|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  | | --- | |  | |  | |  | |  | |
| L3 Cache Simulator Report  ECE 485 / Microprocessor System Design |
| Carmen Ciobanu Deborah Denhart Jeremiah Franke  12/2/2014 |

Table of Contents

L3 Cache Simulator Report 1

1. Purpose 1

2. Cache System Architecture 1

3. Assumptions and Design Considerations 2

3.1 L3 Cache 2

3.2 L2 cache 2

3.3 Shared Memory 3

3.4 Commands 3

4. Design 3

4.1 Trace File Parser 3

4.2 Commands Algorithms 3

4.3 Coherence Protocol (MESIF) 4

4.4. Line Replacement Policy (Pseudo LRU) 6

4.5. L2-L3 Inclusivity 7

4.6. Write Buffer 8

4.7. Memory 8

4.8. Statistics 8

5. Test Plan 8

5.1. Test Plan 8

5.2. Test Cases 8

6. Simulation Results 13

6.1. Trace Files 13

6.2. Results 13

6.3. Usage Statistics 13

# L3 Cache Simulator Report

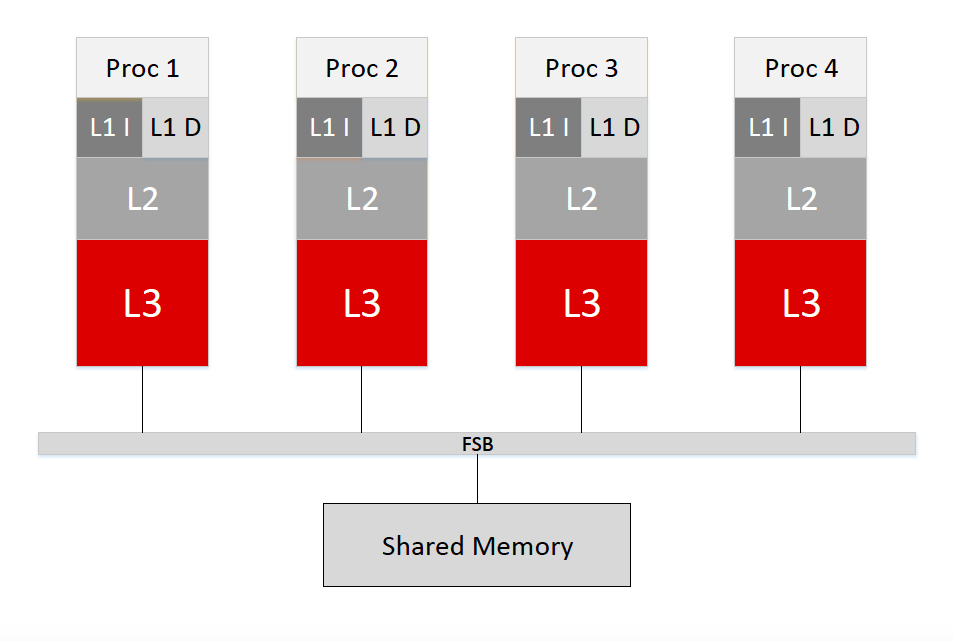
## 1. Purpose

The purpose of the program is to simulate the last level cache (L3) for a processor that can be used with up to three other processors in a shared memory system.

## 2. Cache System Architecture

The system uses multilevel caches:

* Separate L1 instruction and data caches
* Unified L2 instruction and data caches
* Unified L3 instruction and data caches



The L3 caches and memory are communicating on the FSB (Front Side Bus). The L1 and L2 are probably on the same chip as the CPU. L2 might communicate with L3 through a Back Side Bus, or L3 might also be on the same chip as the CPU.

The L1 and L2 caches as well as the DRAM shared memory are not modeled in this program. The program allows for flexible L3 cache configuration, by taking command line arguments. No command line arguments are taken for L2 cache configuration, but several assumptions are made and will be detailed in the Assumptions section of the report.

