1. Cache memory

- Introduction: memory hierarchy
- Main characteristics of cache memory.
- Main design parameters: line, mapping, write strategy and replacement policy

Problem

The processor (CPU) is faster than the memory (difference increases). The processor needs to get data and instructions from memory as fast as possible.

Att.: **the slowest element affects the speed of the whole system!**

Generally, the access time increases with the size of the memory: **small memories** are usually **faster** than big memories. But we need big memories**!**

So, how can we structure the memory systems so that rd/wr operations are as fast as possible?

Types of RAM memories

Static RAM

Fast, but it needs many transistors (4-5) for each memory-bit.

Dynamic RAM

More compact (1 transistor + 1 capacitor per memory bit) but slower than static RAMs.

If the critical parameter is speed: Static RAM. If it is capacity: Dynamic RAM.

Associative memories

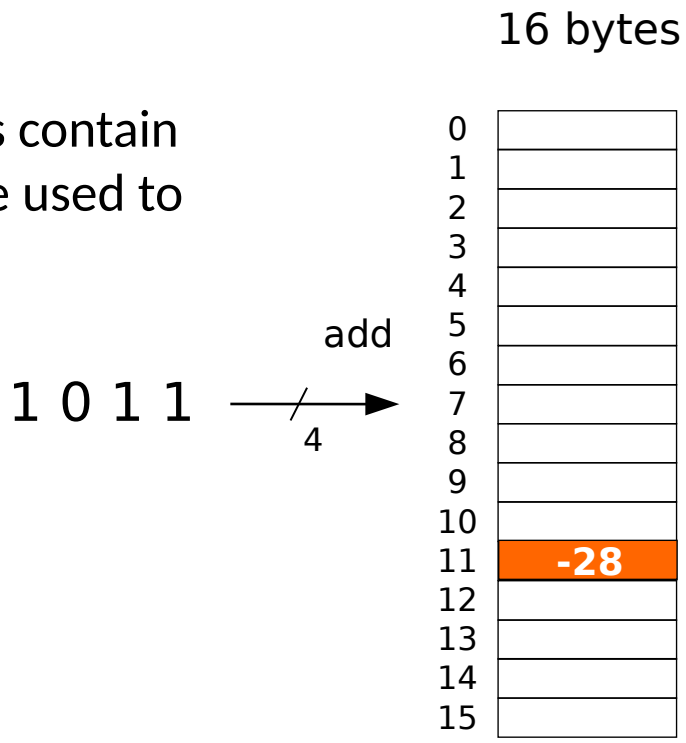
Access is not done using addresses but using content

Given a word the result can be: **yes** it is in memory; on **not** it is not. In addition to the information **associated** to that word: for instance **where it is**. They are more complex than standard RAM memories and they have an special use in CM.

Main memory: structure

Memory cells accessed by their **address** for reading and writing.

The addresses of memories of **P** cells contain $\log_2 P$ bits; addresses of **n** bits can be used to access 2^n words .



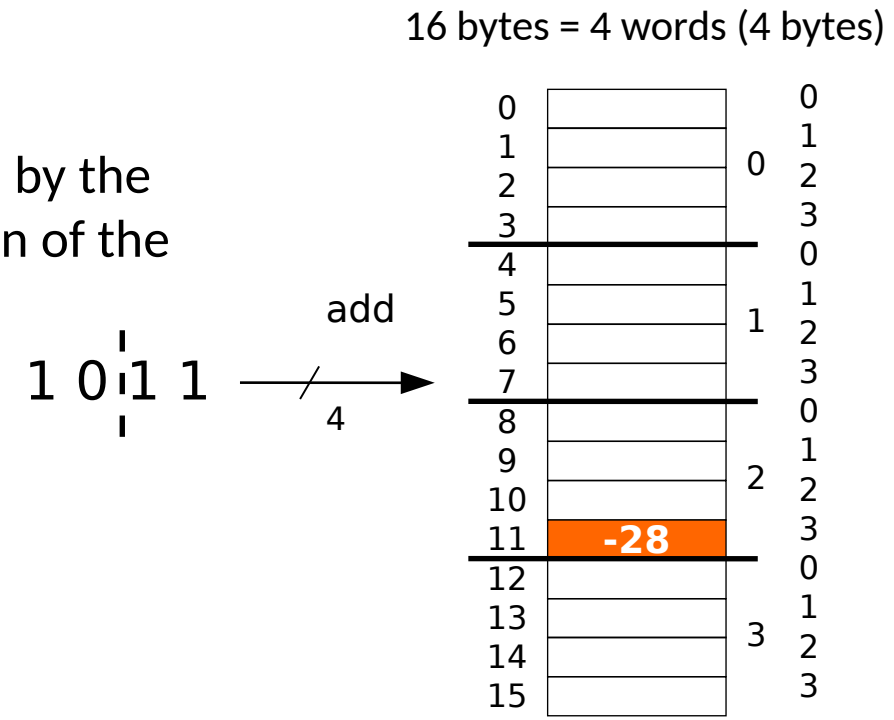
Connections with processors:

- Addresses → address-bus
- Data → data-bus
- Operation (rd/wr) → Control-bus

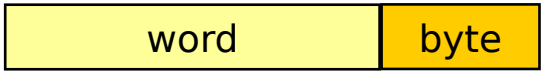
Main memory: structure

Which is the **information-unit** accessed by the processor?
byte (usual) byte addressable
word (4 - 8 bytes)

In general, the addresses generated by the processor indicate the initial position of the word.



Structure of addresses:



- word = add **div** size_word (integer division: 11 / 4 = 2)
- byte = add **mod** size_word (11 mod 4 = 3 rest of the division)

Notes about memory addresses

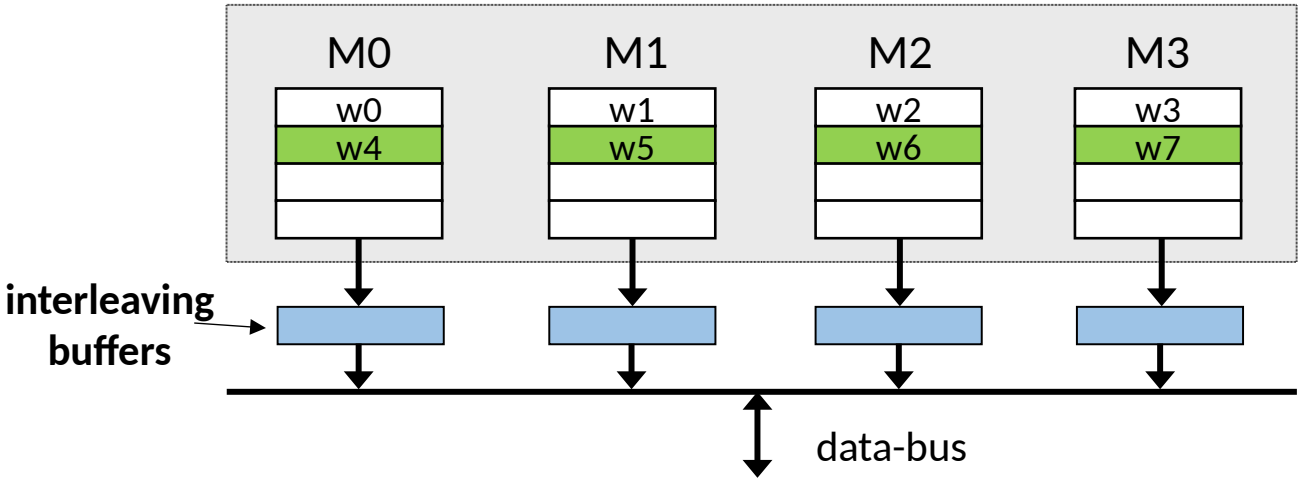
When the program/data are loaded in MM the position where data is located must be decided according to the free space in that moment. Consequently those positions are not fixed. Every time data is accessed, the physical addresses of the programs or data pages will be stored in a table.

That is why the processor uses **logical addresses** instead of concrete physical addresses. Logical addresses need to be **translated** to obtain **physical** (memory) **addresses**. The translation is done using **TLB** (*translation look-ahead buffer*) .

To make it simpler we will work with physical addresses in this subject.

Main memory: structure

The main memory is **interleaved.** Multiple **memory banks** take turns to supply data. Some contiguous words can be accessed **simultaneously** at the **same time**



A ***4-way interleaved*** memory.

The ***N-way*** defines the size of the **data block**.

Main memory: structure

Why is memory interleaved?

Because the data and instructions in memory are **not accessed randomly**!

"90% of the accesses are done in 10% of the code"

Locality:

- **temporal** locality: if a word is used, it will probably be used soon.

t → @ t+Δt → @ (loops, ...)

- **spatial** locality: if a word is used, the next one will probably be a contiguous or nearby word

t → @ t+1 → @+Δ

(sequential execution, accessing vectors, ...)

