ELECTRICAL & COMPUTER
ENGINEERING

Portland State
UNIVERSITY

ECE 585 – Microprocessor System Design
Winter 2021
FINAL PROJECT REPORT

Project Title	Split L1 Cache Design & Simulation
Prepared For	Professor Yuchen Huang
Prepared By	Josiah Sweeney Nguyen Pham Neima Kashani Melinda Van

Date of Submission	03/12/2021

Table of Contents

Introduction

Due to the great disparity between microprocessors and memory performance, caches are used to as a buffer between RAM and CPU to alleviate this gap, therefore, they are normally places nearby processor cores. A cache is a block of memory that stores copies of most frequently used information from main memory location. Whenever an application makes a request, the processor can fetch or store data in the caches instead of going all the way to main memory to retrieve the data needed. Most processors have a hierarchy of multiple cache levels: L1, L2, and L3. L1 caches are normally implemented as a split cache, with separate instruction and data caches. Data is transferred between memory and caches in a block of fixed size, called a cache line. A cache line is made up of a copy of memory data and its memory address, or a tag. The tag is used to compare with the requested memory address by the processor. If the processor finds the requested memory location in the cache, a cache hit occurs. Otherwise, it is a cache miss.

There are several placement policies to map data from memory to the cache. The simplest approach is to assign each memory block to one specific cache line. This is called direct mapping method. The best and most flexible policy is fully associative mapping in which any block of memory can be placed in any cache line. Set associative mapping is an enhanced form of direct mapping. A memory block can be mapped to any line of a specific set. When the cache is full, it follows replacement policies to evict existing data. One of the most popular policies is least recently used (LRU) to replace the least recently accessed data. Another aspect to consider when designing the cache is cache coherence to synchronize shared data stored in multiples cache levels and main memory. Snooping is one of the cache coherency schemes used to keep track of the sharing status of the copies of data across the memory system. The state of the copy of data in the cache can be noted as modified, exclusive, shared, and invalid. This snoopy scheme is knowns as MESI protocol.

To further reinforce our understanding about caches, we were assigned to design and simulate a split L1 cache using C++ programming language for a new 32-bit processor (acting processor) and can be used with up to three processors (shared processors) in the shared memory system. Both caches implement MESI protocol to maintain coherency and inclusivity by communicating with a shared L2 cache whose implementation and simulation is not in the scope of the project. However, we are required to model the interaction between L1 and L2 caches by displaying the appropriate messages for the cache operations. The L1 caches also employ LRU with counter as replacement policy.

Design Specifications

The L1 instruction cache consists of 16K sets of 4-way set associative 64-byte lines. The L1 data cache consists of 16K sets of 8-way set associative 64-byte lines. The data cache implements write allocate policy, except for the first write from memory to the cache, which is write-through. Both caches employ LRU using a counter and MESI protocols and are backed up by a shared L2 cache to maintain cache coherency and inclusivity. This can be accomplished during simulation by displaying the appropriate messages to show the communication between L1 and L2 caches.

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

In addition to those responses, the program is required to keep track and display all the key statistics of both L1 caches:

- Number of cache reads
- Number of cache writes
- Number of cache hits
- Number of cache misses
- Cache hit ratio

The simulation also supports three modes without the need of recompilation:

- Mode 0: Summary of usage statistics and print commands only
- Mode 1: Information from mode 0 and display additional messages from L2
- Mode 2: Information from previous modes and a message for every cache hit

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

n address
where *n* is

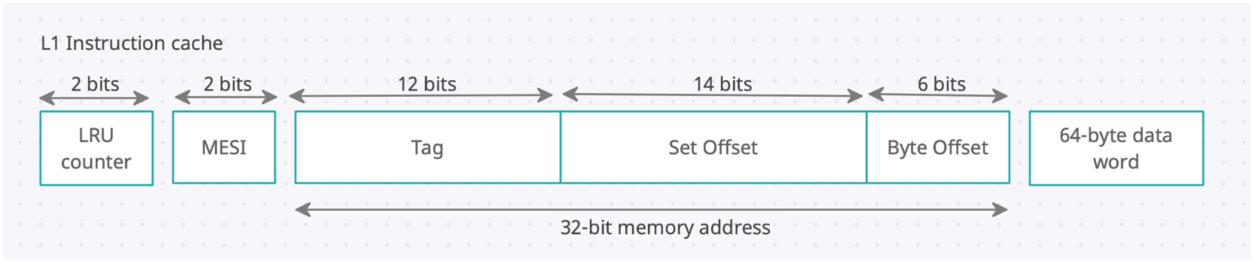
0	read data request to L1 data cache
1	write data request to L1 data cache
2	instruction fetch (a read request to L1 instruction cache)
3	invalidate command from L2
4	data request from L2 (in response to snoop)
8	clear the cache and reset all state (and statistics)

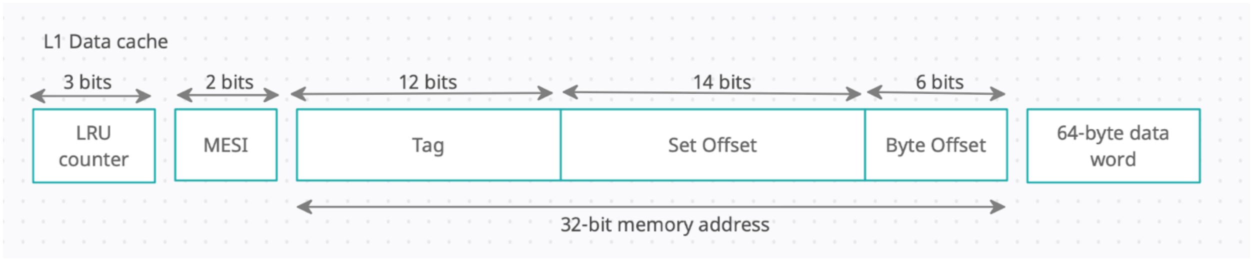
9 print contents and state of the cache (allow subsequent trace activity)

Assumption

All cache lines in the caches are initialized to invalid state every time the program is simulated. For the first time the acting processor sends a read request to the caches, that cache line is updated to exclusive state since there are no other processors share the same copy. All requesting addresses are assumed to be in the main memory system. If there is a read hit on the cache line with an E state, it is assumed that this read is from another processor, so the cache line status is updated to S. However, if the read hit is on the cache line with a M state, it is assumed that the requesting device is the acting processor, and the cache line status remains unchanged. The caches themselves do not handle multiple requests when a transaction spans to multiple cache lines and involves other processors. Those dependent operations are assumed to be held in the shared processors, and the caches are only updated once all those transactions complete. The shared L2 cache is merely a stub in this simulation and is not a part of the project development.