## 3. Assumptions and Design Considerations

### 3.1 L3 Cache

* The maximum L3 cache size is 4294965248 bytes
* The possible L3 line sizes (in bytes) are: 32, 64, 128, 256, 512, 1024, 2048 and is received as command line argument
* L3 number of sets must be in the range: 1 to 65535 and is received as command line argument
* L3 number of lines must be in the range: 1 to 2097151
* L3 associativity will be one of: 4, 8, 16, or 32 and is received as command line argument
* Data is not stored in the cache in this simulation
* Only tags, state bits and LRU bits are stored in the cache in this simulation
* L3 is a write-back cache (implicit for MESIF implementation)

### 3.2 L2 cache

* L2 cache parameters (associativity, line size etc.) will change depending on the changes in the L3 cache parameters in such a way that it is possible to maintain inclusivity with both L3 and L1 caches
* The ratio of L3 line size to L2 line size is 2:1
* L2 is either a write-trough cache or write-through the first time a line is modified (write once)

### 3.3 Shared Memory

* The size of the L3 line is equal to the number of bytes the DIMM can provide in a burst event
* Only one memory access will be needed to bring in an L3 cache line from DRAM

### 3.4 Commands

* In case of a write command from L2 cache, we are assuming all physical addresses generated are pointing to a line that has the permission of being written (not an instruction)

## 4. Design

### 4.1 Trace File Parser

This parses the files to capture the command and the address to work with the command. The parser converts the character array for the hexadecimal address to an unsigned integer, then it uses bit shifting to get the index and the tag bits and saves them as unsigned integers. This then passes the address, tag, index, and command to the commands functions.

### 4.2 Commands Algorithms

#### 4.2.1. Commands 0, 2

Command 0 = Read Request from L2 Data Cache

Command 2 = Read Request from L2 Instruction Cache

We decided that command 0 should be equivalent to command 2, since at level 2 the instruction and data caches are unified. The actions taken as a result of receiving any of the 0 or 2 commands are identical.

#### 4.2.2. Command 1

Command 1 = Write Request L2 Data Cache

A write request coming from the L2 cache can be interpreted in one of three ways:

1. CPU write request to L1, L1 cache miss, L2 cache miss
2. L2 writes evicted line to L3
3. L2 writes modified line to L3 behaving as a write through cache the first time a line is modified

Since there is not enough granularity in the provided commands to capture all of these three different behaviors, we decided command 1 will be interpreted as a CPU write request to L1, followed by a cache miss in both L1 and L2. This behavior is more useful in emitting the correct bus operation and maintaining coherence with the other L3 caches.

#### 4.2.3. Command 3

Command 3 = Snooped Invalidate

#### 4.2.4. Command 4

Command 3 = Snooped Read Request

#### 4.2.5. Command 5

Command 3 = Snooped Write Request

#### 4.2.6. Command 6

Command 3 = Snooped Read With Intent to Modify

#### 4.2.7. Command 8

Command 8 = Clear Cache

Command 8 clears MESIF bits and implicitly invalidates all lines.