CM: main characteristics

Taking into account the locality, why not build a **memory hierarchy**?

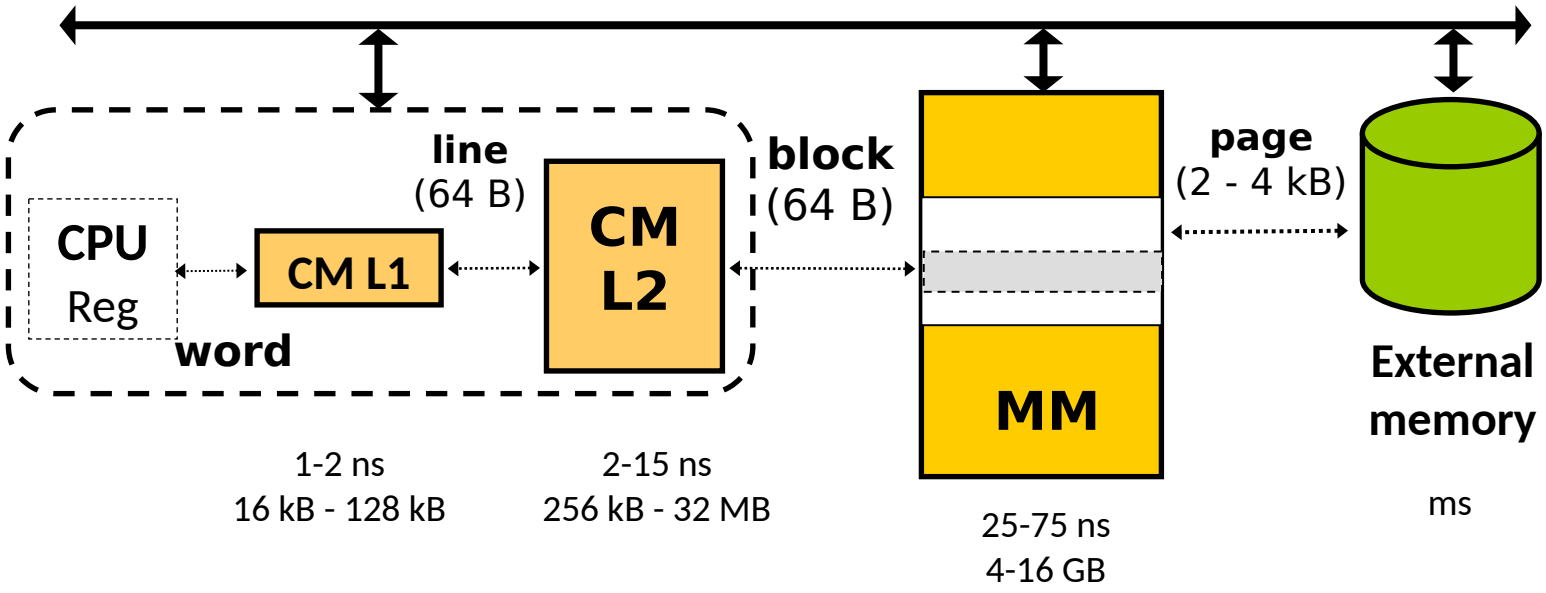
> Higher level: **cache memory** (CM)

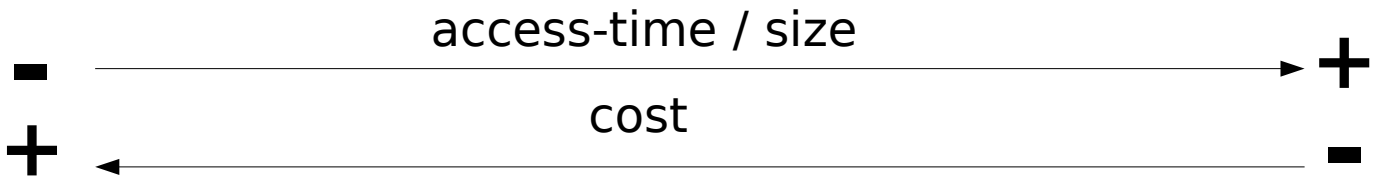
- Small memory / fast access (near). Usually static RAM
- Data and instructions that are (or will be) more used by the processor
- If possible, several levels (L1 (smallest), L2, L3...)
- High number of transistors. Usually L1 (and also L2) On-chip cache faster

> Lower level: **main memory** (MM)

- larger memory (1000 times) / slower (5 – 10 times)
- dynamic RAM

Memory-hierarchy





Inclusion principle

- An upper level is generally a subset of the data contained in the next lower level, So copies of a data block can be in:

L1	1	-	-	-
L2	1	1	-	-
MM	1	1	1	-

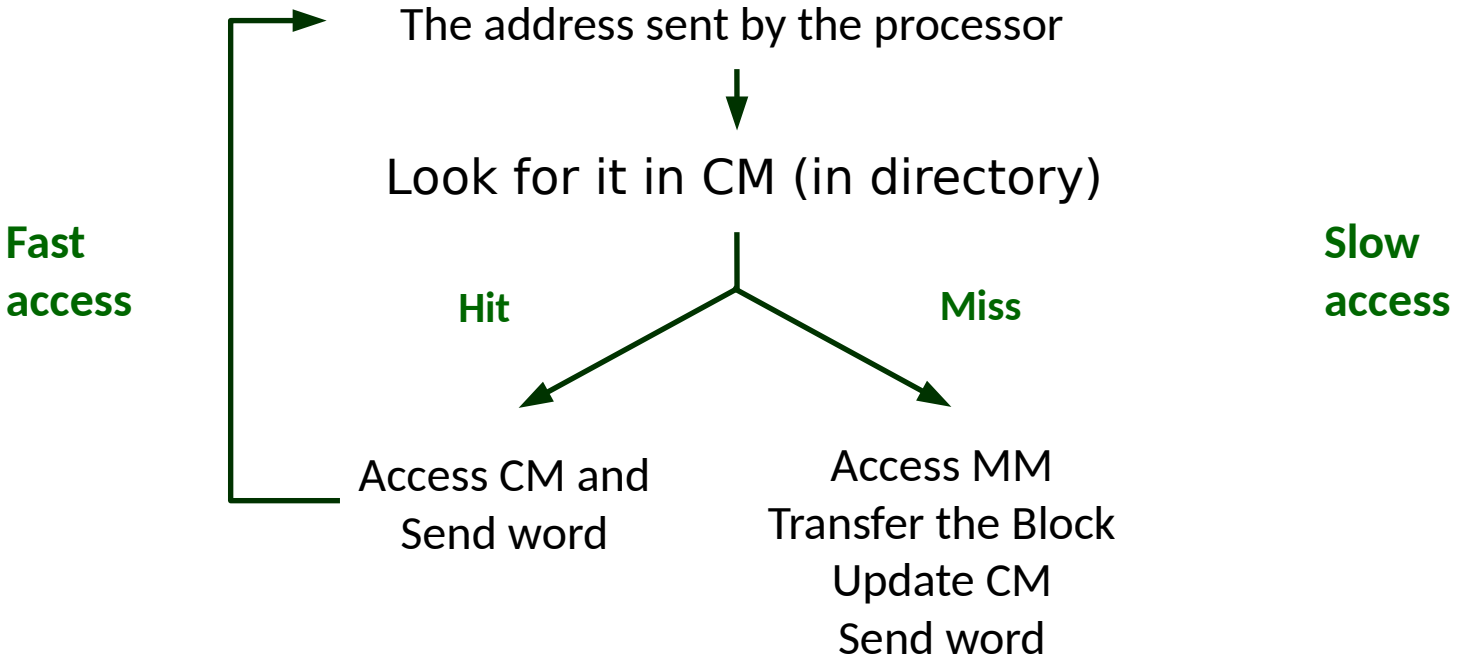
In some of the processors the condition is not fulfilled and L2 is not "inclusive": a block can be in cache L1 without being in L2.

- The processor will search for data and instructions in the "closest" memory. If they are not there, it will go to the next level and so on. So, there will be **different access times** depending on the location of the data.

- Accesses will be fast if the searched data/ instructions are in CM. But not all of them fit in the cache memory! → we will have to **BET**.

CM: main characteristics

- When betting, it is possible to **hit** (the required data/instruction is in the cache) or **miss** (it is not there).
- If the processor wants to read a word



CM: main characteristics

- Memory access-time:

$$T_a = h \times T_{CM} + (1 - h) \times T_{miss}$$

h hit rate (1-h, miss rate)

T_{CM} access-time to cache

T_{miss} time required to find data/instructions in the next level of the hierarchy (it will depend on the hit rate of each level!)

- To be efficient, the hit rate **h** needs to be very **big** (> 95%).

CM: main characteristics

- **Transfer unit** between cache memory and main memory: **line /block**.

