1. Memoria cache

Introducción: jerarquía de memoria.

Características generales de las memorias cache.

Principales parámetros de diseño: bloque, correspondencia, algoritmo de reemplazo y política de escritura.

Problema

El procesador es más rápido que la memoria y esta diferencia de velocidad entre la CPU y la memoria es cada vez mayor.

La CPU debe utilizar la memoria lo más rápido posible para acceder a las instrucciones y a los datos

Atención: **el componente más lento, la memoria, determina la velocidad de todo el sistema**

- El tiempo de acceso a memoria aumenta con el tamaño de la misma: **las memorias pequeñas serán más rápidas**. Sin embargo, necesitamos memorias grandes.

  ¿Cómo estructurar el sistema de memoria para realizar las operaciones (lectura/escritura) lo más rápido posible? Repasemos los conceptos principales.

Tipos de memoria RAM

- **RAM estática:** rápida, pero necesita 4-5 transistores para cada bit de memoria.
- **RAM dinámica:** mayor integración (un transistor por cada bit de memoria), pero más lenta.

Si el parámetro crítico es la velocidad, se utiliza RAM estática; sin embargo, si es la capacidad, mejor RAM dinámica.

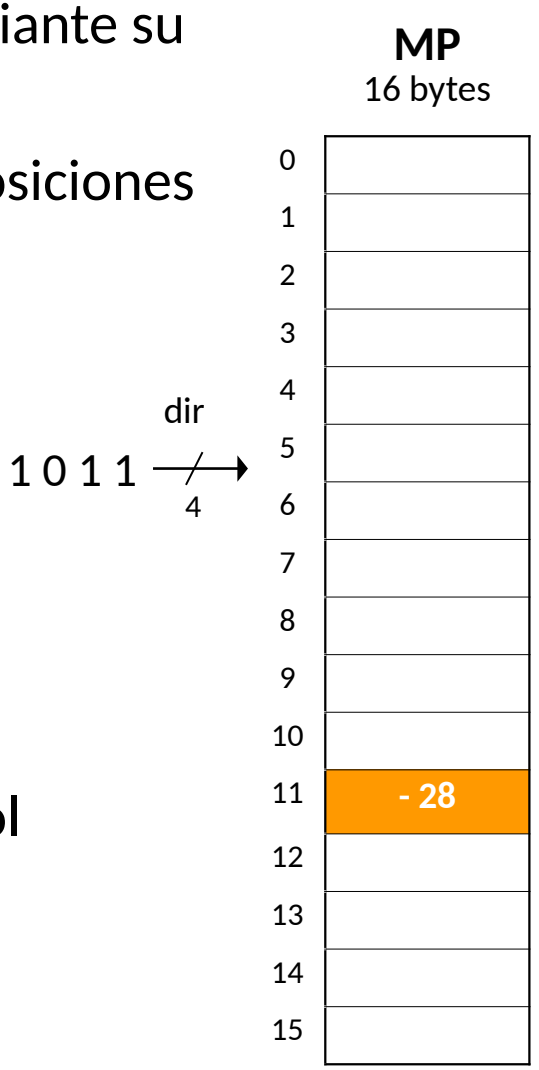
- **Memorias asociativas**

  La búsqueda no se realiza por dirección, sino mediante el contenido. Dada una palabra, el resultado puede ser: **sí**, está en memoria; o **no**, no está (además de la información asociada a esa palabra; por ejemplo, dónde está). Son más complejas que las memorias RAM habituales y tienen un uso especial en las memorias cache.

Estructura de la memoria principal (RAM)

Una posición de memoria se accede mediante su dirección, tanto para leer como escribir.

Las direcciones de una memoria con **P** posiciones son de $\log_2 P$ bits; una dirección de **n** bits direcciona 2^n posiciones.



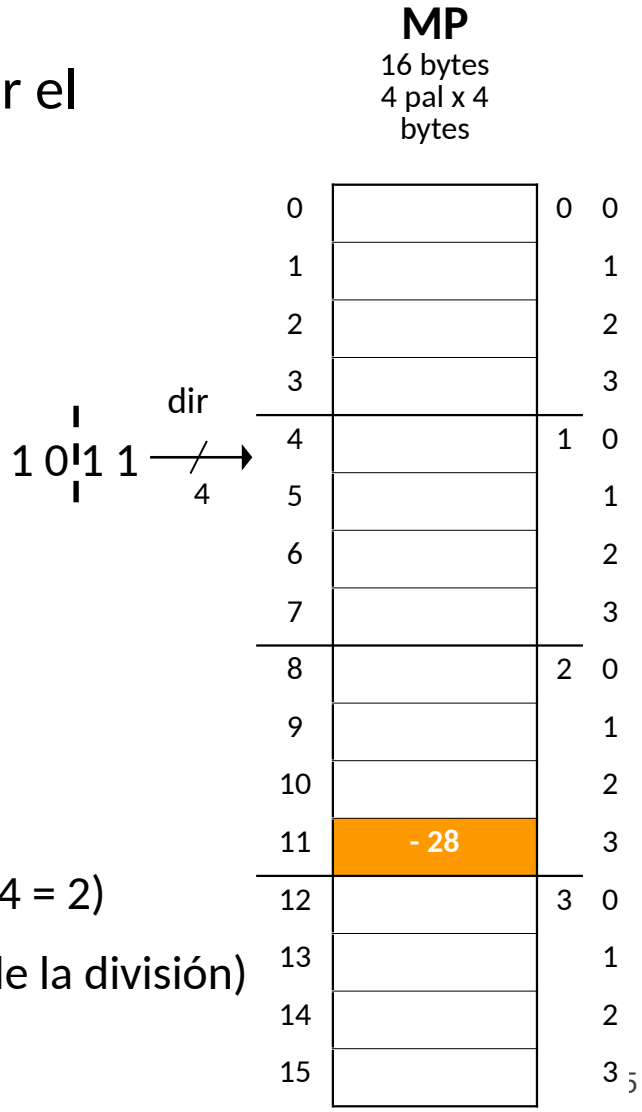
- **Comunicación entre la CPU y la MP**
  - Direcciones → bus de dirección
  - Datos → bus de datos
  - Operación (rd/wr) → bus de control

Estructura de la memoria principal (RAM)

Unidad de información direccionada por el procesador:

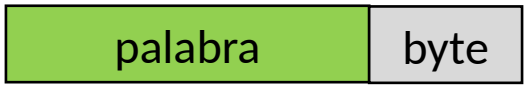
> **byte** (habitual)

> palabra: 4 u **8 bytes**



En general, las direcciones que genera el procesador suelen ser alineadas a la palabra (indican el comienzo de una palabra).

Estructura de la dirección:



palabra = dir **`div`** tam_pal (división entera: 11 / 4 = 2)

byte = dir **`mod`** tam_pal (11 `mod` 4 = 3 resto de la división)

Un apunte sobre las direcciones de memoria

Al cargar el programa/datos en la MP, hay que decidir en qué posiciones se ubican, según qué espacio de memoria está libre en ese momento. Por tanto, las posiciones en las que se carga un programa en memoria no son fijas. El programa cargador decide en qué posiciones físicas se cargan el programa y sus datos.

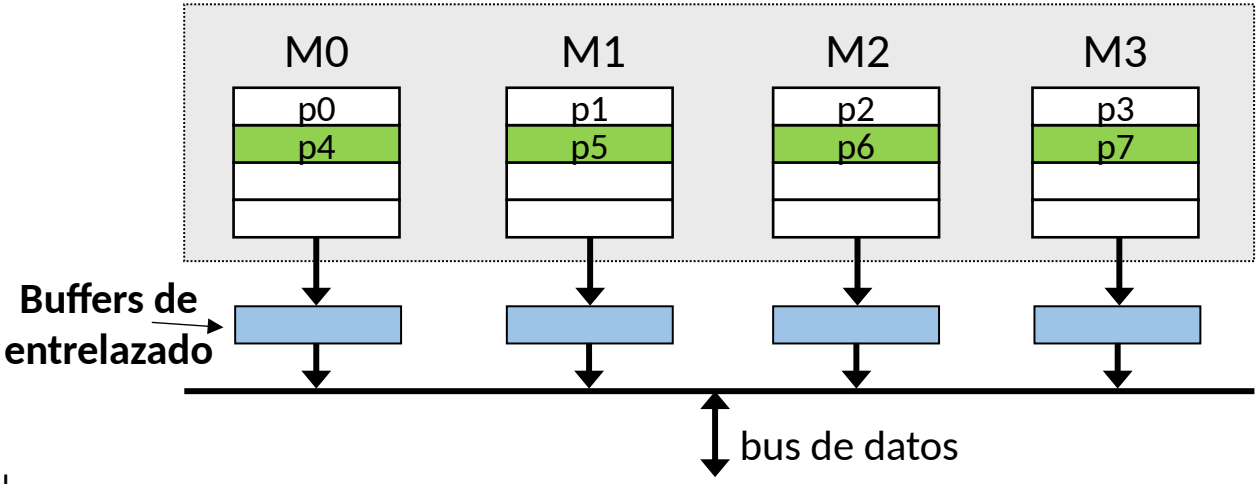
Por ello, el procesador utiliza **direcciones lógicas**, no direcciones físicas concretas. Las direcciones lógicas deben **traducirse** a **direcciones físicas** (de memoria). Esta traducción se realiza a través de un hardware especial: **TLB** (*translation look-ahead buffer*).

Para simplificar, en este tema trabajaremos con **direcciones físicas**.

Estructura de la memoria principal (RAM)

La memoria principal se organiza en varios módulos, de forma que las **direcciones/posiciones se entrelazan** entre los módulos.

De esta forma, **se puede acceder de forma simultánea a direcciones consecutivas**, ya que estarán en módulos diferentes. El contenido de estas direcciones se guarda en los buffers de entrelazado y desde aquí, mediante el bus de datos, se transfieren al procesador



El nivel de entrelazado define el **tamaño del bloque** (en el ejemplo, 4 palabras)

Estructura de la memoria principal

¿Por qué se entrelaza la memoria? El acceso a las instrucciones y a los datos **no es aleatorio**

"El 90% del tiempo de acceso se consume en el 10% del código"

Principio de localidad (*locality*):

- **localidad temporal**: si se utiliza una palabra, seguramente pronto se volverá a utilizar.

