University of Kansas

Department of Electrical Engineering & Computer Science

Technical Report

Cache Simulator

Dilesh Fernando

Cheng-Yeh Lee

Vuong Nguyen

13th April 2017

Revision History

1. Report Date: April 2017
2. Report Type: Final Technical
3. Data Recovered: April 2017
4. Title and Subtitle: Report of EECS 645 - Group 11 on Cache Simulator project
5. Authors: Dilesh Fernando, Cheng-Yeh Lee, Vuong Nguyen
6. Performing Organization: University of Kansas - Department of Electrical Engineering & Computer Science
7. Performing Organization Report Number: N/A
8. Distribution availability Statement: Approved for public release.
9. Number of pages: 18

Abstract

This project describes design and implementation of a L1 Cache Simulator to a gain better understanding of how L1 cache operates in a modern computer. This program simulates the read from cache memory behavior. Cache Simulator is implemented by using C programming language in a Linux environment. Cache Simulator calculates cache hit ration in relation to cache associativity with random replacement policy for cache associativity two and above.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

In this project, basic cache simulator is implemented using C programming language. To simulate cache mechanism in a CPU, Cache Simulator program will take trace address file and cache associativity as inputs and calculate cache metrics, such as hit rate, miss rate, and cache hit ratio.

Chapter 2

Overview of Cache Memory

Cache memory is a small amount of fast memory nearest to the CPU that stores all the recent memory access blocks from main memory. When the CPU tries to read from main memory, first the address is sent to a cache controller to check if it was saved in cache for faster access. If the block was saved in cache, then that particular block is sent to the CPU rather than accessing main memory. Therefore, cache memory has less access time and faster than main memory access.

To determine if the memory block is in cache, the memory address is divided into three sections. The low bits represent the block offset, middle bits represent the line number in cache and the high bits represent tag value. The number of bits in each section will be determined by the cache associativity and block size, which are constants for a particular implementation of a cache system.

Following is a diagram illustrates a 32 bit memory address with direct cache memory system.

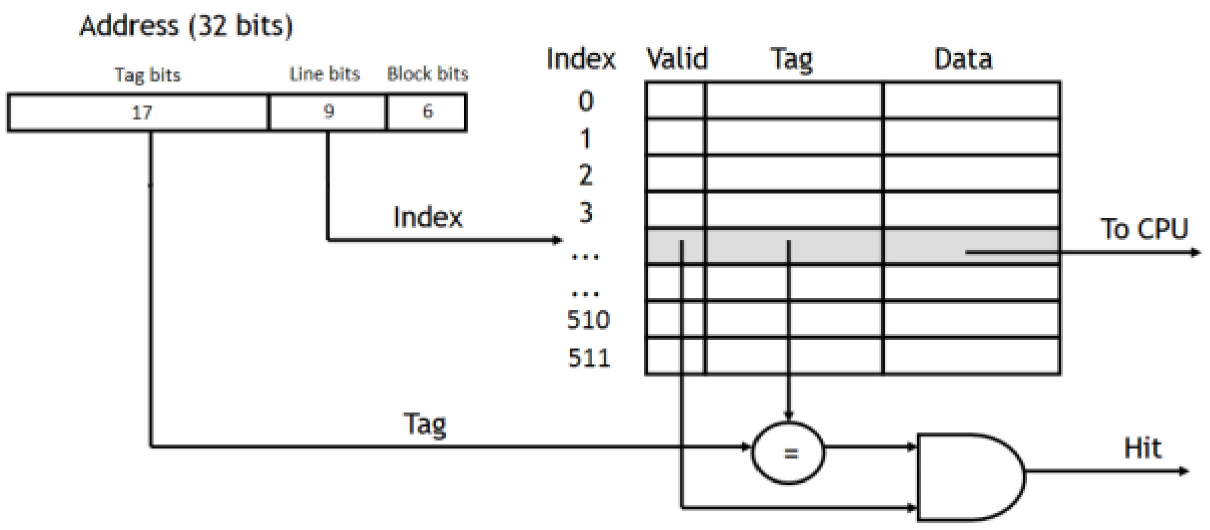

Figure 2.1: Cache Memory Overview

Overview of the Cache Simulator

In this project, Cache Simulator is built to simulate hardware cache memory system. To implement the Cache Simulator following steps are implemented.

