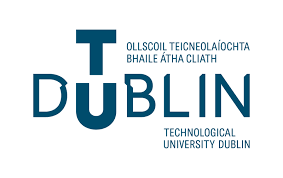
******

**SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING**

**Bachelor of Engineering (Hons) BE in Electrical & Electronic / Comp & Comm Eng**

**Programme Code: (DT021A)**

**<YEAR 4>**

**Name of Module (Computer Architecture 3, COMP4601)**

**Name of Lecturer (Lynch, Raymond)**

Formal Assignment

**Student Name** \_Sajjad Ullah, C17344483\_\_\_\_\_\_\_\_\_\_\_\_\_\_**\_\_\_\_\_\_\_\_\_\_\_**

**Title \_**Simulating 3 Types of Cache\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_

**DECLARATION**

I hereby certify that the material, which is submitted in this assignment, is entirely my own work and has not been submitted for any academic assessment other than as part fulfilment of the assessment procedures.

Signature of student: ……Sajjad ullah……………….

Date: ……28/11/21…………………………

Contents

[Introduction 4](#_Toc89017156)

[Problem Statement 4](#_Toc89017157)

[Cache memory 6](#_Toc89017158)

[Direct Mapped Cache 7](#_Toc89017159)

[2-Way Set Associative Cache 9](#_Toc89017160)

[Associative Cache 11](#_Toc89017161)

[Advantages and Disadvantages compared 13](#_Toc89017162)

[Programming the Cache Simulator 14](#_Toc89017163)

[The Code in Main 15](#_Toc89017164)

[Direct mapped function 17](#_Toc89017165)

[Direct map function flowchart 19](#_Toc89017166)

[2-Way Set Associative function 20](#_Toc89017167)

[2-way Set Associative function flowchart 22](#_Toc89017168)

[Set Associative function 23](#_Toc89017169)

[Set Associative function flowchart 25](#_Toc89017170)

[Modularity 26](#_Toc89017171)

[The initilise\_all\_cache() function 26](#_Toc89017172)

[The user\_cache\_pick() function 26](#_Toc89017173)

[Testing Direct map 28](#_Toc89017174)

[Testing 2-Way Set Associative map 29](#_Toc89017175)

[Testing Set Associative 30](#_Toc89017176)

[Conclusion 30](#_Toc89017177)

[Bibliography 31](#_Toc89017178)

[Appendix A Direct map result 31](#_Toc89017179)

[Appendix B 2-Way Set Associative result 35](#_Toc89017180)

[Appendix C Set Associative result 38](#_Toc89017181)

[Appendix D full program code 41](#_Toc89017182)

[Figure 1 Latency of different memory accesses [1] 6](#_Toc89017145)

[Figure 2 capacity and latency of different storages [2] 6](#_Toc89017146)

[Figure 3 Direct Mapping method [3] 7](#_Toc89017147)

[Figure 4 Direct map implementation [4] 8](#_Toc89017148)

[Figure 5 2-Way Set Associative method [5] 9](#_Toc89017149)

[Figure 6 2-Way Set Associative implementation [4] 10](#_Toc89017150)

[Figure 7 Associative Cache method [6] 11](#_Toc89017151)

[Figure 8 Associative Cache implementation [4] 12](#_Toc89017152)

[Figure 9 Test file for direct map 28](#_Toc89017153)

[Figure 10 Test file for 2-Way Set Associate map 29](#_Toc89017154)

[Figure 11 Test file for Set Associative map 30](#_Toc89017155)

# Introduction

This formal assignment is on the subject of caching, there are three main types of cache designs, these are associative, N-way set associative and direct mapped cache. We are tasked to simulate these three designs using the programming language C++.

# Problem Statement

Write a computer program in C++ that simulates the functioning of a hardware cache during a series of read operations. Do this for 3 types of cache design:

* Associative Cache,
* 2-Way Set Associative Cache and
* Direct mapped Cache

Assume that CPU has a **32-bit address** bus and an **8-bit data bus**. 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 hit should increment a hit counter. A cache miss results in a cache line fill and an increment of a miss counter.

For each test address the program should:

* Output the resulting address in the form of a tag number, a set number, and a byte number.
* It should also indicate a hit or a miss
* the status of the hit and miss counters
* and in the case of a miss the line number of the data loaded from memory.

**Input:**

The input test data should come from a file, consisting of a sequence of addresses. This simulates the behaviour of a CPU as it fetches program instructions from memory. The address can be represented by a 32-bit unsigned integer, split into an upper tag of 16 bits, 14-bit set no. and a 2-bit byte select as follows:

|  |  |  |
| --- | --- | --- |
| TAG (16 Bits) | SET (14 bits) | WORD (2 bits) |

**Processing:**

Model your cache as an array of cache lines. Each cache line should consist of a:

* TAG field containing a single 16-bit value
* VALID field containing a single true/false value
* LRU field containing a single true/false value
* DATA field containing an array of 4 bytes representing a cache line

It is not necessary to model your main memory data values.

You may assume that the addressable unit is a byte, the replacement algorithm is LRU.