  $t \rightarrow @i \gg t+\Delta t \rightarrow @i$ bucles…

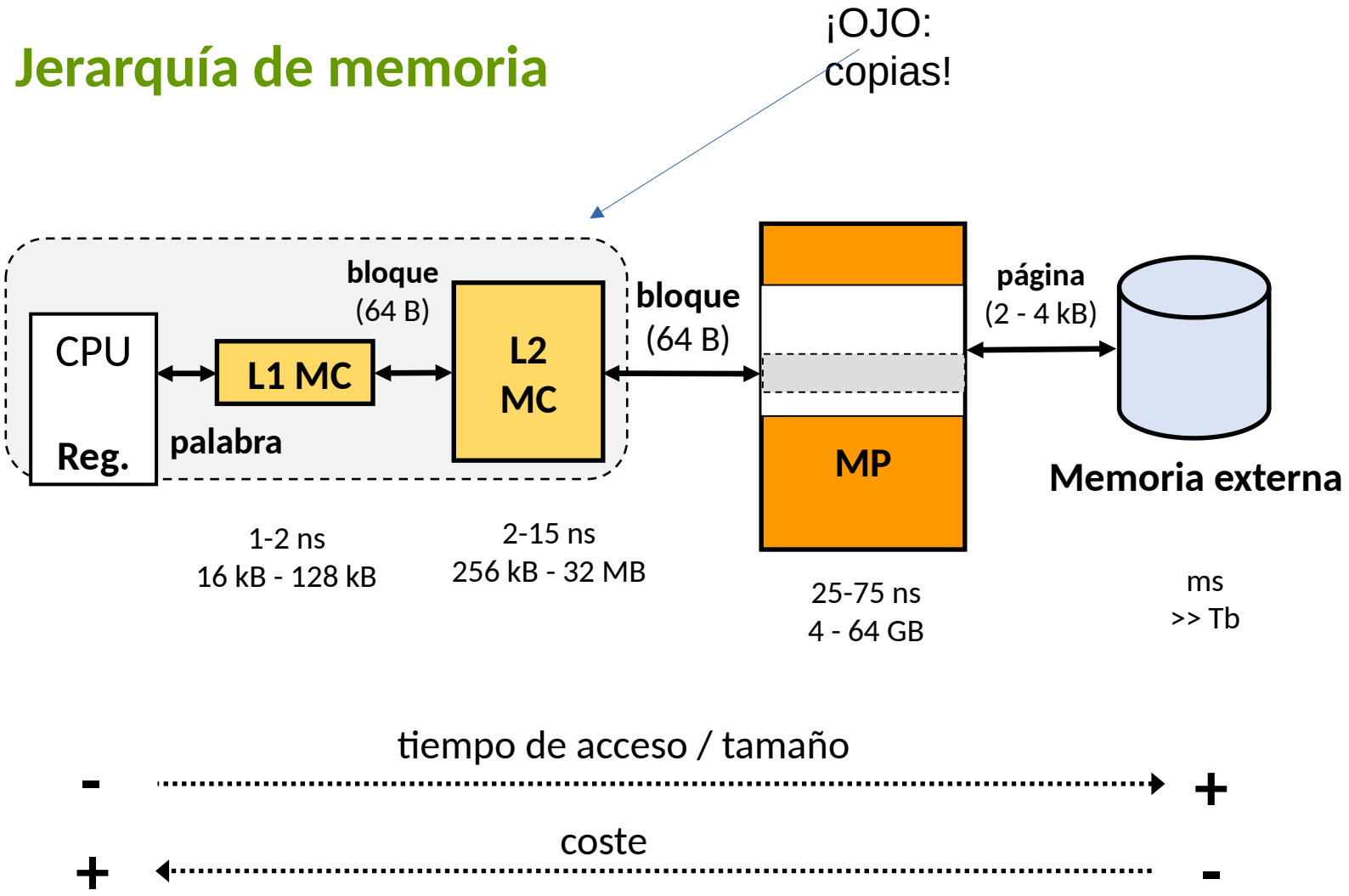
- **localidad espacial**: si se utiliza una palabra, seguramente la siguiente palabra será la palabra consecutiva

  $t \rightarrow @i \gg t+1 \rightarrow @(i+\Delta)$ ejecución secuencial

  acceso a vectores…

- Debido al principio de localidad, el sistema de memoria se organiza en varios niveles, definiendo una **jerarquía de memoria**
  - En los niveles superiores: **memoria cache** (MC)
    - Memoria pequeña pero rápida. RAM estática.
    - Instrucciones y datos que más utiliza el procesador.
    - Repartida en varios niveles (L1, el más pequeño; L2, L3).
    - Dado que se pueden integrar muchos transistores en un chip, L1 y L2 se integran dentro del chip: acceso rápido.
  - En los niveles inferiores: **memoria principal** (MP)
    - Memoria más grande (1.000 veces) pero más lenta (5 – 10 veces).
    - RAM dinámica.

Jerarquía de memoria



Principio de inclusión

El contenido de un nivel de la jerarquía de memoria es un subconjunto del nivel anterior. Por ello, puede haber varias copias de un bloque de datos, tal y como se indica en el esquema.

L1	1	-	-	-
L2	1	1	-	-
MP	1	1	1	-

No obstante, en algunos procesadores no se cumple esta condición y la cache L2 no es inclusiva: un bloque puede estar en L1 sin estar en L2.

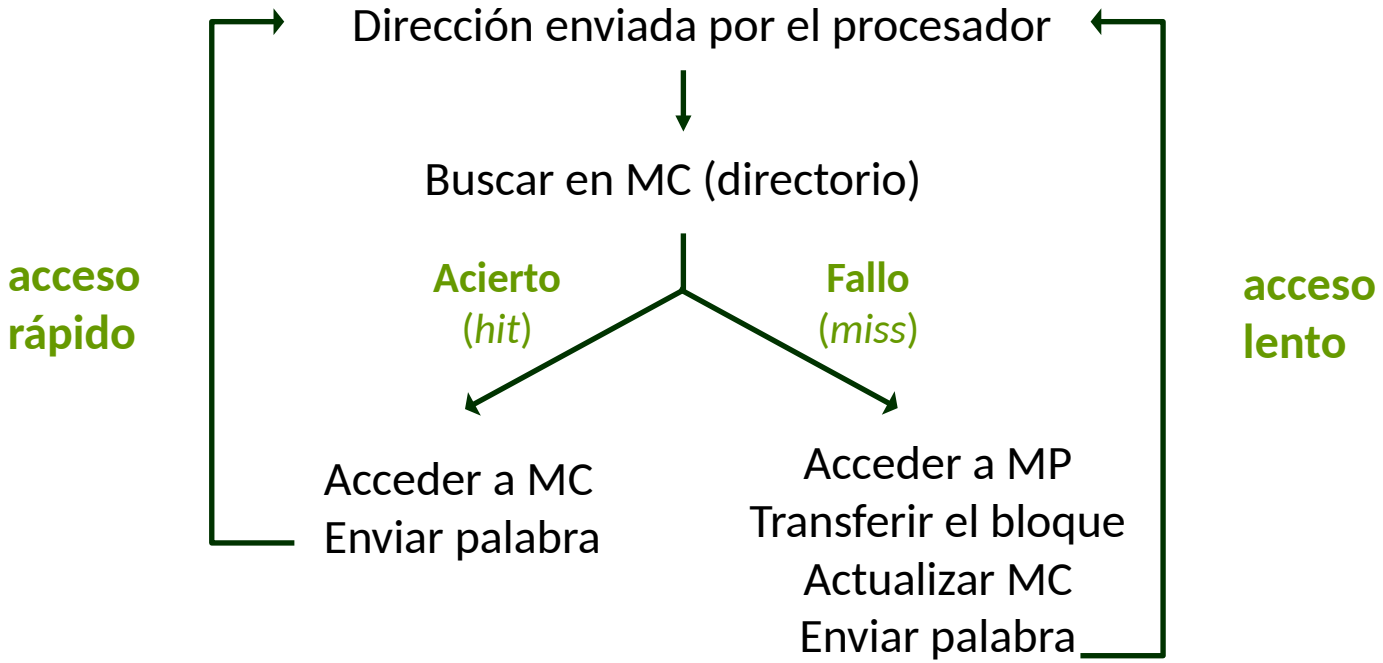
El procesador buscará la información en la **memoria más cercana**. Si no están en ese nivel, buscará en el siguiente nivel y así sucesivamente. Por ello, **el tiempo de acceso a la información es variable**, dependiendo de su localización.

Acceso rápido: la información que se busca está en un nivel cercano (en MC). Pero en la MC no cabe toda la información!

>> Hay que hacer una **APUESTA**: ¿qué se lleva a MC?

- A la hora de buscar la información en la memoria cache, se puede producir un **acierto** (la información está en MC) o un **fallo** (no está en MC).

  Por ejemplo, al leer una palabra:



- En resumen, así se puede expresar el tiempo de acceso a la jerarquía de memoria:

$$T_{acceso} = h \times T_{MC} + (1 - h) \times T_{fallo}$$

h	tasa de acierto (1 – h, tasa de fallos)
T_{MC}	tiempo de acceso a memoria cache
T_{fallo}	tiempo para acceder a la información (instrucción/dato) en el siguiente nivel (en función de la tasa de aciertos de cada nivel)

- **Para que sea eficiente**, la tasa de aciertos, **h**, debe ser **muy alta** (> % 95).

- A la unidad de transferencia entre MC y MP se le llama **bloque** (line)

  Bloque: 2^n palabras consecutivas en memoria.

  Por ejemplo, 64 bytes: 8 palabras de 8 bytes.

- ¿Por qué bloque y no palabra? **¡Localidad!**

Si se transfiere una palabra a MC para usarse en un determinado momento, seguramente el siguiente acceso será a una palabra consecutiva. Eso sí, se trata de una apuesta.

Por ello, se aprovecha la transferencia MP→MC para transferir más de una palabra, casi al mismo coste.

En general, el tamaño del bloque coincide con el entrelazado de la memoria.

Estructura de la memoria cache: directorio / contenido

La dirección de una palabra indica su posición en MP. ¿Pero en la memoria cache, cuál es su dirección?

Una memoria cache tiene dos partes:

- **contenido**: SRAM para almacenar los bloques de datos.
- **directorio**: información acerca de los bloques de datos almacenados en la cache, una palabra por cada bloque de datos: marcas de dirección (tag) y varios bits de control del bloque.

  Dependiendo de la organización de la cache, el directorio puede ser una memoria asociativa (búsqueda por contenido).

