new line should be in shared state

        int new_index, new_tag, i;
        // mask tag & index
        new_index = i_index(address);
        new_tag = i_tag(address);

         // call cache_check
        i = cache_check(&instruction_cache, new_index, new_tag);
        if (i != -1)
        {
                // if hit, update LRU
                update_LRU(&instruction_cache, new_index, new_tag, i);

                instruction_cache.hits++;
                hits++;
                if (mode==DEBUG)
        {printf("Instruction                    cache                    hit
count=%d\n",instruction_cache.hits);
printf("MESI=%d",instruction_cache.set[new_index].line[i].MESI);
        }
        }
        else
        {
                // if miss, find line to evict, replace line, update LRU
                i = find_victim(&instruction_cache, new_index);
                // Write back only when line is modified
                if((instruction_cache.set[new_index].line[i].MESI          ==
MODIFIED)
                        && ((mode == VERBOSE) || (mode == DEBUG)))
                        printf("Write to L2 %x\n",

        i_address(instruction_cache.set[new_index].line[i].tag,
new_index));
                instruction_cache.set[new_index].line[i].tag = new_tag;
                instruction_cache.set[new_index].line[i].MESI              =
SHARED;
                update_LRU(&instruction_cache, new_index, new_tag, i);
                if((mode == VERBOSE) || (mode == DEBUG))
                printf("Read from L2 %x\n", address);

                instruction_cache.misses++;
                misses++;
                if (mode==DEBUG)
       {

printf("MESI=%d\n",instruction_cache.set[new_index].line[i].MESI);
          printf("Instruction                cache                miss
count=%d\n",instruction_cache.misses);
       }
        }
        instruction_cache.reads++;
        reads++;
        if (mode==DEBUG)
       printf("Instruction read count=%d\n",instruction_cache.reads);
```

```
}

// Invalidates a line
void invalidate_line(int address)
{
	int new_tag, new_index, i;

	// Check if line is in data cache
	new_tag = d_tag(address);
	new_index = d_index(address);
	i = cache_check(&data_cache, new_index, new_tag);
	if(i != -1)
	{
		if((data_cache.set[new_index].line[i].MESI            ==
MODIFIED) ||
			(data_cache.set[new_index].line[i].MESI         ==
EXCLUSIVE))
			fprintf(stderr, "Warning: Invalidating modified line
at address %x\n",
			d_address(new_tag, new_index));
		data_cache.set[new_index].line[i].MESI = INVALID;
		if(mode==DEBUG)

printf("MESI=%d\n",data_cache.set[new_index].line[i].MESI);
		return;
	}

	// Check if line is instruction cache
	new_tag = i_tag(address);
	new_index = i_index(address);
	i = cache_check(&instruction_cache, new_index, new_tag);
	if(i != -1)
	{
		instruction_cache.set[new_index].line[i].MESI         =
INVALID;
		if((mode == VERBOSE) || (mode == DEBUG))
			printf("Invalidating address %x\n", address);
```

```
        }
}

// Sends data to L2 (in response to a snoop)
void return_data(int address)
{
        int new_tag, new_index, i;

        // Mask index & tag from address
        new_tag = d_tag(address);
        new_index = d_index(address);

        i = cache_check(&data_cache, new_index, new_tag);
        if((i != -1) &&
                (data_cache.set[new_index].line[i].MESI == MODIFIED))
        {
                if((mode == VERBOSE) || (mode == DEBUG))
                        {
                          printf("Return data to L2 %x\n", address);
                data_cache.set[new_index].line[i].MESI = SHARED;
                        }
                if(mode==DEBUG)

printf("MESI=%d\n",data_cache.set[new_index].line[i].MESI);
        }
}

// Clears all caches and resets statistics
void clear_cache()
{
        hits = 0;
        misses = 0;
        reads = 0;
        writes = 0;


        invalidate_cache(&data_cache);
        invalidate_cache(&instruction_cache);
        //if((mode=DEBUG) || (operation==8))
    //{
      //  printf("Hits=%d\n",hits);
       //printf("Misses=%d\n",misses);
      // printf("Reads=%d\n",reads);
      // printf("Writes=%d\n",writes);
      // printf("Data cache is cleared\n");
       //printf("Instruction cache is cleared\n");
    //}
}

// Prints cache statistics & valid cache content
void print_contents()
{
        // Global statistics
        printf("\nGlobal cache statistics:\n");
        printf("Reads:\t%ld\n", reads);
        printf("Writes:\t%ld\n", writes);
        printf("Hits:\t%ld\n", hits);
        printf("Misses:\t%ld\n", misses);
        printf("Hit    rate:\t%f\n",    ((float)hits    /    ((float)hits    +
(float)misses)));

        // Data cache statistics & contents
        printf("\nData cache statistics:\n");
        printf("Reads:\t%ld\n", data_cache.reads);
        printf("Writes:\t%ld\n", data_cache.writes);
        printf("Hits:\t%ld\n", data_cache.hits);
        printf("Misses\t%ld\n", data_cache.misses);
        printf("Hit       rate:\t%f\n",       ((float)data_cache.hits       /
((float)data_cache.hits +(float)data_cache.misses)));
        printf("\nData ");
        display_cache(&data_cache);

        // Instruction cache statistics & contents
        printf("\nInstruction cache statistics:\n");
```

```
	printf("Reads:\t%ld\n", instruction_cache.reads);
	printf("Writes:\t%ld\n", instruction_cache.writes);
	printf("Hits:\t%ld\n", instruction_cache.hits);
	printf("Misses\t%ld\n", instruction_cache.misses);
	printf("Hit rate:\t%f\n", ((float)instruction_cache.hits /
((float)instruction_cache.hits + (float)instruction_cache.misses)));
	printf("\nInstruction ");
	display_cache(&instruction_cache);
}
```