**Output:**

The output should consist of the following:

* A section which identifies the parameters of the cache and memory being modelled. At a minimum it should list the number of addressable units per line (or block), the number of sets, and the number of lines.
* A section detailing each read as it is performed. For each read processed, you should produce a sub-section of output containing:
  + The resulting address in the form of a tag number, a set number and a byte number.
  + Whether this is a hit or a miss, the status of the hit and miss counters and
  + in the case of a miss the line number of the data loaded from memory.

# Cache memory

A cache is area located typically within the Central Processing Unit (CPU) which serves as fast access for data and instructions. This fast access is important for low latency execution of applications as the latency of data from main memory such as Hard-Disk Drives (HHD) or Random-Access Memory (RAM) can be 10 – 100 times slower.

There are different layers of cache that reside in the CPU there are L1, L2, & L3 they each have different speeds with the fastest being L1 cache having an access latency of up to 1ns.

Table

Description automatically generated

Figure Latency of different memory accesses [1]

As seen in figure 1 the layers 1,2 and 3 access is far better than RAM and significantly outperforms Solid State Drives (SSD) but manufacturing L1, L2 and L3 cache is still more costly, which is why its capacity is very small. Figure 2 shows the varying capacity and latency of different storages.

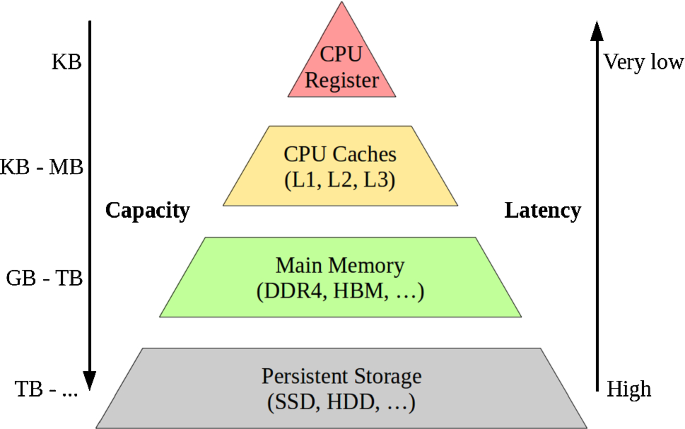


Figure capacity and latency of different storages [2]

Due to the high cost and low capacity of L1-3 caching, designers have created three different caching methods to minimise the chances a CPU will need to request data/address from main memory, which would be slower.

## Direct Mapped Cache

Direct mapping divides an address into three parts: tag bits , set bits , and word bits. The tag bits are used to locate a blocks position in main memory while the set bits also called line bits are used to identify the line of the cache in which the block of interest is stored. Lastly the word bits that are the least significant bits are used to identify the specific words within a block of memory.

Direct mapping assigns each memory block in main memory to a particular line in the cache. If the line is already filled with a memory block and a new bock need to be loaded, then old block is overwritten with the new block.

Diagram

Description automatically generated

Figure Direct Mapping method [3]

In figure 3 assuming a 9-bit set we would have a cache with 512 lines; therefore, the main memory is segmented into blocks of 512 and each of these will directly map into the cache lines 0 – 511.

To implement a direct mapped cache, we first need to view the steps preformed in a direct mapped process.

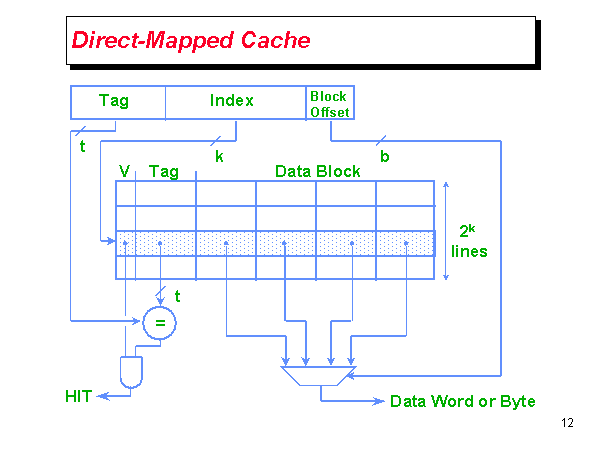


Figure Direct map implementation [4]

The cache will have fields for V (valid) , Tag , Data Block.

Assuming all cache fields are initialised to null.

The Index bits K from the address is used to access the cache line.

The Tag field inside the cache is then compared with the Tag of the address.

The Valid field is checked for True.

If these are True, a HIT has occurred, and the Block Offset is used to multiplex the data from the cache data block.

If these are FALSE, a MISS has occurred, and address is saved in the cache and the valid is set to true.

## 2-Way Set Associative Cache

A N-way set associative cache reduces chances of a cache miss by using N blocks in each set where data mapping to the set can be checked. Each memory address still maps to a specific set, but it can map to any one of N sets. The direct mapped cache design is also known as the one way set associative cache.

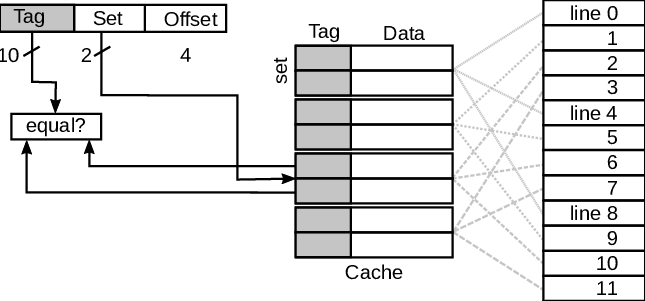


Figure 2-Way Set Associative method [5]

Figure 5 shows a 2-way set associative cache with 8 cache lines, each line 16 bytes. This time the Tag is checked in each of the two sets for a cache hit or miss.

