

**Table of Contents**

[**Overview/Abstract**](#_5uvz7uszdsi7) **3**

[**Problem Statement**](#_qwgj7r9lrvvv) **3**

[**Specifications**](#_w6avj3pcp8nh) **3**

[L1 Instruction Cache](#_fbzwtou67eko) 4

[L1 Data Cache](#_vta991pt3u1b) 4

[**Assumptions**](#_cilnjxnsace7) **4**

[**Addressing**](#_txgfkl6dps50) **5**

[**MESI Protocol**](#_b3mgxxogwy0k) **5**

[**LRU**](#_b1ainj9or876) **5**

[**Pseudocode**](#_lkzq6njbzq70) **6**

[Data Cache:](#_igfdpkb0sqfb) 6

[Read Flow Chart](#_pdj0cgz7r7zk) 6

[Write Flow Chart](#_v1om8s4tve0y) 7

[Invalidate and Snooping](#_9ti9m4czewli) 9

[**Testing**](#_en4mdfyepz5o) **9**

[**Source Code**](#_8itvstxj27ow) **13**

# **Overview/Abstract**

This project required the design and simulation of a split L1 cache for a 32 bit processor. The processor can be used with up to at least three other processors in a shared memory configuration. To ensure cache coherence the design uses a MESI protocol.

The caches had two caches in the processor, instruction and data caches. The instruction cache is four-way set associative consists of 16K sets and 64-byte lines. The L1 data cache is eight-way set associative and contains 16K sets and 64-byte lines. The L1 data cache is write-back using write allocate and is write-back except for the first write to a line which is write-through. Both caches employ LRU replacement policy and are backed by a shared L2 cache. In addition, this cache hierarchy employs inclusivity.

# **Problem Statement**

For the simulation of the L1 cache, there needs to be output printing out a summary of the usage statistics and responses to a command to print contents and state of the cache. In another mode, in addition to the previous print messages, there is also printing from the L2 cache. The main part of the program takes in an argument containing the input file. Then depending on the input of the text file using the following format, while the address is in hexadecimal.

*n address*

Where n is

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)

# **Specifications**

In this project, we have implemented a split L1 for a 32-bit microprocessor which can be used with up to three other processors in a shared memory configuration. The system employs cache coherence by using the MESI protocol. It includes an L1 data cache, which is eight-way set associative and contains 16K sets and 64-bytes lines. The data cache is write-back using write allocate. It also includes a L1 instruction cache, which is a four-way set associative and consists of 16K and 64-bytes lines. The L2 cache has not been modeled explicitly, but it serves as a stub module; we know the L2 cache will know how to get the data. If the data doesn't exist within the L2 cache, it will get it from the L3 cache if it exists, or directly from main memory. The cache hierarchy employs inclusivity. We also maintain various statistics to model the performance of the L1 cache, such as the number of cache hits, misses and the calculate the cache ratio. These statistics are outputted after execution.

|  |  |  |
| --- | --- | --- |
| **Specifications** | L1 Instruction Cache | L1 Data Cache |
| **Set Associativity** | 4-Way | 8-Way |
| **Sets** | 16K | 16K |
| **Cache Line Size** | 64 Bytes | 64 Bytes |
| **Inclusivity Property** | Employs Inclusivity | Employs Inclusivity |

TABLE 1: Project specifications

# **Assumptions**

While working through the project, several assumptions had to be made about the requirements and what the instructions for the cache controller project stated for the purpose of the simulation. Some of our assumptions include:

* **Singular Simulation of L1 Cache**. This means that only the split L1 cache of instruction and data cache read, write, hits, and misses are being recorded. While there are four processors total, for our project we just simulated one processor. The output printing out the results is what the other processors see is happening.
* **The data contained within the addresses is not important** for this project. This concludes that the focus of the project is making sure the commands and addresses are correctly placed within the memory space. Not on what is inside the data and what is being stored.
* **L2 Cache isn’t being simulated**, the messages required for communication between L1 and L2 are printed out just to show what interaction occurs.
* **The cache hierarchy employs inclusivity**, because it’s a requirement.
* **Both caches employ LRU replacement policy** and are backed by a shared L2 cache
* **The cache has a write once policy** because of the L1 data cache being write-back using write allocate. Additionally it’s write-back except for the first write to a line which is write-through.

# **Addressing**

The addressing in this cache controller project was modeled after the project explanation given on D2L. For this reason the address is represented as hexadecimal as the file input. The binary representation of the address is a 32 bit address. 12 of these bits are the tag bits, 14 bits are the index bits, and 6 bits are used for the offset.

