

**SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING**

*College of Engineering and Built Environment*

**Bachelor of Engineering in Computers and Communications**

**Programme Code: (DT 021A) 2014/18**

**Year 4**

**Computer Architecture**

**Raymond Lynch**

**Student Name(s): Aisling Lee**

**Class Group: DT021A/4**

**Student Number C12358536**

**Assignment title** **Simulating a 2-way set associative cache**

**Date issued 28/09/17**

**Date due for return 22/11/2017**

**Date returned 22/11/2017**

Table of Contents

[Introduction: 2](#_Toc499202391)

[Objective: 4](#_Toc499202392)

[Method: 6](#_Toc499202393)

[Code 6](#_Toc499202394)

[Results: 7](#_Toc499202395)

[Outputs 7](#_Toc499202396)

[Conclusion: 8](#_Toc499202397)

[Works Cited 9](#_Toc499202398)

Table of Figures

[Figure 1: Cache mapping types [3] 3](file:///F:\College\DT021A\DT021A4\CompArch3\Lab\FE-%20AssocCache\ALee_FormalElement.docx#_Toc499202399)

[Figure 2: Process solution flow chart 5](#_Toc499202400)

Table of Tables

[Table 1: Maximum performance of memory drive interfaces [2] 2](file:///F:\College\DT021A\DT021A4\CompArch3\Lab\FE-%20AssocCache\ALee_FormalElement.docx#_Toc499202401)

[Table 2: 16 bit binary data break down 3](#_Toc499202402)

Simulating 2-Way Set Associative Cache

# Introduction:

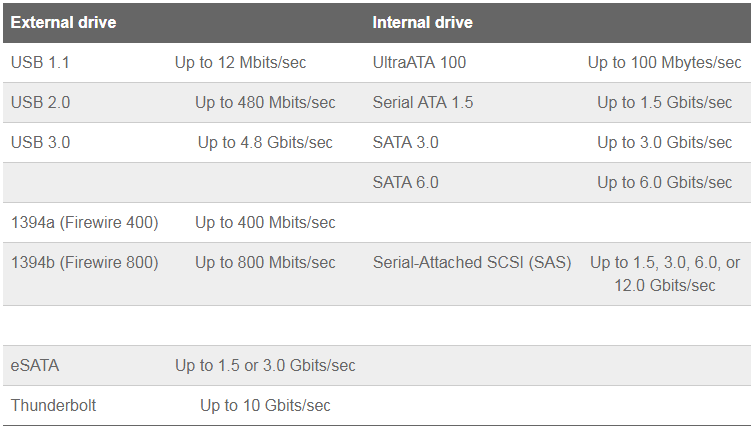
A computers memory structure of its internal storage consists of a hard disk drive (HDD)/ solid state drive (SSD), random access memory (RAM), and cache. Cache memory was developed out of necessity for faster memory retrieval to keep up with the service speeds required by the ever increasing processing speeds of the computers central processing units (CPU). Currently, based on whether the internal storage is HDD, or SSD (and its connection to the CPU) transfer speeds may vary as can be seen in table 1. RAM or DRAM (dynamic RAM) is faster than the drives and with DDR3 and DDR4 operating at similar frequencies (1.6GHz and 2.1GHz respectively) to the clock speeds of many CPUs today but due to its need to be constantly refreshed and CPUs having multiple cores able to make multiple requests, it has has a slower retrieval rate than cache which is SRAM (static RAM). Cache was previously located in close proximity to the CPU however due to advancement in technology parts of it may now be found contained within the chip. Cache is broken into 3 levels, Level 1, 2, and 3 more often referred to as L1, L2, and L3. L1 is located on the CPU chip and, for multi core chips, each core will have its own L1 cache. L1 cache memory is usually set to be between 2KB and 64KB depending on the CPU specification. L1 is broken down into the instruction cache and the data cache each handle the instructions and data for the CPU respectively. For L2 it may be located on or outside the CPU and each core may have its own or they may all share one. For instances where L2 cache is located outside of the CPU it will be connected directly with a high speed bus. Its memory is usually between 256-512 KB but is slower than L1. Lastly L3 which can be found in higher quality processors is located outside of the CPU and is shared amongst all cores within it. It usually has a memory size of 1MB to 8 MB. The cache memory although faster and more efficient is smaller in computers than RAM is due to its power consumption, space and mainly cost as it is quite expensive in comparison to DRAM.