To implement a 2-way set associative cache, we first need to view the steps preformed in a 2-way set associative process.

Diagram

Description automatically generated with medium confidence

Figure 2-Way Set Associative implementation [4]

Each of the 2-way cache will have fields for V (valid) , Tag , Data Block.

Left side cache will be referred to as way 0, right side will be way 1.

Assuming all cache fields are initialised to null.

The Index bits K from the address is used to access the way 0 & 1 cache line.

The Tag field inside the two ways is then compared with the Tag of the address.

The Valid field is checked for True.

If one of the ways returns True, a HIT has occurred, and the Block Offset is used to multiplex the data from the cache data block.

If both of the ways returns FALSE, a MISS has occurred, and address is saved in either of the ways, where the LRU (least recently used ) is also true. This valid is set to true.

## Associative Cache

A fully associative cache contains a single set with B ways, where B is the number of blocks. Here a memory address can map to a block in anywhere. An address will be partitioned into offset and tag fields, note no index field is used here.

The address is simultaneously compared with all stored addresses using comparators. Fully associative caches usually require more a large number of comparators due to the high number of comparisons executed, due to this it typically has the lowest miss rate.

Diagram

Description automatically generated

Figure Associative Cache method [6]

In figure 7 all tags are compared to the block number in parallel. If one tag matches the block number, it is a hit and the corresponding data using the offset is sent.

To implement a fully associative cache, we first need to view the steps preformed in a fully associative process.

Diagram

Description automatically generated

Figure Associative Cache implementation [4]

Each of the 2-way cache will have fields for V (valid) , Tag , Data Block.

Left side cache will be referred to as way 0, right side will be way 1.

Assuming all cache fields are initialised to null.

The Tag field is used to compare all the cache tags in parallel.

The Valid field is checked for True.

If these are True, a HIT has occurred, and the Block Offset is used to multiplex the data from the cache data block.

If these are FALSE, a MISS has occurred, and address is saved in the cache and the valid is set to true.

## Advantages and Disadvantages compared

In Table 1 we have taken an overview of the advantages and disadvantages of the three cache designs.

Table Three cache designs compared

|  |  |  |
| --- | --- | --- |
| **Cache method** | **Advantages** | **Disadvantages** |
| Direct Mapping | simplest type of cache memory mapping,  fast: only tag field is required to match.  least expensive | Each block of main memory maps to a fixed location in the cache,  performance not as good as other methods. |
| Associative Mapping | fast,  Easy to implement | expensive,  need to store address along with the data. |
| Set-Associative Mapping | two or more words can be stored under the same index address,  highest hit-ratio | very expensive as the set size increases. |

# Programming the Cache Simulator

Given the address is split into Tag, Set and Word fields we will use bitwise operator ‘&’ to isolate these fields during the program.

|  |  |  |
| --- | --- | --- |
| TAG (16 Bits) | SET (14 bits) | WORD (2 bits) |

The Tag field is 16-bits, therefore keeping these bits as ‘1’ and rest ‘0’ will yield:

11111111111111110000000000000000 = FFFF0000 in hex

The Set field is 14-bits, therefore keeping these bits as ‘1’ and rest ‘0’ will yield:

00000000000000001111111111111100 = 0000FFFC in hex

The word field is 2-bits, therefore keeping these bits as ‘1’ and rest ‘0’ will yield:

00000000000000000000000000000011 = 00000003 in hex

For the associative cache the tag field will be the sum of Tag (16 Bits) + SET (14 bits) = 30 bits. Therefore, keeping these bits as ‘1’ and rest ‘0’ will yield:

11111111111111111111111111111100 = FFFFFFFC in hex

At the top of main.cpp file declare the tag, set and word fields in hex, used to execute an ‘&’ operation with the input addresses to isolate the relevant fields.

#define TagHex 0xFFFF0000;

#define WordHex 0x00000003;

#define SetHex 0x0000FFFC;

#define FULL\_Assoctiv\_TagHex 0xFFFFFFFC;

To implement each of the 3 cache designs, we will create a *struct* for a cache line containing the LRU, VALID, DATA and TAG fields as shown below:

struct cachesEntry

{

bool LRU;

bool invaild;

unsigned short Tag;

char bytes[4];

};

This will then be used to declare an array of cache lines for direct mapped. Set is 16 bits, 2^16 = 16384 the index size.

const int index\_size = 16384;

struct cachesEntry directmap[index\_size];

struct cachesEntry TwoWayCacheing[2][index\_size];

struct cachesEntry FullyAssociativeMap[index\_size];

There will be individual functions declared to execute the direct map cache design, there will be a 32-bit address passed (type unsigned int) into these functions with no return value.

void DirectMap(unsigned int CPUaddress);

void Twowayset(unsigned int CPUaddress);

void SetAssociate(unsigned int CPUaddress);

## The Code in Main

The fields in the arrays will be initialised to 0 using a *for* loop.