# **MESI Protocol**

MESI is an acronym for Modified, Exclusive, Shared and Invalid. These represent the four states that an individual line in a cache. The Modified (M) state implies the cache line found by a write hit was exclusive and the current processor has modified the data, also it expresses that currently not shared and exclusively owned data have been modified. The Exclusive (E) state indicates that the current cache is not aware of any other cache sharing the same information and the line is unmodified, however if another line contains the same information in the future, it will be allowed and the (E) state will be changed. The Shared (S) state expresses that the hit line is present in more than one cache. The Invalid (I) state indicates that the associated cache line is invalid and holds no valid data and it will always set after a system reset. In some cases where after a line write hit in a cache and the data in the cache and memory are not identical then the data will be written in the memory however, the modifier should observe the other bus master when trying to access the same line, then in this case, the action which be taken to ensure data integrity is called snooping. This protocol is necessary to ensure Cache Coherence as more than one processor can have a copy of common data with right to modify it. Via this protocol, we ensure that data coherence is maintained no matter which of the processors performs a write to the cache.

# **LRU**

The methodology used for replacing data was the LRU used method. The strategy used for our LRU method was that anything less than the current LRU, increments, and anything greater than the current LRU doesn’t change. In the cache controller simulation, we didn’t create a LRU prototype but several functions which take care of the LRU replacement in the cache controller.

# **Pseudocode**

## **Data Cache:**

## **Read Flow Chart**

## 

Read\_data:

First read to find the gap fill in the empty slot

If there is an empty slot, then **fill data MESI state is E**

Else if no empty slot, then search for hit

Call matching\_tag

If matching\_tag ==0 miss

Count\_miss ++

Is there any Invalid state in the cache line ? call search\_invalid\_state function

If search\_invalid\_state ==1 found

evict that slot

**replace data MESI state is E**

call LRU\_data\_update

If searh\_invalid\_state ==0

call search\_LRU

**place the data MESI state is E** recent use (LRU==111)

call LRU\_data\_update

Else if matching\_tag==1 hit

Count\_hit ++

If MESI state is **E** , then MESI state will be **S**

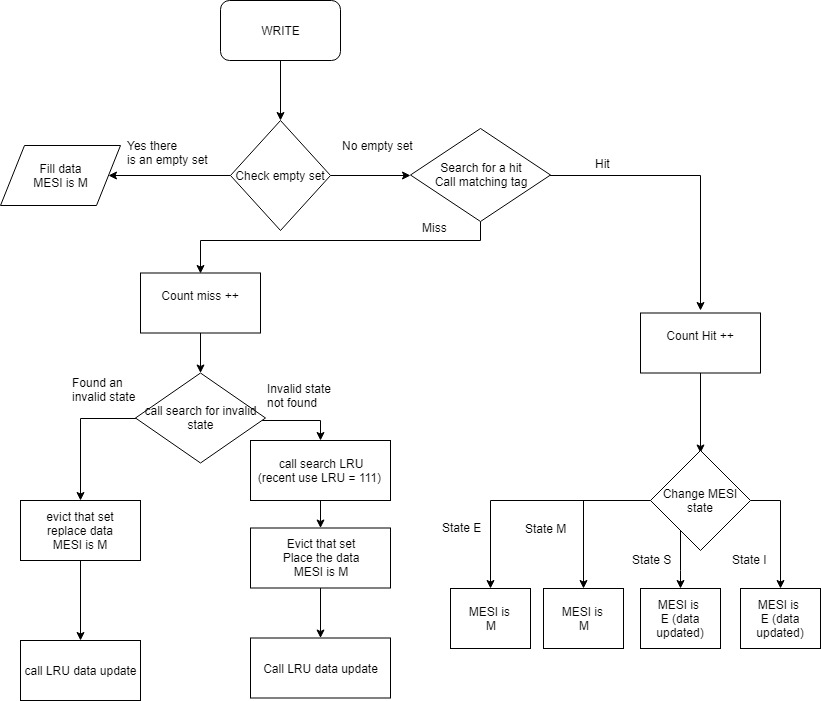
else if MESI state is **M ,** then MESI state will be **M**

else if MESI state is **S ,** then MESI state will be **S** (data the same)

else if MESI state is **I,** then MESI state will be **S** (data updated)

call LRU\_data\_update

## **Write Flow Chart**



Write\_data:

First write to find the gap fill in the empty set

If there is an empty set, then **fill data**

MESI State is M

Else if no empty set, then search for hit

Call matching\_tag

If matching\_tag ==0 miss

count \_miss++

Is there any Invalid state in the cache line ? call search\_invalid\_state function