Table : Maximum performance of memory drive interfaces [2]

The CPU accesses memory by first checking L1 cache. If the CPU finds what it was looking for in the L1 cache it then it is referred to as a hit and returns it to the CPU, if not then we refer to it as a miss and it moves onto the next stage of the memory checking process. This is repeated at all levels from L2 to L3 to RAM and then to the HDD/SDD.

Within this problem we are asked to work with 2-way set associative cache. Set associative is a type of cache mapping technique. There are 3 types, direct mapping, fully-associative mapping, and set-associative mapping. Direct mapping only permits each entry to a specific location in cache, when another piece of data requires the same location the current data is replaced. Fully associative allows allocation of any location in cache to incoming data. Set-associative is a combination of direct mapping and fully associative. Set associative works by breaking down the binary value of the data into categories tag, set number, word number and byte number.

|  |  |  |
| --- | --- | --- |
| Tag | Set No | Word No |
| 8 bits | 6 bits | 2 bit |

Table : 16 bit binary data break down

The set number refers to a set or group of locations in cache that the data is permitted to be in, once its designated set has been located, (within its set) it is permitted to be stored in any of the possible locations. We can see in table 2 how it is broken down. The 2-way aspect denotes the grouping of locations into the sets as seen in figure 1 where the locations are referred to as blocks. Currently there is 2-way, 4-way and 8-way set associative caches from which to model your mapping system. As mentioned before however we will be working with 2-way set associative.

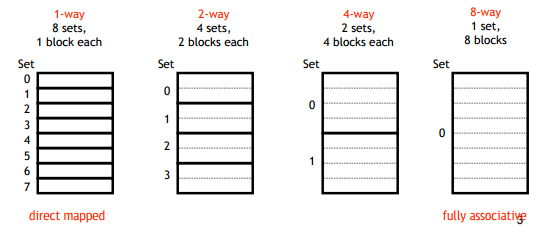


Figure : Cache mapping types [3]

The last thing to consider when analysing cache and how it functions is its replacement policy. After a while we know that the memory blocks will fill up and the issue arises then on what data should be kept in the cache and what should be removed ?The most common replacement policy methods are;

* Optimal replacement: Replace the block least likely to be needed in the future
* FIFO: First in, first out . The block of data longest in the cache is removed
* LRU: Least recently used. Replacement of the block of data which has the longest time between when it was last used.

The most popular replacement policy is LRU as result of a long history of measuring system behaviour. There is no one best algorithm as that requires accurate future prediction which with data access and usage cannot always be done. For some applications FIFO would be optimal however if you’re currently working on something that was admitted to cache a substantially long time ago compared to other recently cached data it would not be ideal to have it redirected to the disk. Likewise with LRU while you may not have used something recently you may require it in the near future in comparison to other data in the cache and thus removing it will delay productivity time.

## Objective:

Write a computer program in C or C++ that simulates the functioning of a 2-way set associative cache during a series of read operations. Assume a CPU with an 8-bit data bus and a 16-bit address bus. Assume instructions are 8 bits wide. The cache will initially be empty. A test file containing addresses is used as a test source. As each address is read from the file it is checked against the cache entries. A cache miss results in a cache line fill. Assume a cache line is 4 bytes. A cache hit should result in outputting of the relevant data entry. Hence the program should output the resulting address/data values and the effect of the operation on the cache. Use a second file to hold data values simulating DRAM. On the next page you can observe figure 2 which outlines the above process in a flowchart form

Figure : Process solution flow chart

# Method:

## Code initial

///////////////////////////////////////////////////////////////////////////////

// Aisling Lee C12358536 //

// DT021A Year 4 //

// Computer Architecture 3 //

// Last modified 24/11/2017 //

///////////////////////////////////////////////////////////////////////////////

//////////////////Library Declarations///////////////////////////////////////

#include <stdio.h>

#include <iomanip>

#include <fstream>

#include <tchar.h>

#include <windows.h>

#include <Winbase.h>

#include <bitset> //bitset - print all bits including 0's

#include <string> //string - data from input file is read in as string

#include <fstream> //fstream - read in external text file

#include <sstream> //sstream - convert string data to hex format

#include <iostream>//iostream - used for printing results

using namespace std;