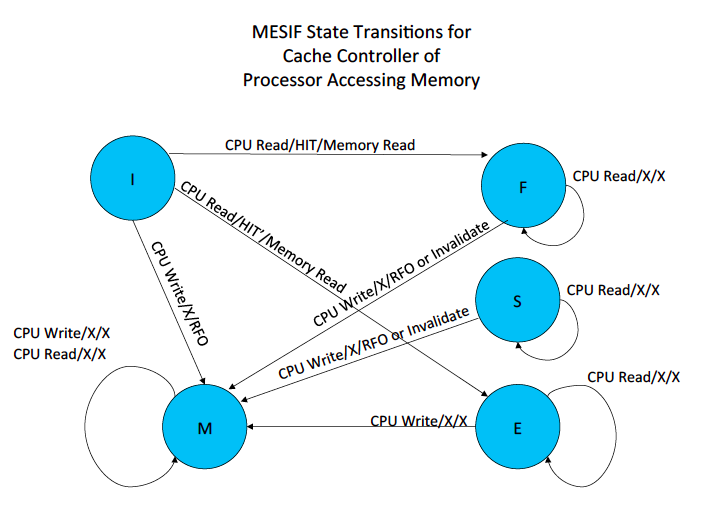
#### 4.2.8. Command 9

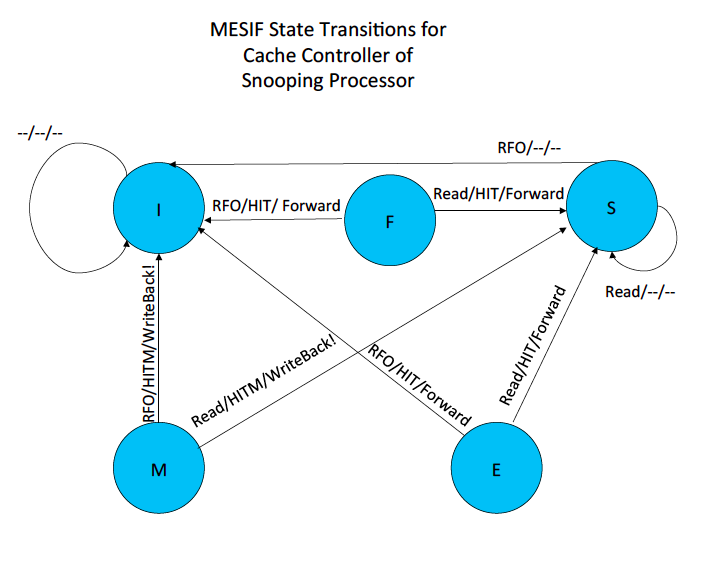
Command 9 = Print Contents and State of Valid Lines

Command 9 displays the tag and MESIF bits of all the valid lines and the pseudo LRU bits of the sets containing at least one valid line.

### 4.3 Coherence Protocol (MESIF)

Stuff



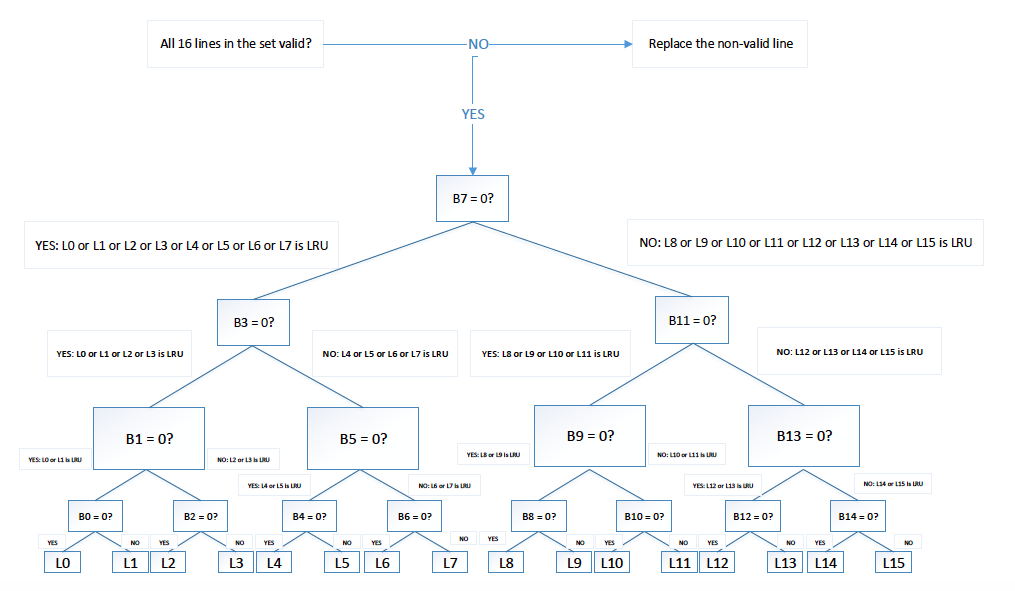


### 4.4. Line Replacement Policy (Pseudo LRU)

The line replacement policy is Pseudo LRU as per requirements. This algorithm uses fewer bits than true LRU methods (n -1 bits, where n is the cache associativity).

For cache accesses resulting in cache hit, the pseudo LRU bits are only written (set or cleared) but not read. For accesses resulting in cache miss, the pseudo LRU bits are read to select a line for eviction (if no invalid lines available), and then written again to indicate current line was most recently used.