If search\_invalid\_state ==1 found

evict that set

**replace data MESI state is M**

call LRU\_data\_update

If search\_invalid\_state ==0

call search\_LRU recent use (LRU==111)

evict that set

**place the data MESI state is M**

call LRU\_data\_update

Else if matching\_tag==1 hit

count\_hit++

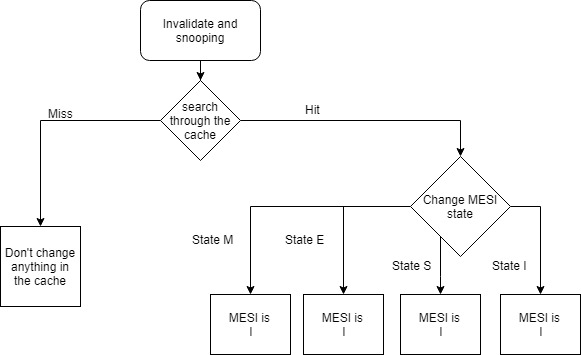
If MESI state is **E** , then MESI state will be **M**

else if MESI state is **M ,** then MESI state will be **M**

else if MESI state is **S ,** then MESI state will be **E** (data updated)

else if MESI state is **I,** then MESI state will be **E** (data updated)

call LRU\_data\_update

****

## **Invalidate and Snooping**

Search through the cache:

If cache hit

If MESI state is **M** , then MESI state will be **I**

else if MESI state is **E ,** then MESI state will be **I**

else if MESI state is **S ,** then MESI state will be **I**

else if MESI state is **I,** then MESI state will be **I**

Else if cache miss

Return

# **Testing**

In order to test the simulation of a cache controller, an input file is passed in which has the instructions on what to do with the address that follows the text. The first tests mimicked the project explanation on D2L with all the commands being read data requests to L1 data cache. Subsequent tests observed the results of passing in the rest of the n commands and making sure there were cache hits whenever the same address was being used.

Testvector:

8 FFFFFFFF

2 974DE100

2 967DE100

2 947DE100

2 967DE100

2 667DE100

2 967DE100

2 967DE100

2 967DE100

0 999DE132

0 999DE132

0 999DE132

0 999DE132

0 999DE132

0 999DE132

0 999DE132

0 999DE132

0 999DE132

0 999DE132

0 999DE132

0 999DE132

0 999DE132

0 999DE132

1 116DE123

1 666DE135

1 333DE12C

0 846DE10C

0 777DE136

1 ABCDE128

0 116DE101

1 100DE101

1 AAADE101

1 EDCDE101

4 0AAADE10

0 999DE132

1 116DE123

1 666DE135

1 333DE12C

4 333DE12C

3 333DE12C

9 FFFFFFFF

Results:

aleolson@ada:~/ECE585$ ./a.out testvector

Resetting Cache Controller...

Mode Selection

Mode 0: Summary of usage statistics and print contents and state of cache

Mode 1: Information from Mode 0 and messages to L2.

Mode: 1

Resetting Cache Controller...

Read from L2 974de100 [Instruction]

Read from L2 967de100 [Instruction]

Read from L2 947de100 [Instruction]

Read from L2 967de100 [Instruction]

Read from L2 667de100 [Instruction]

Read from L2 999de132 [data]

Read from L2 999de132 [data]

Read from L2 999de132 [data]

Read from L2 999de132 [data]

Read from L2 999de132 [data]

Read from L2 999de132 [data]

Read from L2 999de132 [data]

Read from L2 999de132 [data]

Read for Ownership from L2 116de123

Write to L2 116de123 [write-back]

Read for Ownership from L2 666de135

Write to L2 666de135 [write-back]

Read for Ownership from L2 333de12c

Write to L2 333de12c [write-back]

Read from L2 846de10c [data]

Read from L2 777de136 [data]

Read for Ownership from L2 abcde128

Write to L2 abcde128 [write-back]

Read for Ownership from L2 100de101

Write to L2 100de101 [write-back]

Read for Ownership from L2 aaade101

Write to L2 aaade101 [write-back]

Read for Ownership from L2 edcde101

Write to L2 edcde101 [write-back]

Read from L2 999de132 [data]

Read for Ownership from L2 666de135

Write to L2 666de135 [write-back]

Read for Ownership from L2 333de12c

Write to L2 333de12c [write-back]

