Department of Electrical Engineering and Computer Science

COSC201- Computer System Organization

PROJECT _FALL 2021

Design of Trace Driven Direct-mapped Cache Simulator

By:

Rashed Aljaberi

ID: 100053179

Khalid Al Sayed

ID: 100052995

Overview:

In this project, we will explore the effectiveness of caches and the impact of different cache configurations. To do this, we will write a C program that simulates the behavior of a cache.

Introduction:

A good processor could compute an instruction in only a picosecond. So theoretically, it should be able to compute 2 instruction in 2 picoseconds, right? Wrong. The thing stopping it from doing so is the memory gap. Even though a processor could finish a cycle in a picosecond, it can't compute the next instruction until it reaches the processor from main memory. And that time it takes is actually so big, it is more than 1000 times the time of a cycle.

To resolve this issue, we need faster memory. But the problem with faster memory is that it's expensive, forcing us to have lesser capacity. One solution to this problem was to eliminate the time it takes to travel from main memory to the CPU by "bringing the CPU to the memory". This is a concept known as in-memory computing. Right now, there is no known implementation of this concept on a consumer-level computer, and it seems it might happen in the near future. Instead, the solution to the memory gap that is used by most computers is by introducing

intermediate memory levels, each one smaller in capacity but faster than the one below it. Also, each level acts like a 'cache' (A temporary storing destination that is faster) to the level below it. This concept is known as memory hierarchy.

Also keep in mind that the term 'cache' usually refers to the SRAM, which is the cache for the main memory (DRAM). As mentioned, each level is smaller than the one below it. As a result, the CPU has to decide what 'blocks'(a group of consecutive memory addresses also known as pages) to place in what cells in the cache. This is known as mapping. There are a lot of mapping techniques like Direct, Associative, and Set-Associative mapping

Direct mapping is a technique that gives a block in main memory a corresponding cache line depending on the middle bits of the address that makes the 'set' bits. But since main memory is bigger, more than one block could fit in the same cache cell. That's why the most significant bits of the address that make the 'tag' bits are used to identify the addresses saved in the cache. Based on the tag, the CPU could distinguish which block is in the cache. If the tag in the cache matches the one requested by the CPU, then this is a 'hit'. Finally, the least significant bits of the address known as 'offset' are used to identify which byte of the block is requested. If the CPU doesn't find the address in the cache, it will then consider it a 'miss' and bring the entire block that the address is in to the corresponding line. It does that because of the principle of locality, that states that if you needed an address, you will probably need it or need an address near to it in the near future.

Another technique is Associative (or Fully-Associative) mapping, where blocks don't have corresponding cache cells and can be placed in any cell. Because of that, the CPU has to check every single cell until it finds a cell with the corresponding tag. This results in a "miss penalty" (the time lost checking cells to find the corresponding tag). If not found, the CPU replaces one of the cells by the block/page in question. It could choose what cells to replace based on many replacement policies, like random replacement policy or FIFO.

Set-associative technique is similar to Associative but it divides each line to multiple sets. It could be 2-way, 4-way or whatever. This results in a smaller miss rate, but is also more expensive to implement.

Even the cache (SRAM) has multiple levels in it, each faster but smaller than the previous (L1, L2, L3, respectively).

Project Statement:

In this project, we will write a program to simulate a cache. The simulator takes memory addresses as input (as if it was coming from the CPU) and then return the statues of the cache (e.g.: cache hits/misses and cache accesses) or modify it. The simulator is designed for 32-bit memory addresses and 32KB cache sizes with blocks of size 64B. The simulator also uses Direct-mapping technique.

Calculations:

Before designing the simulator, we needed to calculate some parameters: the number of bits for the "block offset", for the 'index', and finally for the 'tag'.

Offset bits: This was found by calculating how many bits we need to represent a block with 64 bytes.

$$2^b=64$$

$$b=6$$

thus, we needed 5 bits for the offset

Index: number of lines or sets was found by dividing the cache size by block size. Then, the index bits were found by calculating how many bits we need to represent such number of lines

$$lines = \frac{32KB}{64B} = \frac{32 \times 1024}{64} = 512$$

$$\text{Index} = 2^i=512$$

$$i=9$$

thus, we needed 9 bits for the index

tag: this was simple found by subtracting the index and offset bit from the total address bits.

$$t=32-9-6$$

$$t=17 \text{ bits}$$

cache hit:

Determining if an address is in the cache is easy and quick. First, we use the index bits to go to the corresponding cache cell. Here is a simple example of a 3-bit address (1 for tag, 1 for index, and 1 for offset). This means we have 8 unique addresses and 4 cache cells. The cache has 2 lines/blocks which means each block has 2 addresses. Each cache cell in this example could correspond to 2 different addresses, or in other words, each line could correspond to 2 different pages/blocks. The cache lines are color coded in this example to show what line corresponds to what page in main memory.