- User to enter the associativity and the name of an address trace input file at command line.
- Setup the data structure to accommodate cache memory.
- Read data file contains trace addresses which simulates CPU generating address in a real system.
- Cache Simulator determines a cache hit or miss.
- If a cache miss occurs, replace the cache block with new data by using a replacement policy.
- Keep an account of hits and misses.
- After all the data is consumed by the Cache Simulator, calculate the cache ratio.
- Output results to the console.

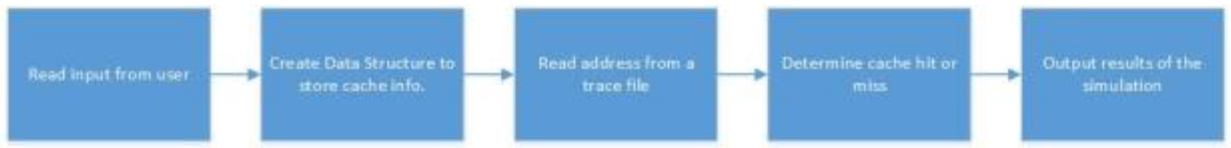
Read input from user
Create Data Structure to store cache info.
Read address from a trace file
Determine cache hit or miss
Output results of the simulation

Figure 2.2: Block Diagram of Cache Simulator

Cache Simulator in Detail

Following steps further illustrates the design approach in building the Cache Simulator.

Command line input from user:

Cache Simulator takes three arguments as inputs from the user. It follows the format shown below.

$ ./CacheSimulator <cache associativity> <name of data file>.

- The first argument is the name of the program.
- The second argument is the number of cache associativity.
- The third argument is the name of the data file.

For example if the user wants to simulate a 1-way set associative cache on the binary file named AddressTrace_FirstIndex.bin, enter the following command.

```
$ ./CacheSimulator 1 AddressTrace_FirstIndex.bin
```

Check the command line inputs are correct:

Cache Simulator uses conditional statements to validate correct inputs from the user. In the event of invalid command line arguments, Cache Simulator prints an error message and suggestion message for correct arguments to the program.

Simulator accepts three arguments as mentioned in the previous section and for cache associativity 1, 2, 4, or 8 are the only valid arguments.

Read associativity and the data file from the user:

After validating the command line arguments, program sets the cache associativity exponential in the following manner:

- If 1-way associativity is given, the program sets the cache associativity exponential to 0.
- If 2-way associativity is given, it sets the cache associativity exponential to 1.
- If 4-way associativity is given, it sets the cache associativity exponential to 2.
- If 8-way associativity is given, it sets the cache associativity exponential to 3.

If a valid file name is provided, the program stores in a variable for later use.

Calculate tag/line size and create array:

Line size and the tag size are computed using the parameters provide to the program.

Line size and cache associativity are used to create a 2-Dimensional array that mimics the cache. The values of the tag and valid bit in the array are initialized to 0.

Open the data file for reading:

The program opens the data file for read only in binary mode and seek to the beginning of the file.

Simulate cache mechanism:

The program uses while loop to read data from the file. Each address is read sizeof(int) which is 4 bytes. Inside the main loop program calculates the line number and the tag by bit-shifting the address value read from file. An integer variable is created to use as a flag that indicates cache hit.

Cache hit or miss:

Inside the main loop, two for loops are utilize to determine cache hit or miss.

The first for loop is to check for cache hit. If the block in the cache is valid and the tag matches the tag that program calculates using the address read from the data file, then it is a hit. The cache hit counter is then increased and the program sets the flag to 1 which indicates the address has been found in the cache.

If the address is not found in the cache, program will execute the second for loop. There is another integer variable that acts as a flag to indicate the cache has been updated. Inside the second for loop, if the block in the cache is invalid and empty, the program increases the cache miss counter and updates the block’s tag and valid variables. The flag is also updated to 1.