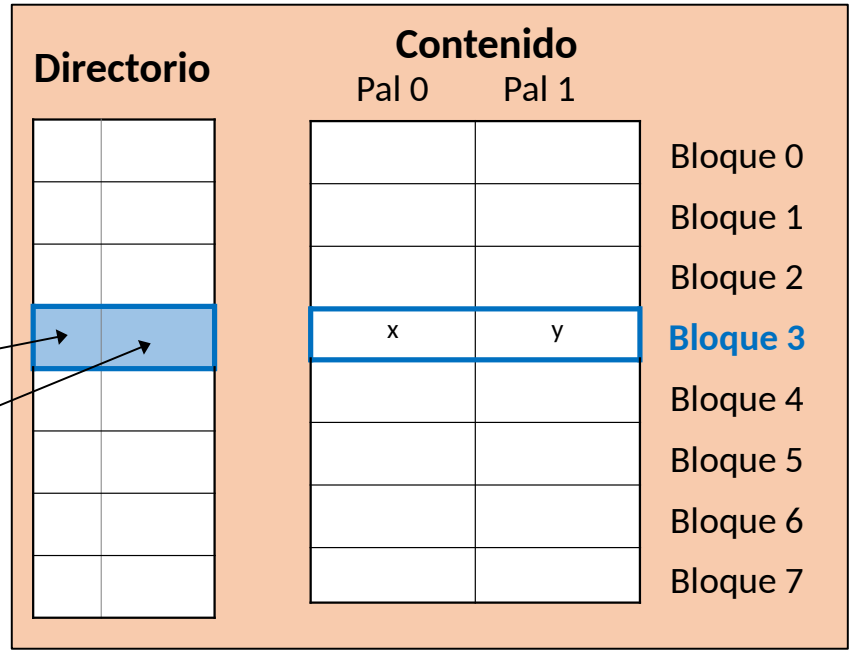
Además, hardware habitual: multiplexores, codificadores …

Estructura de la memoria cache: directorio / datos

Información necesaria para saber si la palabra que se busca está en MC o no.
(SRAM / M. asoc.)

bits de control (ocup/*valid*, ...)

marcas de dirección (***tag***)



- A la hora de diseñar una cache hay que tener en cuenta una serie de parámetros. Veamos los más importantes:

1. Tamaño o capacidad
2. Contenido
3. Bloque
4. Correspondencia: búsquedas y estructura
5. Algoritmo de reemplazo
6. Política de escritura
7. Algoritmo de búsqueda

El **objetivo** siempre es el mismo:

- conseguir la **mayor tasa de acierto** posible
- conseguir el **menor tiempo de acceso** posible

1. Capacidad (tamaño)

A pesar de que cada vez son mayores, la memoria cache es bastante más pequeña que la memoria principal. Por ejemplo, 8 MB de cache, pero 64 GB de memoria principal.

Por tanto, **en la memoria cache no caben todos los bloques de memoria principal**. Por tanto, en algunas ocasiones los bloques que el procesador necesita no estarán en la cache.

En general, cuanto mayor sea la memoria cache, mayor será su tasa de aciertos (**h**).

(siempre teniendo en cuenta el comportamiento del programa)

1. Capacidad

¡**Atención**! El coste también será mayor

>> hay que conseguir un equilibrio entre rendimiento y coste

En los procesadores actuales:

- Dos o tres niveles de cache

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

- *On chip*. Las caches L1 y L2 dentro del chip (más rápidas); L3 dentro o fuera del chip (más lenta), dependiendo de su tamaño.

- Privadas / compartidas. L1 yL2 privadas para cada núcleo (*core*); L3 compartida entre todos los núcleos.

2. Contenido

El procesador accede tanto a **datos** como a **instrucciones**, cuyo patrón de acceso (tasa de aciertos) es diferente. ¿Cómo gestionar esta diferencia?

- **Cache unificada**: datos e instrucciones en la misma cache. Habitual en caches grandes (el tamaño no es problema).

- **Caches separadas:** datos e instrucciones se almacenan en distintas caches >> cache de datos y cache de instrucciones.

  Se pueden utilizar estructuras/parámetros diferentes en las caches; y, sobre todo, **se puede acceder a la vez a datos e instrucciones**.

En general, las caches L1 son separadas (datos e instrucciones), mientras que las caches L2 y L3 son unificadas.

3. Bloque

Bloque: **n palabras consecutivas de memoria;** unidad de transferencia entre MP y MC. ¿De qué tamaño son los bloques?

Por una parte, debido a la localidad espacial, es bueno que los bloques sean grandes: a priori, se cargan palabras que luego se utilizarán, por lo que la tasa de aciertos aumenta.

Pero, no es conveniente que sean demasiado grandes:

- **polución**: podrían no utilizarse todas las palabras del bloque (apuesta), por lo que cargaríamos en la cache información inútil
- caben menos bloques en la cache, por lo que disminuye la tasa de acierto (localidad temporal)
- mayor tiempo de transferencia del bloque

Tamaños habituales: 32 - **64** - **128** bytes

3. Bloque

Dada una dirección de memoria, ¿en qué bloque se encuentra? Dado que las palabras se organizan en bloques, es suficiente dividir la dirección de la palabra con el tamaño del bloque en palabras. Por tanto,

Palabra = dir / tamaño_palabra

Bloque = Palabra / tamaño_bloque_palabras

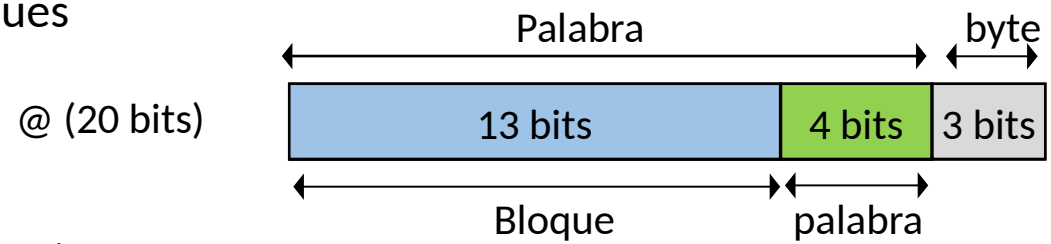
Palabra del bloque = Palabra **mod** tamaño_bloque_palabras

Ejemplo.

MP: 1 MB → direcciones de 20 bits

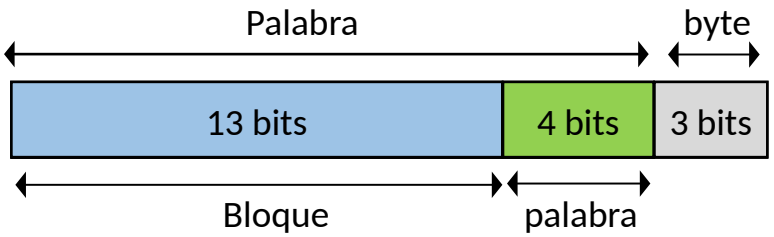
Palabras: 8 bytes (3 bits para indicar un byte) → 128 k palabras

Bloques: 16 palabras (4 bits para indicar una palabra) → 8 k bloques



3. Bloque

@ (20 bits)



MP: 1 MB -- Palabras: 8 bytes (128 k pal) -- Bloques: 16 pal (8 k bloques)

Dirección de memoria: **12536** **0000 0011 0000 1111 1000**

Palabra: dir / 8 = **1567** **0000 0011 0000 1111** 1000

byte (pal): dir **mod** 8 = 0 1567

Bloque: palabra / 16 = dir / 128 = **97** **0000 0011 0000 1111** 1000

Pal (bloq): palabra **mod** 16 = dir **mod** 128 = **15** 97 15 0

3. Bloque

> Tiempo de transferencia de un bloque a memoria cache

Dado que la MP está entrelazada, todas las palabras del bloque se leen a la vez hacia los buffers de entrelazado. Tendremos en cuenta estos tiempos:

- T_{MP} transferencia de una palabra: lectura de MP y transferencia por el bus.
- T_{buff} transferencia de una palabra desde el buffer de entrelazado (previamente ya se ha leído de MP).

Por tanto, **la transferencia de un bloque completo**: primera palabra (T_{MP}) más el resto de palabras desde los bufferes de entrelazado (T_{buff})

$$T_{tb} = T_{MP} + (palabras\_bloque - 1) \times T_{buff}$$

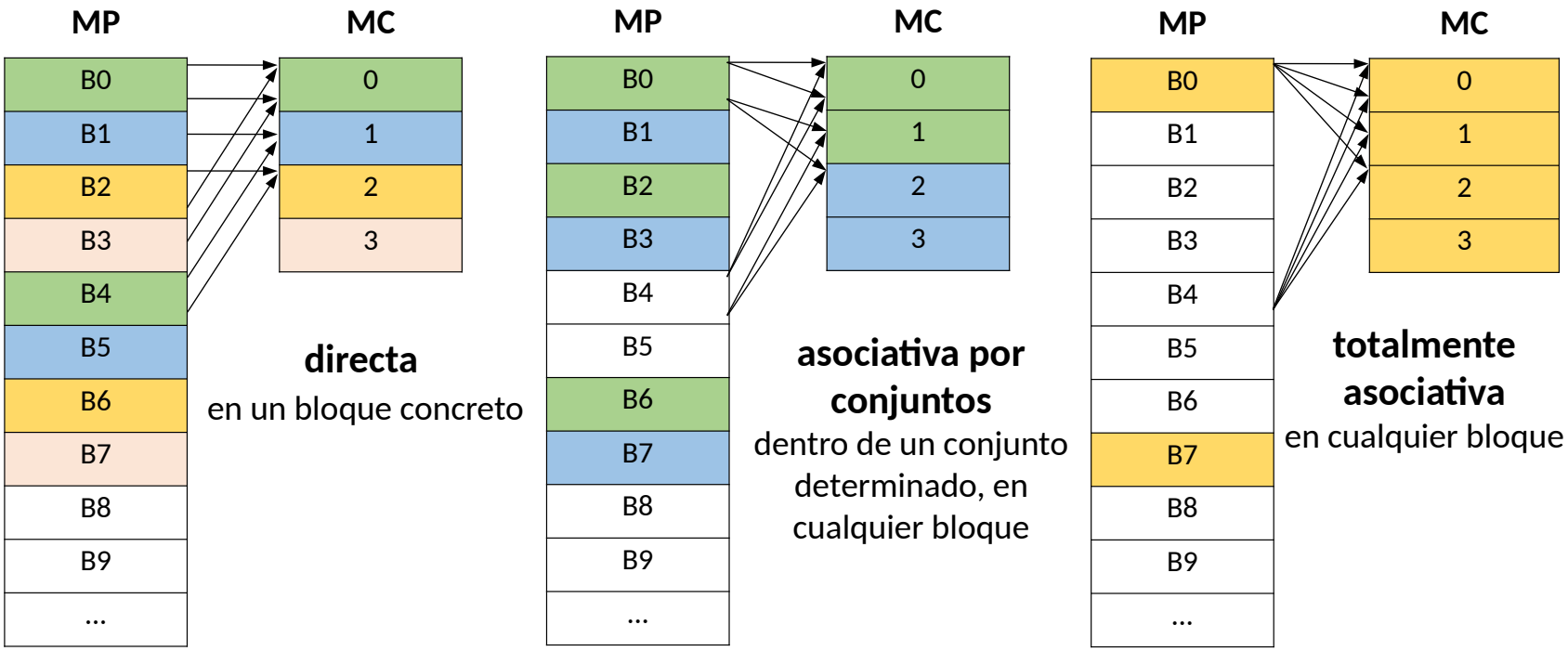
4. Correspondencia (*correspondence*)

En la MC no caben todos los bloques de la MP. Por ello, ¿**dónde** se carga un bloque de MP? ¿**cómo encontrar** ese bloque en MC posteriormente, cuál es su dirección en MC?