Address	Value
000	A
001	B
010	C
011	D
100	E
101	F
110	G
111	H

	0	1
0	A	B
1	G	F

tags
0
1

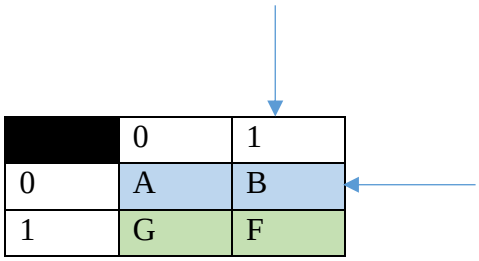
Let's say in this example, the CPU asks for the address '001'. First it will take the index bit. In this case, it is the middle bit (0). It will go that line in the cache that has that index.

	0	1
0	A	B
1	G	F

Then, to make sure that the page in this line is the one we want, we compare the tag. The tag here is the most significant bit (0). If the tag bit matches the tag in the cache line, then we have a hit

tags
0
1

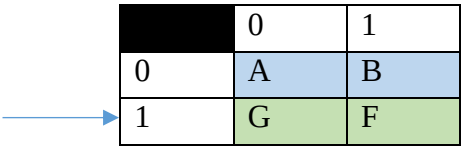
Since it is a hit, we look into the offset bit. Here, the offset is the least significant bit (1). We go to the cell corresponding to the offset and get the value and return it to the CPU



	0	1
0	A	B
1	G	F

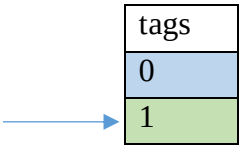
Thus, we return B.

But what if the tag bits didn't match? Let's say this time the CPU was looking for '011'. The index bit here is '1'. So we go to the line with that index.



	0	1
0	A	B
1	G	F

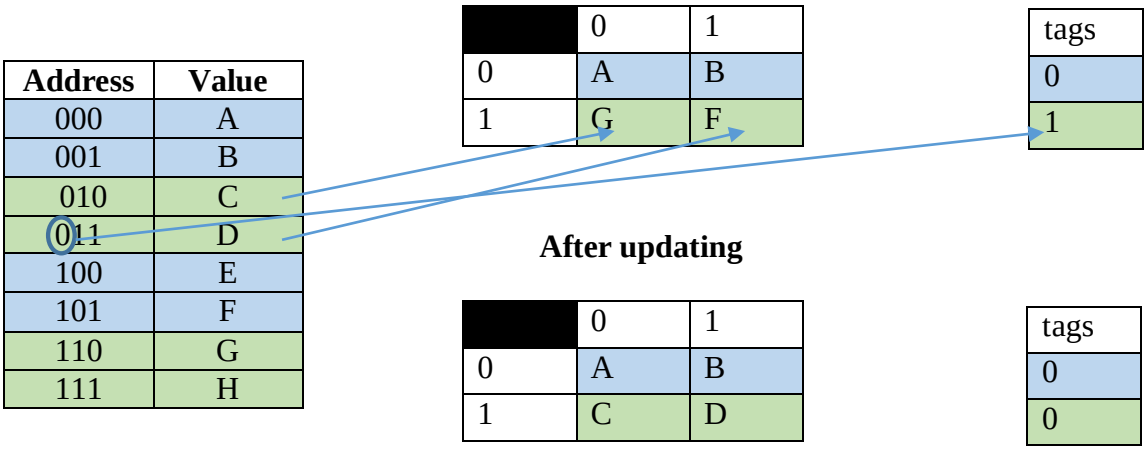
Now we compare the tag. The tag we have is '0' but the tag in the cache is '1'



tags
0
1

So this means that this is a miss and the value in the cache is not the one we are looking for. So what the CPU does is that it goes and fetches the page that contains this address from main memory and copies it to this line. It then updates the tag accordingly.

