Memoria:
Registros, caches y RAM

Arquitectura de Computadores – IIC2343

Yadran Eterovic S. (yadran@uc.cl)

Las CPUs siempre han sido más rápidas que las memorias

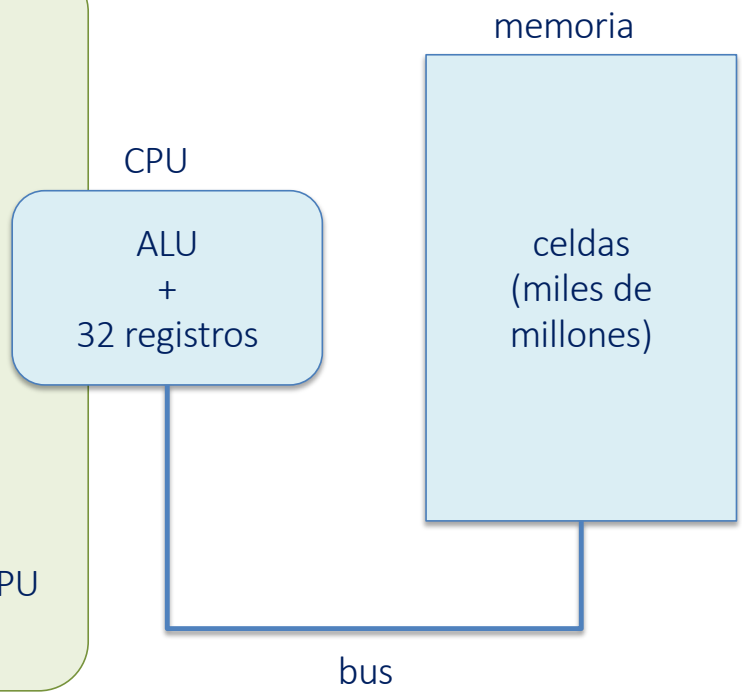
… y la diferencia ha ido aumentando con el tiempo

Parte de la solución es colocar la memoria en el chip de la CPU:

- ya que el acceso a la memoria a través del bus es muy lento

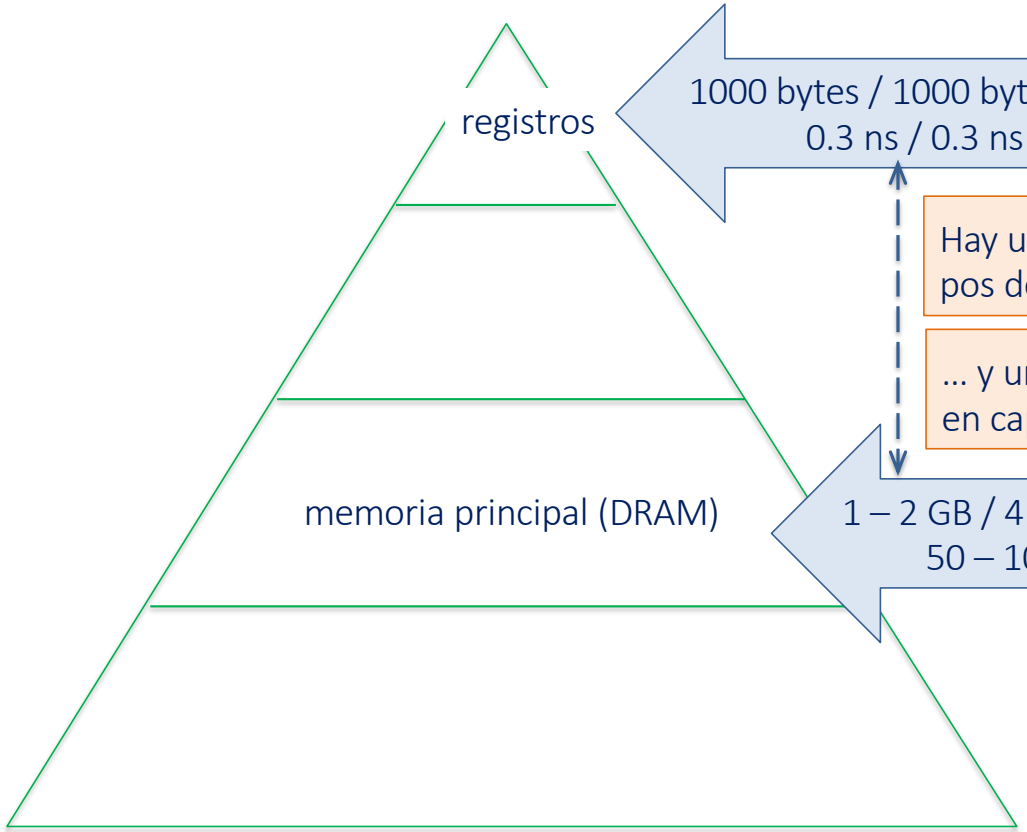
… pero esto significa CPUs más grandes:

- hay límites económicos y prácticos para el tamaño del chip de la CPU



Tamaños y tiempos de acceso típicos (≈ 2019) para los registros y la memoria principal en tres tipos de computadores:

teléfono celular / laptop / servidor



1000 bytes / 1000 bytes / 4000 bytes
0.3 ns / 0.3 ns / 0.2 ns

Hay un factor de 200 a 500 veces en tiempos de acceso a DRAM frente a registros

… y un factor de varios millones de veces en capacidad de almacenamiento

1 – 2 GB / 4 – 16 GB / 32 – 256 GB
50 – 100 ns (tres casos)

El **principio de localidad**, o cómo funcionan los programas en la práctica:

> **los programas accesan una porción relativamente pequeña de su *espacio de direcciones* en un momento cualquiera del tiempo**

Es decir, un programa no accesa todo su código o todos sus datos al mismo tiempo con igual probabilidad

Todas las direcciones de memoria, tanto de las instrucciones del programa como de los datos, a las que el programa podría hacer referencia durante su ejecución

Localidad temporal

Si un programa hace referencia a una instrucción o un dato, entonces probablemente hará referencia a esa misma instrucción o dato pronto

... p.ej., la ejecución de un *loop*

Dirección	Label	Instrucción/Dato
	CODE:	
0x00	start:	MOV CL, [var1]
0x01	while:	MOV AL,[res]
0x02		ADD AL,**[var2]**
0x03		MOV [res],AL
0x04		SUB CL,1
0x05		CMP CL,0
0x06		JNE while
	DATA:	
0x07	var1	3
0x08	**var2**	**2**
0x09	res	0

Localidad espacial

Si un programa hace referencia a un ítem, probablemente pronto se hará referencia a ítemes cuyas direcciones están (numéricamente) cerca

p.ej., ejecución secuencial de instrucciones

... acceso a los elementos de un arreglo

Dirección	Label	Instrucción/Dato
	CODE:	
0x00	start:	MOV SI, 0
0x01		MOV AX, 0
0x02		MOV BX, arreglo
0x03		MOV CL, [n]
0x04	while:	CMP SI, CX
0x05		JGE end
0x06		MOV DX, **[BX + SI]**
0x07		ADD AL, DL
0x08		INC SI
0x09		JMP while
0x0A	end:	DIV CL
0x0B		MOV [prom], AL
	DATA:	
0x0C	**arreglo**	**6**
0x0D		**7**
0x0E		**4**
0x0F		**5**
0x10		**3**
0x11	n	5
0x12	prom	0

El principio de localidad permite implementar una **jerarquía de memorias**:

- múltiples niveles de memoria con diferentes velocidades y tamaños

Las memorias más rápidas están más cerca del procesador, son más caras (por bit) que las memorias más lentas, y por lo tanto son más pequeñas:

- esto hace posible que la mayoría de los accesos a memoria sean rápidos

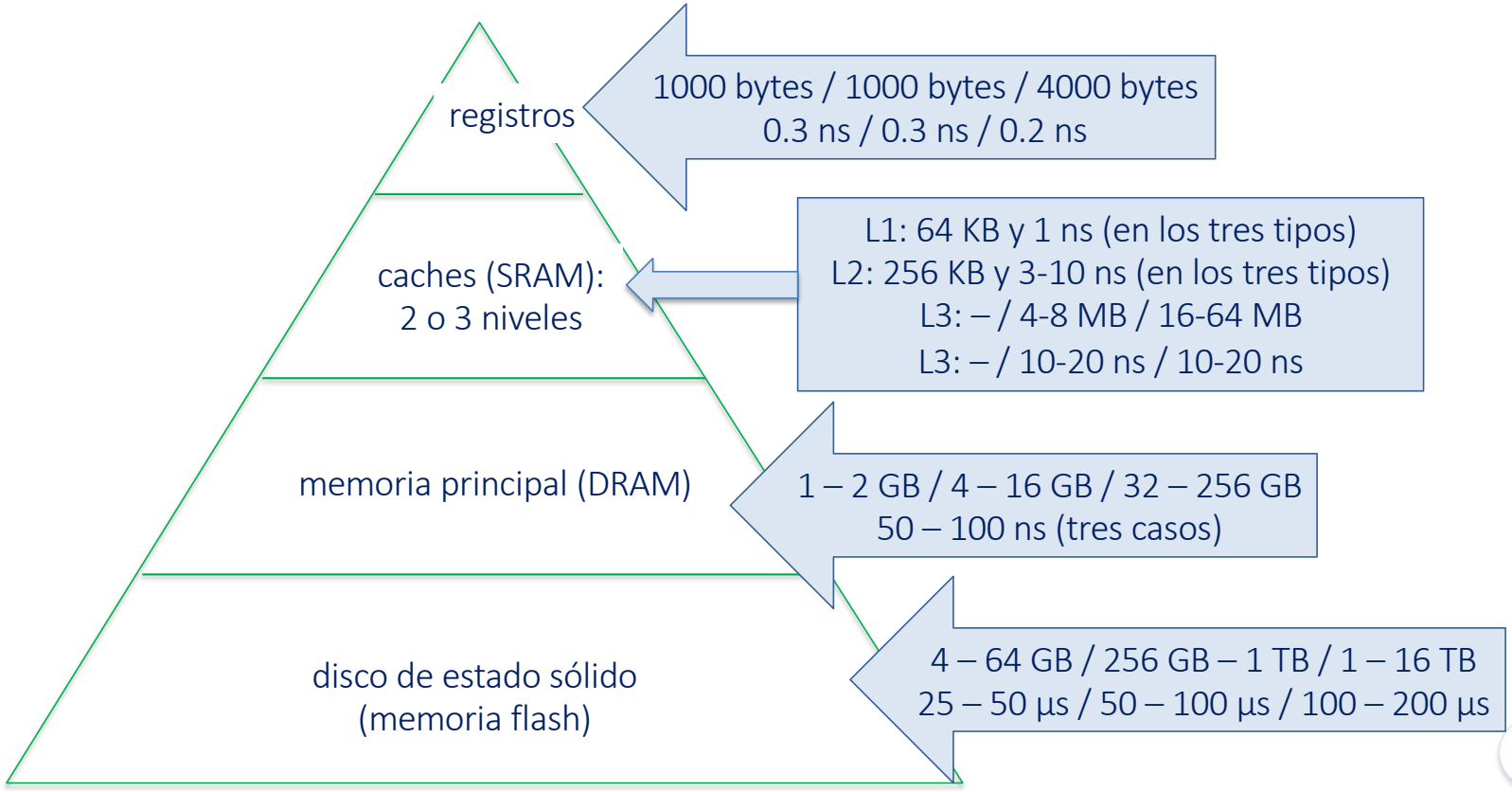
  … y al mismo tiempo tengamos una memoria grande