Tres opciones:

- Un bloque de MP se carga siempre en un bloque fijo (siempre el mismo) de MC: cache **directa** (***direct***).
- Un bloque de MP puede cargarse en cualquier bloque de MC: cache **totalmente asociativa** (***full associative***).
- El espacio de la cache se divide en conjuntos. Un bloque de MP se carga siempre en un conjunto fijo (directa) y, dentro de ese conjunto, en cualquier bloque (asociativa): cache **asociativa por conjuntos** (***set associative***).

4. Tipos de correspondencia



Por tanto, el contenido de la cache puede organizarse de diferentes formas, en función del número de conjuntos y del tamaño de cada conjunto :

N×1 --- N/2×2 N/4×4 N/8×8... --- 1×N

directa asociativas por conjunto totalmente asociativa

4. **Correspondencia:** marcas de dirección (*tag*)

Una posición de cache no identifica un bloque concreto, dado que en esa posición puede cargarse más de un bloque.

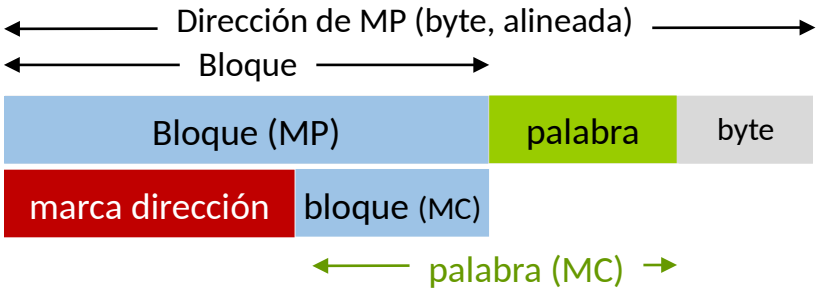
Por tanto, al cargar un bloque en la cache, hay que mantener **información** de dicho bloque en el **directorio**. En todas los accesos a memoria, se debe analizar el directorio para ver si la palabra (el bloque) a la que se quiere acceder está o no en cache.

A dicha información se le llama **marca de dirección** o ***tag***. Es una parte de la dirección del bloque, en función de la correspondencia de memoria cache que se utilice.

Veamos los tres casos posibles.

a Directa

Un bloque se carga siempre en el mismo bloque de MC. Los bits de menos peso del bloque indican su posición en la cache; el resto de los bits se guardan en el directorio, para identificar el bloque que está en esa posición.

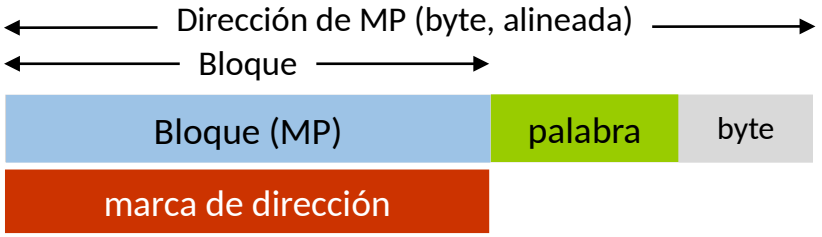


bloque (MC) = Bloque (MP) **mod** num_bloques_cache

tag (direct.) = Bloque (MP) / num_bloques_cache

b Totalmente asociativa

Un bloque puede estar en cualquier posición en la cache; por tanto, hay que guardar todos los bits de la dirección del bloque en el directorio para identificar el bloque que está en esa posición.

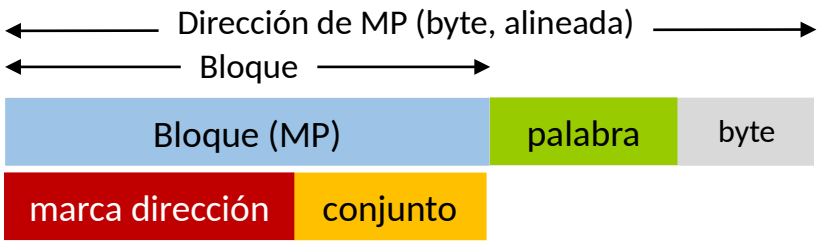


bloque (MC) = cualquiera (¡asociativa!)

tag (direct.) = dirección del bloque

c Asociativa por conjuntos

Un bloque se carga en un determinado conjunto, siempre el mismo (directa), y dentro del conjunto en cualquier posición (asociativa).
Por tanto, los bits de menos peso del bloque identifican el conjunto; el resto de bits, se guardan en el directorio.

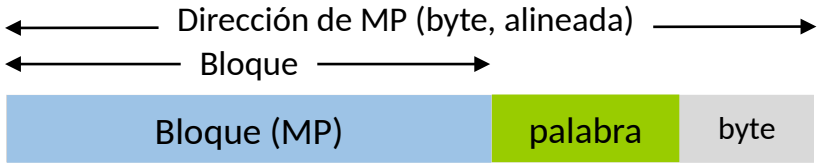


conjunto (MC)= bloque (MP) **mod** num_cjtos_cache

bloque (MC) = cualquiera, dentro del conjunto

tag (direct.) = bloque (MP) / num_cjtos_cache

Recuerda: bits de menos peso → resto de la división
bits de más peso → división



Ejemplo

MP: 64 kB – palabras de 4 bytes (16 k pal) – bloques de 8 palabras (2 k bloques)

Dirección, 16 bits = **25396** 0110 0011 0011 0100]
Palabra, 14 bits = 25396 / 4 = 6349 0110 0011 0011 01
Bloque, 11 bits = 6349 / 8 = **793** 0110 0011 001
Palabra/bloque, 3 bits = 6349 mod 8 = 5 1 01

MC: 1 kB → 256 palabras → **32 bloques × 8 palabras**

a. directa

Bloque(MC), 5 bits = **793** mod 32 = **25** 11001
tag, 6 bits = **793** / 32 = **24** 0110 00
Palabra de MC, 8 bits = 11001 101

b. totalmente asociativa

tag, 11 bits = **793** 0110 0011 001
Palabra de MC, 8 bits (5 direc. + 3) = xxxxx 101

c. asociativa por conjuntos, 8 conjuntos × 4 bloques

conjunto, 3 bits = **793** mod 8 = **1** 001
tag, 8 bits = **793** / 8 = **99** 0110 0011
Palabra de MC, 8 bits (5 direc. + 3) = xxxxx 101

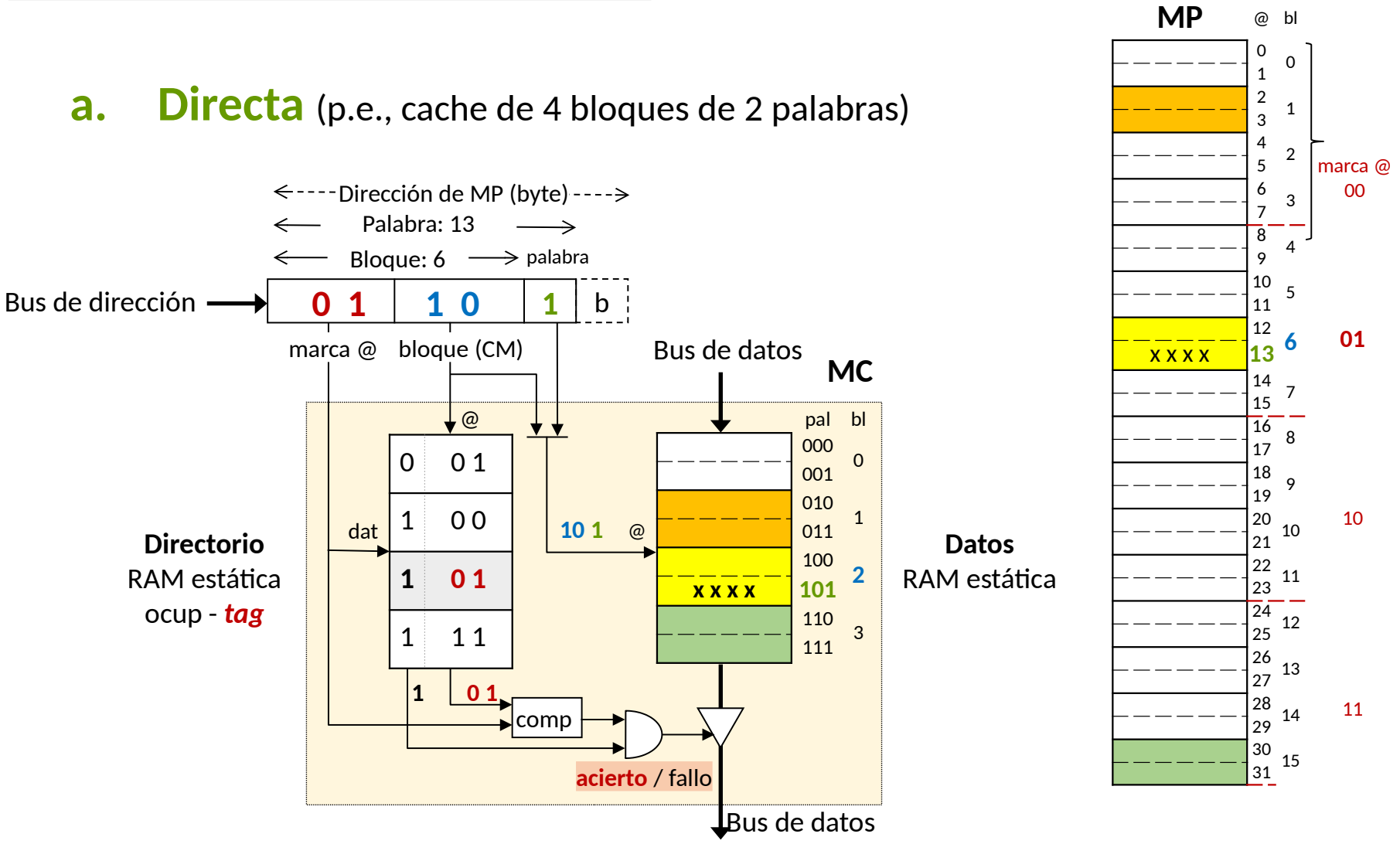
4. Correspondencia: estructura de la cache (directorio + datos)

La memoria cache tiene dos partes: **directorio** y **datos** (contenido). Para almacenar los datos se utiliza una memoria SRAM; sin embargo, el diseño del directorio depende de la correspondencia.