Before updating



Address	Value
000	A
001	B
010	C
011	D
100	E
101	F
110	G
111	H

	0	1
0	A	B
1	G	F

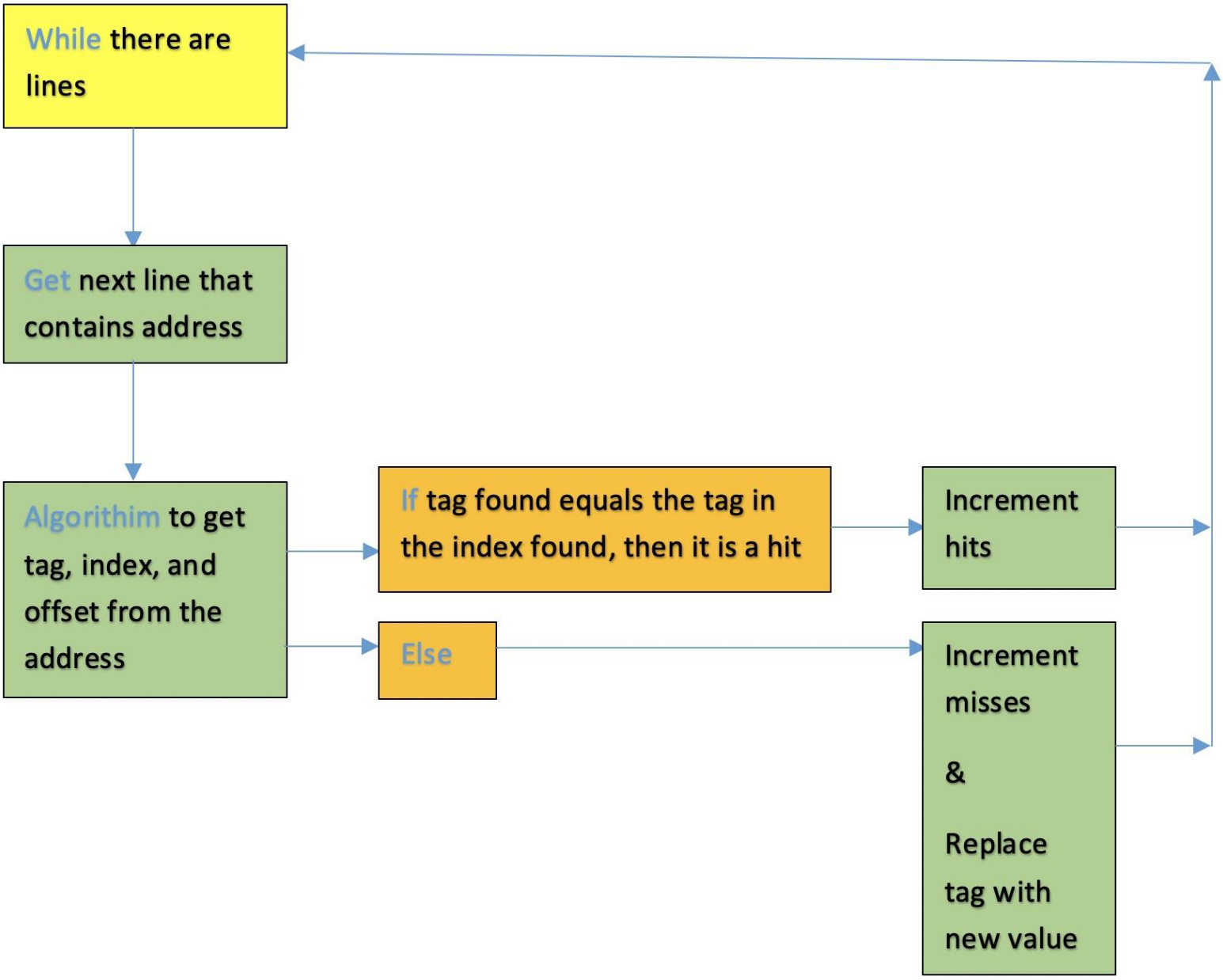
tags
0
1

After updating

	0	1
0	A	B
1	C	D

tags
0
0

Flow chart:


While there are lines
Get next line that contains address
Algorithim to get tag, index, and offset from the address
If tag found equals the tag in the index found, then it is a hit
Increment hits
Else
Increment misses
&
Replace tag with new value