Cache Structure





Cache Operations

The cache operations are handled by the commands extracted from the trace file. Those function

For command 0, it is a read request to L1 data cache. If the memory address is matched with the tag of one of the cache lines, it is a hit. The copy of data is delivered from the data cache to requesting device. If it is a miss, the requesting processor needs to fetch the data from L2 and write-back data to the data cache.

Command 1 is a write request to L1 data cache. If the requesting memory location exists in the data cache, data is write back to the matching tag line. If the status of that cache line is in modified, the processor will write the content of that modified line back to L2 cache before replacing it. If the requesting address does not exist in the cache, the processor will write through to L1 and L2 caches if it is an initial write. Otherwise, the processor will evict the LRU cache line and replace the new block of memory.

Command 2 is an instruction fetch in the instruction cache. The read operation is similar to the read operation described in the data catch. However, there is no write operation allowed in this cache to avoid overwriting instructions in the system. All instructions are fetched from the memory.

Command 3 is a request sent by L2 processor to invalidate a cache line in L1 data cache. All this does is to set the requested cache line to invalid state in order to maintain coherency in the system

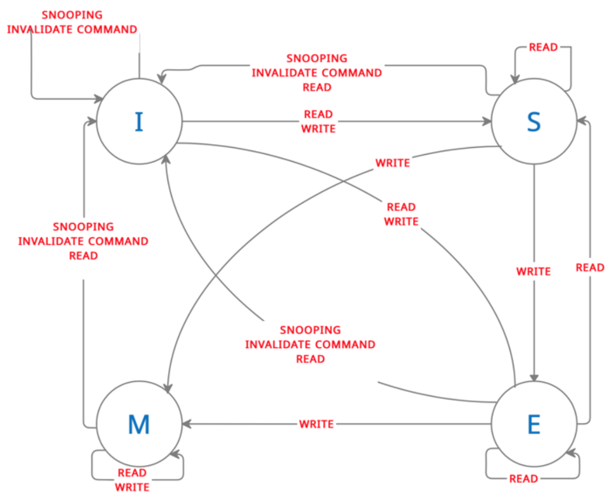
Command 4 is a read for ownership or a response from snooping. The L2 processor makes a read request of a copy of data for ownership. The copy is first written back to main memory and then delivered to the requested processor. The processor then updates the cache line in L2 cache to exclusive state and broadcasts the invalidate signal to other processors. Therefore, the cache line in L1 data cache is updated to the invalidate state. All of these transactions between L2 cache and memory system are assumed to be handle by other processors and not the acting processor. Our processor though needs to update the cache line in data cache to invalid state.

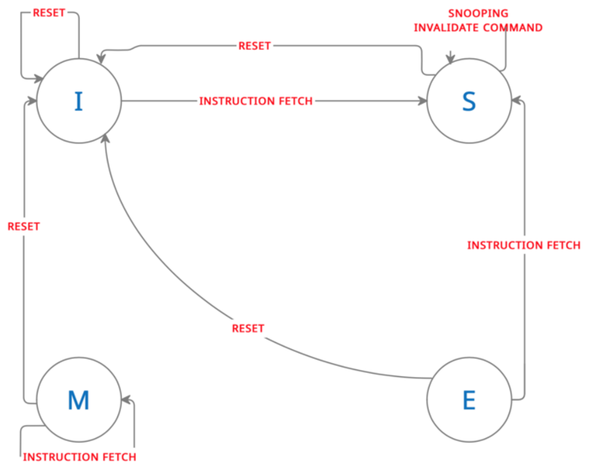
LRU Replacement Policy

For this project, the LRU protocol is implemented by adding an additional counter in front of each cache line in a set to indicate how recently an item is referenced. Whenever a copy of data in cache line is referenced, LRU counter is updated to the largest number. The LRU counters in other cache line in the set is decremented by 1 if they are greater than the previous LRU counter of the most recently used (MRU) cache line. When an eviction occurs, the data of LRU cache line is removed. The LRU counter for the data caches is 3-bit long, and the LRU counter for the instruction caches is 2-bit long. The MRU values for the data and instruction caches are 111 and 11 respectively.

MESI Protocol

The following diagrams are the state transitions described how we implemented MESI protocol in the cache design.


SNOOPING INVALIDATE COMMAND
SNOOPING INVALIDATE COMMAND READ
READ
I
READ WRITE
S
WRITE
READ WRITE
SNOOPING INVALIDATE COMMAND READ
WRITE
READ
SNOOPING INVALIDATE COMMAND READ
M
WRITE
E
READ WRITE
READ



Testing and Verifications

The program was compiled, run, and tested in Dev C++ environment after the initial coding completed. Our test plan was designed to first checked for the cache functionalities separately by creating a dedicated trace file for each operation. Once we were able to ensure the accuracy of cache operations, additional function was added to the cache handle the communication between L1 and L2 caches. Beside from generating messages between L1 and L2 caches when L2 caches was involved, we checked for the status of MESI bits of requested cache line, which should be in invalidate states in all cases. The program itself does not handle any bus snooping protocol or L2 cache implementation since it is out of the scope of this project.

Here is the list of trace files we created to test our caches.