The relevant variables will be declared such as variables to store the number of hits and misses.

int HIT = 0;

int MISS = 0;

for (int i = 0; i < index\_size; i++)

{

directmap[i].invaild = true;

directmap[i].LRU = true;

directmap[i].Tag = true;

FullyAssociativeMap[i].invaild = true;

FullyAssociativeMap[i].LRU = true;

FullyAssociativeMap[i].Tag = true;

}

for (int x = 0; x < 2; x++) {

for (int i = 0; i < index\_size; i++) {

TwoWayCacheing[x][i].invaild = true;

TwoWayCacheing[x][i].LRU = true;

TwoWayCacheing[x][i].Tag = true;

}

}

A file will be read in containing addresses, they will be treated as hex before conversion into *unsigned int* in order to store the full 32-bit address.

Using a ‘while not at end of file’ execution we will read in each address from file and pass this into one of the three cache functions.

filein.open("address\_file.txt", ios::in);

while (!filein.eof()){

filein >> hex >> cpuAddr;

DirectMap(CPUaddr);

//Twowayset(CPUaddr);

//SetAssociate(CPUaddr);

}

filein.close();

After the end of file has been reached, we will have used one of three cache designs, the relevant outputs will have been shown during and after program completion for example the number of hits and misses will be shown at the end of the program:

std::cout << "\n---------------------------------------------" << std::endl;

std::cout << "Total number of hits => " << std::dec << HIT << std::endl;

std::cout << "Total number of misses => " << std::dec << MISS << std::endl;

std::cout << "---------------------------------------------" << std::endl;

## Direct mapped function

As each address is read in it will be passed to the direct map function.

void DirectMap(unsigned int CPUaddress){

The first line of code in the function will be to display the input address to the user as hex, this will assist during debug if the address are not being read in properly.

std::cout << "\n++++++Address has been read in :: " << hex << CPUaddress << std::endl;

The next step is to isolate the tag, set and word fields by ‘&’ with the predeclared hex values.

unsigned int BytesN = CPUaddress & WordHex;

unsigned int SetN = CPUaddress & SetHex;

unsigned int Tag = CPUaddress & TagHex;

The tag and set fields need to be right bit shifted, because the most significant bit has changed.

Example, if isolated TAG address is: 11101111011101010000000000000000

we want to shift the bits so that all the zeros are removed and the 16 bits in the tag should become the most significant bit (MSB), this will become:

1110111101110101 stored in the *unsigned int tag* variable.

Tag = Tag >> 16;

SetN = SetN >> 2;

Next display these field values to user.

std::cout << "\nThe resulting address in the form:"<< std::endl;

std::cout << "BytesN => " << hex<<BytesN << std::endl;

std::cout << "TAG => " << hex<<Tag << std::endl;

std::cout << "SetN => " << hex<<SetN << std::endl;

using an *if* statement we can use *SetN* value to index into to the direct map cache and check if the Tag matches also check if the line is not invalid. If the Tag matches and line is not invalid, a hit has occurred. *Else* we want to store the Tag field which if saved then change the invalid field.

if ((directmap[SetN].Tag == Tag) && (!directmap[SetN].invaild))

{

std::cout << "\nAddress is a Hit " << std::endl;

HIT++;

}

else

{

std::cout << "\nAddress is a Miss " << std::endl;

directmap[SetN].Tag = Tag;

directmap[SetN].invaild = false;

MISS++;

}

Before exiting the function, display the status of Hits and Miss to user:

std::cout << "\nStatus of hits and Miss:"<< std::endl;

std::cout << "Amount of hits => " << std::dec << HIT << std::endl;

std::cout << "Amount of misses => " << std::dec << MISS << std::endl;

}

### Direct map function flowchart

Diagram

Description automatically generated

## 2-Way Set Associative function

As each address is read in it will be passed to the Twowayset function.

void Twowayset(unsigned int CPUaddress){

The first line of code in the function will be to display the input address to the user as hex.

std::cout << "\n++++++Address has been read in :: " << hex << CPUaddress << std::endl;

The next step is to isolate the tag, set and word fields by ‘&’ with the predeclared hex values.

unsigned int BytesN = CPUaddress & WordHex;

unsigned int SetN = CPUaddress & SetHex;

unsigned int Tag = CPUaddress & TagHex;

The tag and set fields need to be right bit shifted, because the most significant bit has changed.

Tag = Tag >> 16;

SetN = SetN >> 2;

Next display these field values to user.

std::cout << "\nThe resulting address in the form:"<< std::endl;

std::cout << "BytesN => " << hex<<BytesN << std::endl;

std::cout << "TAG => " << hex<<Tag << std::endl;

std::cout << "SetN => " << hex<<SetN << std::endl;

Now check the 2 ways for any hits, using a *for* loop, compare the isolated Tag and compare if the invalid field is false, this would mean a hit. Therefore, set the flag ‘flagHM’ to 1 and break from the loop. Else set the ‘flagHM’ to 0.

for (int x = 0; x < way; x++) {

if ((TwoWayCacheing[x][SetN].Tag == Tag) && (TwoWayCacheing[x][SetN].LRU == false )) {

std::cout << "\nAddress is a Hit " << std::endl;

flagHM = 1;

break;

}

else {

flagHM = 0;

}

}

The next if-else statement will increment the hit/miss counter.

if (flagHM == 1) {

HIT++;