  1 line: 2^n contiguos words from memory (ex. 64 bytes: 8 8-byte words, or 4 16-byte words).

- Why a block/line and not a word?

  To take advantage of **locality**:

  > If a word needs to be transferred to CM to be used, the contiguous word will probably be used in the next cycle.
  > Take advantage of the MM-CM transference to transfer more than a word with small cost increase.

CM: main characteristics

Searching for data: the address of a word indicates its position in the MM but which is its position in CM?

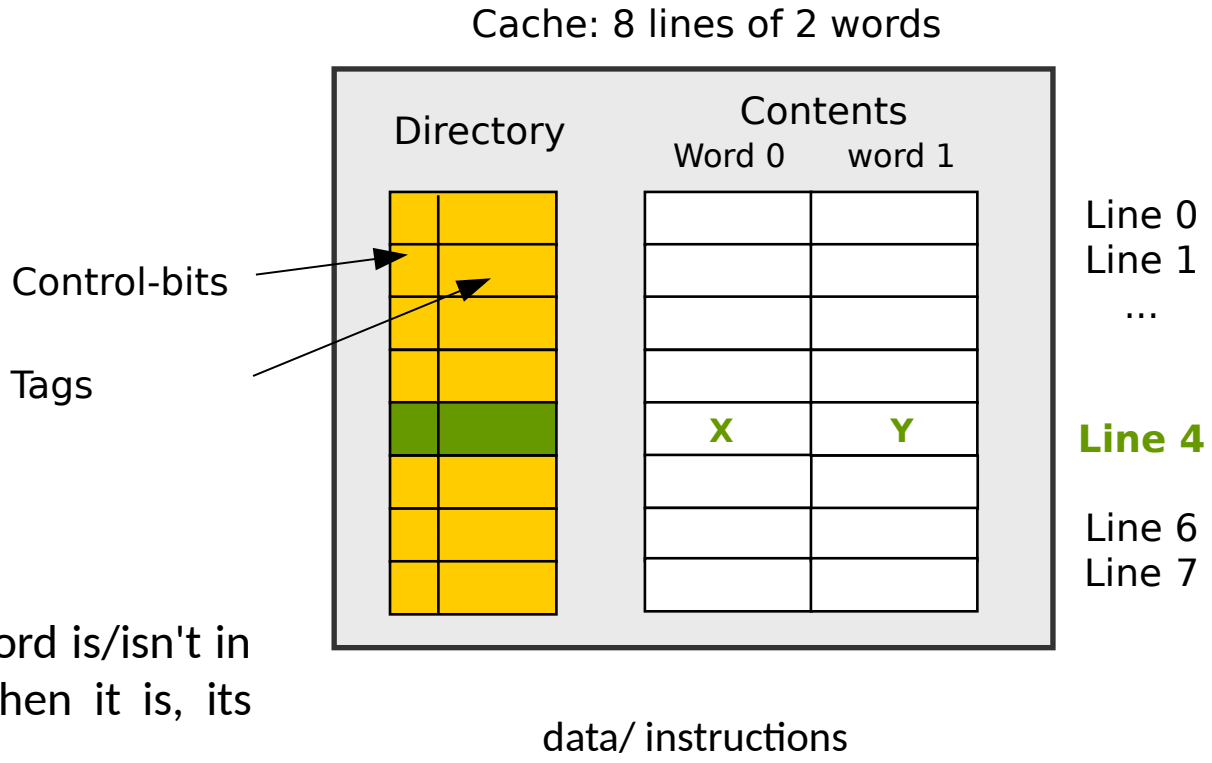
CM components: **directory + content (Data)**

- **Content** SRAM memory to store data blocks.
- **Directory** information about the data blocks stored in the CM: **a word per data-block:** address tags and state of the data.

The directory can be an associative M. Content Addressable M (CAM) / SRAM

In addition: comparers, multiplexers, encoders,...

Cache memory **structure**:



Directory:
the needed word is/isn't in cache. And when it is, its state.

CM: design parameters

- The most important design parameters of a cache memory:

1. Size.
2. Content.
3. Line (Block).
4. Correspondence.
5. Replacement policy.
6. Operations. Write strategy.
7. Search algorithm

Objective: big hit rate, small access time (Tcm)

1. Size

Although size is increasing, cache memories in processors are much smaller than the main memory. For example, 8 MB // 64 GB.

As a consequence, **not all the blocks in the main memory fit in the cache** → many times they won't be in CM !

The bigger the cache is, the greater the hit rate (**h**) will be. It will always depend on the program.

Attention! The cost will also be bigger!

>> a **balance** between cost and efficiency is needed.

1. Size

Current processors:

- 2 or 3 level caches

  L1 cache: 32 - 128 kB
  L2 cache: 256 - 1024 kB
  L3 cache: 4 - 120 MB

- *On chip*

  Caches L1 and L2 are in the same chip the processor is (faster); L3, inside or outside (slower), depending on the size.

- Private / shared.

  L1 and L2 are usually private for each core of the processor; L3, shared for every core.

2. Content

The processor needs to process data and instructions. Access **patterns** are **different** for **data** and **instructions**. Which is the best way to store them?

> **Unified cache**, data and instructions in the same cache.
> Usual strategy for big caches (size not a problem)

> **Distributed cache**, data and instructions in different caches (data cache (DC) and instruction cache (IC)). Each cache can be optimized indenpendently and both can be used at the same time.

Generally L1 caches are distributed but L2 and L3 caches are unified.

3. Line

The amount of information transferred between the main memory and the cache memory.
What size do the blocks/lines have (number of bytes)?

On the one hand, the blocks/lines should be **big**. Spatial locality increases (data that will be used later is stored in cache and **h increases**).

But, on the other hand, they should **not** be **too big**:

- fewer blocks/lines in the cache
- more time required to transfer a block/line
- **pollution**: bring to cache and then, not use it (att: bet)

3. Line

Given a memory address, which is the block to be accessed? As words are organised in blocks it is enough to divide the address of the word with the number of words. So,

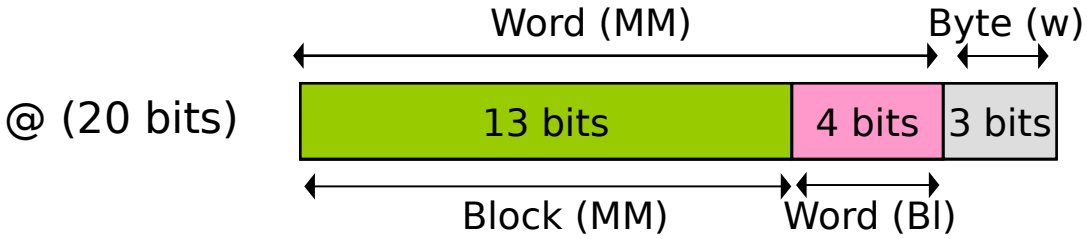
Word = address / word_size

Block = Word / numb_words_block

Word in block = Word **mod** numb_words_block

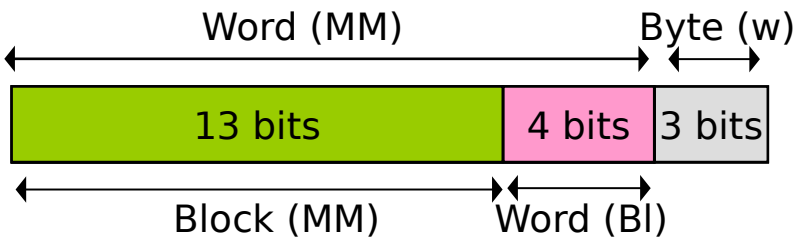
For example:

MM: 1 MB → address: 20 bits
Words: 8 bytes (3 bits to access a concrete byte)-→ 128k words
Blocks: 16 words (4 bits to access a concrete word)-→ 8k blocks



3. Line (block)

@ (20 bits)



For Example. MM: 1 MB; Blocks: 16 words; Words: 8 bytes

Memory address: **12539** **0000 0011 0000 1111 1011**

Word (MM): add **div** **8** = **1567**
Byte(w): add **mod** **8** = 3

0000 0011 0000 1111 1011
1567

Block (MM):word **div** **16** = add **div** 128 = **97**
Word/bl: word **mod** **16** = add **mod** 128 = 15

0000 0011 0000 1111 1011
97 15 3

(the bits used to name the block are usually called *offset*: block - *offset*)

3. Line

Time to transfer a data block to cache memory

As MM is interleaved, all the words of a block are read at the same time. The following times are differentiated:

T_{MM} to transfer a word: read memory and transfer through the bus.

T_{buff} to transfer a word from the interleaving buffer (already read).