- dataR.txt
- dataW.txt
- instructionR.txt
- invalidateCommand.txt
- dataRFO.txt
- instructionAndDataTest.txt
- allOperationsOfCaches.txt

Results

We only showed the data and statistics of our cache for the last trace file allOperationsOfCaches.txt which is also our extensive test. All commands were being tested in this trace file. The trace file and the result file can be found in Appendix B and Appendix C respectively.

Appendix A: Source code

```
/**************************************************************************************
 * cache.cpp
 *
 * Project Team 6: Josiah Sweeney, Melinda Van, Nguyen Pham, Neima Kashini
 * Class: ECE 585
 * Term: Winter 2021
 *
 * This file contains the source code for the simulation of a cache controller with a split
 * 4-way set associative instruction cache and an 8-way set associative data cache.
 *
 ***************************************************************************************/
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <string.h>
#define BYTE 6
#define SET 14
#define TAG 12
#define BYTEMASK 0x0000003F
#define SETMASK 0x000FFFFF
using namespace std;
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
    RESET = 8,                          // Reset cache and clear stats
    PRINT = 9                           // Print the contents of the cache
};
class Cache {
public:
    unsigned int tag;                    // Tag bits
    unsigned int set;                    // Set bits
    unsigned int LRU;                    // LRU bits
    char MESI = 'I';                    // MESI bits
    unsigned int address;               // Address
};
class Cache_stats {
public:
    // Data cache
    unsigned int data_cache_hit;                   // Data cache hit count
    unsigned int data_cache_miss;                   // Data 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 ratio

    // Instruction cache
    unsigned int inst_cache_hit;                  // Instruction cache hit
count
    unsigned int inst_cache_miss;            // Instruction cache miss count
    unsigned int inst_cache_read;                 // Instruction cache read
count
    float inst_ratio;                               // Instruction hit/miss
ratio
};
// Global Mode Instantiation
unsigned int mode = 3;                              // Start at invalid state
// Function declarations
int cache_read(unsigned int addr);
int cache_write(unsigned int addr);
int instruction_fetch(unsigned int addr);
int invalidate_command(unsigned int addr);
int snooping(unsigned int addr);
void reset_cache();
void print_cache();
int file_parser(char *filename);
int data_tag_match(unsigned int tag, unsigned int set);
int instruction_tag_match(unsigned int tag, unsigned int set);
int data_LRU_search(int set);
int instruction_LRU_search(int set);
void data_LRU_update(unsigned int cache_way, unsigned int cache_set,
unsigned int flag);
void instruction_LRU_update(unsigned int cache_way, unsigned int cache_set,
unsigned int flag);
// Instantiation of data and instruction caches
Cache data_cache[8][16384];
Cache instruction_cache[4][16384];
// Instantiation of stats tracker
Cache_stats stats;
/****************************************************************************************

 *                        MAIN CACHE CONTROLLER

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


    // Check for command line test file
    if (argc != 2) {
        cout << "\n\t ERROR: No input file provided.\n";
        exit(1);
    }

    // Initialize the cache at the beginning
    reset_cache();

    // Reading in file name from command line
    char *test_file = argv[1];

    // Select a mode
    cout << "\n\t Select Mode" << endl;
    cout << "Mode 0: Summary of usage statistics and print commands only" <<
endl;
    cout << "Mode 1: Information from Mode 0 with messages to L2 in
addition" << endl;
    cout << "Mode 2: Information from previous modes with information for
every cache hit" << endl;

    do {
        cout << "\nEnter Mode: ";
        scanf("%u", &mode);
    }
    while(mode > 2);

    if (file_parser(test_file)) {
        cout << "\n\t ERROR: Parsing File Failed";
    }


    cout <<"\n\n\t Testing Completed: Closing Program... \n\n\n";

    return 0;
}
/**************************************************************************************

 *                     CACHE SUBFUNCTIONS

 *************************************************************************************/
/* Test file text parser
 * Parses text data in the format of <n FFFFFFFF>
 * Where n is the operation number and FFFFFFFF is the address.
 * Calls appropriate operation based on parsing result
 *
 * Input: string of .txt test file
 * Output: pass = 0, fail != 0
 */
int file_parser(char *filename) {

    char line[1024];
    char temp_op[1];

    unsigned int operation;                         // Operation parsed from
input
    unsigned int address;                           // Address parsed from
input
    FILE *fp;                                      // .txt test file pointer

    if(!(fp = fopen(filename, "r"))) {
        cout << "\n\t ERROR: File Cannot Open" << endl;
    }

    while(fgets(line, sizeof line, fp)) {

        sscanf(line,"%c %x", temp_op, &address);
        if(line[0]=='#'||!strcmp(line,"\n")||!strcmp(line,"
")){
        }
        else {
            operation = atoi(temp_op);
            switch(operation) {
                case READ :
                    if(cache_read(address)) {
                        cout << "\n\t ERROR: L1 Data Cache Read" << endl;
                    }
                    break;

                case WRITE :
                    if(cache_write(address)) {
                        cout << "\n\t ERROR: L1 Data Cache Write" << endl;
                    }
                    break;

                case FETCH :
                    if(instruction_fetch(address)) {
                        cout << "\n\t ERROR: L1 Instruction Cache Fetch" <<
endl;
                    }
                    break;

                case INVAL :
                    if(invalidate_command(address)) {
                        cout << "\n\t ERROR: L2 Cache Invalidate Command" <<
endl;
                    }
                    break;

                case SNOOP :
                    if(snooping(address)) {
                        cout << "\n\t ERROR: L2 Snoop Data Request" << endl;
                    }
                    break;

                case RESET :
                    reset_cache();
                    break;

                case PRINT :
                    print_cache();
                    break;

                default :
                    cout << "\n\t ERROR: Invalid Command" << endl;
                    break;
            }
        }
    }
    fclose(fp);

    return 0;

}
```

```
/* L1 Data Cache Read function
 * This function will attempt to read a line from the data cache
 * On a cache miss, the LRU member is evicted if the data cache is full
 *
 * Input: Address to read from the cache