//////////////////////////////////////////////////////////////////////////////

////////////////// Declaring variables //////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////

/\* ifstream cpu\_address\_list("address.txt"); //Open and read in addresses from "address.txt" file, and attach to cpu\_address

unsigned short cpu\_address;

string text\_line\_in;

unsigned short memoryaddr = 0;

int memoryline = 0;

int hit\_way0 = 0;

int hit\_way1 = 0;

int miss\_way0 = 0;

int miss\_way1 = 0;

int set\_no;

int tag\_addr;

\*/

//////////////////////////////////////////////////////////////////////////////

//////////////////Structure Declarations//////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////

struct cache\_data

{

// Structure representing a single line in the cache

//unsigned char higherCPUAddr; // Higher CPU Address stored in cache

bool validflag; // Valid Flag variabe

bool lruflag; // Least Recently Used Flag variabe

char mem\_loc [4]; // array of four 1 byte memory locations

unsigned char tag; // Stores tag

unsigned char set; // Stores set

}

way0[64], way1[64]; // Create two cache entry structures

struct dram\_block

{

unsigned short dramaddr; //2 byte short for storing address

unsigned char dramdata; //1 byte char for storing data @ above address

} dram\_memory[1024]; //1Kb of DRAM memory declared

//////////////////////////////////////////////////////////////////////////////

//////////////////Function Declarations///////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////

void clear\_cache(cache\_data way0[], cache\_data way1[])

{

// Initilise Valid and LRU Flags

int counter = 0;

while (counter<64) // declare and increment counter value and loop for all 64 elements (64 possible sets)

{

way0[counter].validflag = 0; // reset valid\_flag for way 0

way1[counter].validflag = 0; // reset valid\_flag for way 1

way0[counter].lruflag = 0; // reset lru\_flag for way 0

way1[counter].lruflag = 0; // reset lru\_flag for way 1

counter++;

}

}

void print\_setup ()

{

cout << " \t\tAisling Lee\n";

cout << " \t\tC12358536, DT021A Year 4\n";

cout << " \t\tComputer Architecture 3: Formal element\n";

cout << " \t\t2-Way Set Associative Cache\n";

cout << "--------------------------------------------------------------------|\n";

cout << "| Address | Status | Location (Set #) | Data |\n";

cout << "|----------------|----------------|----------------|----------------|\n";

}

/\*void print\_end ()

{

cout << "|--------------------------------------------------------------------|\n";

cout << " Results of Cache Analysis" << endl;

cout << " Hit Count: Way 0: " << dec << hit\_way0 <<"Way 1: " << dec << hit\_way1 << endl;

cout << " Miss Count: Way 0: " << dec << miss\_way0 << " Way 1: " << dec << miss\_way1 << endl;

}\*/

void dram\_setup()

{

//Populating DRAM with random data spanning from address 0x8000 - 0x8400

int loop\_counter = 0;

while (loop\_counter++ < 1024 )

{

dram\_memory[loop\_counter].dramaddr = 0x8000 + loop\_counter;

dram\_memory[loop\_counter].dramdata = rand() % 255;

}

}

/\*void cache\_hit\_check ()

{

cout << " \n I am in cache hit check\n";

if ((way0[set\_no].tag == tag\_addr) && (way0[set\_no].validflag = 1))

{

cout << " \n I am in cache hit check way0\n";

cout << "| " << hex << cpu\_address << " | Hit " << "| Way 0 "<<dec << set\_no << "| "<< hex<<tag\_addr<<" |\n"<< endl;

//Increment hit counter & swap LRU flags

hit\_way0++;

way0[set\_no].lruflag = 0;

way1[set\_no].lruflag = 1;

}

else if ((way1[set\_no].tag == tag\_addr) && (way1[set\_no].validflag = 1))

{

cout << " \n I am in cache hit check way1\n";

cout << "| " << hex << cpu\_address << " | Hit " << "| Way 1 "<<dec << set\_no << "| "<< hex<<tag\_addr<<" |\n"<< endl;

//Increment hit counter & swap LRU flags

hit\_way1++;

way0[set\_no].lruflag = 1;

way1[set\_no].lruflag = 0;