Cache with replacement policy:

Cache Simulator uses random replacement policy to replace a cache block. When a cache block is full the program will increase the cache miss counter and randomly pick which cache block to be replaced.

Calculate metrics and report:

At the end of the program, a cache report is generated which includes the following information: data file name, cache associativity, size of L1 cache, size of a block, number of lines in cache, number of block bits, number of line bits, number of tag bits, number of cache hits, number of cache misses, and finally the hit ratio.

Following flow chat displays all the steps in Cache Simulator program.

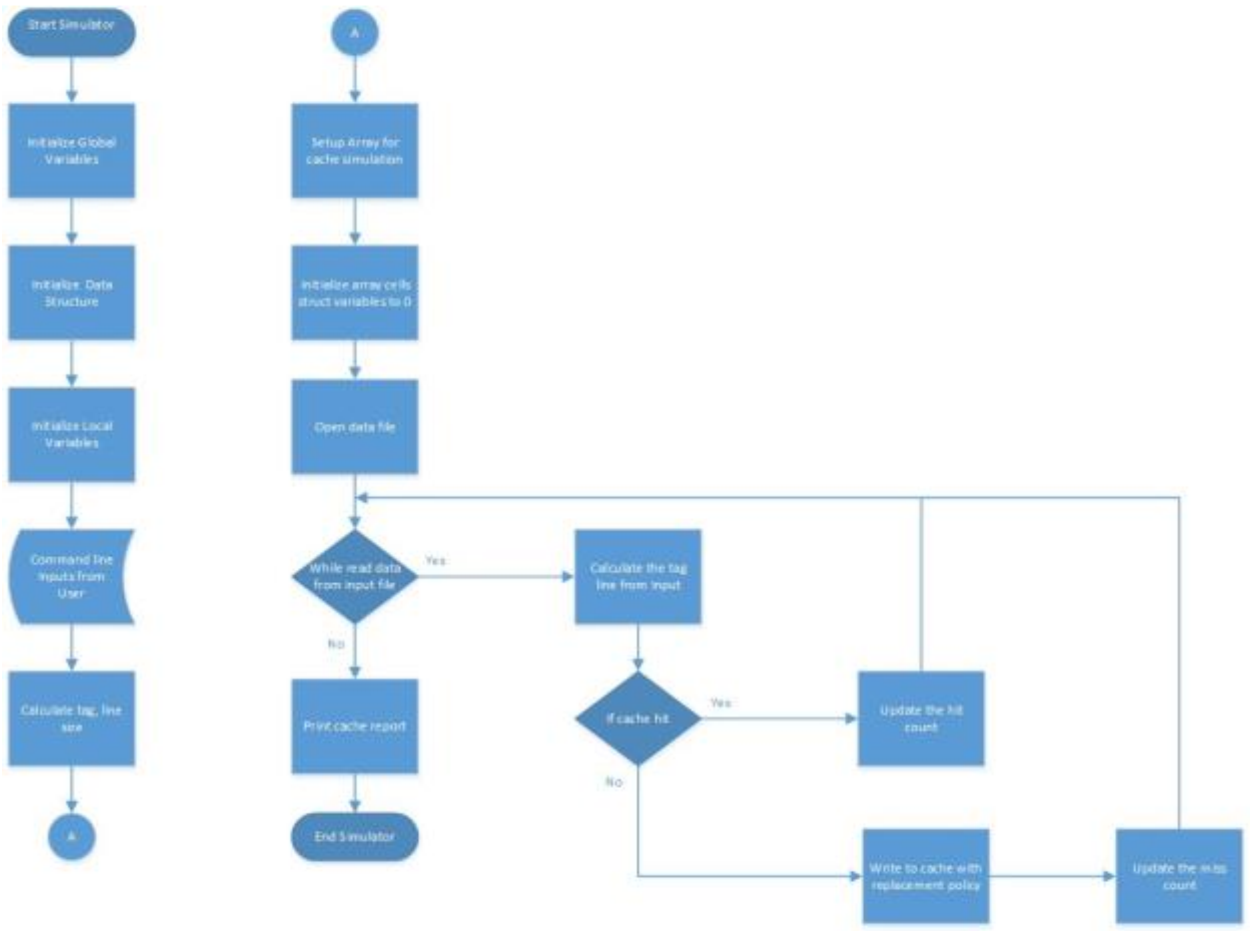
Start Simulator
Initialize Global Variables
Initialize Data Structure
Initialize Local Variables
Command line inputs from User
Calculate tag, line size
A
A
Setup Array for cache simulation
Initialize array cells struct variables to 0
Open data file
While read data from input file
Yes
No
Print cache report
End Simulator
Calculate the tag line from input
If cache hit
Yes
No
Update the hit count
Write to cache with replacement policy
Update the miss count