So, time to transfer the complete data block: first word (T_{MM}) plus the rest of the words of the block (T_{buff})

$$T_{bt} = T_{MM} + (num\_words - 1) \times T_{buff}$$

4. Correspondence (block placement)

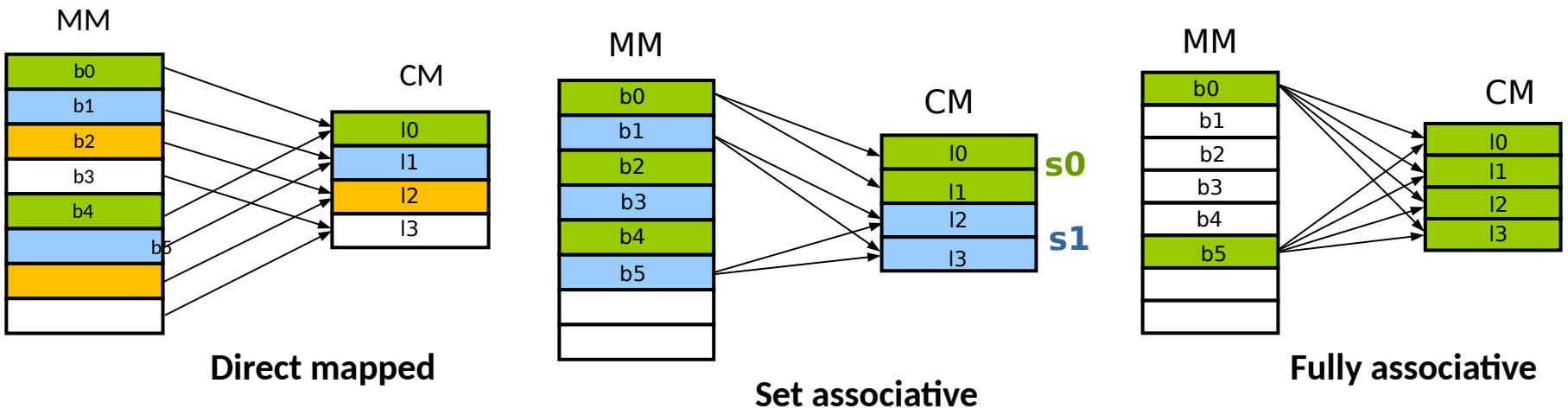
It is not possible to fit every MM block in CM. **Where** to place a new block? Then, **how** will we **search** for that block? Which is its address in CM?
Three options:

- **Fully associative:** blocks can be allocated to any possition.

- **Direct mapped:** each block in a concrete cache position (always the same).

- **Set associative:** The cache is divided in sets each containing some blocks. Each block from MM will be placed in a concrete set, but in any position within the set.

4. Correspondence (block placement)



The **content** of the cache, **N lines**, can be organised in different ways depending on the number and sizes of the sets:

N×1 --- N/2×2 N/4×4 N/8×8… --- 1×N

direct set associative fully associative

4. Correspondence: *tags*

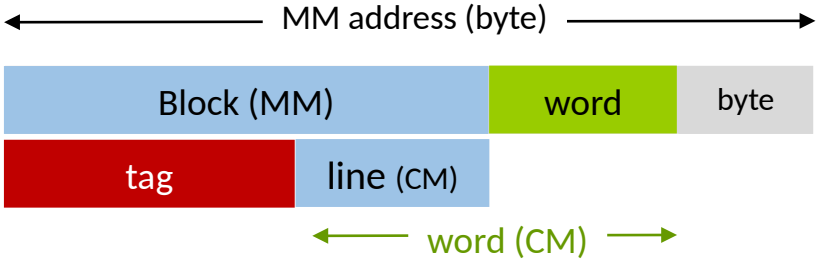
The cache position does not identify a concrete data block. Different blocks can be located in the same position.

When the data block is loaded in a concrete position, **information** about the **block** must be stored in the **directory**. The processor needs to analyse the directory in every memory operation to discover if the desired word is in CM or not.

The information is called ***tag***. It is part of the block address and changes depending on the correspondence option.

a Direct

The block is located in a fixed cache position. Always the same. The last bits of the block address indicate the position in the cache, the rest must be stored in the directory to identify the block stored in that position.

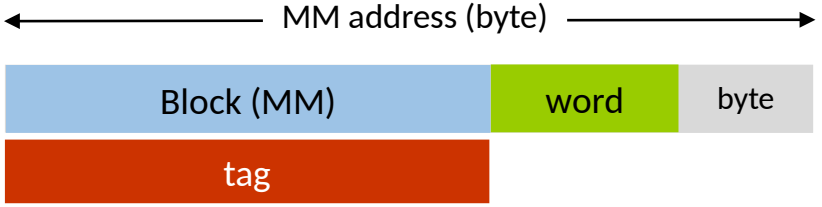


line (CM) = Block (MM) **mod** num_lines_cache

tag (dirctry.) = Block (MM) / num_lines_cache

b Fully associative

The block can be located anywhere; so, the full address of the block must be stored in the directory to know which block is in that position.

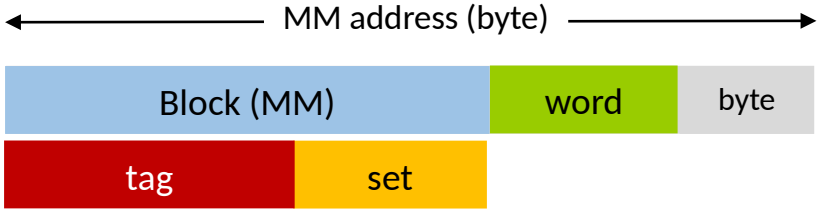


line (CM) = any (associative!)

tag (dirctry.) = complete address of the block

c Set associative

The block goes to a fixed set in cache, always the same (direct mapping), and within the set to any position (associative).
So, the last bits of the block address indicate the set; the rest, must be stored in the directory.

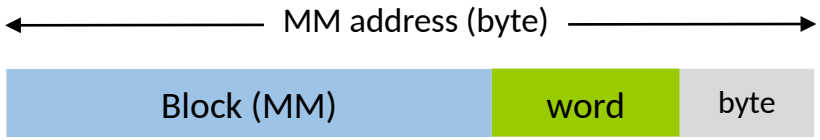


set (CM) = Block(MM) **mod** num_sets_cache

line (CM) = any within the set

tag (dirctry.) = Block(MM) **/** num_sets_cache

Remember: smallest weight bits → rest of the division
biggest weight bits → division



Example

MM: 64 kB -- 4 byte words (16 k words) -- 8 word blocks (2 k blocks)

Address, 16 bits = **25396** 0110 0011 0011 0100
Word, 14 bits = 25396 / 4 = 6349 0110 0011 0011 01
Block, 11 bits = 6349 / 8 = **793** 0110 0011 001
Word in block, 3 bits = 6349 mod 8 = 5 1 01

CM: 1 kB → 256 word → **32 lines × 8 words**

a. direct

line (CM), 5 bits = **793** mod 32 = **25** 11001
tag, 6 bits = **793** / 32 = **24** 0110 00
cache word, 8 bits = 11001 101

b. fully associative

tag, 11 bits = **793** 0110 0011 001
Cache word, 8 bits (5 dirctry. + 3) = xxxxx 101

c. set associative, 8 sets × 4 lines

set, 3 bits = **793** mod 8 = **1** 001
tag, 8 bits = **793** / 8 = **99** 0110 0011
cache word, 8 bits (5 dirctry. + 3) = xxxxx 101