 * Output: pass = 0, fail != 0
 */
int cache_read(unsigned int addr) {

    unsigned int tag = addr >> (BYTE + SET);            // tag = address >>
(byte + set)
    unsigned int set = (addr & SETMASK) >> BYTE; //set=(address & setmask)
>> byte
    unsigned int empty_cache = 0;                       // Flag for empty
cache
    int cache_way = -1;                                 // Cache way in the cache
set

    stats.data_cache_read++;

    // Search for a matching tag
    cache_way = data_tag_match(tag, set);

    if(cache_way >= 0) {                                // Data Cache Hit


        switch (data_cache[cache_way][set].MESI) {
            case 'M' :
                stats.data_cache_hit++;
                data_cache[cache_way][set].tag = tag;
                data_cache[cache_way][set].set = set;
                data_cache[cache_way][set].MESI = 'M';
                data_cache[cache_way][set].address = addr;
                data_LRU_update(cache_way,set,empty_cache);
                // Debug Mode Message
                if(mode == 2) {
                    cout << "Data Cache Hit" << endl;
                }
                break;

            case 'E' :
                stats.data_cache_hit++;
                data_cache[cache_way][set].tag = tag;
                data_cache[cache_way][set].set = set;
                data_cache[cache_way][set].MESI = 'S';
                data_cache[cache_way][set].address = addr;
                data_LRU_update(cache_way,set,empty_cache);
                // Debug Mode Message
                if(mode == 2) {
                    cout << "Data Cache Hit" << endl;
                }
                break;

            case 'S' :
                stats.data_cache_hit++;
```

```
                    data_cache[cache_way][set].tag = tag;
                    data_cache[cache_way][set].set = set;
                    data_cache[cache_way][set].MESI = 'S';
                    data_cache[cache_way][set].address = addr;
                    data_LRU_update(cache_way,set,empty_cache);
                    // Debug Mode Message
                    if(mode == 2) {
                        cout << "Data Cache Hit" << endl;
                    }
                    break;

                case 'I' :
                    stats.data_cache_miss++;
                    data_cache[cache_way][set].tag = tag;
                    data_cache[cache_way][set].set = set;
                    data_cache[cache_way][set].MESI = 'S';
                    data_cache[cache_way][set].address = addr;
                    data_LRU_update(cache_way,set,empty_cache);
                    if (mode > 0) {
                        cout << "Data Cache Miss: Read from L2 " << hex << addr
<< " [Data]" << endl;
                    }
                    break;
            }
        }
        else {                                              // Data Cache Miss
            stats.data_cache_miss++;

            // Check for an empty set in the cache line
            for (int i = 0; cache_way < 0 && i < 8; ++i) {
                if (data_cache[i][set].tag == 4096) {
                    cache_way = i;
                    empty_cache = 1;
                }
            }

            // Place if empty
            if (cache_way >= 0) {
                data_cache[cache_way][set].tag = tag;
                data_cache[cache_way][set].set = set;
                data_cache[cache_way][set].MESI = 'E';
                data_cache[cache_way][set].address = addr;
                // LRU Data Update
                data_LRU_update(cache_way,set,empty_cache);
                // Simulate L2 Cache Read
                if (mode > 0) {
                    cout << "Data Cache Miss: Read from L2 " << hex << addr << "
[Data]" << endl;
                }
            }
            else {
                if (mode > 0) {
                    cout << "Data Cache Miss: Read from L2 " << hex << addr << "
[Data] and evict " << data_cache[cache_way][set].address << endl;
                }
                // Check for a line with invalid MESI data to evict
                for (int j = 0; j < 8; ++j) {
                    if(data_cache[j][set].MESI == 'I') {
                        cache_way = j;
                    }
                    else {
                        cache_way = -1;
                    }
                }
                if (cache_way < 0) {
                    // Search for smallest LRU
                    cache_way = data_LRU_search(set);
                    if (cache_way >=0) {
                        data_cache[cache_way][set].tag = tag;
                        data_cache[cache_way][set].set = set;
                        data_cache[cache_way][set].MESI = 'E';
                        data_cache[cache_way][set].address = addr;
                        data_LRU_update(cache_way,set,empty_cache);
                    }
                    else {
                        cout << "LRU data is invalid" << endl;
                        return -1;
                    }
                }
                else {                    // Else, the invalid member is evicted
                    data_cache[cache_way][set].tag = tag;
                    data_cache[cache_way][set].set = set;
                    data_cache[cache_way][set].MESI = 'E';
                    data_cache[cache_way][set].address = addr;
                    data_LRU_update(cache_way,set,empty_cache);
                }

            }
        }
        return 0;
    }
    /* L1 Data Cache Write function
     * This function will attempt to write an address from the data cache
     * On a cache miss, the LRU member is evicted if the data cache is a miss
     *
     * Input: Address to read from the cache
     * Output: pass = 0, fail != 0
     */
    int cache_write(unsigned int addr) {

        unsigned int tag = addr >> (BYTE + SET);     // tag = address >> (byte +
    set)
        unsigned int set = (addr & SETMASK) >> BYTE; // set=(address &
    setmask)>> byte
        unsigned int empty_cache = 0;                 // Flag for empty cache
        int cache_way = -1;                           // Cache way in the cache
    set

        stats.data_cache_write++;
```

```
// Search for a matching tag
cache_way = data_tag_match(tag, set);

if(cache_way >= 0) {                                    // Data Cache Hit

    switch (data_cache[cache_way][set].MESI) {
        case 'M' :
            stats.data_cache_hit++;
            data_cache[cache_way][set].tag = tag;
            data_cache[cache_way][set].set = set;
            data_cache[cache_way][set].MESI = 'M';
            data_cache[cache_way][set].address = addr;
            data_LRU_update(cache_way,set,empty_cache);
            // Debug Message
            if (mode == 2) {
                cout << "Data Cache Hit" << endl;
            }
            break;

        case 'E' :
            stats.data_cache_hit++;
            data_cache[cache_way][set].tag = tag;
            data_cache[cache_way][set].set = set;