}

}\*/

/\*void cache\_miss()

{

//If way0 is the least recently used path

if (way0[set\_no].lruflag == 1)

{

cout << "| " << hex << cpu\_address << " | MISS " << "| Way 0 "<<dec << set\_no <<" | ";

way0[set\_no].tag = tag\_addr; //update the tag to a new value

//Step through DRAM blocks until requested cache address is found

while (memoryaddr =! cpu\_address)

{

memoryaddr = dram\_memory[memoryline].dramaddr;

memoryline++;

}

//Read data @ cache boundary address & next 3 data lines into cache line

int loop\_counter = 0;

while (loop\_counter++ < 4)

{

way0[set\_no].mem\_loc[loop\_counter] = dram\_memory[memoryline + loop\_counter].dramdata;

cout << hex << way0[set\_no].mem\_loc[loop\_counter] << " |\n"<< endl;

}

//Increment miss counter & swap LRU flags

miss\_way0++;

way0[set\_no].lruflag = 0;

way1[set\_no].lruflag = 1;

}

else if (way1[set\_no].lruflag == 1)

{

cout << "| " << hex << cpu\_address << " | MISS " << "| Way 1 "<<dec << set\_no <<" | ";

way0[set\_no].tag = tag\_addr; //update the tag to a new value

//Step through DRAM blocks until requested cache address is found

while (memoryaddr =! cpu\_address)

{

memoryaddr = dram\_memory[memoryline].dramaddr;

memoryline++;

}

//Read data @ cache boundary address & next 3 data lines into cache line

int loop\_counter = 0;

while ( loop\_counter++ < 4)

{

way0[set\_no].mem\_loc[loop\_counter] = dram\_memory[memoryline + loop\_counter].dramdata;

cout << hex << way0[set\_no].mem\_loc[loop\_counter++] << " |\n"<< endl;

}

//Increment miss counter & swap LRU flags

miss\_way0++;

way0[set\_no].lruflag = 1;

way1[set\_no].lruflag = 0;

}

}\*/

//////////////////Main////////////////////////////////////////////////////

int main()

{

//cout << " \n I am in main\n";

// Set up //

unsigned short cpu\_address;

string text\_line\_in;

unsigned short memoryaddr = 0;

int memoryline = 0;

int hit\_way0 = 0;

int hit\_way1 = 0;

int miss\_way0 = 0;

int miss\_way1 = 0;

int set\_no;

int tag\_addr;

clear\_cache(way0, way1); // Initilises Cache Variables to zero (Except the LRU Flags on way1[] is set to 1)

dram\_setup ();

print\_setup (); // Print out soln details and set up table

ifstream cpu\_address\_list("address.txt"); //Open and read in addresses from "address.txt" file, and attach to cpu\_address

while (getline(cpu\_address\_list, text\_line\_in))

{

//cout << " \n I am in while loop\n";

istringstream(text\_line\_in) >> hex >> cpu\_address; // convert the incoming string line (extracted from the text file) into a hex value and store it in cpu\_address variable

tag\_addr = (cpu\_address & 0xff00); // make all bits zero

tag\_addr >>= 8; //bitshift tag 8 bits

set\_no = cpu\_address & 0x00ff; // make all bits zero

set\_no >>= 2; //bitshift setno 2 bit

cout << " \n I am in cache hit check\n";

if ((way0[set\_no].tag == tag\_addr) && (way0[set\_no].validflag = 1))

{

cout << " \n I am in cache hit check way0\n";

cout << "| " << hex << cpu\_address << " | Hit " << "| Way 0 "<<dec << set\_no << "| "<< hex<<tag\_addr<<" |\n"<< endl;

//Increment hit counter & swap LRU flags

hit\_way0++;

way0[set\_no].lruflag = 0;

way1[set\_no].lruflag = 1;

}

else if ((way1[set\_no].tag == tag\_addr) && (way1[set\_no].validflag = 1))

{

cout << " \n I am in cache hit check way1\n";

cout << "| " << hex << cpu\_address << " | Hit " << "| Way 1 "<<dec << set\_no << "| "<< hex<<tag\_addr<<" |\n"<< endl;

//Increment hit counter & swap LRU flags

hit\_way1++;

way0[set\_no].lruflag = 1;

way1[set\_no].lruflag = 0;