El propósito es ofrecer al programador (la ilusión de) tanta memoria como esté disponible en la tecnología más barata

… pero a la velocidad de acceso de la memoria más rápida

Tamaños y tiempos de acceso típicos (≈ 2019) para los registros y la memoria principal en tres tipos de computadores:

teléfono celular / laptop / servidor

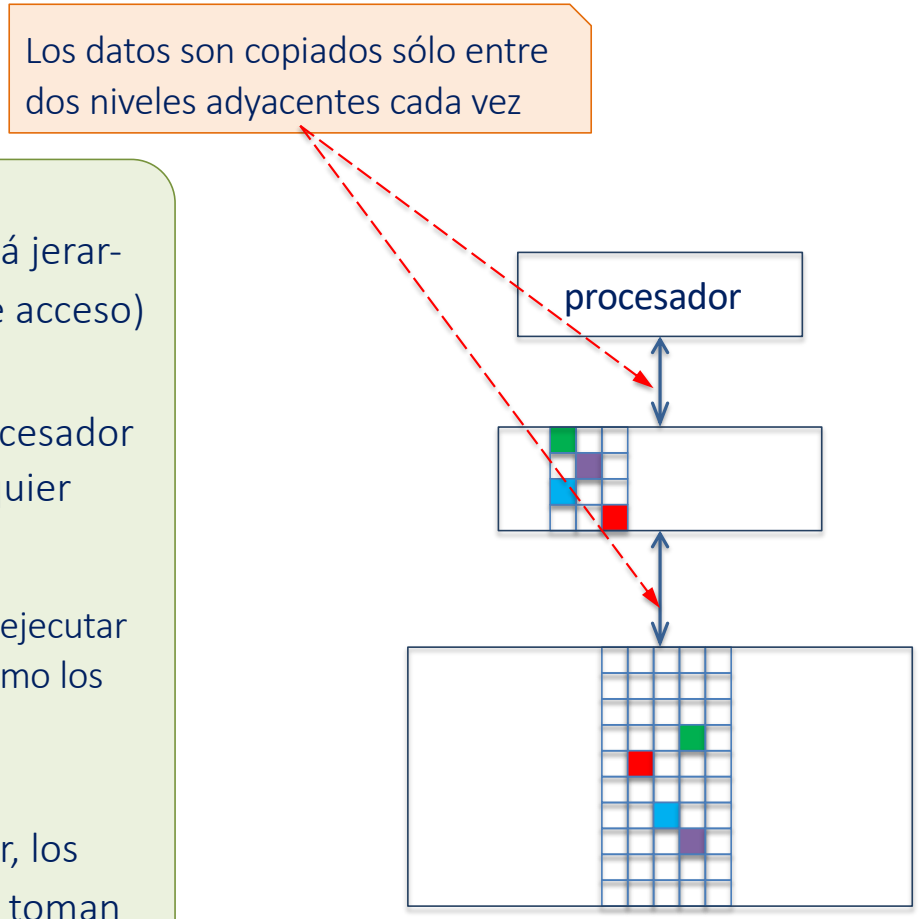


El **contenido** de las memorias también está jerarquizado (no sólo los tamaños y tiempos de acceso)

El contenido de un nivel más cerca del procesador **es un subconjunto** del contenido de cualquier nivel que está más lejos:

- … y el total del contenido necesario para ejecutar un programa —tanto las instrucciones como los datos— está en el nivel más lejano

A medida que nos alejamos del procesador, los accesos a los distintos niveles de memoria toman progresivamente más tiempo



SRAM (*static RAM*, tecnología de semiconductores):

- volátil (se borra si le quitamos la energía eléctrica)
- para caches
- 6 a 8 transistores por bit, igual tiempo de acceso a cualquier dato

0.5-2.5 ns
$500-$1,000

DRAM (*dynamic RAM*, tecnología de semiconductores):

- volátil
- para memoria principal (RAM), debe ser refrescada períodicamente
- un transistor + un circuito capacitor por bit, igual tiempo de acceso a cualquier dato

tiempo de acceso: 50-70 ns
costo por GByte: $3-$6

Flash (tecnología de semiconductores):

- no volátil (permanece, aún si le quitamos la energía eléctrica)
- para memoria secundaria en dispositivos móviles

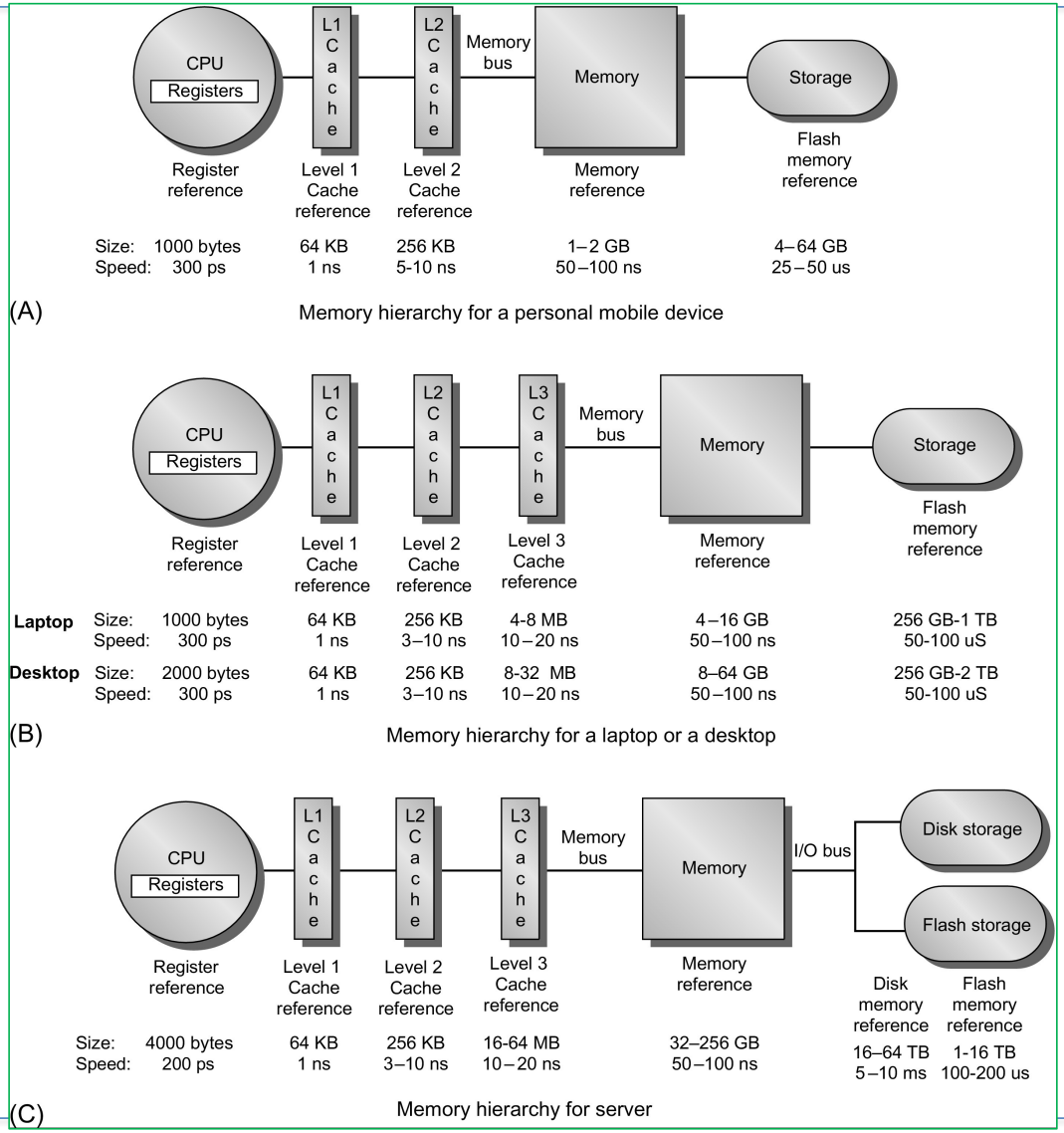
5,000-50,000 ns
$0.06-$0.12

Disco magnético:

- no volátil
- para memoria secundaria en servidores

5,000,000-20,000,000 ns
$0.01-$0.02

J. Hennessy, D. Patterson, "*Computer Architecture: A Quantitative Approach*" (6th ed.) Morgan Kaufman 2019

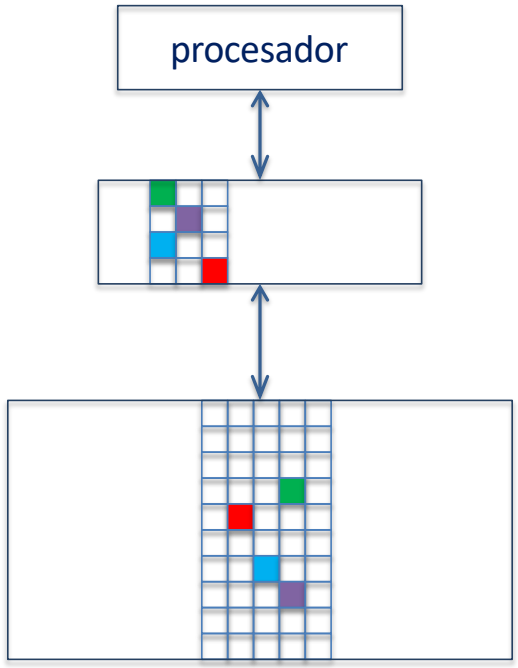


La unidad de información que se copia entre niveles adyacentes, en una jerarquía de dos niveles, se llama:

… **línea** (o **bloque**) —entre cache y memoria principal

… **página** —entre memoria principal y secundaria

Líneas (o bloques) y páginas consisten en múltiples bytes

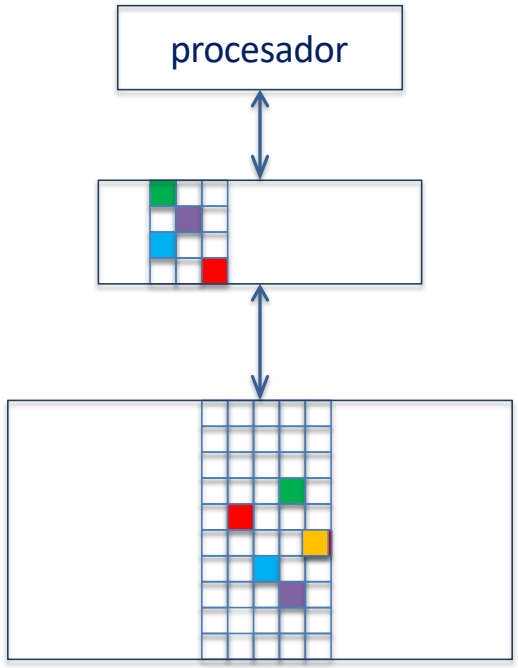


Si el dato pedido por el procesador está en el nivel más cercano → *hit*:

- p.ej., si el dato es ■

… de lo contrario → *miss*:

- p.ej., si el datos es ■
- en este caso, se va a un nivel más abajo en el jerarquía —más lejano del procesador— para traer la línea/página que contenga al dato pedido



Si el dato pedido por el procesador está en el nivel más cercano → *hit*:

- p.ej., si el dato es ■

… de lo contrario → *miss*:

- p.ej., si el datos es ■
- en este caso, se va a un nivel más abajo en el jerarquía —más lejano del procesador— para traer la línea/página que contenga al dato pedido

hit rate: fracción de los accesos a memoria que son encontrados en el nivel cercano —es una medida de desempeño de la jerarquía de memoria

miss rate: 1 – *hit rate*

Si el dato pedido por el procesador está en el nivel más cercano → ***hit***:

- p.ej., si el dato es

… de lo contrario → ***miss***:

- p.ej., si el datos es
- en este caso, se va a un nivel más abajo en el jerarquía —más lejano del procesador— para traer la línea/página que contenga al dato pedido

hit rate: fracción de los accesos a memoria que son encontrados en el nivel cercano —es una medida de desempeño de la jerarquía de memoria

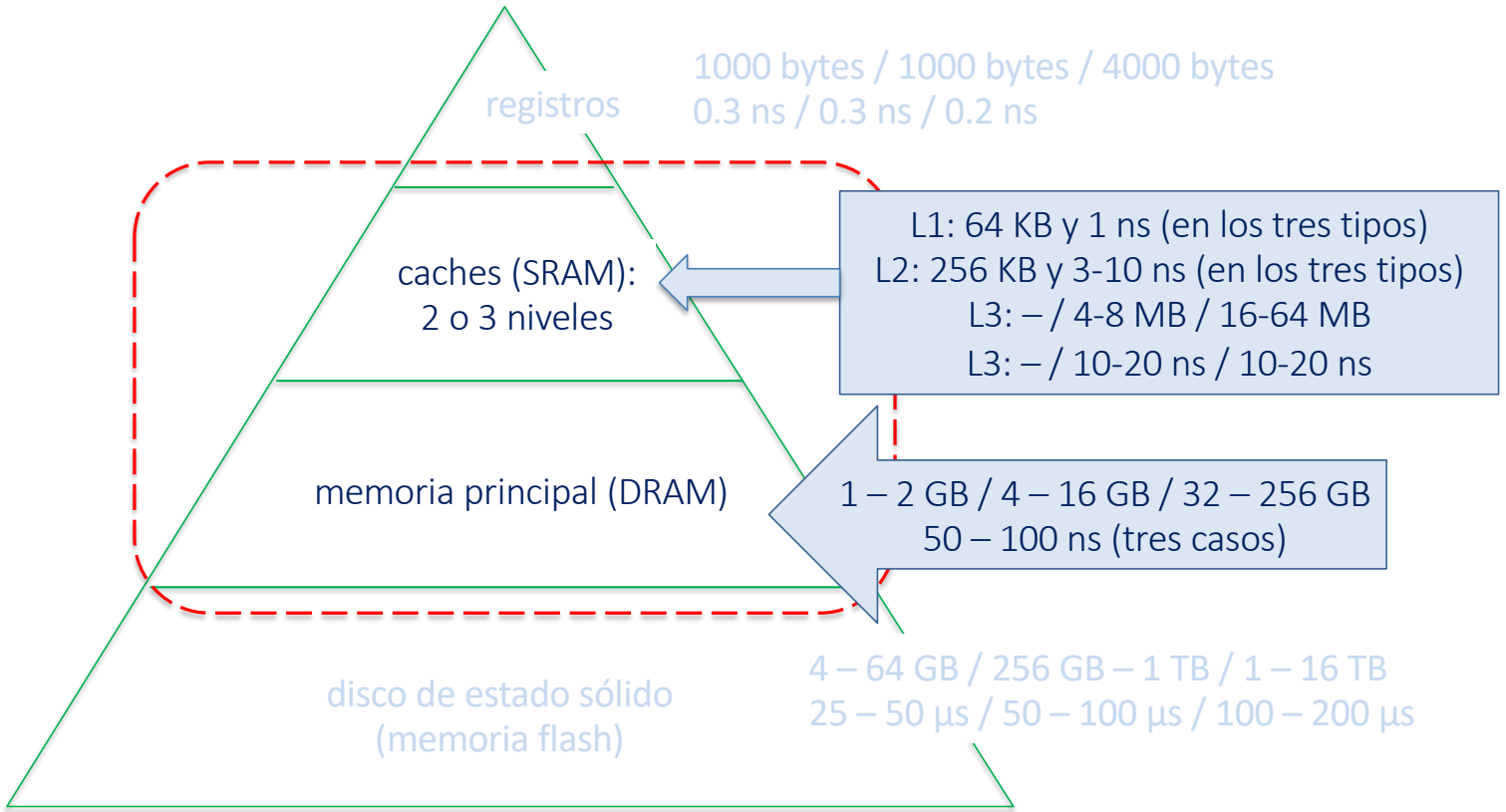
miss rate: 1 – *hit rate*

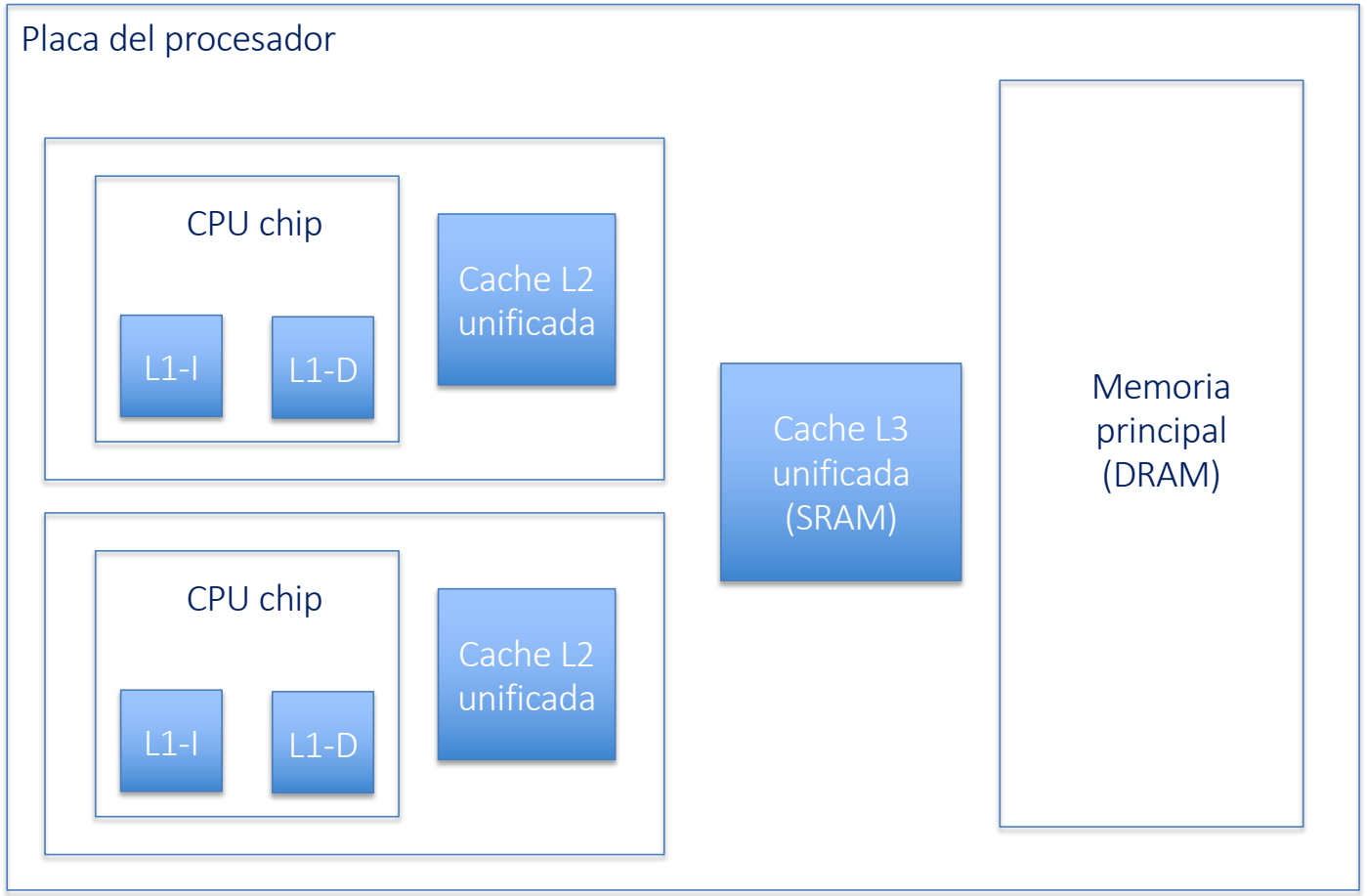
hit time: tiempo que toma el acceso al nivel más cercano de la jerarquía de memoria, incluyendo el tiempo necesario para determinar si el acceso es un *hit* o un *miss*

miss penalty: tiempo que toma reemplazar una línea o página en el nivel más cercano por la línea o página correspondiente del siguiente nivel, más el tiempo que toma entregar el dato específico al procesador