Figure 2.3: Simulator Flow Chat

Chapter 3

Data Structures

There are two main data structures used in cache simulator program, a struct and a two dimensional array.

```
typedef struct _block_t {
     unsigned int m_tag;
     int m_valid;
} block_t;
```

The struct data structure name block_t is implemented to represent a block of memory in a cache memory system. It utilizes two variables to accommodate tag value and valid bit using m_tag and m_vaild integer variables respectively. Variables m_tag and m_vaild facilitates in determining a cache hit or miss.

```
block_t cache[LINE_SIZE][CACHE_ASSOCIATIVITY];
```

Two dimensional array "cache" is type of block_t type and used to represent the entire L1 cache memory in the CPU. The dimensions are determined by command line input, cache associativity entered by the user.

Parameters

`address`

> Trace address read from the data file is stored as a 32bit unsigned integer. This will allow program to store 4 Byte memory addresses.

`ADDRESS_SIZE_EXP 32`

> The number of bits in memory address. Defines as global constant and set to 32. This represents a 32bit memory address.

`BLOCK_SIZE`

> The size of a memory block. Define as global constant and set to 64. Project requirement states that memory block size as 64 Bytes.

`BLOCK_SIZE_EXP`

Exponent of the block size. Defines as a global constant and set to 6. This represents 64 Byte memory block ($2^6 = 64$).

`cache_associativity`

Cache associativity of the cache. Enter as command line argument by user.

`cache_associativity_exp`

Exponent of cache associativity. Determined by cache_associativity.

`cache_hits`

Keep track of number of cache hits.

`cache_misses`

Keep track of number of cache misses.

`CACHE_SIZE`

Total capacity of L1 cache. Defined as a global constant and set to 32768. Project requirements states cache size of 32 MB.

`CACHE_SIZE_EXP 15`

Exponent of cache size. Define as a global constant and set to 15. Project requirements states cache size of 32 MB. Therefore, cache size exponent is 15 (2^{15} = 32 MB).

`ptr_file`

FILE pointer to the address trace data file.

`filename`

Name of the address trace data file. Entered as a command line argument by user.

`line_size`

Number of lines in the cache.

`line_size_exp`

Exponent of line size.

`tag_exp`

Exponent of tag size.

`tag_size`

Size of the tag.

Function Descriptions

`void printCommandFormatMessage()`

This function prints information about the command line arguments in the event command line arguments are entered incorrectly.

`int main(int argc, char* argv[])`

Main line of Cache Simulator program that takes input arguments from the user. *(More on command line arguments please see Compile and Running section.)*

Chapter 4

Calculations

`CACHE_SIZE (1 << CACHE_SIZE_EXP)`

According to project requirements, cache simulator must have cache size of 32 MB. CACHE_SIZE is calculated by left bit shifting CACHE_SIZE_EXP (CACHE_SIZE_EXP = 15).

`BLOCK_SIZE (1 << BLOCK_SIZE_EXP)`

According to project requirements, cache simulator must have a block size of 64 Bytes. BLOCK_SIZE is calculated by left bit shifting BLOCK_SIZE_EXP (BLOCK_SIZE_EXP = 6).