            data_cache[cache_way][set].MESI = 'M';
            data_cache[cache_way][set].address = addr;
            data_LRU_update(cache_way,set,empty_cache);
            // Debug Message
            if (mode == 2) {
                cout << "Data Cache Hit" << endl;
            }
            break;

        case 'S' :
            stats.data_cache_hit++;
            data_cache[cache_way][set].tag = tag;
            data_cache[cache_way][set].set = set;
            data_cache[cache_way][set].MESI = 'E';
            data_cache[cache_way][set].address = addr;
            data_LRU_update(cache_way,set,empty_cache);
            // Debug Message
            if (mode == 2) {
                cout << "Data Cache Hit" << endl;
            }
            break;

        case 'I' :
            stats.data_cache_miss++;
            data_cache[cache_way][set].tag = tag;
            data_cache[cache_way][set].set = set;
            data_cache[cache_way][set].MESI = 'E';
            data_cache[cache_way][set].address = addr;
            data_LRU_update(cache_way,set,empty_cache);
            break;
    }
}
```

```
    else {                                                    // Data Cache Miss
        stats.data_cache_miss++;
        // Simulate L2 Cache RFO message
        if (mode > 0) {
            cout << "Read for Ownership from L2 " << hex << addr << endl;
        }

        // Check for an empty set in the cache line
        for (int i = 0; cache_way < 0 && i < 8; ++i) {
            if (data_cache[i][set].tag == 4096) {
                cache_way = i;
                empty_cache = 1;
            }
        }

        // Place if empty
        if (cache_way >= 0) {
            data_cache[cache_way][set].tag = tag;
            data_cache[cache_way][set].set = set;
            data_cache[cache_way][set].MESI = 'M';
            data_cache[cache_way][set].address = addr;
            // LRU Data Update
            data_LRU_update(cache_way,set,empty_cache);
            // Simulate L2 cache write-through message
            if (mode > 0) {
                cout << "Data Cache Miss: Write to L2 " << hex << addr << "
[Write-Through]" << endl;
            }
        }
        else {
            if (mode > 0) {
                cout << "Data Cache Miss: Write to L2 " << hex << addr << "
[Data] and evict " << data_cache[cache_way][set].address << endl;
            }
            // Check for a line with invalid MESI data to evict
            for (int j = 0; j < 8; ++j) {
                if(data_cache[j][set].MESI == 'I') {
                    cache_way = j;
                }
                else {
                    cache_way = -1;
                }
            }
            if (cache_way < 0) {
                // Search for smallest LRU
                cache_way = data_LRU_search(set);
                if (cache_way >=0) {
                    data_cache[cache_way][set].tag = tag;
                    data_cache[cache_way][set].set = set;
                    data_cache[cache_way][set].MESI = 'M';
                    data_cache[cache_way][set].address = addr;
                    data_LRU_update(cache_way,set,empty_cache);
                }
                else {
                    cout << "LRU data is invalid" << endl;
                    return -1;
                }
            }
            else {        // Else, the invalid member is evicted
                data_cache[cache_way][set].tag = tag;
                data_cache[cache_way][set].set = set;
                data_cache[cache_way][set].MESI = 'M';
                data_cache[cache_way][set].address = addr;
                data_LRU_update(cache_way,set,empty_cache);
            }

        }
    }
    return 0;
}
/* L1 Instruction Cache Read function
 * This function will attempt to read a line from the instruction cache
 * On a cache miss, the LRU member is evicted if the instruction cache is
full
 *
 * Input: Address to read from the cache
 * Output: pass = 0, fail != 0
 */
int instruction_fetch(unsigned int addr) {


    unsigned int tag = addr >> (BYTE + SET);  // tag = address >> (byte +
set)
    unsigned int set = (addr & SETMASK) >> BYTE; // set=(address &
setmask)>> byte
    unsigned int empty_cache = 0;                           // Flag for empty
cache
    int cache_way = -1;                                      // Cache way in
the cache set

    stats.inst_cache_read++;

    // Search for a matching tag
    cache_way = instruction_tag_match(tag, set);    // Search for a missing
tag

    if(cache_way >= 0) {                                // Instruction Cache
Hit

        switch (instruction_cache[cache_way][set].MESI) {
            case 'M' :
                stats.inst_cache_hit++;
                instruction_cache[cache_way][set].tag = tag;
                instruction_cache[cache_way][set].set = set;
                instruction_cache[cache_way][set].MESI = 'M';
                instruction_cache[cache_way][set].address = addr;
                instruction_LRU_update(cache_way,set,empty_cache);
                // Debug Mode Message
                if (mode == 2) {
                    cout << "Instruction Cache Hit" << endl;
                }
                break;

            case 'E' :
                stats.inst_cache_hit++;
                instruction_cache[cache_way][set].tag = tag;
                instruction_cache[cache_way][set].set = set;
                instruction_cache[cache_way][set].MESI = 'S';
                instruction_cache[cache_way][set].address = addr;
                instruction_LRU_update(cache_way,set,empty_cache);
                // Debug Mode Message
                if (mode == 2) {
                    cout << "Instruction Cache Hit" << endl;
                }
                break;

            case 'S' :
                stats.inst_cache_hit++;
                instruction_cache[cache_way][set].tag = tag;
                instruction_cache[cache_way][set].set = set;
                instruction_cache[cache_way][set].MESI = 'S';
                instruction_cache[cache_way][set].address = addr;
                instruction_LRU_update(cache_way,set,empty_cache);
                // Debug Mode Message
                if (mode == 2) {
                    cout << "Instruction Cache Hit" << endl;
                }
                break;

            case 'I' :
                stats.inst_cache_miss++;
                instruction_cache[cache_way][set].tag = tag;
                instruction_cache[cache_way][set].set = set;
                instruction_cache[cache_way][set].MESI = 'S';
                instruction_cache[cache_way][set].address = addr;
                instruction_LRU_update(cache_way,set,empty_cache);
                // Simulate L2 Instruction Cache Read
                if(mode > 0) {
                    cout << "Instruction Cache Miss: Read from L2 " << hex
<< addr << " [Instruction]" << endl;
                }
                break;
        }
    }
    else {              // Instruction Cache Miss
        stats.inst_cache_miss++;