The pseudo LRU bits write functionality is implemented with an algorithm similar to binary search. If we imagine a binary tree of pseudo LRU bits, they are numbered following the binary tree rule, where everything less than the root is found in the left branch tree and everything greater than root is found in the right branch tree. The following diagram illustrated the bit numbering used, for an example associativity of 16:



The write algorithm operates on an array containing all line indices in any set (0 to associativity - 1). It finds recursively the midpoint of the array, corresponding actually to the current bit number to set or clear. The selected bit is set if the currently accessed line number is less than or equal to the bit number, and cleared otherwise. The recursive algorithm ends when the endpoints of the interval for which the midpoint was found become equal.

The pseudo LRU bits read functionality follows the same binary search algorithm. If no invalid lines are available, a line will be selected for eviction. In order to select the pseudo LRU line, the binary search array is traversed using recursive midpoints as described above. The selected bit value is read and the decision to take the right or left array search interval is determined by this bit value.

A 0 bit value will direct the search towards the left interval, while a 1 bit value will direct the search towards the right interval. The index and value of the last queried bit will tell us which of the odd or even numbered line is the pseudo LRU (a 0 bit will return a line number identical to the bit index, while a 1 value bit will return a line number greater with 1 than the bit index).

### 4.5. L2-L3 Inclusivity

One of the project requirements is for cache L3 to maintain inclusivity with the higher level caches. Since we decided to interpret the write command (1) as a CPU write request followed by cache misses in L1 and L2, we are not going to be able to capture L2 writes to L2 due to dirty line evictions or L2 write back for the first time a line is modified. This makes keeping a set of inclusivity bits impractical.

Instead, we chose to send messages from L3 to L2 indicating a set of lines that might need to be invalidated in the L2 cache for all lines in L3, which are either evicted or invalidated (as a result of snooped invalidate and snooped RWIM bus operations). Of course, not all of these lines are in the higher level cache, but without capturing the L2 write back events, there is really no way of telling which line is also in L2.

The result of our design decision will increase the communication traffic between L2 and L3. We are assuming however L2 will receive all messages coming from L3 but will simply disregard the commands associated with addresses, which do not correspond to lines found in both L2 and L3 caches.

Finally, we need to consider the difference in line sizes between L2 and L3. Under “Assumptions”, we stated that the L3 to L2 line size ratio is always assumed to be 2:1. Under this assumption, a line eviction in L3 will have to be followed by a message to L2 indicating two of L2 lines should be invalidated, if resident in L2.

In summary, inclusivity is maintained by taking the following actions:

* Following a read/write request from L2, the desired line is places in both L2 and L3
* If an L3 line was evicted in the process, 2 lines must be invalidated from L2 if these reside in L2
* If an L3 line was invalidated as a result of snooped invalidate and snooped RWIM bus operations, 2 lines must be invalidated from L2 if these reside in L2
* If all L3 lines were invalidated as a result of a clear command, all lines must be invalidated in L2

The process of invalidating lines in L2 cannot be enforced, since we are not simulating L2, but it is signaled in the program by printing appropriate messages to the screen in the non-silent mode. Addresses of lines to be written or invalidated in L2 are provided by L3, along with the required action to be performed by L2.

### 4.6. Write Buffer

The write buffer acts as a converging write buffer for the evicted lines from the cache. This will write to memory every few clock ticks (line reads from the trace file). This will accept insert, clear, check, and trigger commands. Insert inserts data into the write buffer. Clear writes all data to memory. Check walks the cache to see if it can find the address in it. Trigger writes to memory the first address put into the buffer.

### 4.7. Memory

The DRAM shared memory is not modeled in this program. To show when a read from memory or a write to memory is performed, a few memory function stubs are in place in the code to remind us of the current bus operation and to show which address is provided to the DIMMs in order to transfer an entire cache line in a burst.

### 4.8. Statistics

After executing the last instruction read form the trace file, the program will output cache statistics for the current trace file. The number of accesses, reads, writes, hits and misses are incremented by case only when executing commands 0, 1 or 2. The cache hit ratio is expressed as the ratio of the number of cache hits and the number of cache accesses.