}

else {

MISS++;

std::cout << "\nAddress is a Miss " << std::endl;

If address is a miss check the 2 ways for a valid LRU field and save address the there.

for (int x = 0; x < way; x++) {

if (TwoWayCacheing[x][SetN].LRU == true) {

TwoWayCacheing[x][SetN].Tag = Tag;

TwoWayCacheing[x][SetN].invaild = false;

TwoWayCacheing[x][SetN].LRU = false;

TwoWayCacheing[!x][SetN].Tag = true;

break;

}

}

}

Before exiting the function, display the status of Hits and Miss to user:

std::cout << "\nStatus of hits and Miss:"<< std::endl;

std::cout << "Amount of hits => " << std::dec << HIT << std::endl;

std::cout << "Amount of misses => " << std::dec << MISS << std::endl;

}

### 2-way Set Associative function flowchart

Diagram

Description automatically generated

## Set Associative function

As each address is read in it will be passed to the SetAssociate function.

void SetAssociate(unsigned int CPUaddress){

Declare to flags, “flag” and “flagHM”. The first is used to run code if the cache is full. The second is a hit miss flag where 1 is a hit and 0 is a miss, used to increment the hit/miss variables in main.

int flag = 0;

int flagHM = 0;

The first line of code in the function will be to display the input address to the user as hex, this will assist during debug if the address are not being read in properly.

std::cout << "\n+++++Address has been read in :: " << hex << CPUaddress << std::endl;

The next step is to isolate the tag and word fields by ‘&’ with the predeclared hex values.

unsigned int BytesN = CPUaddress & WordHex;

unsigned int Tag = CPUaddress & FULL\_Assoctiv\_TagHex;

The tag field needs to be right bit shifted.

Tag = Tag >> 2;

Next display these field values to user.

std::cout << "\nThe resulting address in the form:" << std::endl;

std::cout << "BytesN => " << BytesN << std::endl;

std::cout << "TAG => " << Tag << std::endl;

Now check the entire cache for any hits, using a *for* loop to check all cache lines, compare the isolated Tag and compare if the invalid field is false, this would mean a hit. Therefore, set the flag ‘flagHM’ to 1 and break from the loop. Else set the ‘flagHM’ to 0 and continue iterating through the cache.

for (int x = 0; x < index\_size; x++) {

if ((FullyAssociativeMap[x].Tag == Tag) && (FullyAssociativeMap[x].invaild == false)){

std::cout << "\nAddress is a Hit " << std::endl;

flagHM = 1;

break;

}

else{

flagHM = 0;

}

}

The next if-else statement will increment the hit/ miss counter.

if (flagHM == 1)

{

HIT++;

}

else if (flagHM == 0)

{

MISS++;

Use the first invalid space in cache to save the address. There is a flag inside the *if* condition which will be 0, if this section never ran, which would mean the cache is full.

for (int x = 0; x < index\_size; x++){

if (FullyAssociativeMap[x].invaild == true){

flag = 1;

std::cout << "\nAddress is a Miss " << std::endl;

FullyAssociativeMap[x].Tag = Tag;

FullyAssociativeMap[x].invaild = false;

break;//exit loop once address saved

}

}

If this flag is 0, the cache is full, therefore we will move all cache contents down one line.

if (flag == 0)

{

for (int x = 0; x < index\_size; x++)//move all cache down

{

FullyAssociativeMap[x+1] = FullyAssociativeMap[x ];

}

Save address to index zero, the top of the cache.

FullyAssociativeMap[0].Tag = Tag;

FullyAssociativeMap[0].invaild = false;

}

}

Before exiting the function, display the status of Hits and Miss to user:

std::cout << "\nStatus of hits and Miss:"<< std::endl;

std::cout << "Amount of hits => " << std::dec << HIT << std::endl;

std::cout << "Amount of misses => " << std::dec << MISS << std::endl;

}

### Set Associative function flowchart

Diagram

Description automatically generated

# Modularity

To add modularity into the program we will encapsulate parts of the code into functions, this will also help to declutter the code in main.

## The initilise\_all\_cache() function

This function will be called from main to initialise the cache fields. We could have only set the invalid field to true and left out the other fields. but for clarity all fields were initialised.

void initilise\_all\_cache() {

for (int i = 0; i < index\_size; i++){

directmap[i].invaild = true;

directmap[i].LRU = true;

directmap[i].Tag = true;

FullyAssociativeMap[i].invaild = true;

FullyAssociativeMap[i].LRU = true;

FullyAssociativeMap[i].Tag = true;

}

for (int x = 0; x < 2; x++) {

for (int i = 0; i < index\_size; i++) {

TwoWayCacheing[x][i].invaild = true;

TwoWayCacheing[x][i].LRU = true;

TwoWayCacheing[x][i].Tag = true;

}

}

}

## The user\_cache\_pick() function

This function will have a return of type int and the argument will also be type int. This will be called from main; it will check the file and if the file is ok the int value will be used to select one of the three cache methods.

The return type will be 0 if the file is not ok or if the user value is outside the range 1-3.

The return type will be 1 if the user value is in the range 1-3.

In main the return type will be used in an *if* statement to run code to display the total hits and misses.

