CACHE

For the computer is easier to check small portions of programs rather than check an entire program. This is the reason why the computers have a small portion of memory called **cache**, this portion of memory is very fast and is very useful for the CPU. Nowadays, computers follow the principle of locality which states that programs access a small portion of a program with the cache. This principle can be extended in:

- Temporal locality: If a portion of the memory was stored in the cache because it was referenced, it will tend to be referenced again soon.
- Spatial locality: If a portion of the memory is stored in the cache because it was referenced, then it is probable that the closer portions of memory are referenced again in the future because they may belong to the same program.

Computers have a **memory hierarchy**, this means that the memory of a computer is divided in levels, the level which is closer to the CPU is also smaller, faster, and more expensive.

In this architecture computers have multiple levels, but the data is usually only at two adjacent levels at a time (memory and cache). The data is stored in the memory as **blocks.** When the processor requests some blocks he "asks" the upper level (in this case the *cache*), if the block asked is in the cache we call that a **hit**, otherwise it will be a **miss**, so if it is a miss the level above (Ram memory) is accessed in order to take the block needed.

From this concepts we have the **hit rate**, which measures the fraction of memory accesses found in the upper level, and the **miss rate**, measures the fraction of memory misses while accessing the memory. From these two concepts we have also a **hit time** which is the time needed to access the upper level, and we have a **miss penalty**, which is the time needed to replace the block in the upper level when we had a *miss*.

$$Miss\ ratio = 1 - hit\ ratio$$

$$Hit\ ratio = \frac{hits}{accesses}$$

The memories in the computer use different technologies, *cache* uses **SRAM Technology**, but **DRAM Technology** is used in *ram memory*.

The *cache* before any reference is empty, but, the more references, the more blocks written in the cache. But where do we store the memory in the cache? Well, we take the *address* in memory in order to place the block in the cache. This cache structure is called **direct mapped**, we usually do this by a simple operation:

$$(BLOCK\ ADDRES)MODULO(NUMBER\ OF\ BLOCKS\ IN\ THE\ CACHE)$$

Usually the *cache*, has a power of 2 entries, so then the indexes of the cache can be computed simply with:

$$log_2(cache\ size\ in\ blocks)$$

So, for example: A 8-block cache uses 3 bits to map his block ($8 = 2^3$), so the cache will have 001, 010 …

If we reference the position in memory 1101**001**, as we can see the last three digits are the ones that say in which "block" (index) of the *cache* is going to be stored that block from the memory. But this means that multiple blocks of memory can be stored in the same block of the *cache*, and how do we know whether a requested word is in the cache or not? For this we use **tags**. The *tags* contain the address information required to identify a memory block. In our case the tag is the information we do not use from the address (in our previous example it will be **1101**001). The other part is called **index**. We also need to know when the *cache*, doesn't have the correct information (this can happen when we turn on the computer), so for this we have just one bit, the **valid bit**, to indicate if the block is valid.

Decimal address of reference	Binary address of reference	Hit or miss in cache	Assigned cache block (where found or placed)
22	10110_{two}	miss (5.6b)	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	miss (5.6c)	$(11010_{two} \bmod 8) = 010_{two}$
22	10110_{two}	hit	$(10110_{two} \bmod 8) = 110_{two}$
26	11010_{two}	hit	$(11010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	miss (5.6d)	$(10000_{two} \bmod 8) = 000_{two}$
3	00011_{two}	miss (5.6e)	$(00011_{two} \bmod 8) = 011_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$
18	10010_{two}	miss (5.6f)	$(10010_{two} \bmod 8) = 010_{two}$
16	10000_{two}	hit	$(10000_{two} \bmod 8) = 000_{two}$

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

Something important in this example is that in the step “c”, we can see that we write in the **010** index of the cache but, in the step “f”, we write again in this index. We can see that in these cases what we must do is **replace** the block stored in that index.

In this example, we were just using memory with words of 6 bits, in the MIPS architecture we have words of 32 bits.