## 5. Test Plan

### 5.1. Test Plan

Stuff

### 5.2. Test Cases

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case No: | 1 | | | | |
| Title: | File Parser Test | | | | |
| Created By: | Jeremiah Franke | | | | |
| Tested by: | Jeremiah Franke | | | | |
| Feature: | File Parser | | | | |
| **Step Description** | | **Expected Result** | **PASS** | **FAIL** | **Actual Results** |
| 1. Compile program in debug mode | | Program should compile without errors. | Pass |  | Compiled without errors |
| 2. Run program with arguments:  #sets = 8192  Line size = 64  Associativity = 16  Trace file = “TestParse.txt” | | Program runs with no errors. | Pass |  | Program Ran |
| 3. Observe output to screen | | Should see output on the screen. | Pass |  | Observed Output |
| 4. Check output lines of Parser – Following Hex:  f08ed4fa  403daa3b  403daa3b  FFFDAA00  FFF5AA00  FFEDAA00  FFC5AA00  FFADAA00  FDFDAA00  FBFDAA00  F7FDAA00  EFFDAA00  8005AA00  E005AA00  F805AA00  402daa3b  401daa3b  400daa3b  413daa3b  f08ed42a  403daa3b  403daa3b  f08ed42a  00aed48a  00aed48a  f08ed42a  403daa3b  403daa3b  f08ed42a  00aed48a  00aed48a | | Address: 4035892474  Address: 1077783099  Address: 1077783099  Address: 4294814208  Address: 4294289920  Address: 4293765632  Address: 4291144192  Address: 4289571328  Address: 4261259776  Address: 4227705344  Address: 4160596480  Address: 4026378752  Address: 2147854848  Address: 3758467584  Address: 4161120768  Address: 1076734523  Address: 1075685947  Address: 1074637371  Address: 1094560315  Address: 4035892266  Address: 1077783099  Address: 1077783099  Address: 4035892266  Address: 11457674  Address: 11457674  Address: 4035892266  Address: 1077783099  Address: 1077783099  Address: 4035892266  Address: 11457674  Address: 11457674 | Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass |  | Address: 4035892474  Address: 1077783099  Address: 1077783099  Address: 4294814208  Address: 4294289920  Address: 4293765632  Address: 4291144192  Address: 4289571328  Address: 4261259776  Address: 4227705344  Address: 4160596480  Address: 4026378752  Address: 2147854848  Address: 3758467584  Address: 4161120768  Address: 1076734523  Address: 1075685947  Address: 1074637371  Address: 1094560315  Address: 4035892266  Address: 1077783099  Address: 1077783099  Address: 4035892266  Address: 11457674  Address: 11457674  Address: 4035892266  Address: 1077783099  Address: 1077783099  Address: 4035892266  Address: 11457674  Address: 11457674 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case No: | 2 | | | | |
| Title: | Address Translation test | | | | |
| Created By: | Jeremiah Franke | | | | |
| Tested by: | Jeremiah Franke | | | | |
| Feature: | Split address into tag, index, offset correctly | | | | |
| **Step Description** | | **Expected Result** | **PASS** | **FAIL** | **Actual Results** |
| 1. Compile program in debug mode | | Program should compile without errors. | Pass |  | Compiled |
| 2. Run program with arguments:  #sets = 8192  Line size = 64  Associativity = 16  Trace file = “TestTrans.txt” | | Program runs with no errors. | Pass |  | Ran |
| 3. Observe output to screen | | Should see output on the screen. | Pass |  | Output Shown |