        // Check  for an empty set in the cache line
        for (int i = 0; cache_way < 0 && i < 4; ++i) {
            if (instruction_cache[i][set].tag == 4096) {
                cache_way = i;
                empty_cache = 1;
            }
        }
        if (cache_way >= 0) {
            instruction_cache[cache_way][set].tag = tag;
            instruction_cache[cache_way][set].set = set;
            instruction_cache[cache_way][set].MESI = 'E';
            instruction_cache[cache_way][set].address = addr;
            // LRU instruction Update
            instruction_LRU_update(cache_way,set,empty_cache);
            // Simulate L2 Instruction Cache Read
            if(mode > 0) {
                cout << "Instruction Cache Miss: Read from L2 " << hex <<
addr << " [Instruction]" << endl;
            }
        }
        else {
            if (mode > 0) {
                cout << "Instruction Cache Miss: Read from L2 " << hex <<
addr << " [Instruction] and evict " <<
instruction_cache[cache_way][set].address << endl;
            }
            // Check for a line with invalid MESI data to evict
            for (int j = 0; j < 4; ++j) {
                if(instruction_cache[j][set].MESI == 'I') {
                    cache_way = j;
                }
                else {
                    cache_way = -1;
                }
            }
            if (cache_way < 0) {
                cache_way = instruction_LRU_search(set);
                if (cache_way >=0) {
                    instruction_cache[cache_way][set].tag = tag;
                    instruction_cache[cache_way][set].set = set;
                    instruction_cache[cache_way][set].MESI = 'E';
                    instruction_cache[cache_way][set].address = addr;
                    instruction_LRU_update(cache_way,set,empty_cache);
                }
                else {
                    cout << "LRU data is invalid" << endl;
                    return -1;
                }
            }
            else {                      // Else, the invalid member is evicted
                instruction_cache[cache_way][set].tag = tag;
                instruction_cache[cache_way][set].set = set;
                instruction_cache[cache_way][set].MESI = 'E';
                instruction_cache[cache_way][set].address = addr;
                instruction_LRU_update(cache_way,set,empty_cache);
            }
        }
    }
    return 0;


}
/* L2 Invalidate Command function
```

```
 * This function will simulate an L2 Invalidate command
 * This will have the MESI bit of the input address set to 'I'
 *
 * Input: Address to invalidate
 * Output: pass = 0, fail != 0
 */
int invalidate_command(unsigned int addr) {

    unsigned int tag = addr >> (BYTE + SET);   // tag = address >> (byte + 
set)
    unsigned int set = (addr & SETMASK) >> BYTE; // set=(address & 
setmask)>> byte


    // Search data cache for a matching tag
    for (int i = 0; i < 8; ++i) {
        if (data_cache[i][set].tag == tag) {
            switch (data_cache[i][set].MESI) {
                case 'M':
                    data_cache[i][set].tag = tag;
                    data_cache[i][set].set = set;
                    data_cache[i][set].MESI = 'I';   // Change MESI bit to 
Invalid
                    data_cache[i][set].address = addr;
                    break;

                case 'E':
                    data_cache[i][set].tag = tag;
                    data_cache[i][set].set = set;
                    data_cache[i][set].MESI = 'I';   // Change MESI bit to 
Invalid
                    data_cache[i][set].address = addr;
                    break;

                case 'S':
                    data_cache[i][set].tag = tag;
                    data_cache[i][set].set = set;
                    data_cache[i][set].MESI = 'I';   // Change MESI bit to 
Invalid
                    data_cache[i][set].address = addr;
                    break;

                case 'I':
                    data_cache[i][set].tag = tag;
                    data_cache[i][set].set = set;
                    data_cache[i][set].address = addr;
                    break;           // Do nothing as state is already 
invalid

                default:
                    return -1;              // Non-MESI state recorded. 
ERROR
            }
        }
    }
    return 0;
}
/* L2 Snooping function
 * This function will simulate an L2 Snoop
 * This function assumes that L2 is telling L1 that the address needs to be
invalidated
 * Therefore the MESI bit will be set to 'I'
 *
 * Input: Address to "snoop"
 * Output: pass = 0, fail != 0
 */
int snooping(unsigned int addr) {

    unsigned int tag = addr >> (BYTE + SET);   // tag = address >> (byte +
set)
    unsigned int set = (addr & SETMASK) >> BYTE;  //set=(address &
setmask)>> byte

    // Search data cache for a matching tag
    for (int i = 0; i < 8; ++i) {
        if (data_cache[i][set].tag == tag) {
            switch (data_cache[i][set].MESI) {
                case 'M':
                    data_cache[i][set].tag = tag;
                    data_cache[i][set].set = set;
                    data_cache[i][set].MESI = 'I';  // Change MESI bit to
Invalid
                    data_cache[i][set].address = addr;
                    break;

                case 'E':
                    data_cache[i][set].tag = tag;
                    data_cache[i][set].set = set;
                    data_cache[i][set].MESI = 'I';  // Change MESI bit to
Invalid
                    data_cache[i][set].address = addr;
                    break;

                case 'S':
                    data_cache[i][set].tag = tag;
                    data_cache[i][set].set = set;
                    data_cache[i][set].MESI = 'I';   // Change MESI bit to
Invalid
                    data_cache[i][set].address = addr;
                    break;

                case 'I':
                    data_cache[i][set].tag = tag;
                    data_cache[i][set].set = set;
                    data_cache[i][set].address = addr;
                    break;          // Do nothing as state is already
invalid

                default:
                    return -1;          // Non-MESI state recorded. ERROR
            }

            if (mode > 0) {
                cout << "Return data to L2 " << hex << addr << endl;
            }
        }
    }
    return 0;

}
/* Cache Reset function
 * This function resets the cache controller and clears statistics
 *
 * Input: Void
 * Output: Void
 */
void reset_cache() {

    cout << "Resetting the Cache Controller and Clearing Stats..." << endl;

    // Clearing the data cache
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 16384; ++j) {
            data_cache[i][j].tag = 4096;
            data_cache[i][j].set = 0;
            data_cache[i][j].LRU = 0;
            data_cache[i][j].MESI = 'I';
            data_cache[i][j].address = 0;
        }
    }


    // Clearing the instruction cache
    for (int k = 0; k < 4; ++k) {