In these cases, to calculate the size of the tag we need this simple formula:

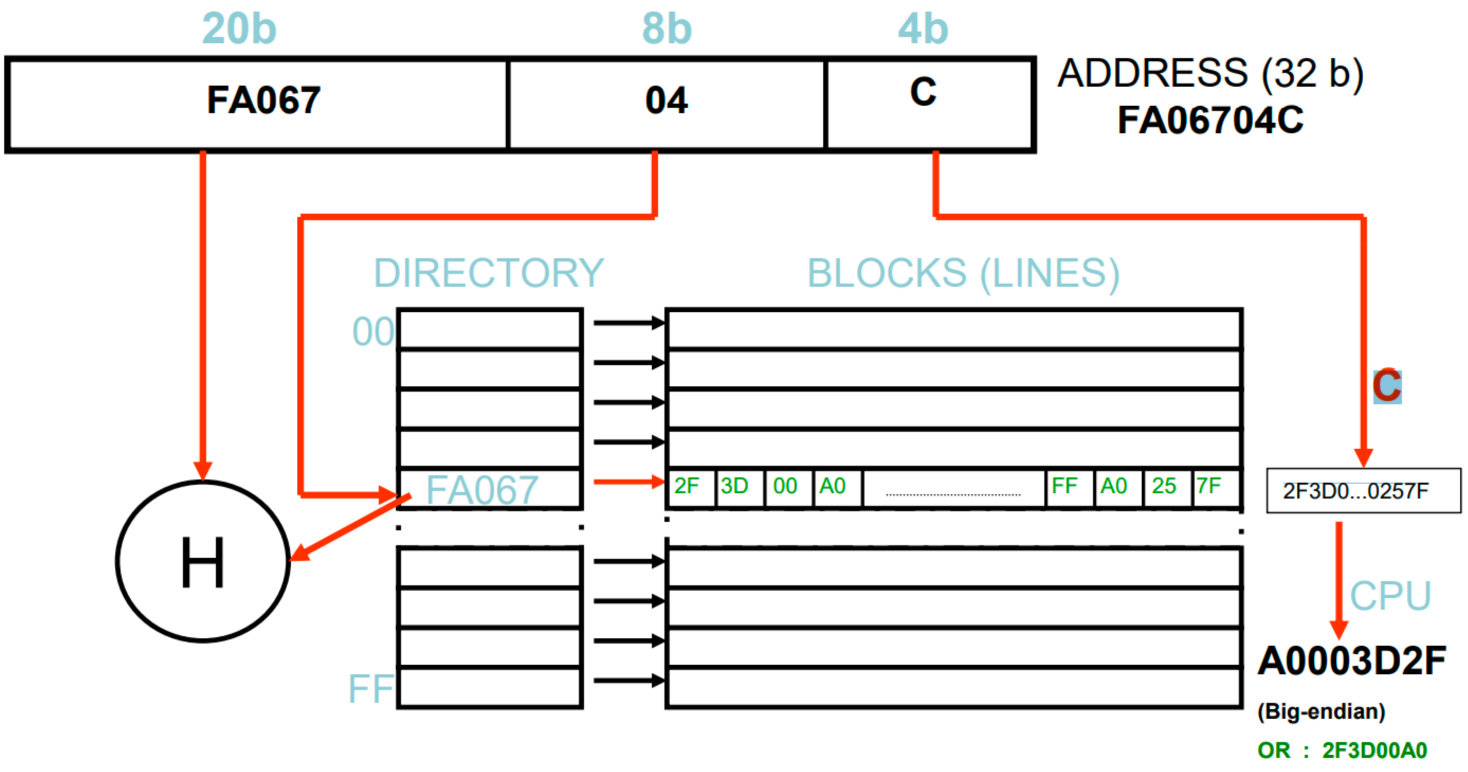
32(or size of each address) – (**n** + **m**)

From the 32 bits of the entire address we subtract, first the n bits used in the index of the cache and the **m** bits used with the offset (offset = $\log_2$ bytes/block). Let us do an example:

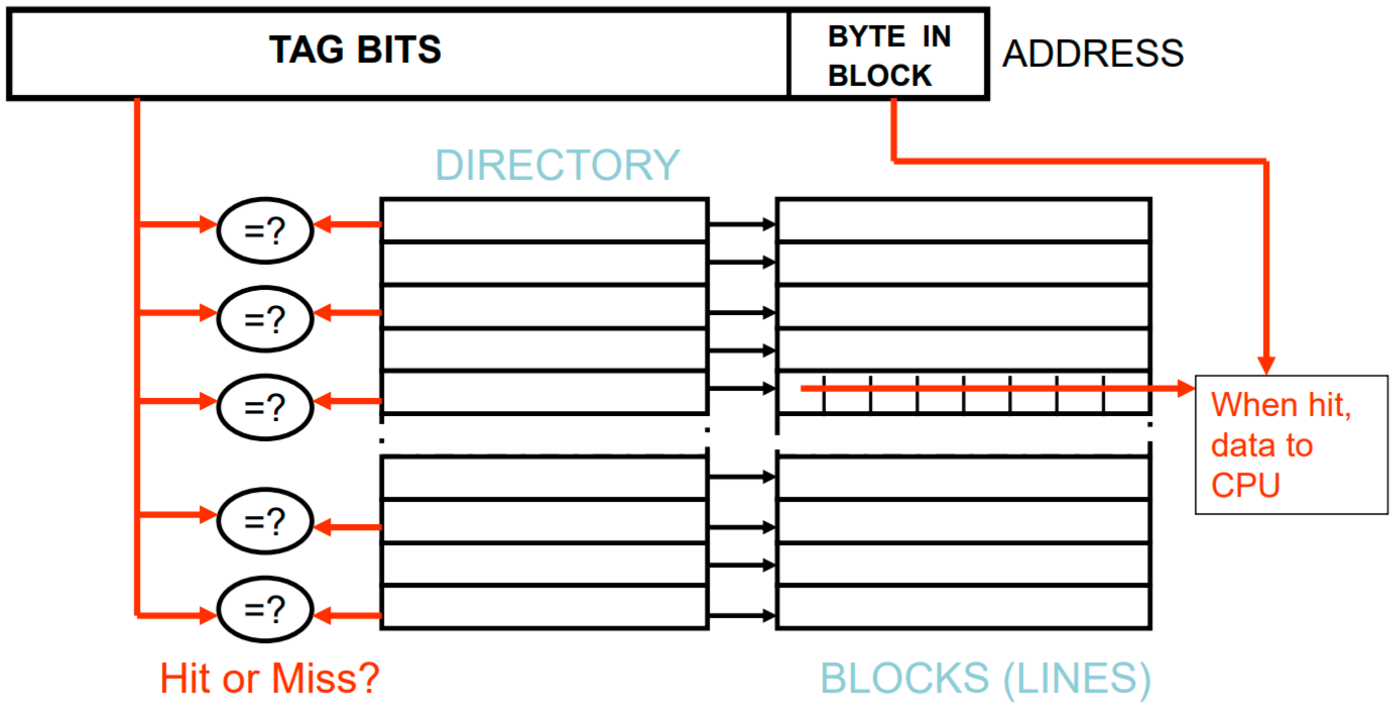
> *Imagine that we have a cache memory of 64 blocks and each block has 16B. We assume that the addresses have 32-bits.*
>
> We need the tag size and to get that we can conclude first that the **index** = 6, why? Because the *cache* has 64 blocks which is 2^6 and $\log_2 2^6 = \mathbf{6}$.
>
> Now we need the offset address. We first need to recognize the **offset**, the offset is the $\log_2$ of the length of each block, so, $16 = 2^4$ and $\log_2 2^4 = \mathbf{4}$.
>
> So now we can compute the tag size, for that we will use our previous formula:
>
> 32 – (6 + 4) = 32 – 10 = **22** bits have the tag.
>
> *To which block index belongs the address 1200?*
>
> In order to know this, we could write the number 1200 in binary and take the last 6 bits, but that will take too much time (and in the exam we don't have much Jaja). So, we can simply do this
>
> 1. First, we divide the address by the size each block. 1200/16 = 75.
> 2. Now we get the module of this number with the number of blocks of the *cache*. 75%64=11
>
> So, the 1200 address is going to be stored in the block 11 of the cache (index = 001011).

One good thing to know is that the bigger the block is, the less miss rate is going to be. But how do processors handle misses? Usually they stall the pipeline until the block is stored. But to improve the performance of the processors we use other methods to store the blocks in the memory.