**In main**

if (user\_cache\_pick(ans)) {

std::cout << "\n---------------------------------------------" << std::endl;

std::cout << "Total number of hits => " << std::dec << HIT << std::endl;

std::cout << "Total number of misses => " << std::dec << MISS << std::endl;

std::cout << "---------------------------------------------" << std::endl;

}

**The function**

int user\_cache\_pick(int a) {

my\_file.open(fileName, ios::in);

if (!my\_file){

std::cout << "Error in reading file: " << std::endl;

return 0;

}

if (a == 1){

while (my\_file >> hex >> CPUaddr)

{

DirectMap(CPUaddr);

}

my\_file.close();

}

else if (a == 2){

while (my\_file >> hex >> CPUaddr)

{

Twowayset(CPUaddr);

}

my\_file.close();

}

else if (a == 3){

while (my\_file >> hex >> CPUaddr)

{

SetAssociate(CPUaddr);

}

my\_file.close();

}

else {

std::cout<<"!!Error you did not select any number between 1-3 "<< std::endl;

return 0;

}

return 1;

}

# Testing Direct map

Using the file “input\_addr\_DM\_9M\_0H” to test the direct map cache we should expect 9 misses and 0 hits in total.

Text

Description automatically generated with medium confidence

Figure Test file for direct map

The result is shown in Appendix A, where we validate that there are 9 Misses and 0 Hits and the program for direct map works as intended.

# Testing 2-Way Set Associative map

Using the file “input\_addr\_2w\_8M\_1H” to test the 2-way set associative map cache we should expect 8 misses and 1 hit in total.

Text

Description automatically generated

Figure Test file for 2-Way Set Associate map

The result is shown in Appendix B, where we validate that there are 8 Misses and 1 Hits and the program for 2-way set map works as intended.

# Testing Set Associative

Using the file “input\_addr\_setassocitive\_7M\_2H” to test the set associative map cache we should expect 7 misses and 2 hit in total.

Graphical user interface, text

Description automatically generated

Figure Test file for Set Associative map

The result is shown in Appendix C, where we validate that there are 7 Misses and 2 Hits and the program for set associative map works as intended.

# Conclusion

We have successfully implemented the three cache designs, tested and validated the code.

Appendix D contains the full program code.

# Bibliography

[1]"Compute Performance – Distance of Data as a Measure of Latency | Formulus Black | In-Memory Storage", *Formulus Black | In-Memory Storage*. [Online]. Available: https://formulusblack.com/blog/compute-performance-distance-of-data-as-a-measure-of-latency/. [Accessed: 28- Nov- 2021].

[2]C. Pohl, K. Sattler and G. Graefe, "Joins on high-bandwidth memory: a new level in the memory hierarchy", *The VLDB Journal*, vol. 29, no. 2-3, pp. 797-817, 2019. Available: 10.1007/s00778-019-00546-z.

[3]"What is Direct Mapping Process in Computer Architecture?", *Tutorialspoint.com*. [Online]. Available: https://www.tutorialspoint.com/what-is-direct-mapping-process-in-computer-architecture. [Accessed: 28- Nov- 2021].

[4]"Direct-Mapped Cache", *People.csail.mit.edu*. [Online]. Available: https://people.csail.mit.edu/devadas/6.004/Lectures/lect18/sld012.htm. [Accessed: 28- Nov- 2021].

[5]Y. Chen, "Secure In-Cache Execution", 2017. [Online]. Available: https://www.researchgate.net/figure/2-way-set-associative-cache-8-cache-lines-in-4-sets-Each-cache-line-is-16-bytes\_fig2\_318083055. [Accessed: 28- Nov- 2021].

[6]L. Rappoport and A. Yoaz, "Computer Structure 2015 – Caches 1 Lihu Rappoport and Adi Yoaz Computer Structure Cache Memory. - ppt download", *Slideplayer.com*, 2015. [Online]. Available: https://slideplayer.com/slide/9853910/. [Accessed: 28- Nov- 2021].

# Appendix A Direct map result

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

# Appendix B 2-Way Set Associative result

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

# Appendix C Set Associative result

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

# Appendix D full program code

#include <iostream>

#include <string>

#include <fstream>

#include <bitset>

#include <math.h>

using namespace std;

#define ADDRESSBUS\_BITS 32

#define DATABUS\_BITS 8

#define TAG\_BITS 16

#define SET\_BITS 14.0

#define WORD\_BITS 2

#define TagHex 0xFFFF0000;

#define WordHex 0x00000003;

#define SetHex 0x0000FFFC;

#define FULL\_Assoctiv\_TagHex 0xFFFFFFFC;

struct cachesEntry

{

bool LRU;

bool invaild;

unsigned int Tag;

char bytes[4];

};

const int index\_size = 16384;

struct cachesEntry directmap[index\_size];

struct cachesEntry TwoWayCacheing[2][index\_size];

struct cachesEntry FullyAssociativeMap[index\_size];

void DirectMap(unsigned int CPUaddress);

void Twowayset(unsigned int CPUaddress);

void SetAssociate(unsigned int CPUaddress);

int user\_cache\_pick(int a);

void initilise\_all\_cache();

int HIT = 0;

int MISS = 0;

unsigned int CPUaddr;

fstream my\_file;

string fileName;

int main()