        for (int l = 0; l < 16384; ++l) {
            instruction_cache[k][l].tag = 4096;
            instruction_cache[k][l].set = 0;
            instruction_cache[k][l].LRU = 0;
            instruction_cache[k][l].MESI = 'I';
            instruction_cache[k][l].address = 0;
        }
    }

    // Resetting all stats for data cache
    stats.data_cache_hit = 0;
    stats.data_cache_miss = 0;
    stats.data_cache_read = 0;
    stats.data_cache_write = 0;
    stats.data_ratio = 0.0;

    // Resetting all stats for instruction cache
    stats.inst_cache_hit = 0;
    stats.inst_cache_miss = 0;
    stats.inst_cache_read = 0;
    stats.inst_ratio = 0.0;
    return;

}
/* Print Cache function
 * This function print information about the caches and their statistics
 *
 * Input: Void
 * Output: Void
 */
void print_cache() {

    // Setting flag for printing
    int set_flag = 0;

    // Update values for cache hits/misses ratio
    stats.data_ratio = 
float(stats.data_cache_hit)/(float(stats.data_cache_miss)+float(stats.data_c
ache_hit));
    stats.inst_ratio = 
float(stats.inst_cache_hit)/(float(stats.inst_cache_miss)+float(stats.inst_c
ache_hit));

    // Printing information for Data Cache
    cout << "\n\n\t\t\tDATA CACHE" << endl;
    for (int i = 0; i < 16384; ++i) {
        for (int j = 0; j < 8; ++j) {
            if(data_cache[j][i].address > 0) {
                if(set_flag == 0){
                    cout << "Set No: " << hex << i << endl;
                    set_flag = 1;
                }
                cout << "Way No: " << dec << j << endl;
                cout << "Address: " << hex << data_cache[j][i].address << "
Tag: " << data_cache[j][i].tag << " Set: "
                << data_cache[j][i].set << " LRU: " << data_cache[j][i].LRU
<< " MESI State: " << data_cache[j][i].MESI << endl;
            }
        }
        set_flag = 0;
    }

    // Printing information for Instruction Cache
    cout << "\n\n\t\t\tINSTRUCTION CACHE" << endl;
    for (int k = 0; k < 16384; ++k) {
        for (int l = 0; l < 4; ++l) {
            if(instruction_cache[l][k].address > 0) {
                if(set_flag == 0){
                    cout << "Set No: " << hex << k << endl;
                    set_flag = 1;
                }
                cout << "Way No: " << dec << l << endl;
                cout << "Address: " << hex << 
instruction_cache[l][k].address << " Tag: " << instruction_cache[l][k].tag
<< " Set: "
```

```
                << instruction_cache[l][k].set << " LRU: " <<
instruction_cache[l][k].LRU << " MESI State: " <<
instruction_cache[l][k].MESI << endl;
            }
        }
        set_flag = 0;
    }

    // Printing Cache Statistics
    cout << "\n\n\t\t\tCACHE STATISTICS" << endl;
    if(stats.data_cache_miss == 0){
        cout << "No Data Cache Transactions Occured" << endl;
    }
    else {
        cout << "\nDATA CACHE:" << endl;
        cout << "Data Cache Hits: " << dec << stats.data_cache_hit <<
"\nData Cache Misses: " << stats.data_cache_miss << "\nData Cache Hit/Miss
Ratio: "
        << stats.data_ratio << "\nData Cache Reads: " <<
stats.data_cache_read << "\nData Cache Writes: " << stats.data_cache_write
<< endl;
    }

    if(stats.inst_cache_miss == 0) {
        cout << "No Instruction Cache Transactions Occured" << endl;
    }
    else {
        cout << "\nINSTRUCTION CACHE:" << endl;
        cout << "Instruction Cache Hits: " << stats.inst_cache_hit <<
"\nInstruction Cache Misses: " << stats.inst_cache_miss << "\nInstruction
Cache Hit/Miss Ratio: "
        << stats.inst_ratio << "\nInstruction Cache Reads: " <<
stats.inst_cache_read << endl;
    }
}
/* Data Cache LRU Update function
 * This function updates the data cache LRU
 *
 * Input: cache_way and cache_set values, flag for whether there is empty
space in the cache
 * Output: Void. Updates LRU values for a given set in the global data cache
class
 */
void data_LRU_update(unsigned int cache_way, unsigned int cache_set,
unsigned int flag) {
    int LRU_current = data_cache[cache_way][cache_set].LRU;

    // If a way is empty, flag = 1
    if(flag == 1){
        for (int i = 0; i < cache_way; ++i){
            --data_cache[i][cache_set].LRU;
        }
    }
    // If a way is NOT empty
    else {
        for (int i = 0; i < 8; ++i) {
            if(LRU_current > data_cache[i][cache_set].LRU) {
                data_cache[i][cache_set].LRU = data_cache[i][cache_set].LRU;
            }
            else
            {
                --data_cache[i][cache_set].LRU;
            }
        }
    }
    data_cache[cache_way][cache_set].LRU = 0x7;
}
/* Instruction Cache LRU Update function
 * This function updates the data cache LRU
 *
 * Input: cache_way and cache_set values, flag for whether there is empty
space in the cache
 * Output: Void. Updates LRU values for a given set in the global
instruction cache class
 */
void instruction_LRU_update(unsigned int cache_way, unsigned int cache_set,
unsigned int flag) {
    int LRU_current = instruction_cache[cache_way][cache_set].LRU;

    // If a way is empty, flag = 1
    if(flag == 1){
        for (int i = 0; i < cache_way; ++i){
            --instruction_cache[i][cache_set].LRU;
        }
    }
    // If a way is NOT empty
    else {
        for (int i = 0; i < 4; ++i) {
            if(LRU_current >= instruction_cache[i][cache_set].LRU) {
                instruction_cache[i][cache_set].LRU =
instruction_cache[i][cache_set].LRU;
            }
            else
            {
                --instruction_cache[i][cache_set].LRU;
            }
        }
    }
    instruction_cache[cache_way][cache_set].LRU = 0x3;

}
/* Data Tag matching function
 * This function compares two data cache tags for equality
 *
 * Input: Tag, Set
 * Output: Void
 */
int data_tag_match(unsigned int tag, unsigned int set) {

    int i = 0;
    // Check for matching tags;
    while (data_cache[i][set].tag != tag) {
        i++;
        if (i > 7) {
            return -1;
        }
    }
    return i;
}
/* Instruction Tag matching function
 * This function compares two instruction cache tags for equality
 *
 * Input: Tag, set
 * Output: Void
 */
int instruction_tag_match(unsigned int tag, unsigned int set) {

    int i = 0;
    // Check for matching tags;
    while (instruction_cache[i][set].tag != tag) {
        i++;
        if (i > 3) {
            return -1;
        }
    }
    return i;

}
/* Data LRU Evict Search function
 * This function finds the member of the cache to resets the cache
controller and clears statistics
 *
 * Input: Set
 * Output: Void
 */
int data_LRU_search(int set) {
    for (int i = 0; i < 8; ++i) {
        if (data_cache[i][set].LRU == 0) {
            return i;
        }
    }
    return -1;
}
/* Cache Reset function
 * This function resets the cache controller and clears statistics
 * Input: Set
 * Output: Void
 */
int instruction_LRU_search(int set) {
    for (int i = 0; i < 4; ++i) {
        if (instruction_cache[i][set].LRU == 0)
            return i;
    }
    return -1;
}
```