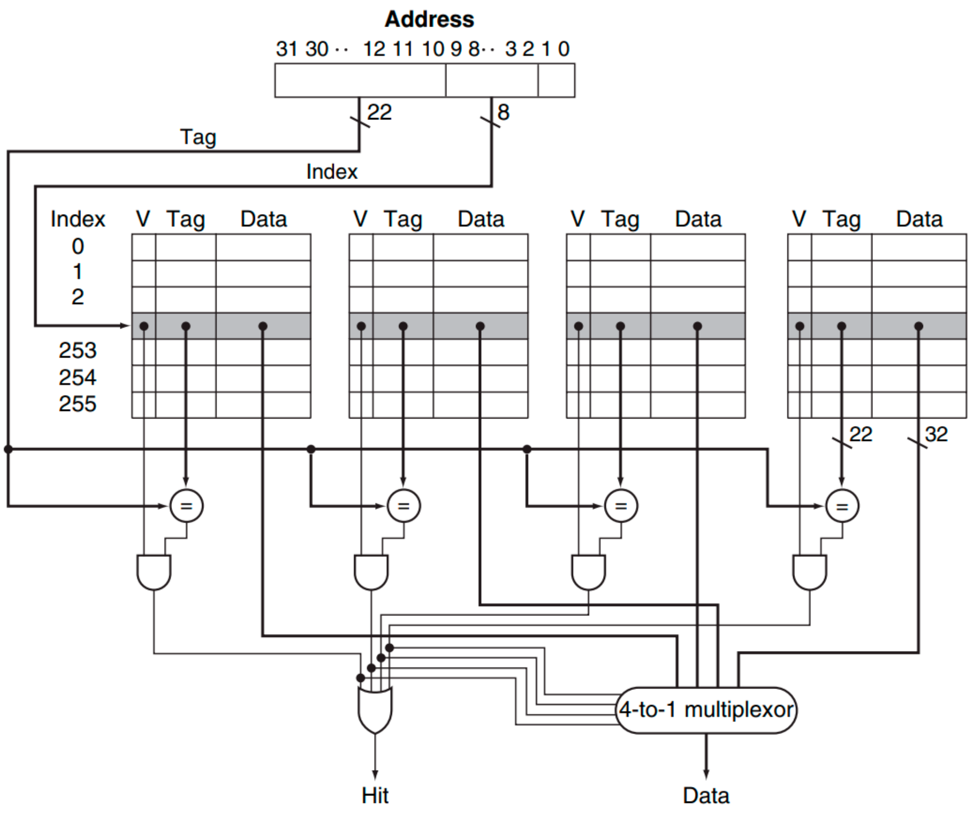
The first technique we already saw it, **direct mapped**.



Another technique is **fully associative**, with this technique any block can be placed anywhere, so this makes the process of searching difficult and requires a lot of hardware because we have to place a comparator with every entry of the cache.



The next technique is **set-associative**, here we have a fixed number of locations where each block can be placed. A set associative cache memory with **n** locations is called *n-way set-associative* cache. A *set-associative* placement combines *direct mapped* and *fully associative*, this is because a block is mapped into a set and then inside this set, we look for the block.



As we can see we have **n** comparators in a **n-way set associative**. We first check the index (as in the *direct mapped*) and then we check every place in that index (as in the *fully associative*).

So, when a miss happens, which block do we have to replace?

- In a *direct mapped* if the index has a block, that block is simply replaced by the new one.
- In a *fully associative* all blocks of the cache are candidates of replacement.
- In a *set-associative* we must choose among the n places in the index, so to choose we use the replacement scheme **last recently used (LRU)**.

LRU consists in replacing the block that has been unused for the longest time, so for example if we have a 2-way set associative we will have 1 bit in each set and that bit will be set whenever that element is referenced.

BLOCK REFERENCED	C_{B0}	C_{B1}	C_{B2}	C_{B3}	STATE	LRU
Initial state	0	0	0	0	Empty blocks	B0,B1,B2,B3
Error cache access	0	1	1	1	B0 full	B1,B2, B3
Error cache access	1	0	2	2	B0,B1 full	B2,B3
Hit in B0	0	1	2	2	B0,B1 full	B2,B3
Error cache access	1	2	0	3	B0,B1,B2 full	B3
Error cache access	2	3	1	0	All blocks full	B1
Hit in B1	3	0	2	1	All blocks full	B0
Error cache access	0	1	3	2	All blocks full	B2
Error cache access	1	2	0	3	All blocks full	B3

As we said before, when a miss happens the CPU stalls the pipeline but, the pipeline is not only stalled when the read occurs, it also happens with the write. So, the time spent while stalling the pipeline due to a miss is:

$$Memory\ stall\ clock\ cycles = (Read\ stall\ cycles + Write\ stall\ cycles)$$

The read stall cycles can be defined as:

$$Read\ stall\ cycles = \frac{Reads}{Program} * Read\ miss\ rate * Read\ miss\ penalty$$

The writes are more complicated because we have also a buffer that generates stalls when it is full:

$$Write\ stall\ cycles = \left(\frac{Writes}{Program} * Write\ miss\ rate * Write\ miss\ penalty\right) + Write\ buffer\ stalls$$

There are few types of writes, the one we just saw is the one that happens when a block is written in the *cache*, **write through**, but what if we have to write in the memory? That is what we call **write back**. This type of writing produces stalls as well:

$$Memory\ stall\ clock\ cycles = \frac{Memory\ accesses}{Program} * Miss\ rate * Miss\ penalty$$

$$= \frac{Instructions}{Program} * \frac{Misses}{Instruction} * Miss\ penalty$$

Another way to fight the miss rate is using the **multilevel caches**, this means that the processor uses more than 1 caches. The other levels are for when a miss happens in the first level cache, if that happens we will access the second level, and the penalty time for that will be just the access time of the second cache. With this architecture the primary cache is often smaller, and it may use smaller block size.