| 4. Check output lines of Translation–Hex:  f08ed4fa  403daa3b  403daa3b  FFFDAA00  FFF5AA00  FFEDAA00  FFC5AA00  FFADAA00  FDFDAA00  FBFDAA00  F7FDAA00  EFFDAA00  8005AA00  E005AA00  F805AA00  402daa3b  401daa3b  400daa3b  413daa3b  f08ed42a  403daa3b  403daa3b  f08ed42a  00aed48a  00aed48a  f08ed42a  403daa3b  403daa3b  f08ed42a  00aed48a  00aed48a | | Tag: 7697 Index: 6995  Tag: 2055 Index: 5800  Tag: 2055 Index: 5800  Tag: 8191 Index: 5800  Tag: 8190 Index: 5800  Tag: 8189 Index: 5800  Tag: 8184 Index: 5800  Tag: 8181 Index: 5800  Tag: 8127 Index: 5800  Tag: 8063 Index: 5800  Tag: 7935 Index: 5800  Tag: 7679 Index: 5800  Tag: 4096 Index: 5800  Tag: 7168 Index: 5800  Tag: 7936 Index: 5800  Tag: 2053 Index: 5800  Tag: 2051 Index: 5800  Tag: 2049 Index: 5800  Tag: 2087 Index: 5800  Tag: 7697 Index: 6992  Tag: 2055 Index: 5800  Tag: 2055 Index: 5800  Tag: 7697 Index: 6992  Tag: 21 Index: 6994  Tag: 21 Index: 6994  Tag: 7697 Index: 6992  Tag: 2055 Index: 5800  Tag: 2055 Index: 5800  Tag: 7697 Index: 6992  Tag: 21 Index: 6994  Tag: 21 Index: 6994 | Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass  Pass |  | Tag: 7697 Index: 6995  Tag: 2055 Index: 5800  Tag: 2055 Index: 5800  Tag: 8191 Index: 5800  Tag: 8190 Index: 5800  Tag: 8189 Index: 5800  Tag: 8184 Index: 5800  Tag: 8181 Index: 5800  Tag: 8127 Index: 5800  Tag: 8063 Index: 5800  Tag: 7935 Index: 5800  Tag: 7679 Index: 5800  Tag: 4096 Index: 5800  Tag: 7168 Index: 5800  Tag: 7936 Index: 5800  Tag: 2053 Index: 5800  Tag: 2051 Index: 5800  Tag: 2049 Index: 5800  Tag: 2087 Index: 5800  Tag: 7697 Index: 6992  Tag: 2055 Index: 5800  Tag: 2055 Index: 5800  Tag: 7697 Index: 6992  Tag: 21 Index: 6994  Tag: 21 Index: 6994  Tag: 7697 Index: 6992  Tag: 2055 Index: 5800  Tag: 2055 Index: 5800  Tag: 7697 Index: 6992  Tag: 21 Index: 6994  Tag: 21 Index: 6994 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case No: | 3 | | | | |
| Title: | Line Selection Test | | | | |
| Created By: | Jeremiah Franke | | | | |
| Tested by: | Jeremiah Franke | | | | |
| Feature: | Select correctly the desired line in a set. Get the line by checking Tag bits. | | | | |
| **Step Description** | | **Expected Result** | **PASS** | **FAIL** | **Actual Results** |
| 1. Compile program in debug mode | | Program should compile without errors. | Pass |  | Compiled |
| 2. Run program with arguments:  #sets = 8192  Line size = 64  Associativity = 16  Trace file = “TestLines.txt” | | Program runs with no errors. | Pass |  | Ran |
| 3. Fill cache with:  f08ed4fa  403daa3b  FFFDAA00  FFF5AA00  FFEDAA00  FFC5AA00  FFADAA00  FDFDAA00  3298A120  4DFABC18  AD12F89B | | Should see output on the screen for print of the cache. | Pass |  | Output Shown |
| 4. Observe output to screen | | Should see output on the screen. | Pass |  | Output Shown |
| 5. Check Line | | 0 | Pass |  | 0 |
| 6. Check Line | | 0 | Pass |  | 0 |
| 7. Check Line | | 1 | Pass |  | 1 |
| 8. Check Line | | 2 | Pass |  | 2 |
| 9. Check Line | | 3 | Pass |  | 3 |
| 10. Check Line | | 4 | Pass |  | 4 |
| 11. Check Line | | 5 | Pass |  | 5 |
| 12. Check Line | | 6 | Pass |  | 6 |
| 13. Check Line | | 0 | Pass |  | 0 |
| 14. Check Line | | 0 | Pass |  | 0 |
| 15. Check Line | | 0 | Pass |  | 0 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case No: | 4 | | | | |
| Title: | Pseudo LRU functional test | | | | |
| Created By: | Carmen Ciobanu | | | | |
| Tested by: | Carmen Ciobanu | | | | |
| Feature: | Pseudo LRU line replacement policy | | | | |
| **Step Description** | | **Expected Result** | **PASS** | **FAIL** | **Actual Results** |
| 1. Compile program in debug mode | | Program should compile without errors |  |  |  |