Code:

```
#include <stdio.h>

void sim(int tags[],int lines){

	FILE *ptr; //pointer to file

	ptr = fopen("trace.txt","r"); //open trace file

	int miss=0,hit=0; //holds total misses and hits

	char address[8]; //holds the addrress aquired from the file

	int i=0; //counter for line in file

	while(i<lines){

		fgets(address,9, ptr); //gets a line from text file and places it in address

		unsigned int n = strtol(address, NULL, 16), //convert address taken from "trace.txt" and placed in 'address' from char to int

		offset=n & 0x3f, //f=get block offset of address

		set=(n>>6)&0x1ff, //get index/set/line/block

		tag=n>>15; //get tag

		if(tags[set]==tag){ //checks if the tags bits match the tag at the index for a hit

			printf("cahce hit for %x at line numebr %d block offset %d\n",n,set,offset);

			hit++; //update hit counter

		}

		else{ //otherwise we got a miss

			printf("cache miss at %x, updating cache by moving respecitve page to line %d\n",n,set);

			miss++; //update miss counter

			tags[set]=tag; //update tag at the index so that the new page is moved

		}

		fgets(address,9, ptr); //after every read, fgets gives a gibbirish value, so this inscures it ignores it and only read every other time.

		i++; //update counter

	}

	fclose(ptr); //close trace file

	printf("there was a total of %d hits and %d misses and %d accesses. the program also updated the cache %d times, the same number as the misses\n",hit,miss,hit+miss,miss);

}

int main(){

	int addresses=100; //numbers of adresses to be computed from trace file

	int tags[512]={}; //holds the tags of all 512 blocks

	sim(tags,addresses);} //runs the simulation
```

```c
#include <stdio.h>

void sim(int tags[],int lines){

        FILE *ptr; //pointer to file
        ptr = fopen("trace.txt","r"); //open trace file
        int miss=0,hit=0; //holds total misses and hits
        char address[8]; //holds the addrress aquired from the file

        int i=0; //counter for line in file
        while(i<lines){

                fgets(address,9, ptr); //gets a line from text file and places it in address

                unsigned int n = strtol(address, NULL, 16), //convert address taken from "trace.txt" and placed in 'address' from char to int
                offset=n & 0x3f, //f=get block offset of address
                set=(n>>6)&0x1ff, //get index/set/line/block
                tag=n>>15; //get tag

                if(tags[set]==tag){ //checks if the tags bits match the tag at the index for a hit
                        printf("cahce hit for %x at line numebr %d block offset %d\n",n,set,offset);
                        hit++; //update hit counter
                }
                else{ //otherwise we got a miss
                        printf("cache miss at %x, updating cache by moving respecitve page to line %d\n",n,set);
                        miss++; //update miss counter
                        tags[set]=tag; //update tag at the index so that the new page is moved
                }

                fgets(address,9, ptr); //after every read, fgets gives a gibbirish value, so this inscures it ignores it and only read every other time.
                i++; //update counter
        }
        fclose(ptr); //close trace file
        printf("there was a total of %d hits and %d misses and %d accesses. the program also updated the cache %d times, the same number as the misses\n",hit,miss,hit+miss,miss);
}

int main(){
        int addresses=100; //numbers of adresses to be computed from trace file
        int tags[512]={}; //holds the tags of all 512 blocks
        sim(tags,addresses); //runs the simulation

}
```

```
rashed@rashed-VirtualBox:~/Desktop/project$ ./a.out
cache miss at d89cfe6e, updating cache by moving respecitve page to line 505
cache miss at c3ee0321, updating cache by moving respecitve page to line 12
cache miss at d0b8b687, updating cache by moving respecitve page to line 218
cache miss at d5ac4c07, updating cache by moving respecitve page to line 304
cache miss at d8a77801, updating cache by moving respecitve page to line 480
cache miss at 8c636b96, updating cache by moving respecitve page to line 430
cache miss at d84a73fc, updating cache by moving respecitve page to line 463
cache miss at cff6892c, updating cache by moving respecitve page to line 36
cahce hit for d84a73fc at line numebr 463 block offset 60
cache miss at d1550308, updating cache by moving respecitve page to line 12
cache miss at 4159dba4, updating cache by moving respecitve page to line 366
cache miss at ccca81c8, updating cache by moving respecitve page to line 7
cache miss at 409d5d95, updating cache by moving respecitve page to line 374
cache miss at c314e10e, updating cache by moving respecitve page to line 388
cache miss at 4026df6b, updating cache by moving respecitve page to line 381
cache miss at 825e8ee3, updating cache by moving respecitve page to line 59
cache miss at 825e8f82, updating cache by moving respecitve page to line 62
cache miss at d8216146, updating cache by moving respecitve page to line 389
cache miss at d8212a4b, updating cache by moving respecitve page to line 169
cache miss at d8736d07, updating cache by moving respecitve page to line 436
cache miss at ccca83ea, updating cache by moving respecitve page to line 15
cache miss at d885f34c, updating cache by moving respecitve page to line 461
cache miss at cf44b17d, updating cache by moving respecitve page to line 197
```

```
cahce hit for cbca968a at line numebr 90 block offset 10
cahce hit for ccfc04c3 at line numebr 19 block offset 3
cahce hit for cbca968a at line numebr 90 block offset 10
cahce hit for d0b8b687 at line numebr 218 block offset 7
cache miss at 421c2c34, updating cache by moving respecitve page to line 176
cahce hit for 421c2c34 at line numebr 176 block offset 52
cahce hit for 421c2c34 at line numebr 176 block offset 52
cahce hit for 421c2c34 at line numebr 176 block offset 52
cahce hit for ccfc04c3 at line numebr 19 block offset 3
cache miss at cf44b2fc, updating cache by moving respecitve page to line 203
cahce hit for d5ac4c07 at line numebr 304 block offset 7
cahce hit for 825e8ee3 at line numebr 59 block offset 35
cahce hit for d5ac4c07 at line numebr 304 block offset 7
cahce hit for cf99cb72 at line numebr 301 block offset 50
cahce hit for cff6980a at line numebr 96 block offset 10
cahce hit for cff69809 at line numebr 96 block offset 9
cahce hit for 4159dba4 at line numebr 366 block offset 36
cache miss at 424d3a21, updating cache by moving respecitve page to line 232
cache miss at ca608c6b, updating cache by moving respecitve page to line 49
cahce hit for d17eb5e5 at line numebr 215 block offset 37
cahce hit for d5ac4c07 at line numebr 304 block offset 7
there was a total of 46 hits and 54 misses and 100 accesses. the program also up
dated the cache 54 times, the same number as the misses
rashed@rashed-VirtualBox:~/Desktop/project$
```