DATA CACHE

---------------------------Way 1-------------------------------

Address: aaade101 Tag: aaa LRU: 5 MESI State: M

---------------------------Way 2-------------------------------

Address: 116de123 Tag: 116 LRU: 2 MESI State: M

---------------------------Way 3-------------------------------

Address: edcde101 Tag: edc LRU: 4 MESI State: M

---------------------------Way 4-------------------------------

Address: 999de132 Tag: 999 LRU: 3 MESI State: E

---------------------------Way 5-------------------------------

Address: 666de135 Tag: 666 LRU: 1 MESI State: M

---------------------------Way 6-------------------------------

Address: 333de12c Tag: 333 LRU: 0 MESI State: I

---------------------------Way 7-------------------------------

Address: abcde128 Tag: abc LRU: 7 MESI State: M

---------------------------Way 8-------------------------------

Address: 100de101 Tag: 100 LRU: 6 MESI State: M

INSTRUCTION CACHE

---------------------------Way 1-------------------------------

Address: 667de100 Tag: 667 LRU: 1 MESI State: E

---------------------------Way 2-------------------------------

Address: 967de100 Tag: 967 LRU: 0 MESI State: S

---------------------------Way 3-------------------------------

Address: 947de100 Tag: 947 LRU: 3 MESI State: E

---------------------------Way 4-------------------------------

Address: 967de100 Tag: 967 LRU: 2 MESI State: E

------------------Statistics Information-------------------

Data Cache Hits: 8 Data Cache Misses: 20

Data Cache Hit/Miss Ratio: 0.400000

Data Cache Reads: 18 Data Cache Writes: 10

Instruction Cache Hits: 3 Instruction Cache Misses: 5

Instruction Cache Hit/Miss Ratio: 0.600000

Instruction Cache Reads: 8

Closing Program...

aleolson@ada:~/ECE585$

# **Source Code**

// CacheController.c

//

// Authors: Alex Olson, Phong Nguyen, Ali Boshehri, Adel Alkharraz, Jennifer Lara

// Class: ECE 585

// Term: Fall 2018

// Group: 11

//

// This file contains the source code for a cache controller of a split 4-way set

// associative instruction cache and an 8-way set associative data cache.

//

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

\* DECLARATIONS

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

#include <stdlib.h>

#include <stdio.h>

#define TAG 12

#define SET 14

#define BYTE 6

#define SETMASK 0x000FFFFF

#define BYTEMASK 0x0000003F

/\* TAG SET BYTE caclulations references from addr

tag = addr >> (BYTE + SET);

set = (addr & SETMASK) >> BYTE;

byte = addr & BYTEMASK;

\*/

/\* Function declarations \*/

int parser(char \*filename);

void reset\_cache\_controller(void);

void LRU\_update(unsigned int set);

int read(unsigned int addr);

int write(unsigned int addr);

int snooping(unsigned int addr);

void print\_cache(void);

int fetch(unsigned int addr);

int invalidate(unsigned int addr);

int matching\_tag\_data(unsigned int tag);

int matching\_tag\_inst(unsigned int tag);

int search\_LRU\_data(void);

int search\_LRU\_inst(void);

int check\_for\_invalid\_MESI\_data(void);

int check\_for\_invalid\_MESI\_inst(void);

void LRU\_instruction\_update(unsigned int set);

void LRU\_data\_update(unsigned int set);

/\* Trace file operations \*/

typedef enum ops {

READ = 0, // L1 cache read

WRITE = 1, // L1 cache write

FETCH = 2, // L1 instruction fetch

INVAL = 3, // Invalidate command from L2 cache

SNOOP = 4, // Data request to L2 cache (response to snoop)

RESET = 8, // Reset cache and clear the statistics

PRINT = 9, // Print the contents of the cache

}OPS;

/\* Cache line set data \*/

typedef struct my\_cache {

unsigned int tag; // Tag bits

unsigned int LRU; // LRU bits

char MESI; // MESI bits

unsigned char data[64]; // 64 bytes of data

unsigned int address; //address

}CACHE;

/\* Keep track of the Cache hits and misses \*/

typedef struct my\_stat {

unsigned int data\_cache\_hit; // Cache hit count

unsigned int data\_cache\_miss; // Cache miss count

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

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

float data\_ratio; // Data hit/miss ration

unsigned int inst\_cache\_hit; // Cache hit count

unsigned int inst\_cache\_miss; // Cache miss count

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

float inst\_ratio; // Data hit/miss ration

}STAT;

// Global mode setting

unsigned int mode = 2; // Begin in invalid state for input check

/\* Instruction and Data Caches \*/ // NOTE: list assumption of testing a single set in final report, 16K sets would take unneeded space

CACHE data\_cache[8];