`CACHE_ASSOCIATIVITY = (1 << CACHE_ASSOCIATIVITY_EXP);`

Cache associativity is calculated by left bit shifting CACHE_ASSOCIATIVITY_EXP.

```
LINE_SIZE_EXP = (CACHE_SIZE_EXP - (CACHE_ASSOCIATIVITY_EXP
                  + BLOCK_SIZE_EXP))
```

LINE_SIZE_EXP is calculated using CACHE_SIZE_EXP, CACHE_ASSOCIATIVITY_EXP, and BLOCK_SIZE_EXP. Following formulas illustrates the calculation.
Lines = Cache Size / (Block Size * Associativity)
Line Exponent = Cache Size Exponent / (Block Size Exponent + Associativity Exponent)
Line Exponent = Cache Size Exponent - (Block Size Exponent + Associativity Exponent)

```
LINE_SIZE = (1 << LINE_SIZE_EXP)
```

Line size is calculated by left bit shifting LINE_SIZE_EXP.

```
TAG_EXP = (ADDRESS_SIZE_EXP - (BLOCK_SIZE_EXP + LINE_SIZE_EXP))
```

TAG_EXP is calculated using ADDRESS_SIZE_EXP, BLOCK_SIZE_EXP, and LINE_SIZE_EXP. Following formulas illustrates the calculation.
Tag bits = Address bits – (Line bits + Block bits)
Tag Exponent = Address Exponent – (Line Exponent + Block Exponent)

```
TAG_SIZE = (1 << TAG_EXP)
```

Tag size is calculated by left bit shifting TAG_EXP.

```
temp = (rand() % ((CACHE_ASSOCIATIVITY - 1) + 1 - 0) + 0)
```

Cache Simulator uses random block replacement policy in replacing cache blocks. This calculation will produce random number between 0 and (CACHE_ASSOCIATIVITY - 1).

```
((double) cache_hits / (cache_hits + cache_misses)) * 100
```

Cache hit ratio is calculated by the following formula.
Cache hit ratio = (cache hits / (cache hits + cache misses)) * 100
The value is casted to a double that will represent a decimal value.

Files

In this project contains single program file, seven address trace data files, and one makefile. Following table lists the file names and a short description of their contents.

File Name	Description
CacheSimlator.c	Main Cache Simulator program
Makefile	Builds the Cache Simulator
AddressTrace_FirstIndex.bin	Address trace file generated by first index fastest memory addressing (Minden).
AddressTrace_LastIndex.bin	Address trace file generated by last index fastest memory addressing (Minden).
AddressTrace_RandomIndex.bin	Address trace file generated by random index memory addressing (Minden).
AddressTrace_Testing_1.bin	Address trace file generator to fill the entire cache from an empty cache, which would result in 100% cache misses. This file is used to validate the computed cache hit ratio is correct.
AddressTrace_Testing_2.bin	Address trace file generator to fill the entire cache from an empty cache and repeat the same values for the second time. This would result in 100% cache miss and 100% cache hit. This file is used to validate the computed cache hit ratio is correct.
AddressTrace_Testing_3.bin	Address trace file generator to fill the entire cache from an empty cache and repeat the same values for two additional times. This would result in 100% cache miss during the filling of an empty and 2 times of 100% cache hits after the same values are repeated twice. This file is used to validate the computed cache hit ratio is correct.
AddressTrace_Testing_4.bin	Address trace file generator with random 7804 address. This file is used to illustrate the effect of associativity in cache.

Table 4.1: List of Project Files and Descriptions

Compile and Running

Cache Simulator is built for Linux environment and complied with GNU GCC complier. Therefore future compilations and execution of the program should be performed in a Linux environment with GNU GCC complier. Follow the instruction below illustrate how to compile and run the Cache Simulator at command prompt.

Compiling:

To compile Cache Simulator run provided Makefile or type following command at the command prompt.

```
$ gcc CacheSimulator.c –o CacheSimulator
```

Running:

To run Cache Simulator enter the following command in given format.

```
$ CacheSimulator <Associativity> <Address Trace File Name>
```

For example, to run the simulator with 8-way set associativity and binary input file named AddressTrace_FirstIndex.bin, enter following command.

```
$ CacheSimulator  8  AddressTrace_FirstIndex.bin
```

Note: All address trace input files must be in binary file format.