El *hit time* es mucho menor que el tiempo que toma tener acceso al siguiente nivel en la jerarquía (componente principal del *miss penalty*)

En primer lugar, vamos a estudiar **la interacción entre la cache y la memoria principal**; si bien actualmente los sistemas tienen dos o tres niveles de caches, vamos a suponer sólo uno

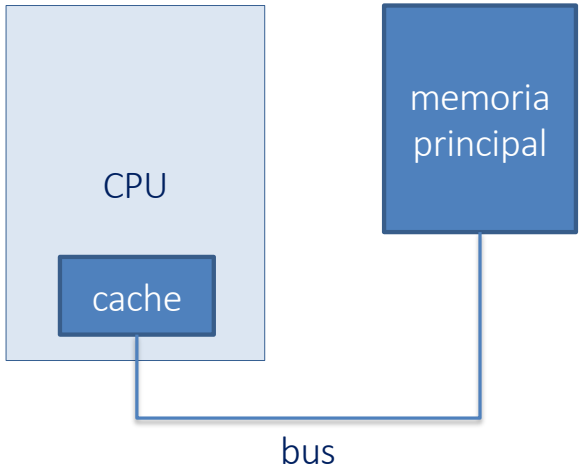



Placa del procesador
CPU chip
L1-I
L1-D
Cache L2 unificada
CPU chip
L1-I
L1-D
Cache L2 unificada
Cache L3 unificada (SRAM)
Memoria principal (DRAM)

Después de los registros, la **memoria cache** —o simplemente la **cache**— ocupa el siguiente nivel de la jerarquía

La *cache* se encuentra en la CPU

… **pero la CPU "no sabe" que existe**



La ubicación *lógica* de la cache es entre la CPU y la memoria

La memoria y la cache se dividen en **líneas** (o **bloques**; ver diap.12), aprovechando la *localidad espacial*:

- **las líneas de la memoria y las líneas de la cache tienen el mismo tamaño**, p.ej., 64 bytes
- cuando ocurre un *cache miss*, toda la línea de la memoria (los 64 bytes) es cargada (copiada) desde la memoria a la cache, no sólo el byte o la palabra a la que se hizo referencia —bajo el supuesto de localidad espacial
- p.ej., si la línea de la cache tiene 64 bytes, una referencia a la dirección 260 va a traer los 64 bytes (una línea) con direcciones 256 al 319

Para describir el funcionamiento de la cache, sólo es necesario entender su comunicación con la CPU y con la memoria principal —veamos

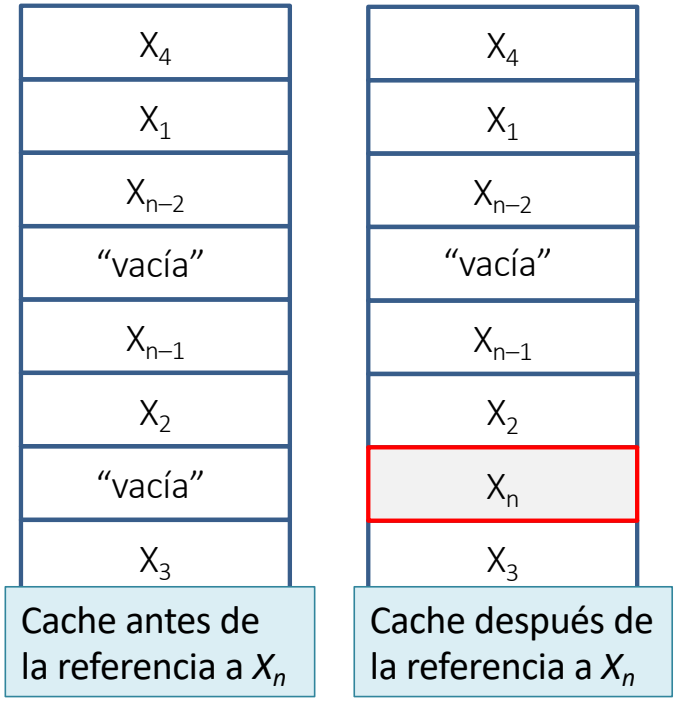
Supongamos, inicialmente, que las solicitudes del procesador son de un byte

… que las líneas contienen también un byte (sólo para simplificar)

… y que la cache tiene 8 líneas (de un byte c/u)*

Dos preguntas:

- ¿cómo sabemos si un dato está en la cache?
- y si está, ¿cómo lo encontramos?



*En la práctica, una cache podría tener 512 líneas de 64 bytes c/u

¿Cómo asignamos las direcciones de memoria
a las líneas de la cache?

Si cada palabra, según su dirección de memoria, puede ir a exactamente un lugar específico en la cache —***direct mapping***

… entonces es fácil encontrar la palabra si ella efectivamente está en la cache

… p.ej., asignamos la ubicación de la línea en la cache —o **índice** de la línea— en base a la dirección (de la línea) de la palabra en la memoria

Direct mapping

En este ej., la memoria tiene 32 bytes:

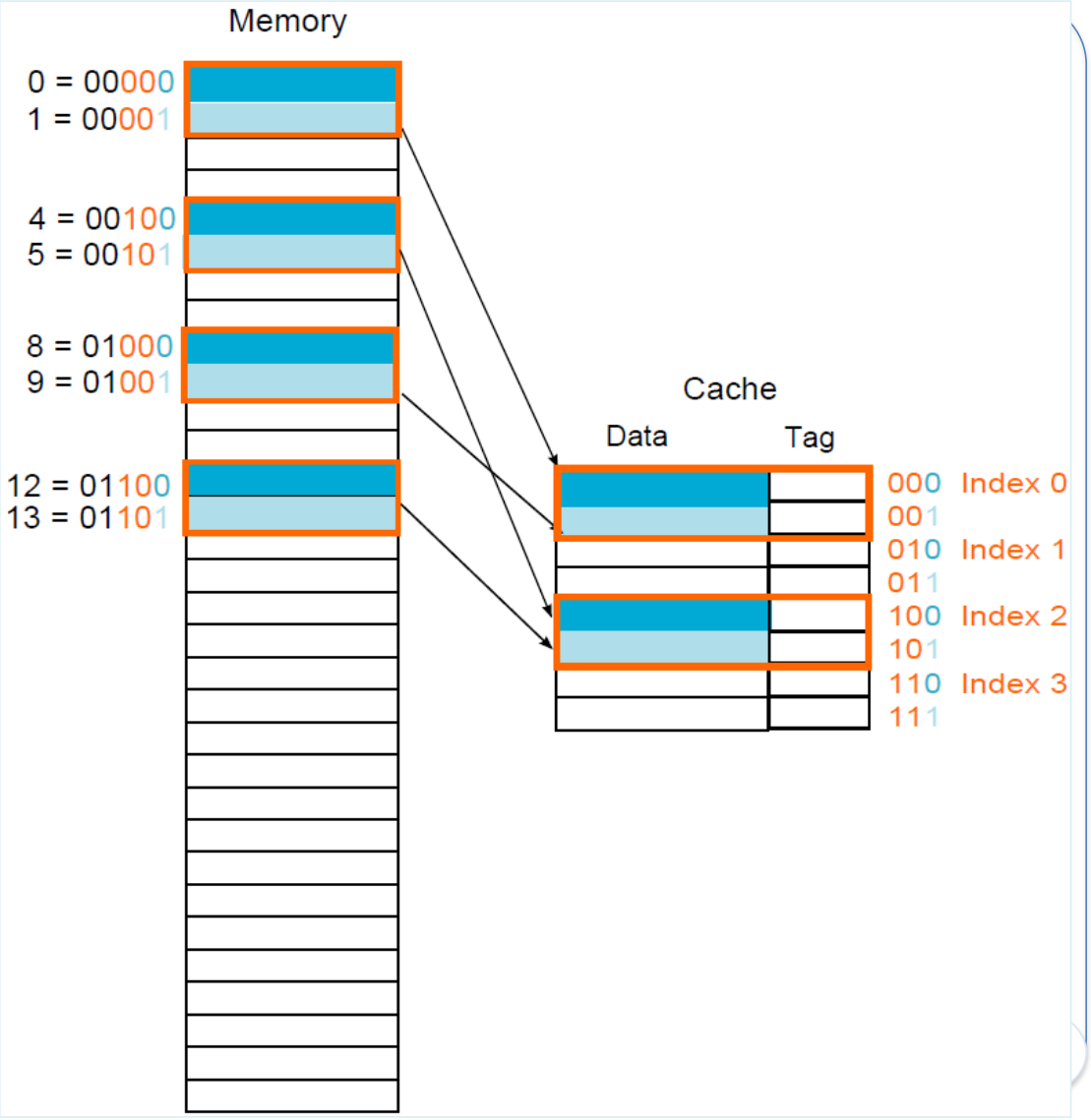
- c/u de las celdas con direcciones 0 (= 00000), 1, 2, …, 31 (= 11111) es un byte