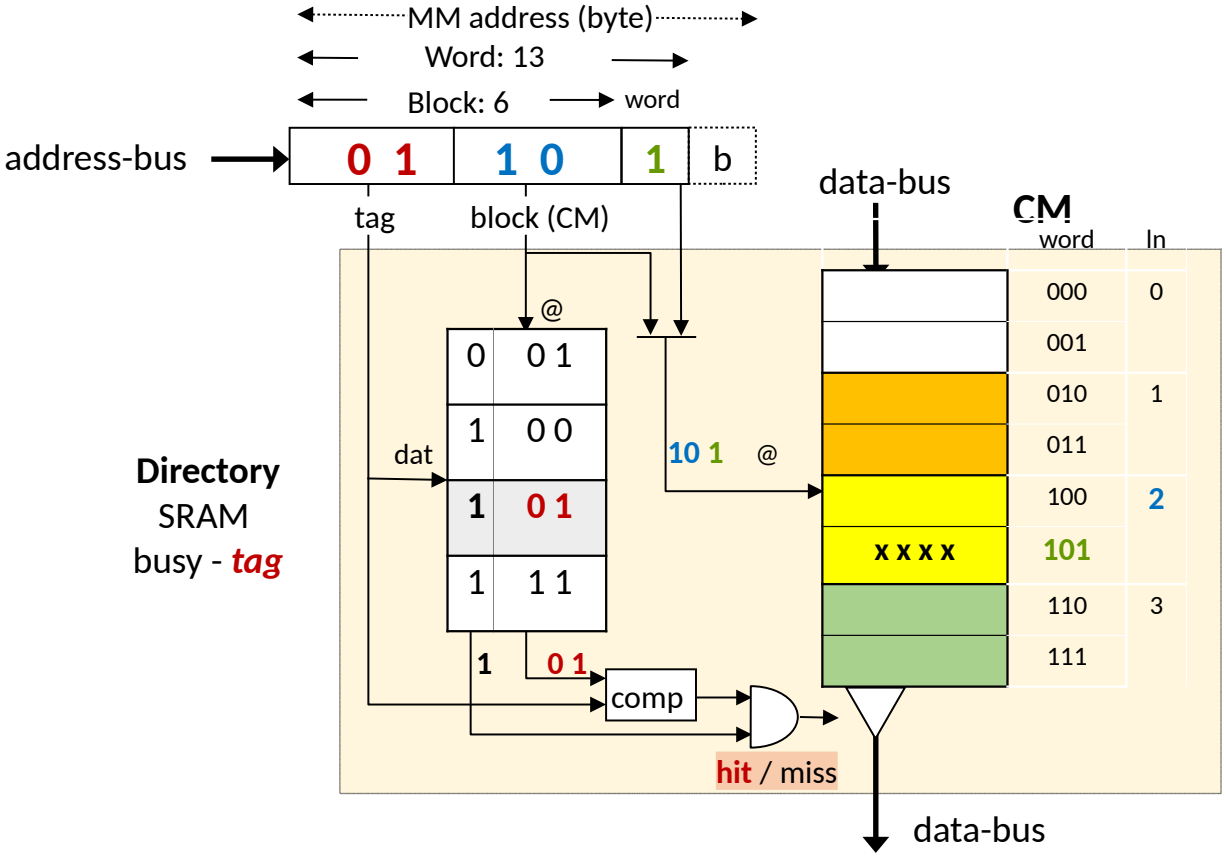
4. **Correspondence:** cache structure (directory + data)

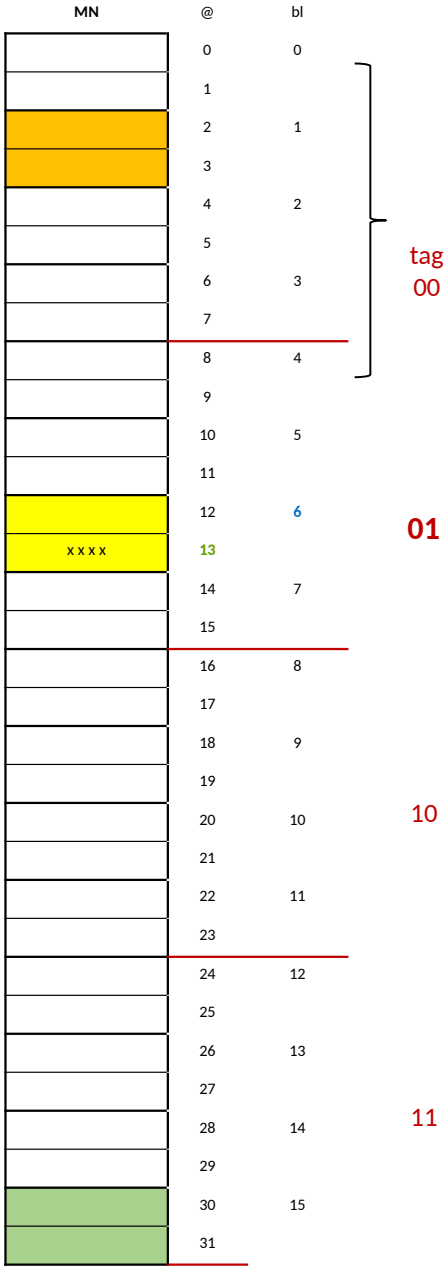
The cache memory has two parts: the **directory** and **data**. A SRAM memory is used for data; however, the design of the directory depends on the correspondence.

In the case of the direct mapped cache it is enough a SRAM memory, because the position of the block is known, but in the case of fully associative caches an associative memory is needed because the block can be in any position. For intermediate cases, different solutions exist depending on the number of sets.

We present a simplified schema of how the cache can be organised in each case.

a. **Direct** (ex., 2 word and 4 line cache)





Search: read the content of the directory in the position corresponding to the block, and compare with the *tag* of the block stored in cache (if the position is busy) and the one we are looking for.

a. Direct

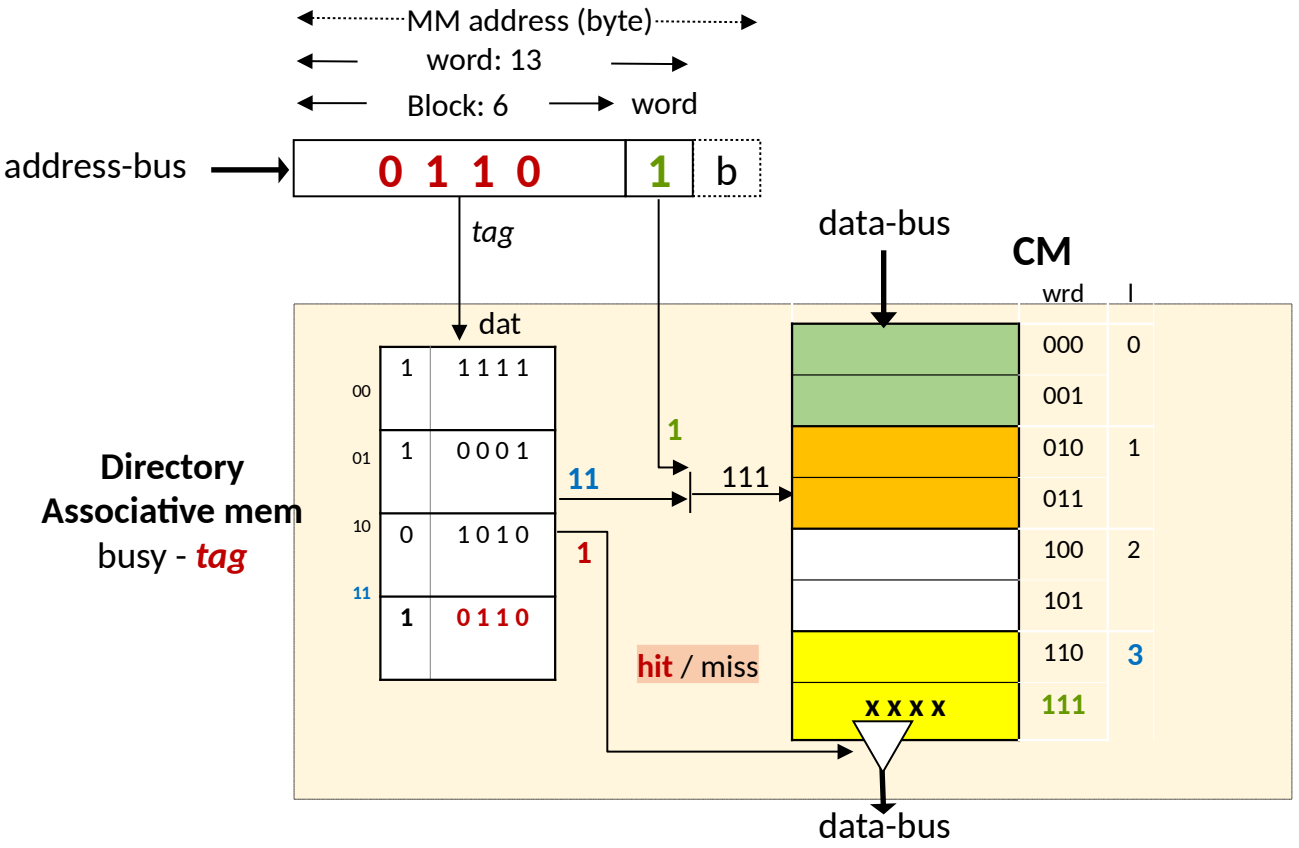
- The directory and the data memory are SRAM and can be accessed **simultaneously**, because where should the block be is known. The structure is **cheap** and **fast**.