En el caso de cache directa, es suficiente una memoria SRAM, dado que se conoce dónde se almacena el bloque. Sin embargo, en la cache totalmente asociativa es necesaria una memoria asociativa, dado que el bloque puede estar en cualquier posición. En el caso intermedio, asociativa por conjuntos, existen diversas soluciones, en función del número de conjuntos.

Veamos a continuación cómo se organiza el directorio de la cache en cada caso.

a. Directa (p.e., cache de 4 bloques de 2 palabras)



Búsqueda: leer en contenido del directorio en la posición del bloque; comparar el tag del bloque que está en la cache (si la posición está ocupada) con el tag del bloque que se busca

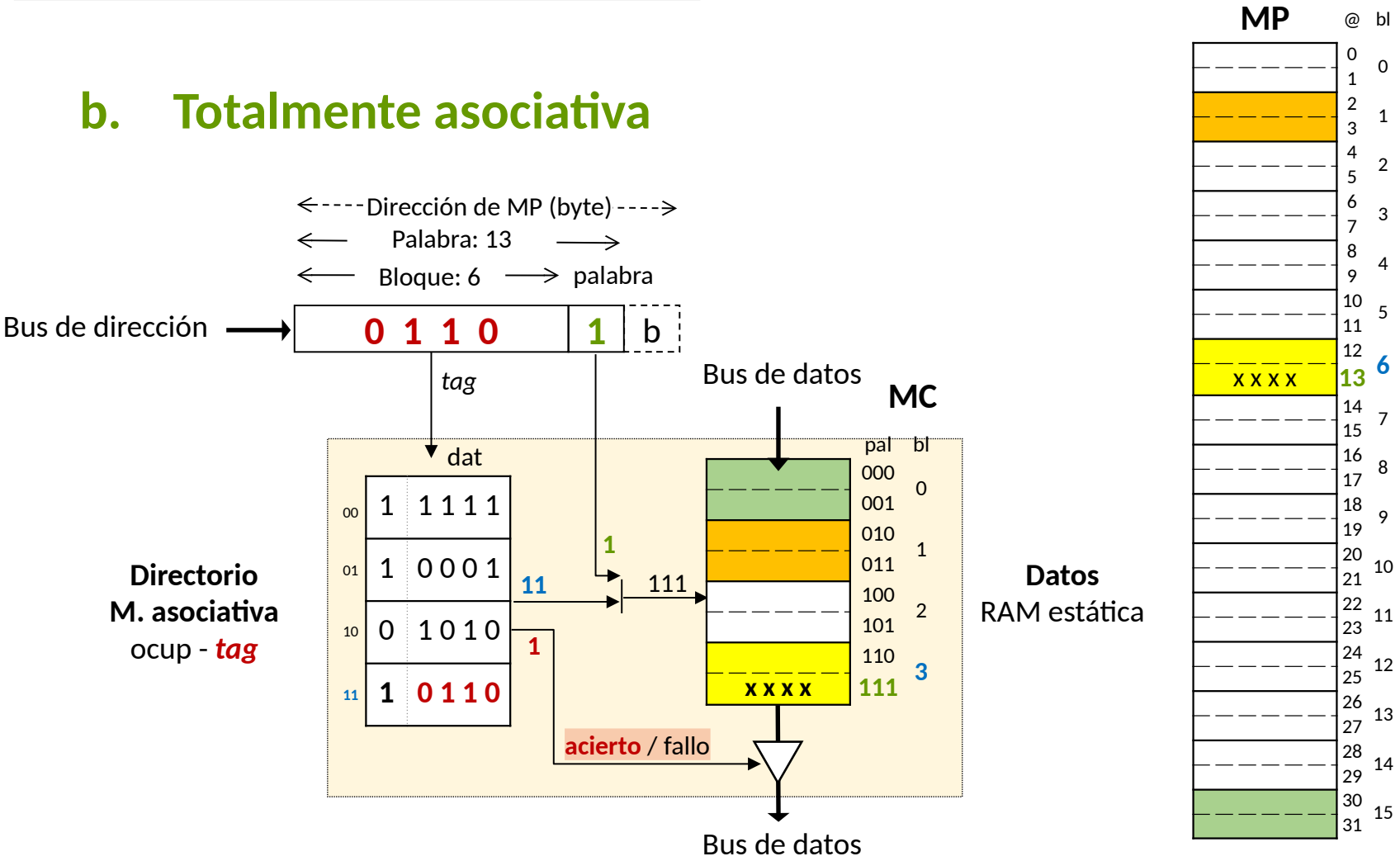
a. Directa

- El directorio y la memoria de datos (contenido) son RAM estáticas y pueden accederse **en paralelo**, dado que se conoce la ubicación del bloque. Se trata de una estructura **barata** y **rápida**.

- Puede no gestionar adecuadamente el espacio de memoria. A un bloque de MP le corresponde un bloque fijo de MC; si está ocupado, hay que reemplazar ese bloque para meter otro (a pesar de que haya sitio en la MC!)

  Por ello, la tasa de aciertos, **h**, puede ser más **baja** de la esperada.

b. Totalmente asociativa



Búsqueda: buscar el tag del bloque en el directorio (memoria asociativa); la respuesta es si ese tag está o no está, y, si está, en qué posición está. Una vez obtenida la posición, se puede acceder a la RAM de datos.

b. Totalmente asociativa

- El directorio es una **memoria asociativa**, que es mas compleja que una SRAM, y también **más cara**.
- El principal problema es que **son necesarios dos accesos secuenciales**. Primero al directorio para ver si el bloque está o no en la cache y en qué posición está, y un segundo a la SRAM de contenidos, una vez conocida la posición del bloque: **no se trata de una solución rápida.**
- Por contra, **gestiona muy bien el espacio de memoria**: hasta llenar la MC, siempre hay sitio para un bloque de MP.
  → tasa de aciertos, **h, grande**

c. Asociativa por conjuntos

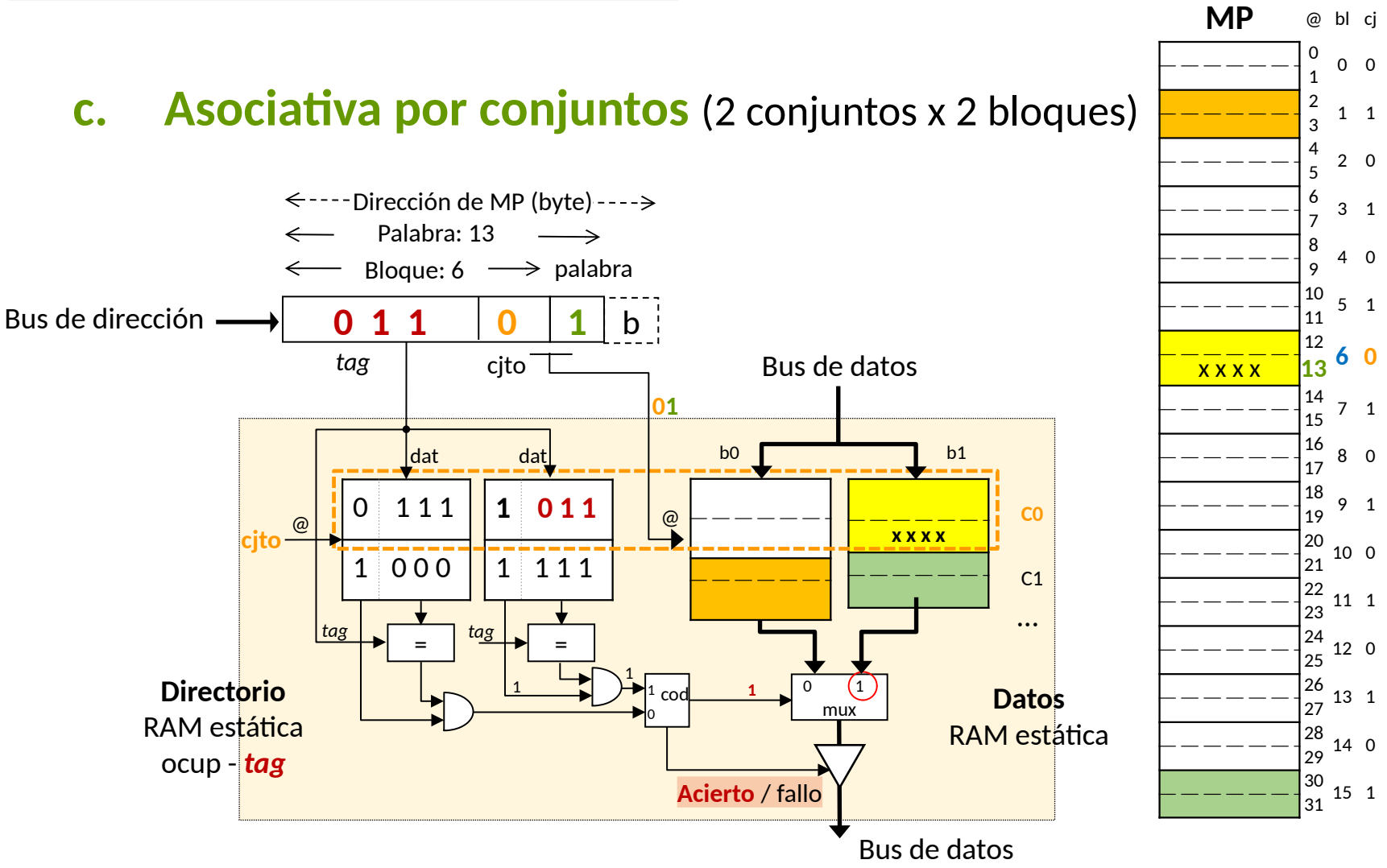
Se trata del caso general; las dos anteriores pueden verse como extremos de este caso general: en la cache totalmente asociativa, 1 conjunto con n bloques; en la cache directa, n conjuntos con un único bloque. Por tanto, la estructura de la cache puede ser una mezcla de las dos anteriores.