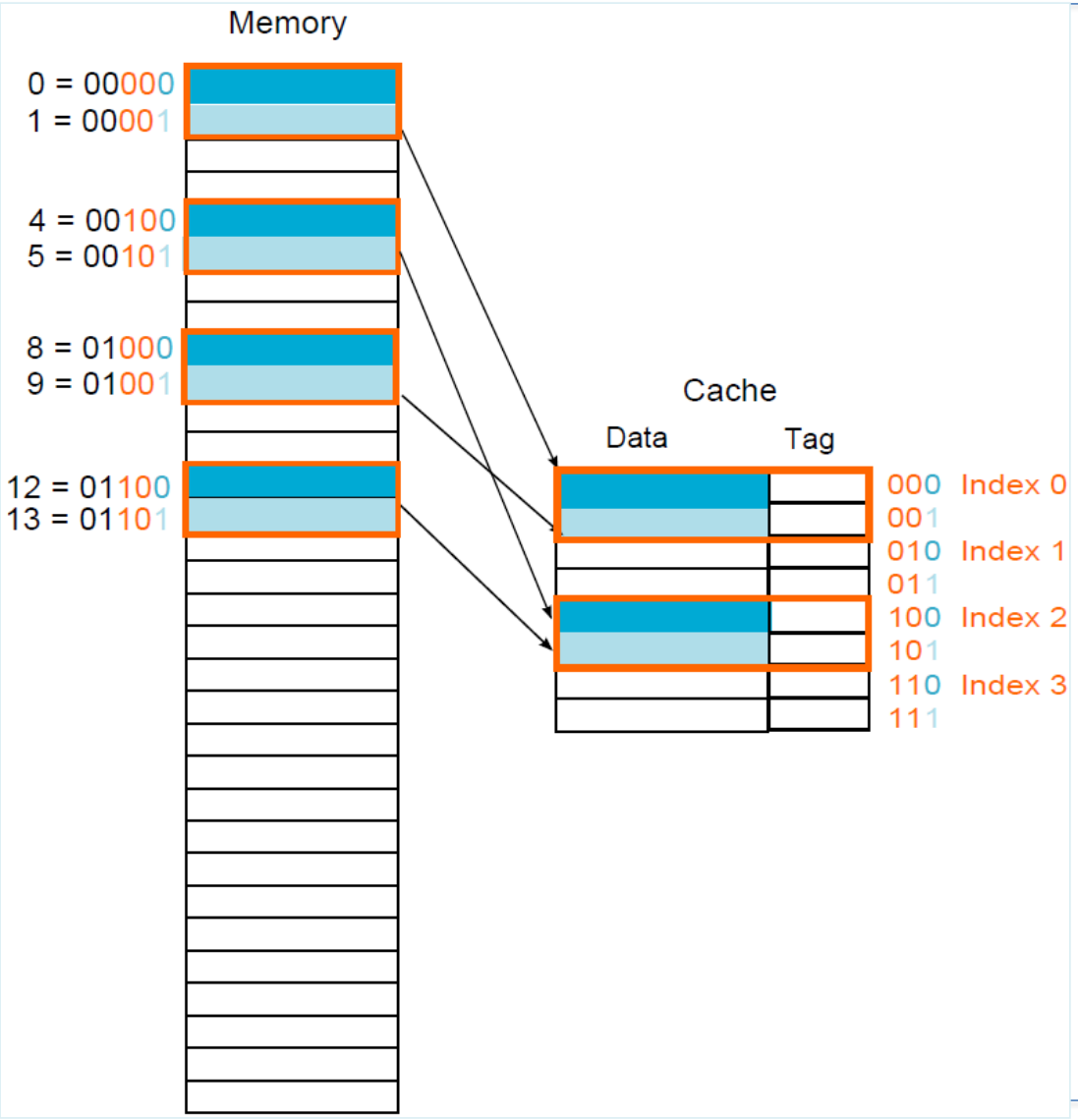
… y las **líneas de memoria** son de dos bytes c/u:

- los dos bytes en las direcciones 0 (= 00000) y 1 (= 00001) forman una línea

  … y los dos bytes en las direcciones 2 (= 00010) y 3 (= 00011) forman otra línea, etc.

- [… pero los dos bytes en las direcciones 1 (= 00001) y 2 (= 00010) **no forman** una línea]





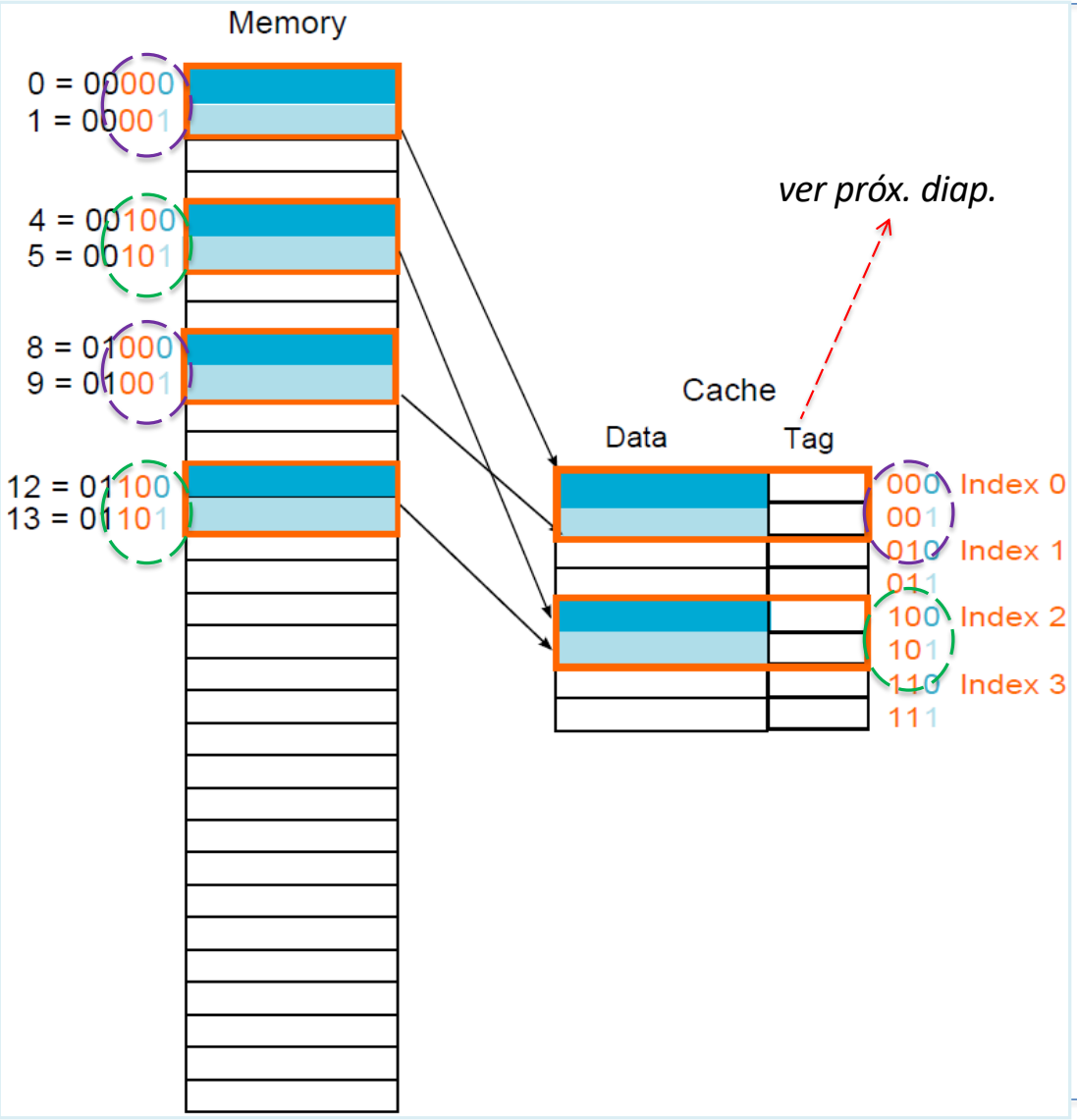
Direct mapping

La cache está organizada en **líneas** (**de cache**) del mismo tamaño que las líneas de la memoria (esto es siempre así), es decir, en este caso, de dos bytes c/u:

- en este ej., la cache tiene una capacidad de 4 líneas (de dos bytes c/u)

... y cada línea tiene un **índice**, un número correlativo correspondiente a la posición relativa de la línea dentro de la cache:

- las líneas de la cache del ej. tienen índices 0 (= 00), 1 (= 01), 2 (= 10), y 3 (= 11)



Direct mapping (“*mapeo*” *directo*):

índice = (*dirección de memoria*) modulo *N*

… en que *N* = número de bytes en la cache

En el ej. se ilustra el mapeo de algunas líneas de memoria a las líneas de la cache

El cálculo del índice se simplifica si *N* es una potencia de 2:

- p.ej., si *N* = 8, entonces todas las direcciones de memoria que terminan en …001 van a parar a la línea de la cache cuyo índice es **00**1
- … y todas las direcciones que terminan en …101 van a parar a la línea cuyo índice es **10**1

Direct mapping

Cada línea de la cache puede contener (el contenido de) varias líneas de memoria diferentes (sólo que no al mismo tiempo):

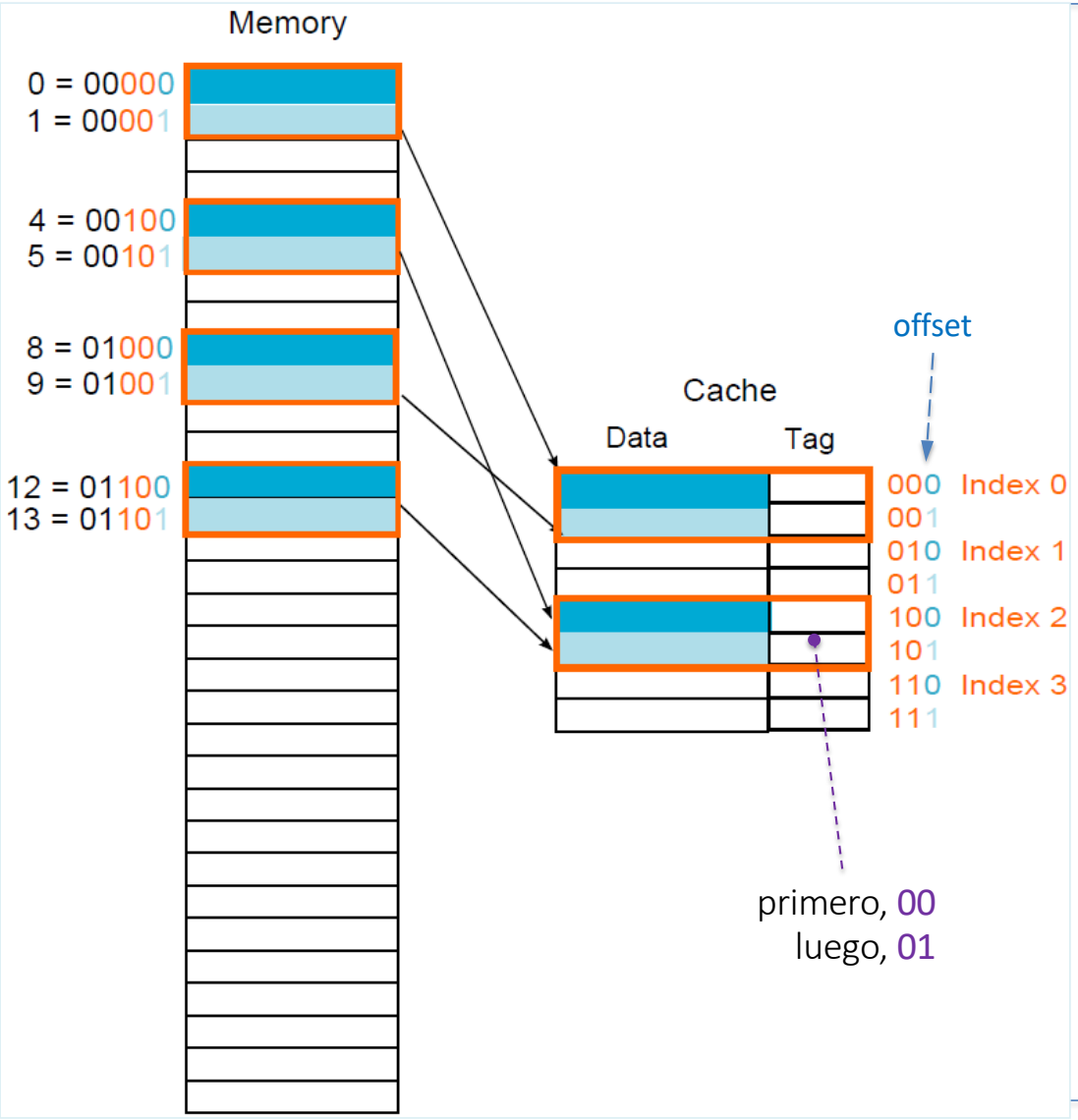
- todas aquellas líneas de la memoria para las cuales el valor calculado como