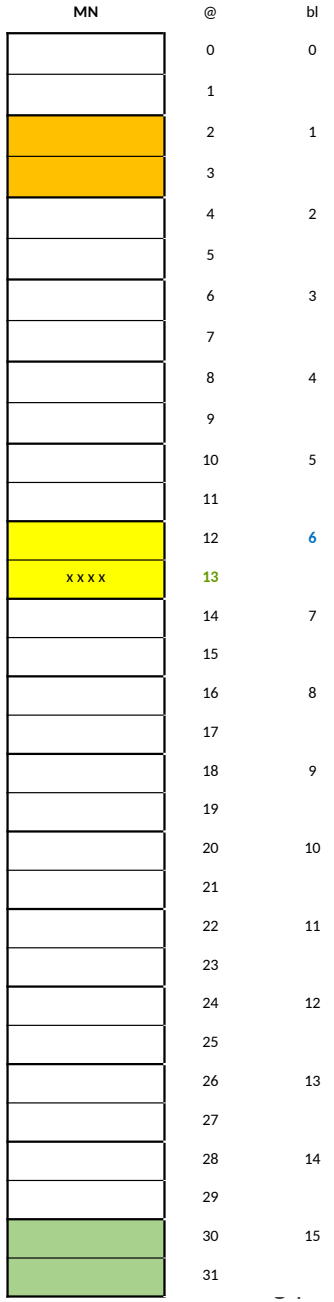
  Although the access can be simultaneous, if a write operation (a modification) is going to be performed it will only be possible after having the response of the directory (hit/miss).

- But **the memory space** (always scarce) **is not administered in the best way**. Each block must be stored in a fixed position; if it is busy, the block will need to be replaced (even if there is space in other lines).

  Consequently, the hit rate, **h**, will not be as big as expected.

b. Fully associative





Search: search for the content in the directory (associative memory): the response is **yes** or **no**, and when positive, in which **position** it is.

b. Fully associative

- The directory is an **associative memory**, so that searches can be done according to content. Associative memories are more complex than SRAM (all position's contents need to be compared with the searched word), and, consequently, **more expensive**. But the main problem is the following.

- **Two accesses must be done:** one in the directory to know where is the block placed, and then, data must be read/written: **it is not the fastest solution**.

- The **memory space** used to store data/instructions is **very well administered**: until the cache is full, there is always space for a new block.

  → hit rate, **h, big**

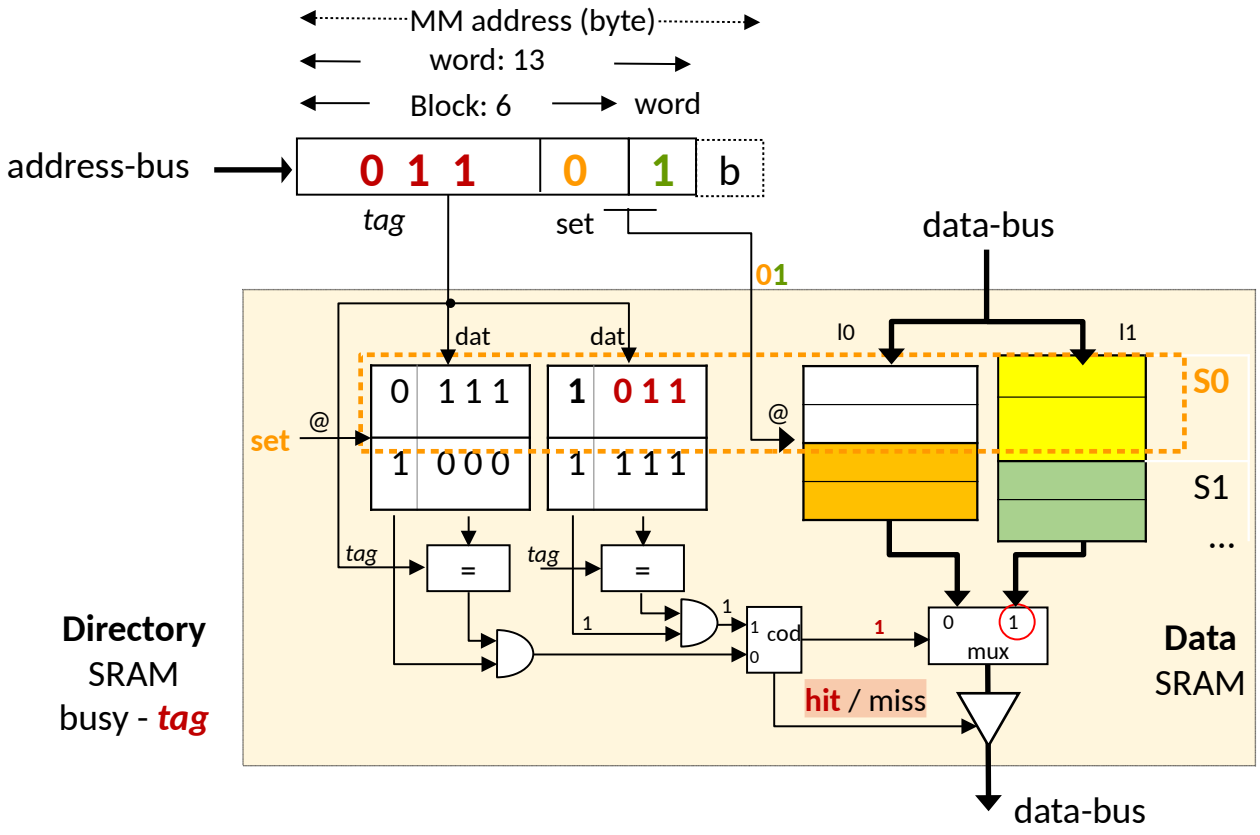
c. Set associative

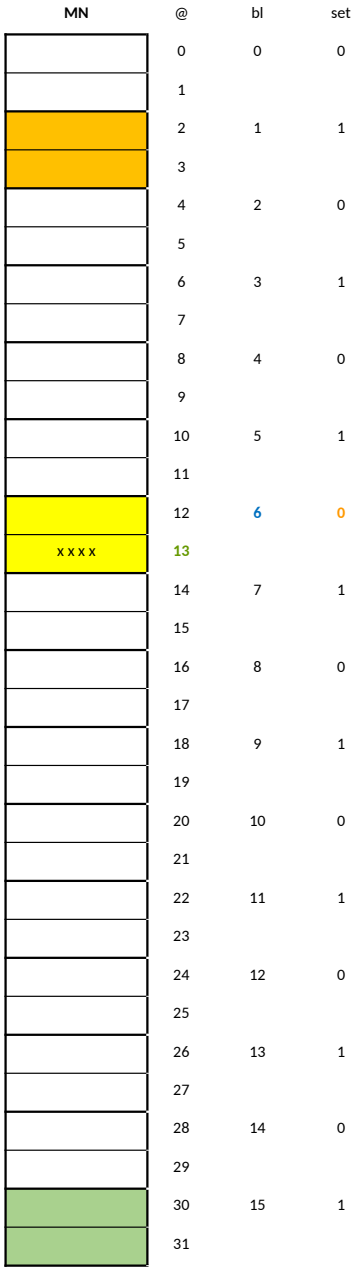
This is the general case: in fully associative there is a single set with N lines, and in the direct mapped, N sets with a single line. So, the structure of the cache can be a mix of the two previous cases.

Normally, **the size of the sets** (number of options) **is small** (2-16), and the **number of sets big**. For instance, 1024 sets of 4 lines. When this is the case, the main problem of the associativity can be avoided: the need of two sequential accesses because the position of the block is unknown.

With this aim, more than a memory module are used for data, and the lines within the sets are interleaved in the modules. This makes possible to read a word in all lines at the same time, and then, select one of them at the end (as in the previous cases, to write a word we need to wait the answer of the directory).

c. **Set associative** (2 sets with 2 lines)





Search: read simultaneously a given word in every line of a set (and the directory); depending on the response of the directory, the required word can be selected.

c. Set associative

- The cache memory is divided in sets where any free position can be used. The size of the sets is called *way*. Ex.: 8 *way associative* --> sets of 8 lines.

- **Compromise** between direct and fully associative: as fast as direct mapped and the higher hit rate offered by associativity:

  - access time, similar to direct mapped
  - hit rate (h), similar to fully associative

- What is used today?

  - set associative (4 - 16 line sets)

  for instance, i7 L1: 8 *way*; L2: 4 *way*; L3: 16 *way*

4. **Correspondence** - summary

Different MM blocks can be placed in a concrete position of CM. So, in addition to data more information needs to be stored in the cache, in the **directory**, so that blocks can be identified: the ***tag***. Before any memory operation is performed, the *tag-s* in the directory must be analysed to find out if the block is in cache or not.

Depending on the strategy used to administer the space, the design and the information to store in the directory vary. In the direct mapped the directory is a SRAM; on the contrary, in the fully associative an associative memory. Direct mapped caches are fast but the hit rate is not high; fully associative memories have bigger hit rates, but are slower.

Set associative memories are a compromise: the hit rates are similar to fully associative and in order to achieve the speed of direct mapped options the number of lines in the set (*way*) are not high; it is enough with 4 - 16 lines to achieve good results.

4. Correspondence (summary)

In a concrete position of the cache, different blocks from MM can be placed. This means that **more information is required in the CM's directory**, so that blocks can be identified. Before executing a rd/wr, the directory needs to be explored to decide if the block is in the cache or not. **Tags** depend on the structure of the cache:

- **Fully associative**: (it can be in any position) the whole address of the block.
- **Direct**: (cyclically distributed), “address of the block div number of lines”.
- **Set associative**: (sets cyclically distributed), “address of the block div number of sets”.