| 2. Run program with arguments:  #sets = 8192  Line size = 64  Associativity = 16  Trace file = “TestLRU.txt” | | Program runs, no errors |  |  |  |
| 3. Observe output to screen | | Should see program output |  |  |  |
| 4. Check first 16 output lines | | M: Available invalid line 0  M: Available invalid line 1  M: Available invalid line 2  M: Available invalid line 3  M: Available invalid line 4  M: Available invalid line 5  M: Available invalid line 6  M: Available invalid line 7  M: Available invalid line 8  M: Available invalid line 9  M: Available invalid line 10  M: Available invalid line 11  M: Available invalid line 12  M: Available invalid line 13  M: Available invalid line 14  M: Available invalid line 15 |  |  |  |
| 5. Check next output line | | H |  |  |  |
| 6. Check next output line | | H |  |  |  |
| 7. Check next output line | | M: Evict line 0 |  |  |  |
| 8. Check next output line | | H |  |  |  |
| 9. Check next output line | | M: Evict line 4 |  |  |  |
| 10. Check next output line | | H |  |  |  |
| 11. Check next output line | | H |  |  |  |
| 12. Check next output line | | H |  |  |  |
| 13. Check next output line | | M: Evict line 14 |  |  |  |
| 14. Check next output line | | M: Evict line 2 |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case No: | 5 | | | | |
| Title: | MESIF Coherence Test | | | | |
| Created By: |  | | | | |
| Tested by: |  | | | | |
| Feature: | Cache coherence | | | | |
| **Step Description** | | **Expected Result** | **PASS** | **FAIL** | **Actual Results** |
| 1. | |  |  |  |  |
| 2. | |  |  |  |  |
| 3. | |  |  |  |  |
| 4. | |  |  |  |  |
| 5. | |  |  |  |  |
| 6. | |  |  |  |  |
| 7. | |  |  |  |  |
| 8. | |  |  |  |  |
| 9. | |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case No: | 6 | | | | |
| Title: | Inclusivity Test | | | | |
| Created By: | Carmen Ciobanu | | | | |
| Tested by: | Carmen Ciobanu | | | | |
| Feature: | L2-L3 inclusivity | | | | |
| **Step Description** | | **Expected Result** | **PASS** | **FAIL** | **Actual Results** |
| 1. | |  |  |  |  |
| 2. | |  |  |  |  |
| 3. | |  |  |  |  |
| 4. | |  |  |  |  |
| 5. | |  |  |  |  |
| 6. | |  |  |  |  |
| 7. | |  |  |  |  |
| 8. | |  |  |  |  |
| 9. | |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case No: | 7 | | | | |
| Title: | Statistics functional test | | | | |
| Created By: | Carmen Ciobanu | | | | |
| Tested by: | Carmen Ciobanu | | | | |
| Feature: | Display cache statistics at the end of current trace file | | | | |
| **Step Description** | | **Expected Result** | **PASS** | **FAIL** | **Actual Results** |
| 1. Compile program in silent mode | | Program should compile without errors |  |  |  |
| 2. Run program with arguments:  #sets = 8192  Line size = 64  Associativity = 16  Trace file = “TestStatistics.txt” | | Program runs, no errors |  |  |  |
| 3. Read screen output | | Should display Cache Architecture parameters followed by Cache Statistics parameters |  |  |  |
| 4. Check cache capacity value | |  |  |  |  |
| 5. Check cache number of sets | | 1892 |  |  |  |
| 6. Check line size | | 64 |  |  |  |
| 7. Check total # of lines | |  |  |  |  |
| 8. Check cache associativity | | 16 |  |  |  |
| 9. Check cache # reads | |  |  |  |  |
| 10. Check cache # writes | |  |  |  |  |
| 11. Check cache # accesses | |  |  |  |  |
| 12. Check cache hit ratio | |  |  |  |  |
| 13. Check cache # hits | |  |  |  |  |
| 14. Check cache # misses | |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case No: | 8 | | | | |