  (*dirección de uno de los bytes de la línea*) modulo *N*

  … es el mismo

… ⟹ necesitamos una manera de indicar cuál línea de la memoria está efectivamente en la cache:

- usamos un ***tag***, correspondiente al resto de los bits (los más significativos) de la dirección de memoria



Direct mapping

P.ej., si los bytes de la memoria que ocupan la tercera línea ($2_{10} = 10_2$) de la cache tienen las direcciones 00100 y 00101, entonces el *tag* correspondiente va a ser 00 :

- el bit menos significativo, 0 o 1, es el ***offset*** del byte dentro de la línea
- si más adelante los bytes que ocupan la misma tercera línea de la cache pasan a ser los bytes con direcciones 01100 y 01101, entonces el *tag* va a ser 01

Direct mapping

Finalmente, para saber si el contenido de la línea es válido, usamos un bit adicional —**bit de validación**:

- cuando el procesador recién parte o cuando un programa empieza a ejecutarse, la información en la cache no es válida

En las próx. diaps., vemos el comportamiento de nuestra cache de 4 líneas y dos bytes/línea, inicialmente "vacía" (es decir, contenido inicial inválido ⟹ bit *valid* = 0)

… cuando la secuencia de accesos a la memoria es a las direcciones de 12, 13, 14, 4, 12 y 0

inicial

índ. línea	offset	valid	tag	dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	0		
	1	0		
11	0	0		
	1	0		

Direct mapping

P.ej., cache de 4 líneas y dos bytes/línea:
acceso a direcciones de memoria 12, 13, 14, 4, 12 y 0

después del acceso a la dirección **12** = 01100

índ. línea	off set	valid	tag	dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	01	mem[12]
	1	1	01	mem[13]
11	0	0		
	1	0		

después del acceso a la dirección **13** = 01101

índ. línea	off set	valid	tag	dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	01	mem[12]
	1	1	01	mem[13]
11	0	0		
	1	0		

Direct mapping

después del acceso a la dirección **14** = 01110

índ. línea	off set	valid	tag	dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	01	mem[12]
	1	1	01	mem[13]
11	0	1	01	mem[14]
	1	1	01	mem[15]

después del acceso a la dirección **4** = 00100

índ. línea	off set	valid	tag	dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	00	mem[4]
	1	1	00	mem[5]
11	0	1	01	mem[14]
	1	1	01	mem[15]

Direct mapping

después del acceso a la dirección 12 = 01100

índ. línea	off set	valid	tag	dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	01	mem[12]
	1	1	01	mem[13]
11	0	1	01	mem[14]
	1	1	01	mem[15]

después del acceso a la dirección 0 = 00000

índ. línea	off set	valid	tag	dato
00	0	1	00	mem[0]
	1	1	00	mem[1]
01	0	0		
	1	0		
10	0	1	01	mem[12]
	1	1	01	mem[13]
11	0	1	01	mem[14]
	1	1	01	mem[15]

Las memorias están organizadas en palabras (de 4 u 8 bytes), en que la dirección de una palabra es la dirección de uno de sus bytes (el de dirección numéricamente menor):

- p.ej., las direcciones de las palabras de 4 bytes son 0, 4, 8, 12, 16, ...

... y las *líneas* de la memoria (para los efectos de la cache) son de varias palabras consecutivas (2, 4, 8, 16, …):

- ... y por lo tanto de varios bytes consecutivos

P.ej., en una cache de 64 líneas, con índices 0 a 63, en que cada línea tiene 16 bytes

… ¿cuál es el índice de la línea a la que pertenece la dirección de memoria 1200?

Respuesta: 11

Manejo de *hits* y *misses*				
tipo de cache	*hit*		*miss*	
instrucciones (L1-I)	todo sigue como si nada		la **unidad de control** de la cache debe *hacer algo* (diaps. 35 y 36)	
datos (L1-D)	en **LOADs**, todo sigue como si nada	en **STOREs**, hay que mantener la consistencia entre cache y memoria ⟹ protocolo write-through o write-back (diaps. 38, 39 y 40)	en **LOADs**, la unidad de control de la cache debe *hacer algo* (diap. 37)	en **STOREs**, hay que mantener la consistencia entre cache y memoria ⟹ política write-allocate o non-write-allocate (diap. 41)

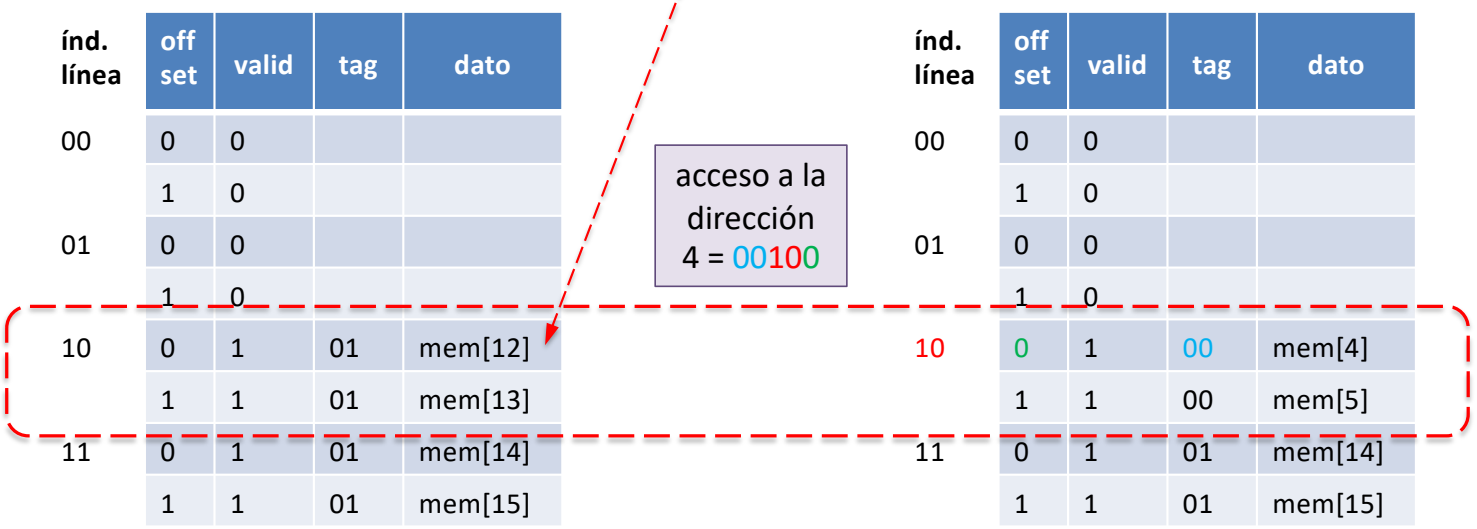
Cuando hay que "hacer algo", el manejo de la situación ocurre colaborativamente entre la **unidad de control** de la cache, la *Control Unit* del procesador, y otro controlador a cargo de transferir datos de la memoria a la cache

"*Hacer algo*": Cuando el acceso a la cache resulta en un *miss*, hay que traer desde la memoria la información que falta:

- si es **una instrucción**, que va a la cache de instrucciones (L1-I), entonces la nueva línea de instrucciones simplemente reemplaza a la línea que está en la cache (diaps. 35 y 36)
- si es **un dato**, que va a la cache de datos (L1-D), entonces la nueva línea de datos reemplaza a la línea que está en la cache

  … pero hay que tener cuidado con los datos en la línea que va a ser reemplazada

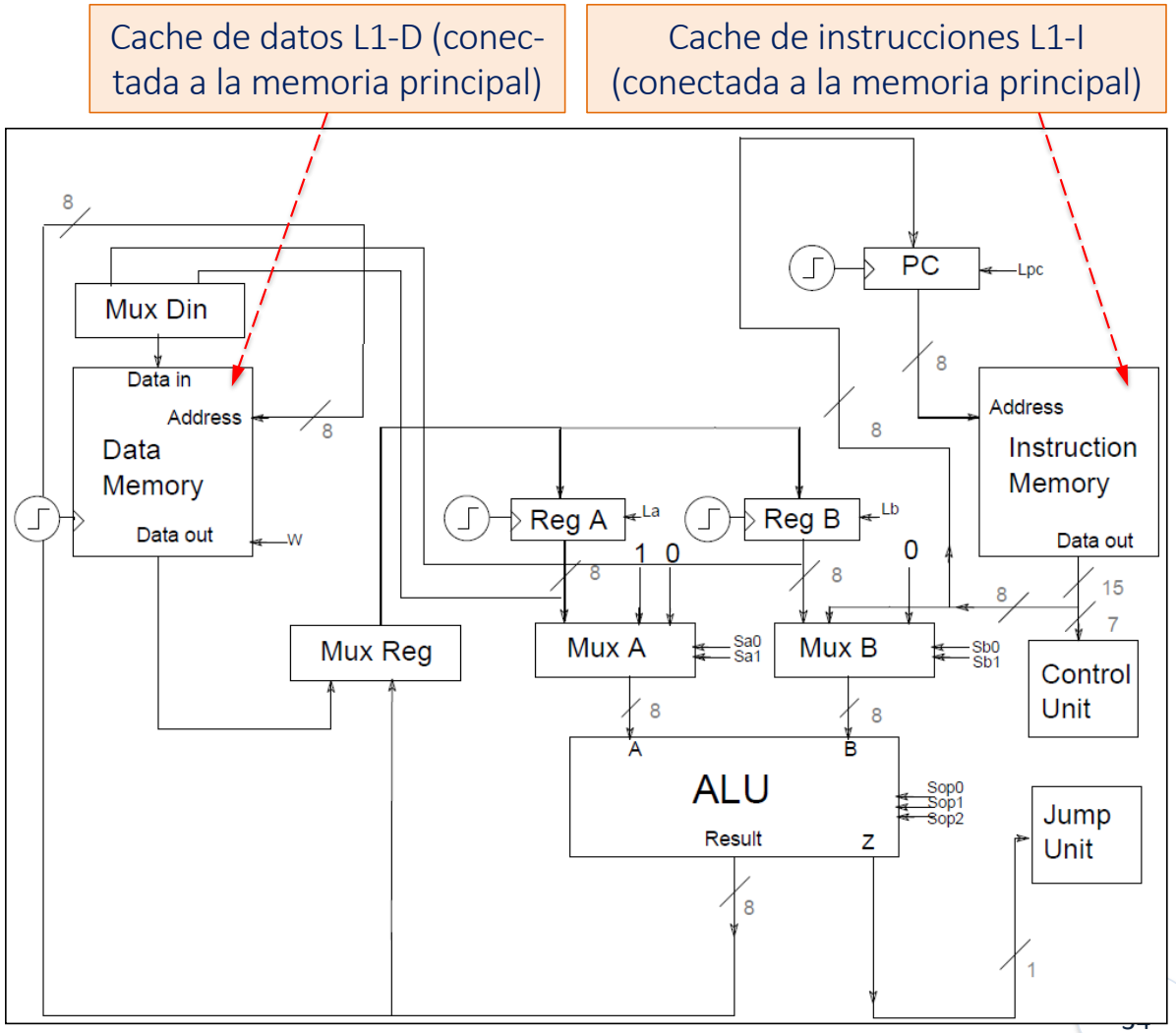
  … ya que pudieron haber sido modificados mientras estaban en la cache (diaps. 37 a 40)