CACHE instruction\_cache[4];

//For keeping track of Stats

STAT stats;

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

\* CACHE CONTROLLER

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

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

// Check for command line argument

if (argc != 2) {

printf("\n\tERROR: No input file provided.\n\t\tUsage: ./a.out filename.txt\n");

exit(1);

}

// Initialize the caches once at the beginning

reset\_cache\_controller();

// Read in file name from command line

char \*input\_file = argv[1];

// Select a mode

printf("\n\tMode Selection\n");

printf("Mode 0: Summary of usage statistics and print contents and state of cache\n");

printf("Mode 1: Information from Mode 0 and messages to L2.\n");

do {

printf("\nMode: ");

scanf("%u", &mode);

} while (mode > 1);

// Test parsing function

if (parser(input\_file))

printf("\n\tERROR: parsing file\n");

printf("\n\n\n\tClosing Program...\n\n\n\n");

return 0;

}

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

\* FUNCTIONS

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

/\* Text file parser

\* Parses ascii data in the format of <n FFFFFFFF> where

\* n is the operation number and FFFFFFFF is the address.

\* Calls appropiate operation based on parsing result.

\*

\* Input: string of .txt trace file to parse

\* Output: pass=0, fail=nonzero

\*/

int parser(char \*filename) {

unsigned int op; // Parsed operation from input

unsigned int addr; // Parsed address from input

FILE \*fp; // .txt file pointer

// Open the file for reading

if (!(fp = fopen(filename, "r")))

printf("\n\tERROR: opening file\n");

// Read the file and decode the operation and address

while (fscanf(fp, "%d %x", &op, &addr) != EOF) {

switch(op) {

case READ:

if (read(addr))

printf("\n\tERROR: L1 data cache read");

break;

case WRITE:

if (write(addr))

printf("\n\tERROR: L1 data cache write");

break;

case FETCH:

if (fetch(addr))

printf("\n\tERROR: L1 instruction cache fetch");

break;

case INVAL:

if (invalidate(addr))

printf("\n\tERROR: L2 cache invalidate");

break;

case SNOOP:

if (snooping(addr))

printf("\n\tERROR: L2 data request from snoop");

break;

case RESET: reset\_cache\_controller();

break;

case PRINT: print\_cache();

break;

default: printf("\n\tERROR: invalid trace number\n");

return -1;

}

}

// Close the file

fclose(fp);

return 0;

}

/\* Reset the cache controller

\*

\* Input: Void

\* Output: Void

\*/

void reset\_cache\_controller(void)

{

int i;

printf("\n\t Resetting Cache Controller...\n\n");

// Clear the instruction cache

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

instruction\_cache[i].tag = 0;

instruction\_cache[i].LRU = 0;

instruction\_cache[i].MESI = 'I';

}

// Clear the data cache

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

data\_cache[i].tag = 0;

data\_cache[i].LRU = 0;

data\_cache[i].MESI = 'I';

data\_cache[i].address = 0;

}

// Reset all statistics

stats.data\_cache\_hit = 0;

stats.data\_cache\_miss = 0;

stats.data\_cache\_read = 0;

stats.data\_cache\_write = 0;

stats.data\_ratio = 0;

stats.inst\_cache\_hit = 0;

stats.inst\_cache\_miss = 0;

stats.inst\_cache\_read = 0;

stats.inst\_ratio = 0;

return;

}

/\* This function will update the instruction cache LRU

\* Anything less than current LRU will increment

\* Anything that greater than current LRU doesn't change

\* MRU: 000 LRU: 111

\*

\* Input: Line set number to start LRU comparison at

\* Output: Void

\*/

void LRU\_instruction\_update(unsigned int set)

{

int current\_LRU = instruction\_cache[set].LRU;

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

if (instruction\_cache[i].LRU <= current\_LRU)

instruction\_cache[i].LRU++;

}

instruction\_cache[set].LRU = 0;

return;

}

/\* This function will update the data cache LRU

\* Anything less than current LRU will increment

\* Anything that greater than current LRU doesn't change

\* MRU: 000 LRU: 111

\*

\* Input: Line set number to start LRU comparison at

\* Output: Void

\*/

void LRU\_data\_update(unsigned int set)

{

int current\_LRU = data\_cache[set].LRU;

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

if (data\_cache[i].LRU <= current\_LRU)

data\_cache[i].LRU++;

}

data\_cache[set].LRU = 0;

return;

}

/\* This function will attempt to read a line from the cache

\* On a cache miss, the LRU member is evicted if the cache is full

\*

\* Input: Address to read from cache

\* Output: Void

\*/

int read(unsigned int addr)