En general, el **tamaño del conjunto** (grado de asociatividad, *way*) es **pequeño** (2-16) y el **número de conjuntos** es **grande**. Por ello, se puede evitar el problema que conlleva la asociatividad: la necesidad de realizar dos accesos secuenciales, dado que no se conoce la ubicación del bloque.

Para ello, se utiliza más de un módulo de memoria y se entrelazan los bloques de los conjuntos. De esta forma, es posible acceder a todas las bloques (y palabras) de un conjunto y, finalmente, escoger uno de ellos (para las escrituras, como en los casos anteriores, hay que esperar a recibir la respuesta del directorio).

Veamos un ejemplo.

c. Asociativa por conjuntos (2 conjuntos x 2 bloques)



Búsqueda: acceder a la vez a todos los bloques de un determinado conjunto (y al directorio). En función de la respuesta del acceso al directorio, escoger la palabra de la RAM de datos.

c. Asociativa por conjuntos

- El contenido se divide en conjuntos, dentro de los que se puede utilizar cualquier posición libre. Al número de bloques del conjunto se le llama grado de asociatividad (way). Por ejemplo, 8 *way associative* --> conjuntos de 8 bloques.

- Se trata de un **compromiso** entre las correspondencias directa y la totalmente asociativa:

  - tiempo de acceso, similar a la directa
  - tasa de aciertos (h), similar a la totalmente asociativa

- ¿Qué se utiliza actualmente?

  - asociativa por conjuntos (conjuntos de 4 - 16 bloques)

  por ejemplo, i7 L1: 8 *way*; L2: 4 *way*; L3: 16 *way*

4. **Correspondencias** - resumen

Un bloque de MC puede alojar a varios bloques de MP. Por tanto, además de los datos, hay que mantener información (**marcas de dirección** o ***tag***) para identificar los bloques existentes en MC: **directorio**. En cualquier acceso a la cache, hay que analizar el directorio para ver si el bloque accedido está o no en la cache.

En función de la correspondencia, el diseño y la información que hay que guardar en el directorio es distinta. En la cache directa, el directorio es un SRAM; en la totalmente asociativa, es necesaria una memoria asociativa. Las caches directas son rápidas, pero su tasa de aciertos, h, no es la más elevada; en la totalmente asociativa h es más elevada, pero su acceso es más lento.

Las caches asociativas por conjunto son un compromiso entre las dos anteriores; h similar a la totalmente asociativa y un tiempo de acceso similar a las caches directas. La cantidad de bloques (*way*) no suele ser alta (con 4-16 se logran buenos resultados)

5. Reemplazo

Se produce un fallo en MC, por lo que hay que cargar un nuevo bloque de MP, pero ¿si no hay sitio en MC (en la posición que le corresponde al nuevo bloque)?

En este caso, antes de cargar el nuevo bloque, hay que quitar uno de la memoria cache. ¿Qué bloque se sustituye?

- Si la cache es directa no hay elección. El bloque a sustituir está totalmente definido, dado que un bloque de MP siempre se carga en un bloque fijo de MC.
- Si hay asociatividad, hay que elegir un bloque de toda la cache (totalmente asoc.) o de un conjunto (asoc. por conjuntos).

A la hora de elegir el bloque, ¿qué estrategia utilizar? Es decir, ¿qué **algoritmo de reemplazo** se utiliza? Existen dos tipos de algoritmos:

5. Reemplazo

> No tienen en cuenta el comportamiento del programa.

- **Aleatorio** (*random*): cualquier bloque

  Simple / poco optimizado (h pequeña)

- **FIFO**: el bloque que más tiempo lleva en cache ("más antiguo").

Para determinar cuál es el bloque más antiguo se utiliza un contador de n bits en el directorio; en cada acceso, se actualiza el valor del contador (n = log num_bloques). Cuando hay que reemplazar un bloque, se elige el bloque con mayor contador (más antiguo) y se actualizan los contadores de todos los bloques. Por ejemplo, para 4 bloques:

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. Reemplazo

> Tienen en cuenta el comportamiento del programa.

- **LRU**: el bloque que **más tiempo lleva** en la MC (o en el conjunto) sin ser utilizado.

  Se usan contadores y hay que actualizarlos en cada acceso, no sólo en los fallos (0 al último accedido – k – y +1 a los contadores de 0 a k-1)

  Difícil de implementar (salvo que el tamaño del conjunto sea de 2 bloques), pero consigue la mayor tasa de aciertos.

Actualmente: LRU (con 2 bloques por conjunto) o, generalmente, variaciones de LRU (*pseudo-LRU*).

La información de reemplazo debe guardarse también en el directorio: ocup – **reemp** -- *tag*

6. Operaciones

El procesador realiza dos operaciones en la cache: **lectura** (`load`) o **escritura de una palabra** (`store`). La palabra debe estar en cache para poder realizar la operación; por tanto, el primer paso será analizar el directorio.

Lecturas (rd)

Es una operación simple. Tras la búsqueda en el directorio:

- **Acierto**. La palabra está en la cache y se accede desde la cache, de forma rápida.
- **Fallo.** La palabra no está en la cache. Por tanto, hay que transferir desde MP el **bloque** que tiene la palabra (ver apartado de optimizaciones)

En general, para no perder tiempo, se puede empezar la lectura a la vez que la búsqueda

6. Operaciones: política de escritura

Escrituras (wr)

Las escrituras son complejas, dado que suponen una actualización de la información. Al utilizar memoria cache, se trabaja con **copias de la información** (MC y MP). Por tanto, ¿dónde se escribe? ¿cómo se gestionan las copias de la misma información?

Hay dos tipos de escritura:

- ***write-through***: cuando se escribe una palabra se actualizan todas las copias de la jerarquía de memoria (MC y MP).

- ***write-back***: la palabra sólo se escribe en MC. La MP no se actualiza hasta que se reemplace el bloque de la MC

6. Política de escritura: *write-through* (WT)

- Las **copias** de los bloques son siempre **coherentes** (misma copia), pero la latencia de escritura es mayor, dado que siempre hay que actualizar la MP.

  En general, en lugar de escribir directamente en MP se utiliza un buffer de escritura intermedio; el controlador de memoria se encargará de ir actualizando la MP (ver apartado de optimizaciones).

- Se utilizan dos estrategias principales:

  a. Escribir tanto en **MC** como en **MP** (*write allocation*).

  b. Se escribe en **MP**, pero no en MC; si el bloque estaba en MC, se anula y, si no estaba, no se transfiere de MP.

  Ejemplo: al hacer C[i]=A[i]+B[i], ¿merece la pena llevar el vector C a la MC?

6. Política de escritura: *write-back* (WB)

- Dado que las escrituras se hacen sólo en MC, la información **no va a ser coherente**: en las escrituras habrá dos copias distintas en MC y MP.

  El valor válido será el de MC.

  Consecuencia: a la hora de reemplazar un bloque de MC, si está modificado, debe actualizarse en MP.

  Para saber si un bloque de MC ha sido modificado hay que incluir un **bit de control** en el directorio de MC: **modificado** (***dirty***).

  direct. >> ocup -- mod -- reem -- *tag*

  ATENCIÓN: al actualizar una palabra en MC, ese bloque queda modificado, y al reemplazarlo se actualiza todo el bloque en MP, no sólo la palabra modificada.

6. Política de escritura: *write-back* (WB)

El bloque a guardar en memoria se deja en un buffer de escritura; el controlador de memoria se encargará de escribirlo luego en memoria, aprovechando los momentos que el bus de datos esté libre (ver las optimizaciones más adelante).

En general, esta estrategia es más adecuada que WT. No obstante, se vuelve a hacer una apuesta:

- si se actualiza continuamente un bloque de MC antes de reemplazar el bloque, se minimiza el tiempo de acceso y el tráfico del bus entre MP y MC, dado que se actualizará únicamente una vez la MP (al reemplazar el bloque).

- pero si el reemplazo se produce nada más actualizar el bloque una única vez, habrá que actualizar todo el bloque (no sólo esa palabra) en MP, dado que el bit *dirty* afecta a todo el bloque.

> Es habitual que los distintos niveles de cache (L1, L2, y L3) utilicen estrategias diferentes.

6. Política de escritura

Cuidado con las copias: no es difícil saber dónde está la copia a utilizar… siempre que todas las operaciones estén bajo una única unidad de control

En sistemas de un solo núcleo o core eso es lo habitual, a excepción de un caso: las operaciones de entrada/salida (E/S), cuando el controlador de DMA carga datos externos… pero, ¿dónde? ¿en MP? ¿y si esos datos de llevan a cache?

Para resolver ese problema hay dos opciones:

- hacer *flush* en la cache: los datos nuevos se cargan en MP, y las copias que pueda haber en cache de bloques de datos de entrada/salida se anulan (basta con poner ocup.=0 en el directorio).
- los bloques de datos que se usen en E/S no se llevan nunca a MC: cuando se necesite un dato (rd/wr), hay que ir siempre a MP.

7. Algoritmo de búsqueda

Para minimizar el tiempo de acceso, se puede adelantar la búsqueda de la información en MP (*prefetch*), aprovechando la localidad espacial.

> Cuando el procesador utiliza el bloque i, se transfiere en bloque i+1 desde MP a MC (si no está ya en MC).