índ. línea	off set	valid	tag	dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	01	mem[12]
	1	1	01	mem[13]
11	0	1	01	mem[14]
	1	1	01	mem[15]

índ. línea	off set	valid	tag	dato
00	0	0		
	1	0		
01	0	0		
	1	0		
10	0	1	00	mem[4]
	1	1	00	mem[5]
11	0	1	01	mem[14]
	1	1	01	mem[15]

Revisitando el computador básico de la primera parte del curso, podemos imaginar que la *Data Memory* de entonces corresponde a la cache *L1-D* de ahora (y similarmente para la *Instruction Memory* y la *L1-I*).

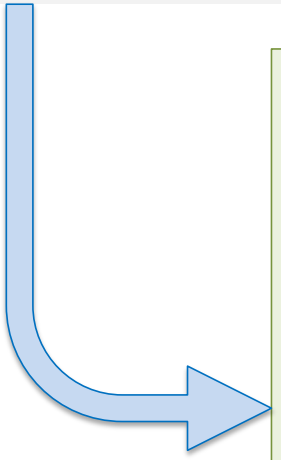


Manejo de un cache *miss* en la cache de instrucciones —la próxima instrucción que hay que ejecutar no está en la cache de instrucciones

… $\Longrightarrow$ **hay que ir a buscarla a la memoria**

Manejo de un cache *miss* en la cache de instrucciones —la próxima instrucción que hay que ejecutar no está en la cache de instrucciones

… ⟹ **hay que ir a buscarla a la memoria**



1. Enviar el valor original del *PC* (= *PC* actual – 4) a la *Address* de la memoria
2. Pedirle a la memoria que ejecute una lectura y esperar (varios ciclos) hasta que la lectura se complete —el procesador queda "en pausa"
3. Escribir en la cache:
   - – la línea completa que fue leída desde de la memoria
   - – los bits más significativos de la dirección (de la línea) en el *tag* de la línea
   - – un 1 en el bit de validez de la línea
4. Reiniciar la ejecución de la instrucción desde el comienzo ⟹ produce un nuevo *fetch* de la instrucción (que ahora sí va a estar en la cache: es un *hit*)

El manejo de la cache de datos, L1-D, es más complejo

Los datos en la cache pueden ser leídos —mediante *loads*:

- ***hit*** → el datos es simplemente entregado a la CPU (y todo sigue como si nada)
- ***miss*** → el manejo es similar al manejo de un *miss* de una instrucción

… y también pueden ser escritos —mediante *stores*— durante su permanencia en la cache:

- **hay que ver qué hacer tanto cuando el *store* es un *hit* como cuando es un *miss***

El manejo de la cache de datos, L1-D, es más complejo

Los datos en la cache pueden ser leídos —mediante *loads*:

- ***hit*** → el datos es simplemente entregado a la CPU (y todo sigue como si nada)
- ***miss*** → el manejo es similar al manejo de un *miss* de una instrucción

… y también pueden ser escritos —mediante *stores*— durante su permanencia en la cache:

- **hay que ver qué hacer tanto cuando el *store* es un *hit* como cuando es un *miss***

(diap. 41)

Al ejecutar un *store* en la cache de datos, **si es un *hit***, entonces el dato es escrito en la cache

Ahora bien, si el dato fuera escrito sólo en la cache y no en la memoria

… entonces la cache y la memoria se volverían **inconsistentes** (recordemos que el contenido de la cache es un subconjunto del contenido de la memoria)

Hay dos protocolos para manejar las **inconsistencias**:

- *write-through* (diap. 39)
- *write-back* (diap. 40)

Dos protocolos para manejar las inconsistencias cuando un *store* en la cache de datos es un *hit*:

write-through *write-back*

Write-through: siempre escribimos el dato tanto en la cache como en la memoria

Simple … pero el desempeño no es muy bueno:

- supongamos un ciclo por instrucción (sin *misses*)
- todo *store* escribe el dato en memoria (además de la cache), tomando, p.ej., 100 ciclos
- si el 10% de las instrucciones son stores, entonces el número de ciclos por instrucción ahora es 1 + 100 × 10% = 11

El desempeño se puede mejorar usando un *buffer* de escritura

Dos protocolos para manejar las inconsistencias cuando un *store* en la cache de datos es un *hit*:

write-through *write-back*

Write-through: siempre escribimos el dato tanto en la cache como en la memoria

Simple … pero el desempeño no es muy bueno:

- supongamos un ciclo por instrucción (sin *misses*)
- todo *store* escribe el dato en memoria (además de la cache), tomando, p.ej., 100 ciclos
- si el 10% de las instrucciones son *stores*, entonces el número de ciclos por instrucción ahora es 1 + 100 × 10% = 11

El desempeño se puede mejorar usando un *buffer* de escritura

Write-back: escribir el dato inicialmente sólo en la cache

… y sólo cuando la línea que contiene al dato modificado "salga" de la cache (porque es reemplazada por otra línea traída desde la memoria) se actualiza la línea correspondiente en la memoria

Mejora el desempeño, si los stores ocurren a menudo

… es más complejo de implementar que *write-through*:

- p.ej., necesita saber si la línea que va a ser reemplazada fue modificada mientras estuvo en la cache (*dirty bit*)

Si el *store* en la cache es un *miss*, entonces hay dos políticas posibles:

1) ***Write-allocate***: primero traemos la línea desde la memoria a la cache

   … y luego escribimos el dato en la cache y también en la memoria:

   - más común en caches *write-back*

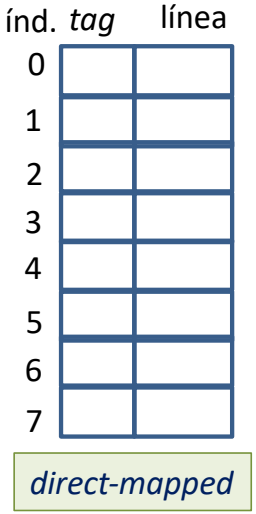
2) ***No-write-allocate***: sólo escribimos el dato en la memoria (y no lo traemos a la cache):

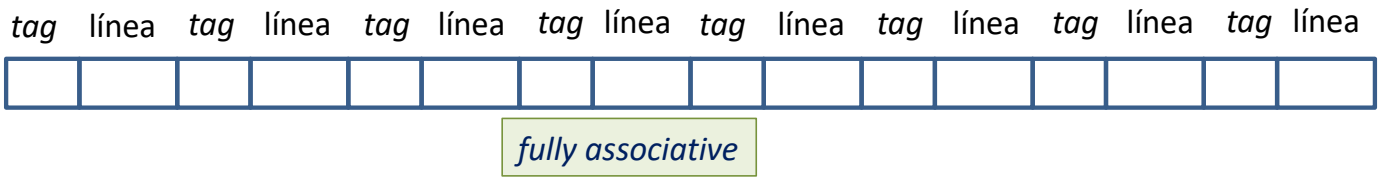
   - más común en caches *write-through*

Almacenamiento en la cache

En lugar de *direct-mapping*, el otro extremo en esquemas de asignación de líneas de memoria a líneas de cache es ***fully associative***:

- una línea de la memoria puede ir **a cualquier** línea de la cache
- → la cache puede contener simultáneamente varias líneas que, bajo *direct-mapping*, tendrían que competir por sólo un lugar en la cache
- para encontrar la línea en la cache, **hay que mirar** (los tags de) **todas las líneas**
- por desempeño, **la búsqueda se hace en paralelo**, encareciendo el costo del hardware necesario



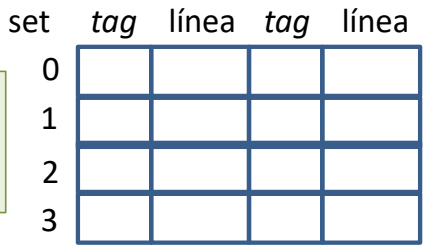


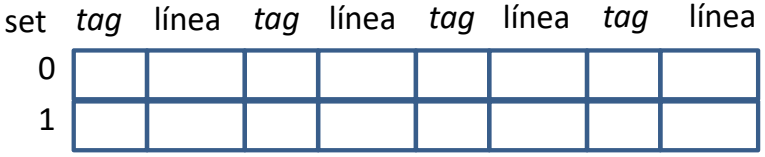
En una cache ***set associative***, hay un número fijo (> 1) de ubicaciones en donde puede estar una línea

… si hay *n* ubicaciones posibles para cada dirección de memoria —un conjunto, o ***set***, de *n* líneas— se habla de una cache ***n-way set associative***:

- en lugar de buscar la dirección en toda la cache, como en *fully-associative*, aquí se la busca sólo entre *n* líneas
- → la cache puede contener simultáneamente varias líneas de memoria que, bajo *direct-mapping*, tendrían que competir por sólo un lugar (una línea) en la cache
- *2-way* y hasta *8-way* funcionan bien en la práctica