{

unsigned int tag; // Cache tag decoded from incoming address

int set = -1; // Set in the cache line

int i = 0;

stats.data\_cache\_read++;

// Cast the tag here so we can search the tag hit

tag = addr >> (BYTE + SET);

// Check for an empty set in the cache line

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

// Check for an empty set

if (data\_cache[i].tag == 0) {

set = i;

}

}

// Place the empty position

if (set >= 0) {

data\_cache[set].tag = tag;

data\_cache[set].MESI = 'E';

LRU\_data\_update(set);

data\_cache[set].address = addr;

stats.data\_cache\_miss++;

// Simulate L2 cache read

if (mode == 1)

printf("\n\tRead from L2 %x [data]", addr);

}

else { // no gap then search for hit/miss

set = matching\_tag\_data(tag); // Search for a matching tag first

if (set < 0) { // Miss

stats.data\_cache\_miss++;

// Check for a line with an invalid state to evict

set = check\_for\_invalid\_MESI\_data();

if (set < 0) { // If no invalid states, evict LRU

set = search\_LRU\_data();

if (set>=0) {

data\_cache[set].tag = tag;

data\_cache[set].MESI = 'E';

LRU\_data\_update(set);

data\_cache[set].address = addr;

}

else {

printf("LRU data is invalid");

return -1;

}

}

else { // Else, evict the invalid member

data\_cache[set].tag = tag;

data\_cache[set].MESI = 'E';

LRU\_data\_update(set);

data\_cache[set].address = addr;

}

// Simulate L2 cache read

if (mode == 1)

printf("\n\tRead from L2 %x [data]", addr);

}

else { // Hit

stats.data\_cache\_hit++;

switch (data\_cache[set].MESI) {

case 'M':data\_cache[set].tag = tag;

data\_cache[set].MESI = 'M';

LRU\_data\_update(set);

data\_cache[set].address = addr;

break;

case 'E':data\_cache[set].tag = tag;

data\_cache[set].MESI = 'S';

LRU\_data\_update(set);

data\_cache[set].address = addr;

break;

case 'S':data\_cache[set].tag = tag;

data\_cache[set].MESI = 'S';

LRU\_data\_update(set);

data\_cache[set].address = addr;

break;

case 'I':data\_cache[set].tag = tag;

data\_cache[set].MESI = 'S';

LRU\_data\_update(set);

data\_cache[set].address = addr;

break;

}

}

}

return 0;

}

/\* This function will attempt to write an address from the cache.

\* On a cache miss, the LRU member is evicted if the cache is miss.

\*

\* Input: Address to read from cache

\* Output: Void

\*/

int write(unsigned int addr)

{

unsigned int tag; // Cache tag from incoming address

int set = -1; // Set from the cache line

int i = 0;

stats.data\_cache\_write++;

// Cast the tag here so we can search the tag hit

tag = addr >> (BYTE + SET);

// Check for an empty set in the cache line

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

// Check for an empty set

if (data\_cache[i].tag == 0) {

set = i;

break;

}

}

// Place the empty position

if (set >= 0) {

data\_cache[set].tag = tag;

data\_cache[set].MESI = 'M';

data\_cache[set].address=addr;

stats.data\_cache\_miss++;

LRU\_data\_update(set);

// Simulate L2 cache write-through

if (mode == 1)

printf("\n\tWrite to L2 %x [write-through]", addr);

}

else { // no gap then search for hit/miss

set = matching\_tag\_data(tag); // Search for a matching tag first

if (set < 0) { // Miss

stats.data\_cache\_miss++;

//Simulate L2 cache RFO

if (mode == 1)

printf("\n\tRead for Ownership from L2 %x", addr);

set = check\_for\_invalid\_MESI\_data();

// If no invalid states, evict LRU

if (set < 0) {

set = search\_LRU\_data();

if (set >= 0) {

//Simulate L2 cache write-back

if (mode == 1)

printf("\n\tWrite to L2 %x [write-back]", addr);

data\_cache[set].tag = tag;

data\_cache[set].MESI = 'M';

data\_cache[set].address=addr;

LRU\_data\_update(set);

}

else {

printf("ERROR: LRU data is invalid");

return -1;

}

}

// Else, evict the invalid member

else {

data\_cache[set].tag = tag;

data\_cache[set].MESI = 'M';

data\_cache[set].address = addr;

LRU\_data\_update(set);

}

}

else { // Hit

stats.data\_cache\_hit++;

switch (data\_cache[set].MESI) {

case 'M':data\_cache[set].tag = tag;

data\_cache[set].MESI = 'M';

data\_cache[set].address = addr;