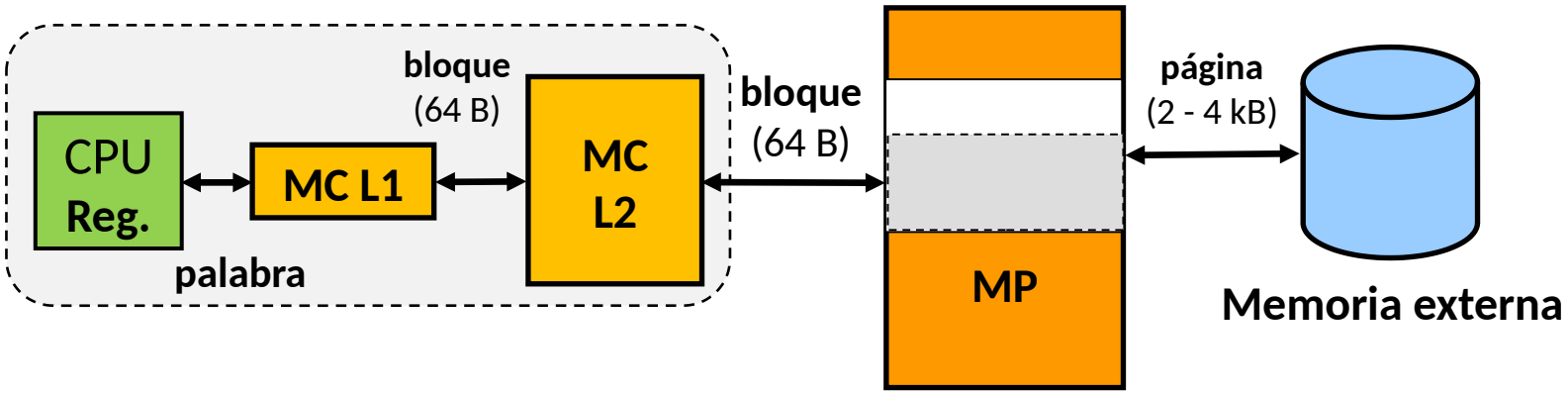
ATENCIÓN: **apuesta** → mayor tráfico en el bus, polución de la MC.

- *prefetch* ***always***: al acceder al bloque i, se trae siempre el bloque i+1 (no está ya en MC)

- *prefetch* ***on miss***: si al acceder al bloque i se produce fallo, se traen a MC los bloques i e i+1. (¿Es lo mismo que traer un bloque el doble de grande?)

La tasa de fallos puede minimizarse entre un 40% y un 80%.

Latencia de las operaciones en memoria



En general, el **tiempo de acceso medio** a la jerarquía de memoria puede expresarse de esta:

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

h_i: tasa de acierto del nivel i T_i: tiempo de acceso al nivel i

Latencia de las operaciones con memoria

Es difícil modelar al detalle la latencia de una determinada operación dado que interviene diferente hardware de forma simultánea en las operaciones, y no acaban a la vez. El modelo depende del punto de vista en que se realice: desde el punto de vista del procesador, de la cache o de la memoria principal.

Además, la casuística es diversa: acierto/fallo, reemplazos, escritura WB/WT… y se pueden realizar optimizaciones a diferentes niveles.

Por ello, es difícil plantear un modelo completo. Para simplificar, tendremos en cuenta el siguiente modelo de tiempo de acceso y tráfico en el bus, desde el punto de vista del sistema de memoria (en general, el procesador puede avanzar más rápidamente, en cuanto tenga el dato). **Vamos a simplificar mucho el modelo**, para realizar los cálculos de forma sencilla. En la realidad, el modelo es más complejo.

- **Tiempo de transferencia de un bloque a MC (**o desde MC**)**

  Tal y como se ha presentado, el tiempo de transferencia de un bloque se puede modelar de esta forma: primera palabra (T_{MP}) + resto palabras del bloque desde los buffers de entrelazado (T_{buff}).

  $$T_{tb} = T_{MP} + (palabras\_bloque - 1) \times T_{buff}$$

- **Tiempo de lectura de una palabra y tráfico en el bus**

  (T_{MC} = tiempo de acceso a memoria cache)

  - Acierto: T_{MC} La palabra está en MC; es la operación más rápida.
    Tráfico en el bus entre MC y MP: no existe.

  - Fallo: $T_{MC} + T_{tb}$ Hay que transferir el bloque desde MP.
    Tráfico en el bus: un bloque (MP > MC).

- **Tiempo** de **escritura** de una palabra - ***write-back*** (= lectura)
  - Acierto: T_{MC} La palabra está en cache. Tráfico en el bus: no existe.
  - Fallo: $T_{MC} + T_{tb}$ La palabra no está en cache. Transferir bloque desde MP
    Tráfico en el bus: un bloque (MP > MC)

  (+ reemp.) $T_{MC} + 2T_{tb}$ Si hay reemplazo de un bloque **modificado** en MC, hay que transferir ese bloque en MP antes de cargar el nuevo bloque en MC
  Tráfico en el bus: 2 bloques (MC > MP; MP > MC)

- **Tiempo** de **escritura** de una palabra - ***write-through***
  - Acierto: $T_{MC} + T_{MP}$ La palabra está en MC; hay que actualizar la MP y actualizar/anular la palabra en MC (operaciones simultáneas en MP y MC)
    Tráfico en el bus: una palabra (> MP).
  - Fallo: $T_{MC} + T_{MP}$ Actualizar MP y no traer el bloque a MC (*no write allocation*)
    Tráfico en el bus: una palabra (> MP)

    $T_{MC} + T_{MP} + T_{tb}$ Actualizar MP y traer el bloque a MC (*write allocation*)
    Tráfico: una palabra (> MP) + un bloque MP > MC)

Optimizaciones para minimizar la latencia de las operaciones

Es fundamental un uso óptimo de la memoria cache para lograr una ejecución eficiente de los programas. Por una parte, es importante **maximizar** la **tasa de acierto** y, por otra, **minimizar** el **tiempo de acceso** en caso de fallo.

Además, es importante minimizar el **tráfico en el bus** entre el procesador y el sistema de memoria: la transmisión de datos puede saturar la capacidad del bus (por ejemplo, en el caso de que se produzcan muchos fallos seguidos) y, en consecuencia, los tiempos de espera pueden llegar a ser elevados.

Veamos algunas optimizaciones básicas.

- Minimizar el tiempo de acceso en caso de fallo:

  a. Para leer/escribir un dato hay que realizar una búsqueda en la cache y, en caso de fallo, acceder a MP.

  Para minimizar el tiempo de búsqueda, las operaciones se inician tanto en la memoria cache como en memoria principal. Posteriormente, si es un acierto en cache, se anula la operación iniciada en MP; si se trata de un fallo en cache, no se pierde el tiempo utilizado en el acceso ya iniciado a MP.

  b. En caso de fallo en cache, a la hora de transferir el bloque de MP, la transferencia no se inicia desde la primera palabra del bloque, sino desde la palabra requerida por el procesador:

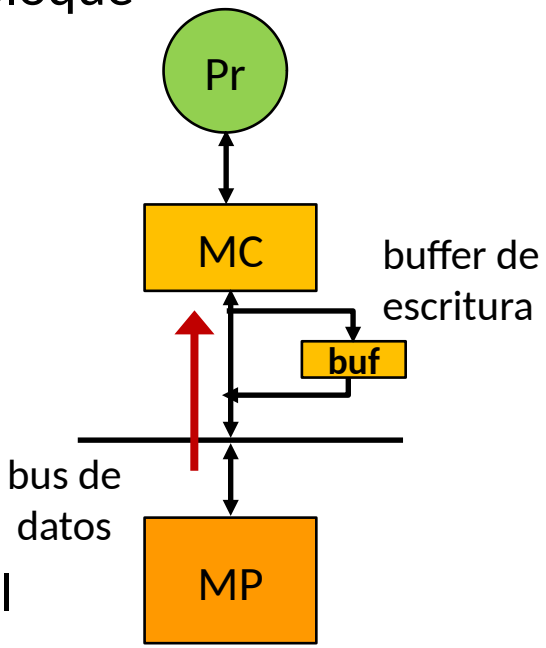
  $A_0\ A_1\ \mathbf{A_2}\ldots\ A_{n-1}$ > $\mathbf{A_2}\ldots\ A_{n-1}\ A_0\ A_1$

  De esta forma, el procesador puede seguir con la ejecución tan pronto como reciba la palabra; entretanto, se cargan el resto de palabras del bloque en la cache.

c. En la escritura WB, en caso de reemplazo de un bloque modificado, antes de cargar el nuevo bloque en la cache, puede ser necesario transferir el bloque a reemplazar (si está modificado) a MP.

La operación se realiza de esta forma:

- se copia el bloque de cache en un buffer interno: **buffer de escritura** (esta copia es rápida).
- **se transfiere desde MP el nuevo bloque**, para que el procesador siga trabajando.
- cuando el bus esté libre, se transfiere a MP el bloque almacenado en el buffer de escritura.



d. En los procesadores avanzados, cuando se produce un fallo en la cache, la cache no se bloquea: el procesador "aparca" la instrucción hasta que se solucione el fallo y sigue con otra instrucción.

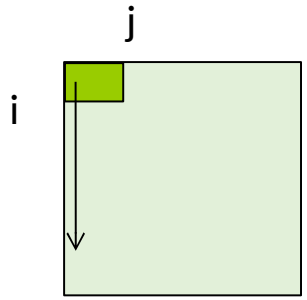
- Por otra parte, como ya hemos visto, para aumentar la tasa de acierto:
  - Caches lo más grandes posibles
  - Bloques no muy pequeños (cuidado con los bloques grandes)
  - Asociatividad / algoritmo de reemplazo

  - E intentar mejorar la localidad en los accesos, tanto espacial como temporal:
    - *prefetch*
    - **optimizar el código** (programador o compilador)

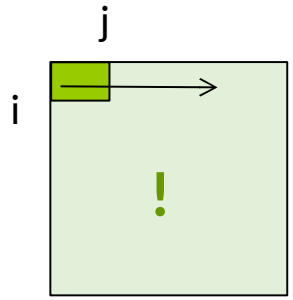
- Algunos ejemplos de optimización de código

  a. Intentar realizar los accesos en palabras consecutivas para aprovechar los bloques (cambiar orden de bucles)

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




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



¡Atención! C: las matrices se guardan por filas en memoria; Fortran: por columnas.

- Algunos ejemplos de optimización del código

b. Intentar reutilizar los datos.

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

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

En los dos bucles se utilizan los bloques de A y B, pero en el segundo caso hay que volverlos a leer (y quizá ya no estén en MC)

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

Mejor.

Ahora los elementos de A y B se utilizan en la misma iteración para las dos instrucciones. Es suficiente una única lectura de memoria para cada vector.

- Algunos ejemplos de optimización del código

c. Intentar minimizar el número de accesos a memoria. En este caso, la mejora no es debida a una mayor tasa de acierto, sino al menor uso de la memoria.