}

else

{

//If way0 is the least recently used path

if (way0[set\_no].lruflag == 1)

{

cout << "| " << hex << cpu\_address << " | MISS " << "| Way 0 "<<dec << set\_no <<" | ";

way0[set\_no].tag = tag\_addr; //update the tag to a new value

//Step through DRAM blocks until requested cache address is found

while (memoryaddr =! cpu\_address)

{

memoryaddr = dram\_memory[memoryline].dramaddr;

memoryline++;

}

//Read data @ cache boundary address & next 3 data lines into cache line

int loop\_counter = 0;

while (loop\_counter++ < 4)

{

way0[set\_no].mem\_loc[loop\_counter] = dram\_memory[memoryline + loop\_counter].dramdata;

cout << hex << way0[set\_no].mem\_loc[loop\_counter] << " |\n"<< endl;

}

//Increment miss counter & swap LRU flags

miss\_way0++;

way0[set\_no].lruflag = 0;

way1[set\_no].lruflag = 1;

}

else if (way1[set\_no].lruflag == 1)

{

cout << "| " << hex << cpu\_address << " | MISS " << "| Way 1 "<<dec << set\_no <<" | ";

way0[set\_no].tag = tag\_addr; //update the tag to a new value

//Step through DRAM blocks until requested cache address is found

while (memoryaddr =! cpu\_address)

{

memoryaddr = dram\_memory[memoryline].dramaddr;

memoryline++;

}

//Read data @ cache boundary address & next 3 data lines into cache line

int loop\_counter = 0;

while ( loop\_counter++ < 4)

{

way0[set\_no].mem\_loc[loop\_counter] = dram\_memory[memoryline + loop\_counter].dramdata;

cout << hex << way0[set\_no].mem\_loc[loop\_counter++] << " |\n"<< endl;

}

//Increment miss counter & swap LRU flags

miss\_way0++;

way0[set\_no].lruflag = 1;

way1[set\_no].lruflag = 0;

}

}

/\*cout << " \ncpu\_address\n" << cpu\_address;

cout << " \ntag\_addr\n" << tag\_addr;

cout << " \nset\_no\n" << set\_no;

if ((way0[set\_no].tag == tag\_addr) || (way0[set\_no].tag == tag\_addr)) cache\_hit\_check();

else cache\_miss();\*/

}

//print\_end();

cout << "|--------------------------------------------------------------------|\n";

cout << " Results of Cache Analysis" << endl;

cout << " Hit Count: Way 0: " << dec << hit\_way0 <<"Way 1: " << dec << hit\_way1 << endl;

cout << " Miss Count: Way 0: " << dec << miss\_way0 << " Way 1: " << dec << miss\_way1 << endl;

system ("pause"); //Pauses the console

return 0;

}

## Code simplified

///////////////////////////////////////////////////////////////////////////////

// Aisling Lee C12358536 //

// DT021A Year 4 //

// Computer Architecture 3 //

// Last modified 24/11/2017 //

///////////////////////////////////////////////////////////////////////////////

//////////////////Library Declarations///////////////////////////////////////

#include <stdio.h>

#include <iomanip>

#include <fstream>

#include <tchar.h>

#include <windows.h>

#include <Winbase.h>

#include <bitset> //bitset - print all bits including 0's

#include <string> //string - data from input file is read in as string

#include <fstream> //fstream - read in external text file

#include <sstream> //sstream - convert string data to hex format

#include <iostream>//iostream - used for printing results

using namespace std;

//////////////////////////////////////////////////////////////////////////////

//////////////////Structure Declarations//////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////

struct cache\_data

{

// Structure representing a single line in the cache

//unsigned char higherCPUAddr; // Higher CPU Address stored in cache

bool validflag; // Valid Flag variabe

bool lruflag; // Least Recently Used Flag variabe

char mem\_loc [4]; // array of four 1 byte memory locations

unsigned char tag; // Stores tag

unsigned char set; // Stores set

}

way0[64], way1[64]; // Create two cache entry structures

struct dram\_block

{

unsigned short dramaddr; //2 byte short for storing address

unsigned char dramdata; //1 byte char for storing data @ above address

} dram\_memory[1024]; //1Kb of DRAM memory declared

//////////////////////////////////////////////////////////////////////////////

//////////////////Function Declarations///////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////

void print\_setup ()