LRU\_data\_update(set);

break;

case 'E':data\_cache[set].tag = tag;

data\_cache[set].MESI = 'M';

data\_cache[set].address = addr;

LRU\_data\_update(set);

break;

case 'S':data\_cache[set].tag = tag;

data\_cache[set].MESI = 'E';

data\_cache[set].address = addr;

LRU\_data\_update(set);

break;

case 'I':data\_cache[set].tag = tag;

data\_cache[set].MESI = 'E';

data\_cache[set].address = addr;

LRU\_data\_update(set);

break;

}

}

}

return 0;

}

/\* Read an instruction in from the instruction cache

\* Note: Since the instruction cache only reads, MESI state M is not possible

\*

\* Input: address to read

\* Output: pass=0, fail=nonzero

\*/

int fetch(unsigned int addr)

{

unsigned int tag; // Cache tag decoded from incoming address

int set = -1; // Set in the cache line

int i = 0;

stats.instruction\_cache\_read++;

// Cast the tag here so we can search the tag hit

tag = addr >> (BYTE + SET);

// Check for an empty set in the cache line

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

// Check for an empty set

if (instruction\_cache[i].tag == 0) {

set = i;

}

}

// Place the empty position

if (set >= 0) {

instruction\_cache[set].tag = tag;

instruction\_cache[set].MESI = 'E';

LRU\_data\_update(set);

instruction\_cache[set].address = addr;

stats.instruction\_cache\_miss++;

// Simulate L2 cache read

if (mode == 1)

printf("\n\tRead from L2 %x [data]", addr);

}

else { // no gap then search for hit/miss

set = matching\_tag\_inst(tag); // Search for a matching tag first

if (set < 0) { // Miss

stats.data\_cache\_miss++;

// Check for a line with an invalid state to evict

set = check\_for\_invalid\_MESI\_data();

if (set < 0) { // If no invalid states, evict LRU

set = search\_LRU\_data();

if (set>=0) {

instruction\_cache[set].tag = tag;

instruction\_cache[set].MESI = 'E';

LRU\_data\_update(set);

instruction\_cache[set].address = addr;

}

else {

printf("LRU data is invalid");

return -1;

}

}

else { // Else, evict the invalid member

instruction\_cache[set].tag = tag;

instruction\_cache[set].MESI = 'E';

LRU\_data\_update(set);

instruction\_cache[set].address = addr;

}

// Simulate L2 cache read

if (mode == 1)

printf("\n\tRead from L2 %x [data]", addr);

}

else { // Hit

stats.instruction\_cache\_hit++;

switch (instruction\_cache[set].MESI) {

case 'M':instruction\_cache[set].tag = tag;

instruction\_cache[set].MESI = 'M';

LRU\_data\_update(set);

instruction\_cache[set].address = addr;

break;

case 'E':instruction\_cache[set].tag = tag;

instruction\_cache[set].MESI = 'S';

LRU\_data\_update(set);

instruction\_cache[set].address = addr;

break;

case 'S':instruction\_cache[set].tag = tag;

instruction\_cache[set].MESI = 'S';

LRU\_data\_update(set);

instruction\_cache[set].address = addr;

break;

case 'I':instruction\_cache[set].tag = tag;

instruction\_cache[set].MESI = 'S';

LRU\_data\_update(set);

instruction\_cache[set].address = addr;

break;

}

}

}

return 0;

}

/\* Invalidate an L2 command

\*

\* Input: address to invalidate

\* Output: pass=0, fail=nonzero

\*

\*/

int invalidate(unsigned int addr)

{

//Assumption that L2 is telling L1 that this address needs to be invalidated.

//Therefore we just have to set MESI bit to 'I'

unsigned int tag = addr >> (BYTE + SET);

int i;

for (i = 0; i < 8; ++i) { //goes through data cache

if (data\_cache[i].tag == tag) { //Compares to find tag of address to invalidate

switch (data\_cache[i].MESI) {

case 'M': data\_cache[i].MESI = 'I'; //changes MESI bit set to invalidate

break;

case 'E': data\_cache[i].MESI = 'I'; //changes MESI bit set to invalidate

break;

case 'S': data\_cache[i].MESI = 'I'; //changes MESI bit set to invalidate

break;

case 'I': return 0; //do nothing, already invalid

default: return -1; // Non-MESI state

}

}

}

return 0;

}

/\* Evict the invalid member

\*

\* Input: void

\* Output: error = -1 or set index of matching tag

\*/

int search\_LRU\_data(void)