| Title: | Write Buffer Test | | | | |
| Created By: | Jeremiah Franke | | | | |
| Tested by: | Jeremiah Franke | | | | |
| Feature: | Merging Write Buffer | | | | |
| **Step Description** | | **Expected Result** | **PASS** | **FAIL** | **Actual Results** |
| 1. Compile program in debug mode | | Program should compile without errors. | Pass |  | Compiled |
| 2. Run program with arguments:  #sets = 8192  Line size = 64  Associativity = 16  Trace file = “TestLines.txt” | | Program runs with no errors. | Pass |  | Ran |
| 3. Fill cache with:  FFFDAA00  FFF5AA00  FFEDAA00  FFC5AA00  FFADAA00  FDFDAA00  FBFDAA00  F7FDAA00  EFFDAA00  8005AA00  E005AA00  F805AA00  F005AA00  F105AA00  F505AA00  FD05AA00  FF05AA00  FF45AA00  BF45AA00  AF45AA00 | | Should see output on the screen for print of the cache. | Pass |  | Output Shown |
| 4. Turn off Trigger | | Trigger Off | Pass |  | Trigger Off |
| 5. Observe output to screen from printing cache | | Should see output on the screen. | Pass |  | Output Shown |
| 6. Observe output when running Insert by setting MESIF bits to MODIFIED and eviction to TRUE | | Insert:  FF05AA00 | Pass |  | Insert:  FF05AA00 |
| 7. Observe output when running Insert by setting MESIF bits to MODIFIED and eviction to TRUE | | Insert:  FF45AA00 | Pass |  | Insert:  FF45AA00 |
| 8. Observe output when running Insert by setting MESIF bits to MODIFIED and eviction to TRUE | | Insert:  BF45AA00 | Pass |  | Insert:  BF45AA00 |
| 9. Observe output when running Insert by setting MESIF bits to MODIFIED and eviction to TRUE | | Insert:  AF45AA00 | Pass |  | Insert:  AF45AA00 |
| 10. Turn Trigger on. | | Trigger On | Pass |  | Trigger On |
| 11. Observe output when running Trigger by passing 5 to the function in the trigger variable. | | Memory Write:  FF05AA00 | Pass |  | Memory Write:  FF05AA00 |
| 12. Observe output when running Trigger by passing 5 to the function in the trigger variable. | | Memory Write:  FF45AA00 | Pass |  | Memory Write:  FF45AA00 |
| 13. Observe output when running Trigger by passing 5 to the function in the trigger variable. | | Memory Write:  BF45AA00 | Pass |  | Memory Write:  BF45AA00 |
| 14. Observe output when running Trigger by passing 5 to the function in the trigger variable. | | Memory Write:  AF45AA00 | Pass |  | Memory Write:  AF45AA00 |
| 15. Rerun Program and follow instructions up to Insert. | | Same as previous | Pass |  | Same as Previous |
| 16. Observe output when running Check by passing FF05AA00 to the function in the address variable. | | Memory Here:  FF05AA00 | Pass |  | Memory Here:  FF05AA00 |
| 17. Observe output when running Trigger by passing FF45AA00 to the function in the address variable. | | Memory Here:  FF45AA00 | Pass |  | Memory Here:  FF45AA00 |
| 18. Observe output when running Trigger by passing BF45AA00 to the function in the address variable. | | Memory Here:  BF45AA00 | Pass |  | Memory Here:  BF45AA00 |
| 19. Observe output when running Trigger by passing AF45AA00 to the function in the address variable. | | Memory Here:  AF45AA00 | Pass |  | Memory Here:  AF45AA00 |
| 20. Observer output when running clear at the end of the program. | | Memory Write:  FF05AA00  Memory Write:  FF45AA00  Memory Write:  BF45AA00  Memory Write:  AF45AA00 | Pass |  | Memory Write:  FF05AA00  Memory Write:  FF45AA00  Memory Write:  BF45AA00  Memory Write:  AF45AA00 |

## 6. Simulation Results

### 6.1. Trace Files

Stuff

### 6.2. Results

Stuff

### 6.3. Usage Statistics