{

cout << " \t\tAisling Lee\n";

cout << " \t\tC12358536, DT021A Year 4\n";

cout << " \t\tComputer Architecture 3: Formal element\n";

cout << " \t\t2-Way Set Associative Cache\n";

cout << "--------------------------------------------------------------------|\n";

cout << "| Address | Status | Location (Set #) | Data |\n";

cout << "|----------------|----------------|----------------|----------------|\n";

}

//////////////////Main////////////////////////////////////////////////////

int main()

{

// Set up //

unsigned short cpu\_address;

string text\_line\_in;

unsigned short memoryaddr = 0;

int memoryline = 0;

int hit\_way0 = 0;

int hit\_way1 = 0;

int miss\_way0 = 0;

int miss\_way1 = 0;

int set\_no;

int tag\_addr;

int counter = 0;

// Initilises Cache Variables to zero (Except the LRU Flags on way1[] is set to 1)

while (counter++<64) // declare and increment counter value and loop for all 64 elements (64 possible sets)

{

way0[counter].validflag = 0; // reset valid\_flag for way 0

way1[counter].validflag = 0; // reset valid\_flag for way 1

way0[counter].lruflag = 0; // reset lru\_flag for way 0

way1[counter].lruflag = 0; // reset lru\_flag for way 1

}

print\_setup (); // Print out soln details and set up table

//Populating DRAM with random data spanning from address 0x8000 - 0x8400

int loop\_counter = 0;

while (loop\_counter++ < 1024 )

{

dram\_memory[loop\_counter].dramaddr = 0x8000 + loop\_counter;

dram\_memory[loop\_counter].dramdata = rand() % 255;

}

ifstream cpu\_address\_list("address.txt"); //Open and read in addresses from "address.txt" file, and attach to cpu\_address

while (getline(cpu\_address\_list, text\_line\_in))

{

//cout << " \n I am in while loop\n";

istringstream(text\_line\_in) >> hex >> cpu\_address; // convert the incoming string line (extracted from the text file) into a hex value and store it in cpu\_address variable

tag\_addr = (cpu\_address & 0xff00); // make all bits zero

tag\_addr >>= 8; //bitshift tag 8 bits

set\_no = cpu\_address & 0x00ff; // make all bits zero

set\_no >>= 2; //bitshift setno 2 bit

cout << " \n I am in cache hit check\n";

if ((way0[set\_no].tag == tag\_addr) && (way0[set\_no].validflag = 1))

{

cout << " \n I am in cache hit check way0\n";

cout << "| " << hex << cpu\_address << " | Hit " << "| Way 0 "<<dec << set\_no << "| "<< hex<<tag\_addr<<" |\n"<< endl;

//Increment hit counter & swap LRU flags

hit\_way0++;

way0[set\_no].lruflag = 0;

way1[set\_no].lruflag = 1;

}

else if ((way1[set\_no].tag == tag\_addr) && (way1[set\_no].validflag = 1))

{

cout << " \n I am in cache hit check way1\n";

cout << "| " << hex << cpu\_address << " | Hit " << "| Way 1 "<<dec << set\_no << "| "<< hex<<tag\_addr<<" |\n"<< endl;

//Increment hit counter & swap LRU flags

hit\_way1++;

way0[set\_no].lruflag = 1;

way1[set\_no].lruflag = 0;

}

else

{

//If way0 is the least recently used path

if (way0[set\_no].lruflag == 1)

{

cout << "| " << hex << cpu\_address << " | MISS " << "| Way 0 "<<dec << set\_no <<" | ";

way0[set\_no].tag = tag\_addr; //update the tag to a new value

//Step through DRAM blocks until requested cache address is found

while (memoryaddr =! cpu\_address)

{

memoryaddr = dram\_memory[memoryline].dramaddr;

memoryline++;

}

//Read data @ cache boundary address & next 3 data lines into cache line

int loop\_counter = 0;

while (loop\_counter++ < 4)

{

way0[set\_no].mem\_loc[loop\_counter] = dram\_memory[memoryline + loop\_counter].dramdata;

cout << hex << way0[set\_no].mem\_loc[loop\_counter] << " |\n"<< endl;