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

en el bucle j se lee siempre el mismo valor

```
for i...
{
  x = A[i];
  for j...
    ... = ... + x;
}
```

el valor se lee una vez y se guarda en un registro

se utiliza n veces, sin acceder a memoria

- Las velocidades de los procesadores y de las memorias son muy diferentes. Para intentar minimizar este gap se utilizan las memorias cache; de hecho, es el principal componente para acelerar la ejecución de los programas
- El acceso a instrucciones y datos no es aleatorio (localidad)

  > la memoria se organiza formando una jerarquía.

  Se utilizan 2 o 3 niveles de cache (L1, L2…). La cache L1 suele ser separada: instrucciones y datos. Recordad: el contenido de cada nivel de memoria es un subconjunto del siguiente.

- Se trata de una apuesta: se trae a la cache la información que se supone que va a ser utilizada por el procesador en ese momento (bloque).

  Un bloque contiene n palabras consecutivas de memoria.

- Existen tres tipos de correspondencias (que determinan la ubicación de los bloques de memoria en la cache):

  - Directa: en un determinado bloque de cache.

  - Totalmente asociativa: en cualquier bloque libre.

  - Asociativa por conjuntos: en un determinado conjunto (directa) y, dentro del conjunto, en cualquier bloque (asociatividad).

    Grado de asociatividad (*way*): número de bloques del conjunto. Ejemplo: *4 way associative*

    Ésta es la opción más utilizada hoy en día.

- A la hora de utilizar la memoria cache, los fallos se clasifican en tres tipos:

  - Carga: al utilizar las instrucciones/datos la primera vez (carga de la información).

    > bloques de datos grandes

  - Capacidad: todos los bloques de datos no entran en la cache.

    > caches más grandes

  - Correspondencia: hay que reemplazar el bloque para cargar en esa posición un nuevo bloque (a pesar de que la cache no esté llena).

    > conjuntos más grandes

[en inglés, “tres C-s”: *compulsory*, *capacity* y *conflict*]

- Además de la información, la cache tiene un **directorio** para realizar la búsqueda de los bloques. Contenido del directorio:

  [ocupado - modificado - LRU - *tag* o marca de dirección]

  tag: información necesaria para encontrar un bloque de MP en la cache, en función de la correspondencia

- Dos opciones en caso de escritura:

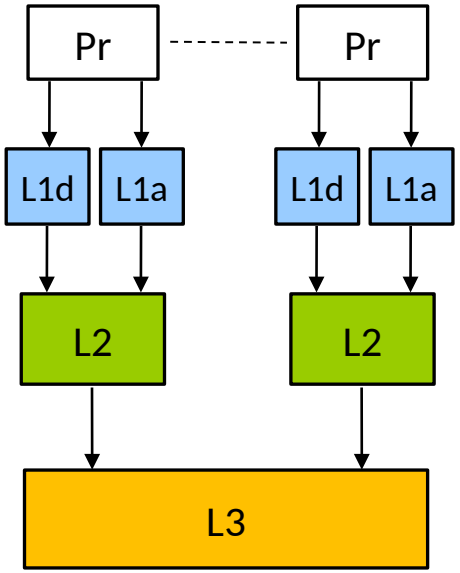
  WT: se escribe siempre en MP (palabra); las copias de la información son coherentes (actualizando o anulando la MC).

  WB: se escribe siempre en MC. Posteriormente, al reemplazar un bloque modificado, se actualiza dicho bloque (entero) en MP.

- Se pueden realizar diferentes optimizaciones (*prefetch*, buffers de escritura, compilador…). Objetivo: minimizar el tiempo de ejecución de los programas.

- Hoy en día los procesadores tienen más de un núcleo de ejecución (multicore).

  La memoria cache se organiza en tres niveles. Normalmente, L1 y L2 son caches privadas en cada núcleo, mientras que la cache L3, mucho más grande, suele ser compartida por todos los núcleos (puede estar dentro o fuera del chip).



> Gama alta: IBM **Power9** (24 *core*, 8.000 millones transistores, bloque = 128 bytes)

L1: 32 kB + 32 kB (por núcleo) 8 *way*, pseudo-LRU WT, *no write-allocate*
L2: 512 kB (2 núcleos), inclusiva 8 *way*, LRU WB, *write allocate*
L3: 120 MB (compartida) 20 *way*, "*sophisticated replacement policy*"

> Gama media: Intel **i7-7700** (Kaby Lake, ~2.000 millones trans., 4 *core*, Bl = 64 bytes)

L1: 32 kB + 32 kB (por núcleo) 8 *way*, TreePLRU WB
L2: 256 kB (por núcleo), no inclusiva 4 *way*, QLRU WB
L3: 8 MB (compartida), inclusiva 16 *way*, *Adaptive* WB

> Móviles: ARM **Cortex A8** (bloque = 64 byte)

L1: separada, 16 edo 32 kB 4 *way*, random WB, *no write allocate*
L2: unificada: 0, 128 kB… 1 MB 8 *way*

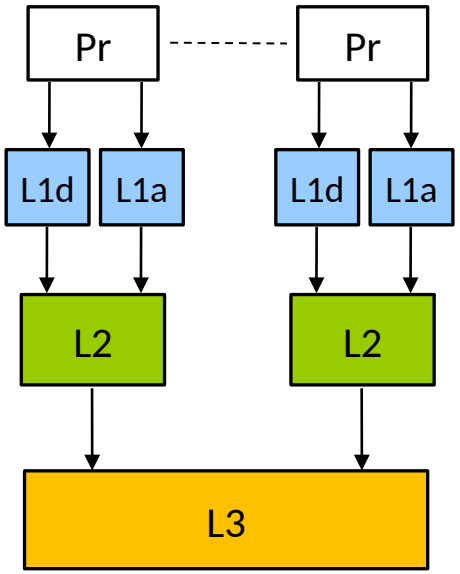
Comentario: la tecnología evoluciona rápido; esos datos son sólo una referencia

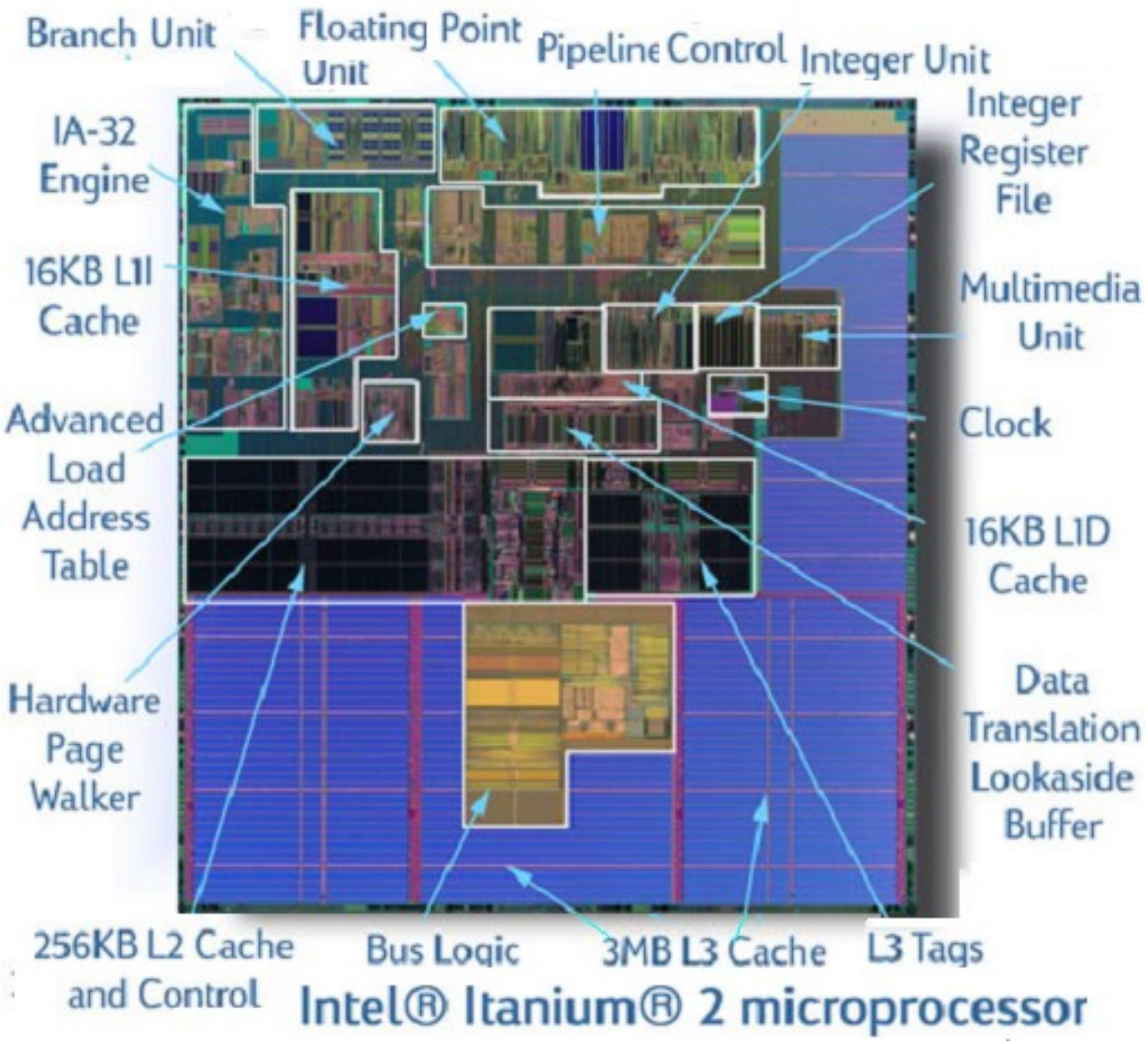
- Hemos analizado la estructura y uso general de la memoria cache, pero hay **más técnicas** para conseguir **aumentar el rendimiento**, por ejemplo, para mejorar el tiempo de acceso.

- Además, surgen **nuevos problemas** en procesadores con más de un *core*.

  Una variable puede estar en varias caches L1. ¿Qué ocurre si se modifica en una de ellas? ¿Cómo sabrá el resto de núcleos que esa variable se ha modificado?

  Este problema de **coherencia de datos** se soluciona mediante hardware especial (***snoopy***). Se estudiará en la asignatura **PAR**.




Branch Unit
Floating Point Unit
Pipeline Control
Integer Unit
IA-32 Engine
Integer Register File
16KB L1I Cache
Multimedia Unit
Advanced Load Address Table
Clock
16KB L1D Cache
Hardware Page Walker
Data Translation Lookaside Buffer
256KB L2 Cache and Control
Bus Logic
3MB L3 Cache
L3 Tags
Intel® Itanium® 2 microprocessor