{

int ans;//to pick one of the cache designs

initilise\_all\_cache(); //to zero

fileName = "input\_addr\_setassocitive\_7M\_2H.txt";

std::cout << "\nFile used will be :" << fileName<< std::endl;

std::cout << "\n|----------------Which cacheing method would you like to go with ?-----------------|" << std::endl;

std::cout << "\n|-----(1)--DirectMapping--(2)--2-way-set-associative--(3)--full-set-associative----|" << std::endl;

std::cout << "\nYour input ==> ";

std::cin >> ans;

if (user\_cache\_pick(ans)) {

std::cout << "\n---------------------------------------------" << std::endl;

std::cout << "Total number of hits => " << std::dec << HIT << std::endl;

std::cout << "Total number of misses => " << std::dec << MISS << std::endl;

std::cout << "---------------------------------------------" << std::endl;

}

}

void DirectMap(unsigned int CPUaddress)

{

std::cout << "\n+++++++++++++++++++++++++++++++++++++Address has been read in :: " << hex << CPUaddress << std::endl;

//isolating the tag, set, wrd field by preforming a bitwise AND '&' operation

unsigned int BytesN = CPUaddress & WordHex; //bitwise AND the WordHex with cpu address, save to BytesN with size 32 bits

unsigned int SetN = CPUaddress & SetHex; //bitwise AND the SetHex with cpu address, save to SetN with size 32 bits

unsigned int Tag = CPUaddress & TagHex; //bitwise AND the TagHex with cpu address, save to Tag with size 32 bits

//right shift the tag and SetN

Tag = Tag >> 16; //eg, 11101111011101010000000000000000 >> 16

//tag => 00000000000000001110111101110101

SetN = SetN >> 2;

std::cout << "\nThe resulting address in the form:"<< std::endl;

std::cout << "BytesN => " << hex<<BytesN << std::endl; //testing

std::cout << "TAG => " << hex<<Tag << std::endl; //testing

std::cout << "SetN => " << hex<<SetN << std::endl; //testing

std::cout << "\n=================================================HIT=========MISS======= " << std::endl;

//using the setN value to access the relevent cache line

//checking if the tag matched and its not invalid

if ((directmap[SetN].Tag == Tag) && (!directmap[SetN].invaild))

{

std::cout << "\nAddress is a Hit " << std::endl;

HIT++;

}

else

{

//if address is invalid, its a miss, save the tag and flip the invalid.

std::cout << "\nAddress is a Miss " << std::endl;

directmap[SetN].Tag = Tag;

directmap[SetN].invaild = false;

MISS++;

}

std::cout << "\n======================================================================== " << std::endl;

std::cout << "The line number of the data loaded from memory is " << SetN << std::endl;

std::cout << "\nStatus of hits and Miss:"<< std::endl;

std::cout << "Amount of hits => " << std::dec << HIT << std::endl;

std::cout << "Amount of misses => " << std::dec << MISS << std::endl;

}

void Twowayset(unsigned int CPUaddress)

{

int way = 2; //2-way, declared for future modification for N-way set associative

int flagHM = 0;//a hit miss flag, 0 = miss, 1 = hit

std::cout << "\n+++++++++++++++++++++++++++++++++++++Address has been read in :: " << hex << CPUaddress << std::endl;

//isolating the tag, set, wrd field by preforming a bitwise AND '&' operation

unsigned int BytesN = CPUaddress & WordHex; //bitwise AND the WordHex with cpu address, save to BytesN with size 32 bits

unsigned int SetN = CPUaddress & SetHex; //bitwise AND the SetHex with cpu address, save to SetN with size 32 bits

unsigned int Tag = CPUaddress & TagHex; //bitwise AND the TagHex with cpu address, save to Tag with size 32 bits

//right shift the tag and SetN

Tag = Tag >> 16;

SetN = SetN >> 2;

std::cout << "\nThe resulting address in the form:" << std::endl;

std::cout << "BytesN => " << BytesN << std::endl;

std::cout << "TAG => " << Tag << std::endl;

std::cout << "SetN => " << SetN << std::endl;

std::cout << "\n=================================================HIT=========MISS======= " << std::endl;

//using 'for' loop to access way 0 and way 1 (these are the 2-way)

//during the for loop using setN to access cacheline and check if tag and lru conditions met

for (int x = 0; x < way; x++) {

if ((TwoWayCacheing[x][SetN].Tag == Tag) && (TwoWayCacheing[x][SetN].LRU == false )) {

//if conditions met, its a hit, set the flagHM = 1 (means hit ) and break from loop

std::cout << "\nAddress is a Hit " << std::endl;

flagHM = 1;

break;

}

else {

flagHM = 0;

}

}

//check flagHM value, if its 1 means hit, then increment hit counter

if (flagHM == 1) {

HIT++;

}

else {

// else means its 0 ie a miss, then increment miss counter

MISS++;

std::cout << "\nAddress is a Miss " << std::endl;

//using 'for' go into way 0 & 1 and find the least reciantly used (LRU) cache line and save the address there

for (int x = 0; x < way; x++) {

if (TwoWayCacheing[x][SetN].LRU == true) {

TwoWayCacheing[x][SetN].Tag = Tag;

TwoWayCacheing[x][SetN].invaild = false;

TwoWayCacheing[x][SetN].LRU = false;

TwoWayCacheing[!x][SetN].Tag = true;

break;