}

//Increment miss counter & swap LRU flags

miss\_way0++;

way0[set\_no].lruflag = 0;

way1[set\_no].lruflag = 1;

}

else if (way1[set\_no].lruflag == 1)

{

cout << "| " << hex << cpu\_address << " | MISS " << "| Way 1 "<<dec << set\_no <<" | ";

way0[set\_no].tag = tag\_addr; //update the tag to a new value

//Step through DRAM blocks until requested cache address is found

while (memoryaddr =! cpu\_address)

{

memoryaddr = dram\_memory[memoryline].dramaddr;

memoryline++;

}

//Read data @ cache boundary address & next 3 data lines into cache line

int loop\_counter = 0;

while ( loop\_counter++ < 4)

{

way0[set\_no].mem\_loc[loop\_counter] = dram\_memory[memoryline + loop\_counter].dramdata;

cout << hex << way0[set\_no].mem\_loc[loop\_counter++] << " |\n"<< endl;

}

//Increment miss counter & swap LRU flags

miss\_way0++;

way0[set\_no].lruflag = 1;

way1[set\_no].lruflag = 0;

}

}

}

//print\_end();

cout << "|--------------------------------------------------------------------|\n";

cout << " Results of Cache Analysis" << endl;

cout << " Hit Count: Way 0: " << dec << hit\_way0 <<"Way 1: " << dec << hit\_way1 << endl;

cout << " Miss Count: Way 0: " << dec << miss\_way0 << " Way 1: " << dec << miss\_way1 << endl;

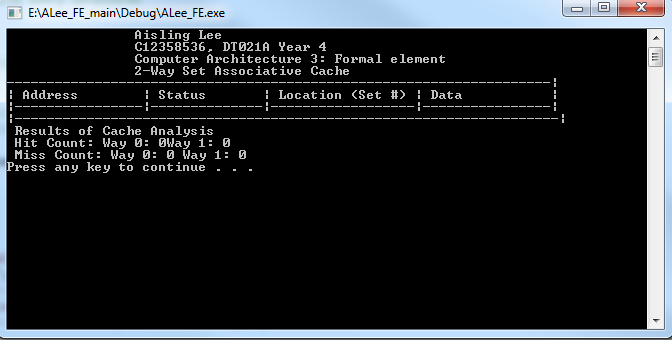
system ("pause"); //Pauses the console

return 0;

}

# Results:

## Outputs



The system seems to skip the while loop as I am unable to print out any statements from within it. This could be a syntax error or an include error in regards to reading in from the text file therefore the getline function is unable to execute as desired.

# Conclusion:

Unfortunately I was unable to successfully run the cache checking system and I have still yet to figure out why. However it still serves as an illustrative point on how LRU caching systems work. As previously mentioned cache and its memory blocks are so highly valuable for performance speeds and that no one algorithm solves the hit and miss issue.

I would predict initially a high number of misses due to the cache starting out as empty (initialised to 0 as seen in code). After the cache would have filled up we would then begin to see more hits due to the locations being entirely populated with data from the file. The hit rate could also be improved by having a more memory allocated to the cache set size, and lines.

Caching is important in the exponentially growing field of computers as we consistently see technologies getting fast, more reliable and smaller. Therefore methods to handle the higher and faster requests are crucial.

# Works Cited

|  |  |
| --- | --- |
| [1] | C. Dirik and B. Jacob, “The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization,” in *ISCA '09 Proceedings of the 36th annual international symposium on Computer architecture*, Austin, 2009. |
| [2] | Seagate, “How fast should an external drive be?,” Seagate, 2016. [Online]. Available: http://knowledge.seagate.com/articles/en\_US/FAQ/172213en?language=en-gb. [Accessed 21 Nov 2017]. |
| [3] | University of Washington, “Review 2-way set associative implementation,” [Online]. Available: https://courses.cs.washington.edu/courses/cse378/09au/lectures/cse378au09-19.pdf. [Accessed 22 Nov 2017]. |
| [4] | B. Jacob, S. W. Ng and D. T. Wang, Memory Systems: Cache, DRAM, Disk, Denise E.M. Penrose, 2008. |
| [5] | D. A. Patterson and J. L. Hennessy, Computer Organization and Design, Morgan Kaufmann, 2014. |