Appendix B: Trace File

“allOperationsOfCaches.txt”

```
0 11145678
0 32145678
0 39845674
1 abc45674
1 48545678
0 32145672
0 11145674
1 39845677
1 cde4ef7a
1 abc45677
2 11145678
2 32145678
2 39845674
2 83445674
3 11145678
2 83445674
2 3984567a
3 abc45674
4 32145678
4 cde4ef7a
0 7d345678
1 fff4565e
9
8
9
```

Appendix C: Test Result

```
Resetting the Cache Controller and Clearing Stats...

      Select Mode
Mode 0: Summary of usage statistics and print commands only
Mode 1: Information from Mode 0 with messages to L2 in addition
Mode 2: Information from previous modes with information for every cache hit

Enter Mode:2
Data Cache Miss: Read from L2 11145678 [Data]
Data Cache Miss: Read from L2 32145678 [Data]
Data Cache Miss: Read from L2 39845674 [Data]
Read for Ownership from L2 abc45674
Data Cache Miss: Write to L2 abc45674 [Write-Through]
Read for Ownership from L2 48545678
Data Cache Miss: Write to L2 48545678 [Write-Through]
Data Cache Hit
Data Cache Hit
Data Cache Hit
Read for Ownership from L2 cde4ef7a
Data Cache Miss: Write to L2 cde4ef7a [Write-Through]
Data Cache Hit
Instruction Cache Miss: Read from L2 11145678 [Instruction]
Instruction Cache Miss: Read from L2 32145678 [Instruction]
Instruction Cache Miss: Read from L2 39845674 [Instruction]
Instruction Cache Miss: Read from L2 83445674 [Instruction]
Instruction Cache Hit
Instruction Cache Hit
Return data to L2 32145678
Return data to L2 cde4ef7a
Data Cache Miss: Read from L2 7d345678 [Data]
Read for Ownership from L2 fff4565e
Data Cache Miss: Write to L2 fff4565e [Write-Through]


                    DATA CACHE
Set No: 1159
Way No: 0
Address: 11145678 Tag: 111 Set: 1159 LRU: 3 MESI State: I
Way No: 1
Address: 32145678 Tag: 321 Set: 1159 LRU: 2 MESI State: I
Way No: 2
Address: 39845677 Tag: 398 Set: 1159 LRU: 4 MESI State: M
Way No: 3
Address: abc45674 Tag: abc Set: 1159 LRU: 5 MESI State: I
Way No: 4
Address: 48545678 Tag: 485 Set: 1159 LRU: 1 MESI State: M
Way No: 5
Address: 7d345678 Tag: 7d3 Set: 1159 LRU: 6 MESI State: E
Way No: 6
Address: fff4565e Tag: fff Set: 1159 LRU: 7 MESI State: M
Set No: 13bd
Way No: 0
```

```
Address: cde4ef7a Tag: cde Set: 13bd LRU: 7 MESI State: I


                    INSTRUCTION CACHE
Set No: 1159
Way No: 0
Address: 11145678 Tag: 111 Set: 1159 LRU: 0 MESI State: E
Way No: 1
Address: 32145678 Tag: 321 Set: 1159 LRU: 1 MESI State: E
Way No: 2
Address: 3984567a Tag: 398 Set: 1159 LRU: 3 MESI State: S
Way No: 3
Address: 83445674 Tag: 834 Set: 1159 LRU: 2 MESI State: S


                    CACHE STATISTICS

DATA CACHE:
Data Cache Hits: 4
Data Cache Misses: 8
Data Cache Hit/Miss Ratio: 0.333333
Data Cache Reads: 6
Data Cache Writes: 6

INSTRUCTION CACHE:
Instruction Cache Hits: 2
Instruction Cache Misses: 4
Instruction Cache Hit/Miss Ratio: 0.333333
Instruction Cache Reads: 6


                    DATA CACHE


                    INSTRUCTION CACHE


                    CACHE STATISTICS
No Data Cache Transactions Occured
No Instruction Cache Transactions Occured


Testing Completed: Closing Program...
```