Chapter 5

Testing

Three address trace file are generated to test the cache hit ratio calculation is correct. By creating input address trace file with known cache hit ratio, it would serve as a benchmark to validate the accuracy of the cache ratio calculation performed by the Cache Simulator.

AddressTrace_Testing_1.bin is created by generating 512 addresses that have the same tag value and a unique line number. Addresses contain in this file would fill the entire cache. Running this file as input to the Cache Simulator with an associativity of one, would result in 512 cache misses. Therefore, Cache Simulator will produce a cache hit ratio of 0%.

AddressTrace_Testing_2.bin is created by repeating context of the AddressTrace_Testing_1.bin file twice. This would create a file with 1024 trace addresses, with first 512 addresses in the first half of the file is identical to the second half of the file. Running this file as input to the Cache Simulator with an associativity of one, would result in 512 cache misses and 512 cache hits. Therefore, Cache Simulator will produce a cache hit ratio of 50%.

AddressTrace_Testing_3.bin is created by repeating context of the AddressTrace_Testing_1.bin file three times. This would create a file with 1536 trace addresses, with first 512 addresses repeated twice in the rest of the file. Running this file as input to the Cache Simulator with an associativity of one, would result in 512 cache misses and 1024 cache hits. Therefore, Cache Simulator will produce a cache hit ratio of 66.67%.

By running files mention above as input, was able to determine that cache hit ratio calculated by the Cache Simulator is correct.

Results

AddressTrace_FirstIndex.bin, AddressTrace_LastIndex.bin, and AddressTrace_RandomIndex.bin are files provided as part of the project. The cache hit ratio results are display in the table below. Increasing cache associativity for each file did not improve its cache hit ratio.

Files AddressTrace_Testing_1.bin, AddressTrace_Testing_1.bin , and AddressTrace_Testing_3.bin displays the results that were except for these files since the results were engineered into the file.

AddressTrace_Testing_4.bin, which contains random trace addresses, shows improvement in cache hit ratio as the associativity is increased.

Table below displays the cache hit ratio and associativity for each file.

File Name	1-way	2-way	4-way	8-way
AddressTrace_FirstIndex.bin	0.00	0.00	0.00	0.00
AddressTrace_LastIndex.bin	93.75	93.75	93.75	93.75
AddressTrace_RandomIndex.bin	0.01	0.01	0.01	0.01
AddressTrace_Testing_1.bin	0.00	0.00	0.00	0.00
AddressTrace_Testing_2.bin	50.00	50.00	50.00	50.00
AddressTrace_Testing_3.bin	66.67	66.67	66.67	66.67
AddressTrace_Testing_4.bin	14.12	19.35	27.71	29.48

Table 5.1: Cache Hit Ratio by Input File

Chapter 6

Member Contribution

Dilesh Fernando:

- Program design and coding.
- Creating test file and testing.
- Creating figures, tables, and report writing.

Cheng-Yeh Lee:

- Validate the results.
- Program testing.
- Part of the report writing.

Vuong Nguyen:

- Program testing.
- Editing and formatting the report.

Conclusion

In future designs, creating address trace files with predetermine cache hit ratios will accelerate your development time. These files can be used to validate the program calculation since you are already aware of the result. This will aid in verifying the correctness of the program.

Also future designers should pay attention to, Big/Little endian format of the computer system. In this design, because the 32bit address was read as an unsigned integer, the problem was avoided (Adiga).

This project was a good learning exercise to better understand the inner workings of cache memory systems. Even though Cache Simulator is not a fully functioning cache memory system with write through, write back, and more sophisticated cache replacement polices that you find in modern CPUs, this project gave some insights to the complex nature of these systems.

References

Adiga, Harsha. "Writing endian-independent code in C." *IBM developerworks*, 24 Apr. 2007, www.ibm.com/developerworks/aix/library/au-endianc/. Accessed 13 Apr. 2017.