What if we had 2 levels:

If we had 2 levels of cache instead of 1 (L1,L2), then the smaller faster one (L1) will act as a cache for the other one. Or another way of viewing it is that L2 will act as an intermediate between the cache and the main memory so that if there was a cache miss, we can still go to a place to check if the data might be there and then bringing it to the cache without suffering huge latency. This slightly changes the code by adding one more stop to check before going to main memory if we got a miss. If the second level also gives us a miss, then we go to main memory and grab the data then bring it to L2 and then from L2 to L1.

In Direct-mapping, there can be only 1 miss for 1 level cache. Since we have 2 levels, it could get 2 misses and this doubles the number of misses we can get. Additionally, now we have to move the data twice instead of once (once from main memory to L2 and once from L2 to L1) so this will increase the time it takes to move data. So the increase in number of possible misses and time it takes to move data will result in an increase in the miss penalty.

Hit rate:

```
there was a total of 821650 hits and 178350 misses and 1000000 accesses. the pro
gram also updated the cache 178350 times, the same number as the misses
```

In 1 level cache our simulator got 821,650 hits when we tried for 1,000,000 accesses. So to find hit rate we divide hits by total accesses.

$$hit\ rate = \frac{821{,}650}{1000000} = 0.82165 = 82.17\%$$

This puts the hit rate at 82.17%. As stated in the previous paragraph, if we had 2 levels the miss rate will go up, so our hit rate will go down.

Conclusion:

Cache and memory hierarchy is a very important concept introduced to solve the memory gap problem and create faster computing devices. In this project, we were able to show what a cache normally refers to, how there are different implementations of cache memory with different techniques, and how the CPU maps them. We then designed a simulator that simulates a 1-level cache that uses direct mapping and we explained how direct mapping works with an example. We also made a flow chart that simplifies the implementation of our program. Finally, we discussed how adding another level of cache will affect our simulator and what the new miss penalty would be.