5. Replacement

After a cache miss, a data block needs to be placed in cache, but the corresponding position is busy. Before loading the block, another one needs to be deleted. Which line will be deleted?

- In the case of direct correspondence there is no doubt since each block can only go in a position.
- When any kind of associativity is used, a block needs to be selected within the whole cache or within a set.

5. Replacement

Two types of algorithms:

> They do not take into account the behaviour of the program.

- **Random**: any
  easy / not optimized (h)

- **FIFO**: the “oldest” line in cache.
  It is enough to use a counter per line in the directory.

(n = log num_lines). The value of the counters is updated as blocks are charged. When a block needs to be replaced, the one with the biggest value (for instance) is selected, and all the values are updated. For instance with 4 lines:

b2 - 00	b2 - 01	b2 - 10	**b2 - 11**	**b6 - 00**
	b4 - 00	b4 - 01	b4 - 10	**b4 - 11**
		b1 - 00	b1 - 01	b1 - **10**
			b7 - 00	b7 - **01**

5. Replacement

> They take into account the behaviour of the program.

- **LRU**: the Least Recently Used line in cache. Difficult to implement (if A != 2), but greater hit rates .

  This case also uses counters but every access requires updating, not only misses. However it obtains the greatest hit rates because very used blocks are not replaced.

Current processors: LRU variants or *pseudo*-LRU

Information for replacements (counters...) must be stored in directories; so, this will be the content of the directory: busy -- **repl** -- *tag*

6. Operations

Two usual operations in cache: **reading a word** (load) and **writing a word** (store). In both cases, the word has to be in the cache so that the operation is performed. So the first step will be to explore the directory.

Read (rd)

- **rd/hit**: the word is in cache and it is directly obtained.

- **rd/miss:** the word is not in cache. The block containing the word has to be transferred from Main Memory.

In general the exploration of the directory and the reading can start at the same time.

6. Operations: write strategy

Write (wr).
Writes are more complex because they modify data. Since we use cache, we work with **copies** (CM, MM). How are these two copies administered?

Two write strategies:

- ***write-through***: when a word is written in cache, the MM is also written. Both are always consistent.

- ***write-back***: The word is written only in cache; the MM is not updated until the block is replaced in cache (finishing main program, context exchange)

6. Write strategy: *write-through* (WT)

- The copies are **always coherent** , but **write latency is bigger** because the MM has to be written in every case.
  Write buffers are used so that the writing is done when the bus is free.

  Two options:
  - **Updating CM** (the write operation is performed in MM and CM)
  - **Not updating CM** (only MM is updated. If the block was in CM it is deleted)

6. Write strategy: *write-back* (WB)

- **Copies are not updated**. In some cases, the truth value is not in the MM but in the Cache memory.

  Consequence: a line can not be directly deleted from cache; if it has been modified, the MM needs to be updated.

Need of a **control-bit** to know whether the line is modified or not: ***dirty***.

directory >> busy -- **dirty** -- repl -- *tag*

ATT: when a word is modified, **the whole line is marked as modified** and written in the MM when required.

6. Write strategy: *write-back* (WB)

The block to be stored in memory is stored in a write-buffer; the memory controller will be in charge of writing it in memory when the data bus is idle (see optimisations).

- This strategy is usually more adequate.
  Anyway, we are still **betting:**
  - if write operations are repeated one after the other before replacing the line, we will save time and bus traffic. The MM will only be written once, when the line disappears (or at the end).
  - If after a single write operation the line needs to be replaced, the whole line (every word instead of a single one) will need to be written, because *dirty* is related to the whole line.
- It is usual to use different write strategies in different level caches. L1 and L2 (L3).

6. Write strategy

- **ATT** to copies: it is not difficult to know where is the copy that must be used … if all the operations are controlled by a single control unit.

- That's common in processors with a single core unless in a single case: **input/output** operations, where DMA controller loads external data… but, where? In MM? And if the copies of these variables are in CM?

  2 options to face the problem:

  - ***flush*** (deleting the information of CM) copies of input/output data-blocks in CM are invalidated (it is enough to update busy=0 in the directory), new data are uploaded in MM.

  - not to transfer to CM I/O data blocks: all accesses (rd/wr) are done in MM.

7. Search algorithm

Due to spatial locality, access time can be reduced prefetching data.

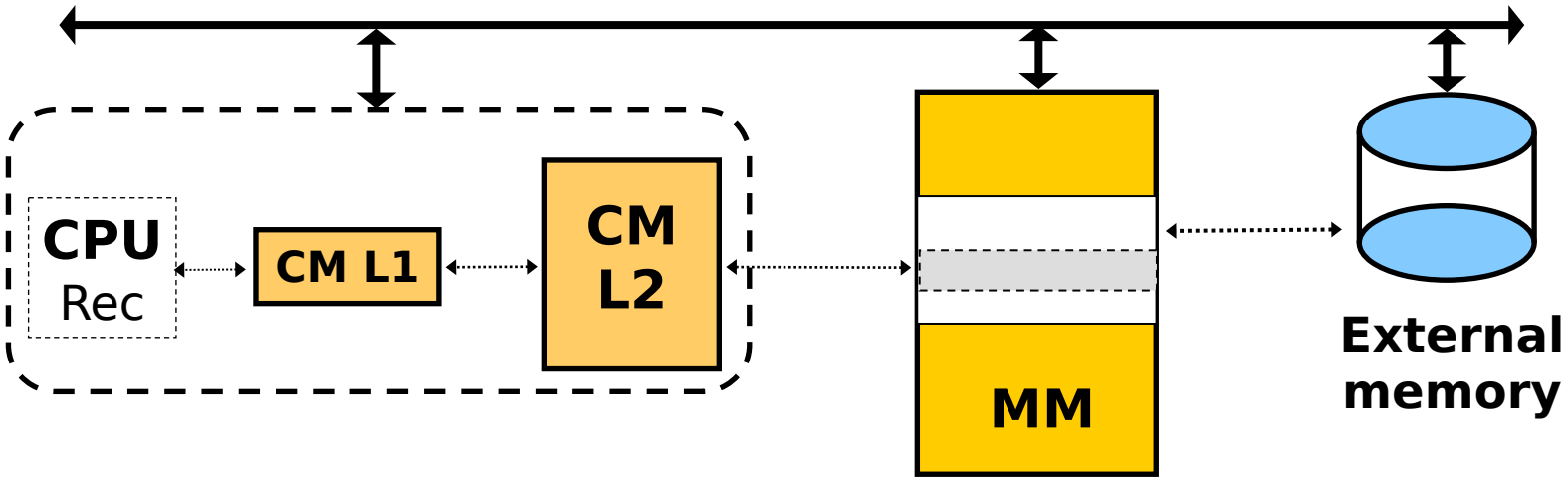
> When the processor requires block i, block i+1 is also transferred from MM to CM.
ATT: **bet** → more traffic in the bus, pollution in the cache

> When to prefetch the block?

- *prefetch **always***: when there is an acces to block i, carry block i+1.
- *prefetch **on miss***: when the access to block i is a miss, carry blocks i and i+1.

Miss rate can be reduced 40% - 80%

Summary



In a memory hierarchy, the **average access time** can be represented in the following way: :

$$T_a = h_1 \times T_1 + (1-h_1) \times [h_2 \times T_2 + (1-h_2) [...]]$$

h_i hit rate in level i T_i access time in level i

Latencies of memory operations:

It is difficult to model precisely the latency of a memory operation because many devices participate at the same time in the operations and they do not finish at the same time. This makes the situation different depending on the viewpoint: processor, cache memory or main memory.

In addition, there are different situations -hits, misses, replacements, *wt*, *wb*, correspondence...-, and the processes required for each of them can be optimised at different levels.

We will show a **simple model**, where times are calculated always from the memory-system point of view and until the complete operation is finished (the processor can often start earlier). Many **simplifications will be done** to make easy calculations; real cases are more complex.

(in addition, as we will see in the next topic the execution of the instructions is pipelined!).

CM operation latency

- **Time to transfer a data block to cache** (or the other way round)

  the time to transfer a **data-block** can be modelled next way: first word (T_{MM}) plus the rest of the words of the block from the interleaving buffer (T_{buff})

  $$T_{bt} = T_{MM} + (num\_words - 1) \times T_{buff}$$

- Time to **read** a word and data traffic in the bus

  (T_{CM} = average time to access cache memory)

  - Hit: T_{CM} the word is in cache; fastest operation.
    No data traffic.

  - Miss: $T_{CM} + T_{bt}$ Search for it and it is not there. Transfer the data block containing the word from MM.
    Data traffic: one block (MM > CM).