Minden, Gary J. "Address Trace Files." *Index of /~gminden/Computer_Architecture/Software*, Gary Minden, 10 Apr. 2017, www.ittc.ku.edu/~gminden/Computer_Architecture/Software/. Accessed 13 Apr. 2017.

Appendix

CacheSimulator.c

```
//******************************************************************************
//
//      Program:                CacheSimulator.c
//      Author:                 Dilesh Fernando <fernando.dilesh@gmail.com>
//                              Cheng-Yeh Lee <chengyeh90@gmail.com>
//                              Vuong Nguyen <nptvuong2912@gmail.com>
//      Date:                   2017-04-04
//      Description:   A program to simulate L1 cache.
//
//******************************************************************************

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Define global cache constants
//Address Size
#define ADDRESS_SIZE_EXP 32

//Size of the cache
#define CACHE_SIZE_EXP 15
#define CACHE_SIZE (1 << CACHE_SIZE_EXP)

//Block Size
#define BLOCK_SIZE_EXP 6
#define BLOCK_SIZE (1 << BLOCK_SIZE_EXP)

/**
 * Cache line Data Structure
 * Following structure will hold all the data associated with line in cache.
 */
typedef struct _block_t {
        unsigned int m_tag;
        int m_valid;
} block_t;

/**
 * Helper function to print command line argument error message
 */
void printCommandFormatMessage() {
        printf("Please enter command in following format.\n");
        printf("$ ./CacheSimulator <cache associativity> <name of data file>\n");
        printf("For example: $ ./CacheSimulator 1 AddressTrace_LastIndex.bin\n");
}

/**
 * Main program that simulate cache mechanism
 */
int main(int argc, char* argv[]) {

        // Random number generation variables
        time_t t;
        srand((unsigned) time(&t)); // use current time as seed value

        // Setup variables for the program
        int cache_associativity_exp = 0,
            cache_associativity = 0,
                line_size_exp = 0,
                line_size = 0,
                tag_exp = 0,
                tag_size = 0;

        char * filename; // Name of command line input filename
        FILE *ptr_file;

        unsigned int address; // address read from data file
        int cache_hits = 0,  // keep track of cache hits
            cache_misses = 0; // keep track of cache misses

        // Check for command line arguments
        // Check command line arguments are less than required number
        if (argc < 2) {
                printf("ERROR: Insufficient command line arguments.\n");
                printCommandFormatMessage();
                return (1);
        }

        // Check command line arguments are more than required number
        if (argc > 3) {
                printf("ERROR: Too many command line arguments.\n");
                printCommandFormatMessage();
                return (1);
        }

        // Right number of command line arguments
        if (argc == 3) {
                // Check command line cache associativity argument
                if (atoi(argv[1]) == 0) {
                        printf(
                                        "ERROR: Error in cache associativity command line
argument.\n");
                        return (1);
                } else if (atoi(argv[1]) == 1) {
                        cache_associativity_exp = 0;
                } else if (atoi(argv[1]) == 2) {
                        cache_associativity_exp = 1;
                } else if (atoi(argv[1]) == 4) {
                        cache_associativity_exp = 2;
                } else if (atoi(argv[1]) == 8) {
                        cache_associativity_exp = 3;
                } else {
                        // Command line argument cache associativity have failed.
                        printf("Error: Cache associativity must be 1,2,4, or 8.\n");
                        printCommandFormatMessage();
                        return (1);
                }

                // Check command line input file argument
                if (argv[2] != NULL) {
                        filename = argv[2];
                } else {
                        // Command line argument filename have failed.
                        printf("Error: Error in input file name command line argument.\n");
                        printCommandFormatMessage();
                        return (1);
                }
        } else {
                // Command line arguments have failed.
                printf("ERROR: Error in command line arguments.\n");
                printCommandFormatMessage();
                return (1);
        }

        // Calculate tag and line size
        cache_associativity = (1 << cache_associativity_exp);

        line_size_exp =
                        (CACHE_SIZE_EXP - (cache_associativity_exp + BLOCK_SIZE_EXP));
        line_size = (1 << line_size_exp);

        tag_exp = (ADDRESS_SIZE_EXP - (BLOCK_SIZE_EXP + line_size_exp));
        tag_size = (1 << tag_exp);

        /**