{

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

if (data\_cache[i].LRU == 0x7)

return i;

}

return -1;

}

/\* Evict the invalid member

\*

\* Input: void

\* Output: error = -1 or set index of matching tag

\*/

int search\_LRU\_inst(void)

{

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

if (instruction\_cache[i].LRU == 0x3)

return i;

}

return -1;

}

/\* Compare two cache tags for equality

\* One function is used for data cahches and the other is for instruction caches

\* Input: Decoded address tag

\* Output: error=-1 or set index of matching tag

\*/

int matching\_tag\_data(unsigned int tag)

{

int i = 0;

while (data\_cache[i].tag != tag) {

i++;

if (i > 7) {

return -1;

}

}

return i;

}

int matching\_tag\_inst(unsigned int tag)

{

int i = 0;

while (instruction\_cache[i].tag != tag) {

i++;

if (i > 3) {

return -1;

}

}

return i;

}

/\* Search for Invalid MESI states within a set

\*

\* Input: void

\* Output: Index of Invalid state, negative if no invalid states

\*/

int check\_for\_invalid\_MESI\_data(void)

{

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

if (data\_cache[i].MESI == 'I')

return i;

}

return -1;

}

/\* Search for Invalid MESI states within a set

\*

\* Input: void

\* Output: Index of Invalid state, negative if no invalid states

\*/

int check\_for\_invalid\_MESI\_inst(void)

{

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

if (instruction\_cache[i].MESI == 'I')

return i;

}

return -1;

}

/\* Check the bus for other processors reading/writing addresses of interest

\*

\* Input: Address of interest

\* Output: pass=0, fail=nonzero

\*/

int snooping(unsigned int addr)

{

//Total of 4 states, Modified, Exclusive, Shared, Invalid

//Assumption that L2 is telling L1 that this address needs to be invalidated.

//Therefore we just have to set MESI bit to 'I'

unsigned int tag = addr >> (BYTE + SET);

int i = 0;

// Check the data cache

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

if(data\_cache[i].tag == tag) { //Compares to find tag of address to invalidate

switch (data\_cache[i].MESI) {

case 'M': data\_cache[i].MESI = 'I'; //changes MESI bit set to invalidate

case 'E': data\_cache[i].MESI = 'I'; //changes MESI bit set to invalidate

case 'S': data\_cache[i].MESI = 'I'; //changes MESI bit set to invalidate

case 'I': return 0; //do nothing, already invalid

}

// Simulate returning data to L2 cache

if (mode == 1)

printf("\n\tReturn data to L2 %x", addr);

}

}

return 0;

}

/\* Print the contents of the current cache lines

\*

\* Input: cache address to print cache line metadata (Tag, LRU, MESI) for

\* Output: pass=0, fail=nonzero

\*/

void print\_cache(void)

{

//Input argument later is a mode for what to print. For right now mode = 0

int i = 0; //To go through cache

printf("\n\n\n\t DATA CACHE");

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

printf("\n---------------------------Way %d-------------------------------\n",i+1);

printf(" Address: %x Tag: %x LRU: %u MESI State: %c \n",

data\_cache[i].address, data\_cache[i].tag, data\_cache[i].LRU, data\_cache[i].MESI);

}

printf("\n\n\t INSTRUCTION CACHE");

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

printf("\n---------------------------Way %d-------------------------------\n",i+1);

printf(" Address: %x Tag: %x LRU: %u MESI State: %c \n",

instruction\_cache[i].address, instruction\_cache[i].tag, instruction\_cache[i].LRU, instruction\_cache[i].MESI);

}

//Update values for cache hits and misses

stats.data\_ratio = (float)stats.data\_cache\_hit / (float)stats.data\_cache\_miss;

stats.inst\_ratio = (float)stats.inst\_cache\_hit / (float)stats.inst\_cache\_miss;

printf("\n\n------------------Statistics Information-------------------\n");

printf(" Data Cache Hits: %u Data Cache Misses: %u \tData Cache Hit/Miss Ratio: %f \n Data Cache Reads: %u Data Cache Writes: %u\n",

stats.data\_cache\_hit, stats.data\_cache\_miss, stats.data\_ratio, stats.data\_cache\_read, stats.data\_cache\_write);

printf(" Instruction Cache Hits: %u Instruction Cache Misses: %u \tInstruction Cache Hit/Miss Ratio: %f \n Instruction Cache Reads: %u\n",

stats.inst\_cache\_hit, stats.inst\_cache\_miss, stats.inst\_ratio, stats.inst\_cache\_read);

return;

}