}

}

}

std::cout << "\n======================================================================== " << std::endl;

std::cout << "The line number of the data loaded from memory is " << SetN << std::endl;

std::cout << "SetN => " << SetN << std::endl;

std::cout << "\nStatus of hits and Miss:" << std::endl;

std::cout << "Amount of hits => " << std::dec << HIT << std::endl;

std::cout << "Amount of misses => " << std::dec << MISS << std::endl;

}

void SetAssociate(unsigned int CPUaddress)

{

std::cout << "\n+++++++++++++++++++++++++++++++++++++Address has been read in :: " << hex << CPUaddress << std::endl;

//isolating the tag, wrd field by preforming a bitwise AND '&' operation

unsigned int BytesN = CPUaddress & WordHex; //bitwise AND the WordHex with cpu address, save to BytesN with size 32 bits

unsigned int Tag = CPUaddress & FULL\_Assoctiv\_TagHex; //bitwise AND the TagHex with cpu address, save to Tag with size 32 bits

/\*

FULL\_Assoctiv\_TagHex defind at top as "#define FULL\_Assoctiv\_TagHex 0xFFFFFFFC;" //11111111111111111111111111111100

we have added the old tag (16-bits) + set (14-bits) fields to create the new tag field for fully associative

FULL\_Assoctiv\_TagHex field of 30 bits

\*/

//right shift the tag

Tag = Tag >> 2;

std::cout << "\nThe resulting address in the form:" << std::endl;

std::cout << "BytesN => " << BytesN << std::endl;

std::cout << "TAG => " << Tag << std::endl;

std::cout << "\n=================================================HIT=========MISS======= " << std::endl;

int flag = 0;// will be 0 if there was no avalible sapce in cache to store address, ie cache is full

int flagHM = 0;// hit miss flag, 1 = hit, 0 = miss

//checking for any hit in cache, if there is a hit set the flagHM flag to 1 and break from loop

for (int x = 0; x < index\_size; x++) {

if ((FullyAssociativeMap[x].Tag == Tag) && (FullyAssociativeMap[x].invaild == false)){

std::cout << "\nAddress is a Hit " << std::endl;

flagHM = 1;

break;

}

else{

flagHM = 0;

}

}

//using flagHM to increment the hit/ miss counter

if (flagHM == 1)

{

HIT++;

}

else if (flagHM == 0)

{

MISS++;

//finding the first invalid space in cache and save address there

for (int x = 0; x < index\_size; x++){

if (FullyAssociativeMap[x].invaild == true){

//using another flag called "flag", will be 0 if this "if" condtion never ran meaning address was never saved

flag = 1;

std::cout << "\nAddress is a Miss " << std::endl;

FullyAssociativeMap[x].Tag = Tag;

FullyAssociativeMap[x].invaild = false;

break;//exit loop once address saved

}

}

//if "flag" == 0, means it was never saved in cache, therefore we execute FIFO policy

if (flag == 0)

{

//move all cache lines down one

//therefore the last index will be removed

for (int x = 0; x < index\_size; x++)//move all cache down

{

//move all down one

FullyAssociativeMap[x+1] = FullyAssociativeMap[x ];

}

//once cache lines are moved down

//save the address to the top, ie index 0

FullyAssociativeMap[0].Tag = Tag;

FullyAssociativeMap[0].invaild = false;

}

}

std::cout << "\n======================================================================== " << std::endl;

std::cout << "\nAmount of hits => " << std::dec << HIT << std::endl;

std::cout << "Amount of misses => " << std::dec << MISS << std::endl;

}

void initilise\_all\_cache() {

//initilise the cache for directmap & FullyAssociativeMap together as they are 1-D array

//could have only set the invalid to true, and left out the other fields.

//but for clarity all fields were initialised

for (int i = 0; i < index\_size; i++){

directmap[i].invaild = true;

directmap[i].LRU = true;

directmap[i].Tag = true;

FullyAssociativeMap[i].invaild = true;

FullyAssociativeMap[i].LRU = true;

FullyAssociativeMap[i].Tag = true;

}

//TwoWayCacheing declared as a 2-d array so use 2 'for' loops to initilise the fields

for (int x = 0; x < 2; x++) {

for (int i = 0; i < index\_size; i++) {

TwoWayCacheing[x][i].invaild = true;

TwoWayCacheing[x][i].LRU = true;

TwoWayCacheing[x][i].Tag = true;

}

}

}

int user\_cache\_pick(int a) {

my\_file.open(fileName, ios::in);

if (!my\_file){

std::cout << "Error in reading file: " << std::endl;

return 0;

}

if (a == 1)

{

while (my\_file >> hex >> CPUaddr) //converting hex to decimal int cpuaddr

{

DirectMap(CPUaddr);

}

my\_file.close();

}

else if (a == 2)

{

while (my\_file >> hex >> CPUaddr) //converting hex to decimal int cpuaddr

{

Twowayset(CPUaddr);

}

my\_file.close();

}

else if (a == 3)

{

while (my\_file >> hex >> CPUaddr) //converting hex to decimal int cpuaddr

{

SetAssociate(CPUaddr);

}

my\_file.close();

}

else {

std::cout << "!!Error you did not select any number between 1-3 " << std::endl;

return 0;

}

return 1;

}