         Setup the array that mimics cache.
         */
        block_t cache[line_size][cache_associativity];

        // Initialize array values to zero
        for (int i = 0; i < line_size; ++i) {
                for (int j = 0; j < cache_associativity; ++j) {
                        cache[i][j].m_tag = 0;
                        cache[i][j].m_valid = 0;
                }
        }

        // Open file in read mode
        ptr_file = fopen(filename, "rb");

        // Check the file is valid
        if (!ptr_file) {
                printf("Error: Unable to Open File!\n");
                return (1);
        }

        // Set file cursor to the beginning of the file
        fseek(ptr_file, 0, SEEK_SET);

        // Main loop, read data from file and insert to cache
        while (fread(&address, sizeof(int), 1, ptr_file) == 1) {

                // bit-shift the value read from file to calculate
                // line number and tag
                unsigned int tag = address >> (line_size_exp + BLOCK_SIZE_EXP);
                unsigned int line = ((address << tag_exp) >> tag_exp) >> BLOCK_SIZE_EXP;

                int isFound = 0; // Auxiliary variable indicate cache hit

                // Check for cache hit
                for (int r = 0; r < cache_associativity; ++r) {
                        if ((cache[line][r].m_valid == 1)
                                        && (cache[line][r].m_tag == tag)) {
                                cache_hits++; // increase the hit counter
                                // cache is not updated since it was hit
                                isFound = 1; // address has been found in cache
```

```
                                break;
                        }
                }

                // Address is not found in cache will result a cache miss
                if (isFound == 0) {
                        int isCacheUpdated = 0; // Auxiliary variable indicate cache has been
updated

                        // Find the first invalid or empty block and write.
                        // This will only occur  starting from a empty cache
                        for (int s = 0; s < cache_associativity; ++s) {
                                if ((cache[line][s].m_valid == 0)
                                                && (cache[line][s].m_tag == 0)) {
                                        // Found the first invalid block
                                        cache_misses++; // increase the miss count

                                        // Update the block
                                        cache[line][s].m_tag = tag;
                                        cache[line][s].m_valid = 1;

                                        isCacheUpdated = 1; // Set auxiliary variable
                                        break;
                                }
                        }

                        // Cache already full, now have to replace a block
                        if (isCacheUpdated == 0) {
                                cache_misses++; // increment the miss count

                                // Replace the cache block
                                // Cache is replaced in random order
                                int temp = (rand() % ((cache_associativity - 1) + 1 - 0) + 0);

                                // Update the block
                                cache[line][temp].m_tag = tag;
                                cache[line][temp].m_valid = 1;
                        }
                }
        } // end while

        // Print cache report
        printf("\n");
        printf("======================================================\n");
        printf("                      CACHE REPORT\n");
        printf("======================================================\n");
        printf("Data filename           : %s\n", filename);
        printf("Cache Associativity      : %d\n", cache_associativity);
        printf("Size of L1 Cache         : %d\n", CACHE_SIZE);
        printf("Size of a Block          : %d\n", BLOCK_SIZE);
        printf("Number of lines in cache : %d\n", line_size);
        printf("Number of Block bits     : %d\n", BLOCK_SIZE_EXP);
        printf("Number of Line bits      : %d\n", line_size_exp);
        printf("Number of Tag bits       : %d\n", tag_exp);
        printf("------------------------------------------------------\n");
        printf("Number of cache hits     : %d\n", cache_hits);
        printf("Number of cache misses   : %d\n", cache_misses);
        printf("Cache hit ratio          : %2.2f%\n",
                        ((double) cache_hits / (cache_hits + cache_misses)) * 100);
        printf("======================================================\n");

        // Close file
        if (fclose(ptr_file) != 0) {
                printf("Error: File Not Closed!\n");
        }

        return (0);
}
```

Makefile

```
all: CacheSimulator.c
	gcc CacheSimulator.c -o CacheSimulator

clean:
	rm -f *~ CacheSimulator
```