CM operation latency

> Time to **write** a word - *write-back* (= reading)

- Hit: T_{CM} the word is in cache. No data traffic.
- Miss: $T_{CM} + T_{bt}$ Search and it is not there. The data block needs to be trasferred.
  Data-traffic: one block (MM > CM).
  (+ repl.) $T_{CM} + 2T_{bt}$ The block needs to be charged in the position of a modified block
  >> the replaced block must be transferred to MM.
  Data-traffic: two blocks (MM > CM; CM > MM).

> Time to **write** a word - *write-through*

- Hit: $T_{CM} + T_{MM}$ the word is in CM; MM must also be updated.
  (att: it can be done at the same time)
  Data-traffc: one word (> MM).
- Miss: $T_{CM} + T_{MM}$ Do not transfer the block (*no write allocation/ not upd. CM*)
  Data-traffic: one word (> MM)
  $T_{CM} + T_{MM} + T_{bt}$ Transfer the block (*write allocation/ updating CM*)
  Data-traffic: one word (> MM) + one block (MM > CM)

CM Optimisations

- Fast accesses to the CM are important to achieve efficient executions. On the one hand, the hit rate has to be maximised. And moreover, the time to access to data on misses has to be minimised.

- Some optimisations can be done to minimise the time, required on misses. For instance:

Minimising the data-traffic between the processor and MM is good: the capacity of the bus to transfer data can be saturated (for instance when many contiguous misses happen) and consequently waiting times become greater.

Minimise the time required on misses:

a. To read/write data it must be searched in CM, and on misses, MM must be accessed.

Not to lose time searching for data, operations start at the same time in **cache memory and main memory**. Then, if it is a hit in cache, the operation started in MM is invalidated; in case of miss, no time is lost.

b A word that is not in cache is required: we need to transfer the corresponding block. The block contains n words but the transferring does not start from the first word but from the word required by the processor:

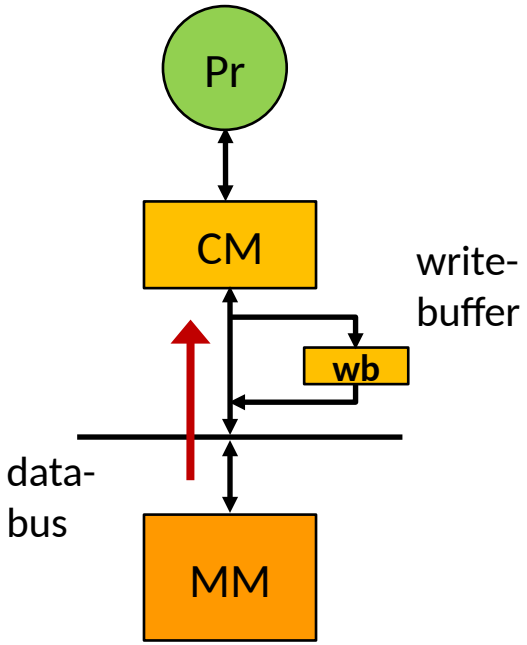
$A_0\ A_1\ \mathbf{A_2}\ldots\ A_{n-1}$ > $\mathbf{A_2}\ldots\ A_{n-1}\ A_0\ A_1$

Minimise the time required on misses:

c A block has to be loaded in cache, but a modified block is in its corresponding position, so MM has to be updated.

The operation is executed in the following order:

- Copy the block in the cache in an write buffer (fast)
- transfer the new block so that the processor starts working
- when possible, transfer the block from the buffer to MM



d In advanced processors, the cache is not blocked on misses: if possible, the processor delays the corresponding instruction and starts with a new one.

CM Optimisations

On the other hand,

Strategies to increase hit rates:

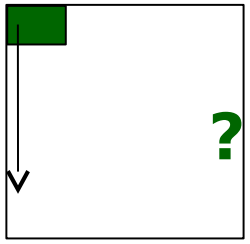
- caches as big as possible
- blocks not too small but careful with pollution
- To have more than one option to place blocks and adequate replacement-policy
- improve locality (time/space)

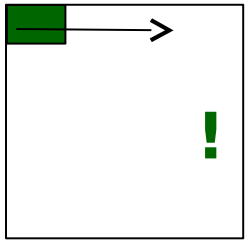
Options to improve locality (temporal and space)

- prefetch
- optimise the code (programmer or compiler)

Some examples to optimise the code:

a Try to access to contiguous words to use all the words in a block. (change the order of the loops)





b Try to "reuse" data

```
for i...
  A[i] = B[i] * B[i];

for i...
  C[i] = B[i] + A[i];
```

Blocks from A and B are used in both loops. But they must be read again in the second (might be no any more in cache).

```
for i...
{
  A[i] = B[i] * B[i];
  C[i] = B[i] + A[i];
}
```

Better

Elements of vectors A and B are used in the same iteration. It is enough to read them once.

Some examples to optimise the code:

- C Try to minimise the number of memory accesses. It is better not to use memory than high hit rates.

```
for i
for j
  ... = ... + A[i]
```

Same value read once and again

```
for i
  X = A[i]    in a register
  for j
    ... = ... + X
```

Read once, store in a register and use it once and again without reading memory.

CM Summary

- The speed of the processor and the memory are very different. Cache memory is the main device to make program executions faster.

- Instructions/data are not randomly used (locality) > memory hierarchy.

  2 - 3 cache levels are organised (L1, L2...). In level L1, the cache is usually distributed: data and instructions.

  The content of each memory level is a subset of the next level.

- Bet: transfer to the cache the data that will likely be used the sooner and the more, by blocks.

  A block contains n contiguous words; for instance, 8 8-byte words, 64 bytes.

CM Summary

- Three types of correspondences (where to locate a block):

  Direct mapped: to a known position.

  Fully associative: to any free position.

  Set associative: to a known set (direct) and within the set to any free position (associative).

- 3 types of misses:

  Load: the first time instruction/data are loaded.

  Capacity: all the data does not fit in cache.

  Correspondence: a block has been removed from the cache to load a new one (although the cache is not full).

CM Summary

- Added to data, cache memories have a directory to find blocks (where are they?). Busy – dirty - tags

  tag: information required to find a block, depends on the correspondence.

- Two write strategies:

  WT: writes in cache and MM: coherent copies.

  WB: writes only in cache. When replacing a block, MM must be updated.

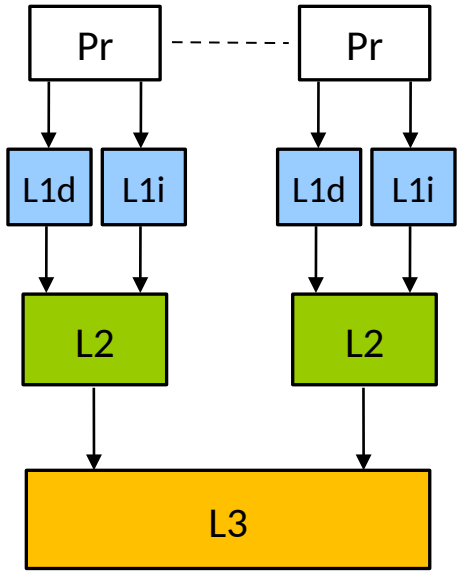
  In misses, two options: transfer the block to cache (*write allocation*) or not.

- Many possible optimisations (prefetch, write-buffers, compiler...). Aim: to minimise the execution-time of the programs.

CM Summary

- Nowadays, average or high level processors have many cores in the chip.

  The cache memory is divided in three levels. Usually, L1 and L2 caches are private for each of the cores, and L3, a lot bigger, is shared for every core (on-chip or off-chip).



CM Summary

> High standard: IBM **Power9** (24 *core*, 8.000 million transistors, line = 128)

L1: 32 kB + 32 kB (core), 8-way set associative

L2: 256 kB (core), 8-way set associative.

L3: 120 MB (shared), 20-way set associative.

> Medium standard: Intel **i7-7700** (Kaby Lake, ~2.000 million trans., 4 *core*, line = 64)

L1: 32 kB + 32 kB (core), 8-way set associative.

L2: 256 kB (core), 4-way set associative.

L3: 8 MB (shared), 16-way set associative.

> Movile devices: ARM **Cortex A8**

L1: unified, 16 or 32 kB; 4-way set associative

L2: unified: 0, 128 kB... 1 MB, 8-way set associative.

CM Summary

We have analysed the basic structure and use of cache memories but there are **many more techniques** to increase efficiency, thus, reduce access times. The design of the cache memory affects severely to the performance of a processor.

In addition **new problems** appear when more than a core is used.

As a variable can be in more than one L1 cache, what happens if it is modified in one of them? How will the rest of the cores know that it has been changed?

The problem of the data coherence is solved using special controllers called **snoopy** and will be analysed in subject **PAR/EHP**.