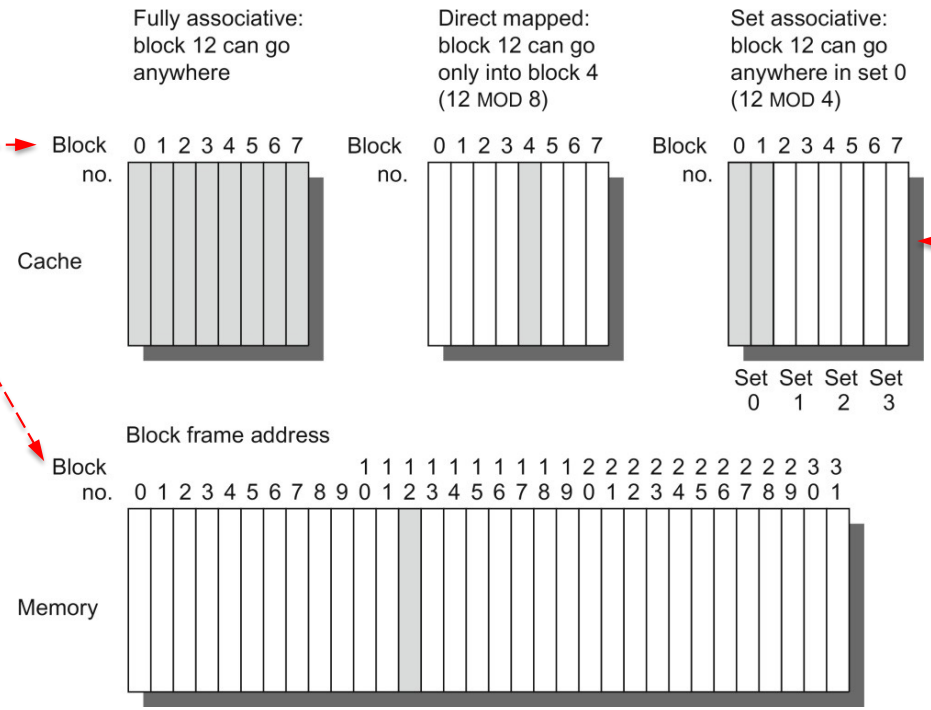
2-*way set associative*:
2 líneas de la memoria en cada set de la cahe





4-*way set associative*:
4 líneas de la memoria en cada set de la cache

block = línea



En ***n-way set associative***, una línea de la memoria puede ir en sólo un *set* de la cache; pero el *set* puede tener entre 2 (n = 2, como en la figura) y, p.ej., hasta 16 (n = 16) líneas

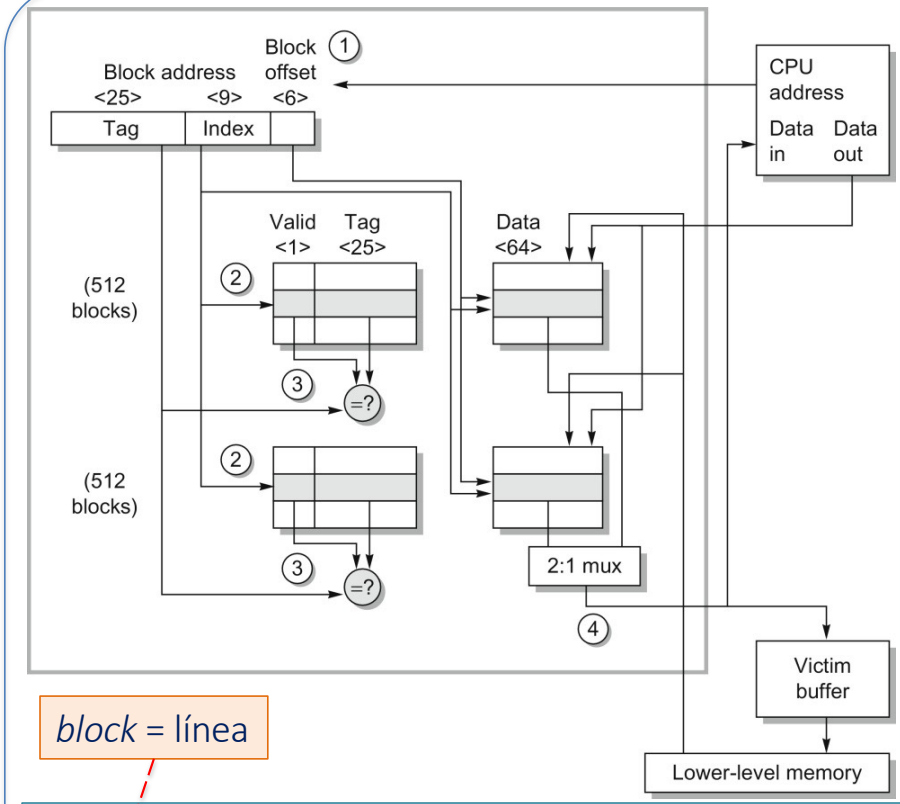
J. Hennessy, D. Patterson, "*Computer Architecture: A Quantitative Approach*" (6th ed.) Morgan Kaufman 2019

Figure B.2 This example cache has eight block frames and memory has 32 blocks. The three options for caches are shown left to right. In **fully associative**, block 12 from the lower level can go into any of the eight block frames of the cache. With **direct mapped**, block 12 can only be placed into block frame 4 (12 modulo 8). **Set associative**, which has some of both features, allows the block to be placed anywhere in set 0 (12 modulo 4). With two blocks per set, this means block 12 can be placed either in block 0 or in block 1 of the cache. Real caches contain thousands of block frames, and real memories contain millions of blocks. The set associative organization has four sets with two blocks per set, called two-way set associative. Assume that there is nothing in the cache and that the block address in question identifies lower-level block 12.

Posibles **políticas de reemplazo** de líneas en esquemas *n-way set associative* y *fully associative*:

- Bélády: se saca la línea que se usará más lejos en el futuro; óptimo no alcanzable en la práctica
- *First-in First-out* (FIFO): el primero en entrar es el primero en salir; simple, pero su desempeño no es muy bueno
- ***Least Recently Used*** (**LRU**): se saca la línea con mayor tiempo sin accesos; complejo, requiere timestamps; en general, el de mejor rendimiento —localidad temporal
- *Random*: muy rápido y con rendimiento algo inferior a LRU

J. Hennessy, D. Patterson, "*Computer Architecture: A Quantitative Approach*" (6th ed.) Morgan Kaufman 2019

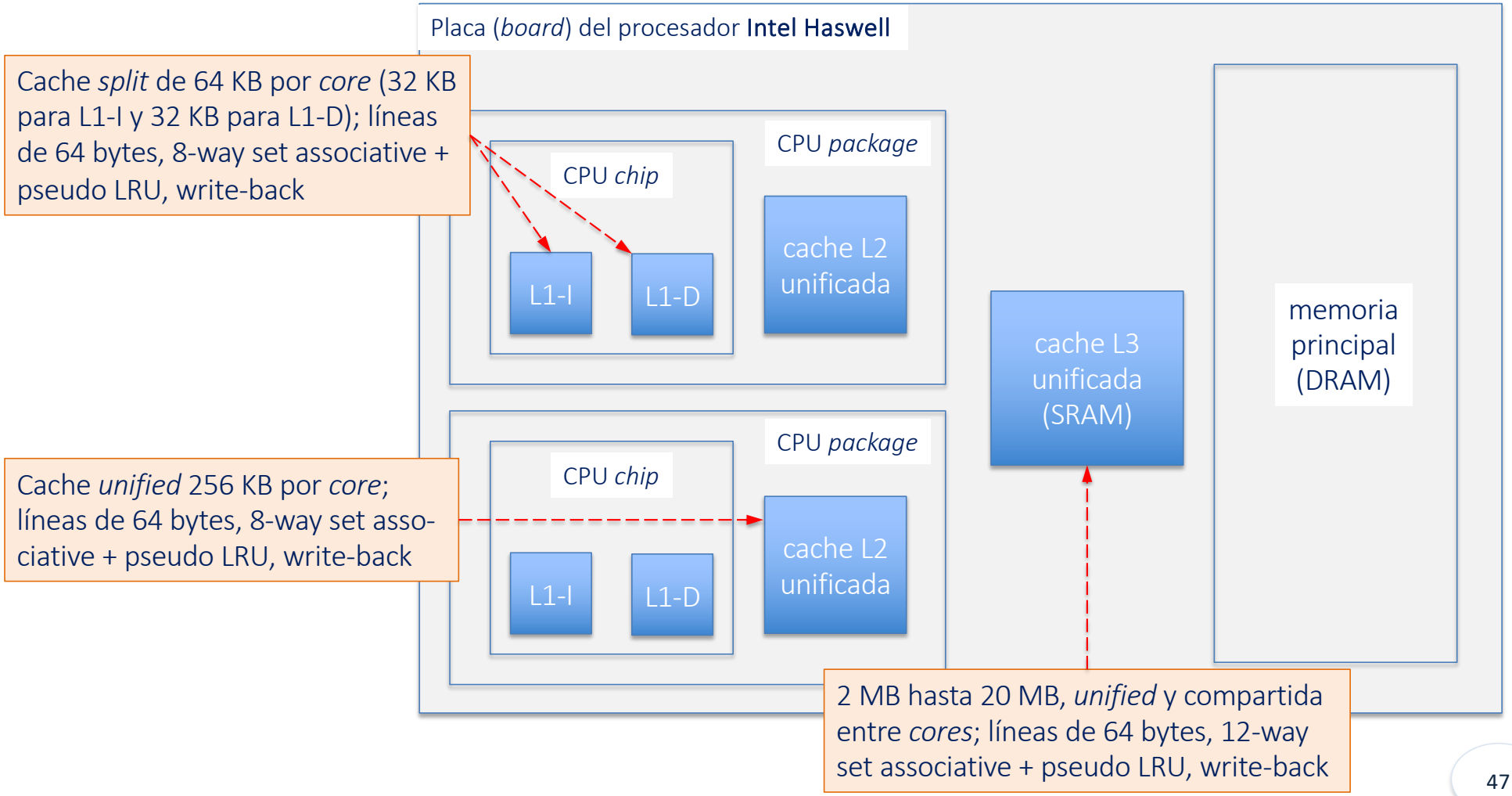


block = línea

*Además, usa reemplazo LRU, *write back* y *write allocate* en los *write misses*

**Diap. 80

Figure B.5 The organization of the data cache in the Opteron microprocessor. The 64 KiB cache is two-way set associative with 64-byte blocks. The 9-bit index selects among 512 sets*. The four steps of a read hit, shown as circled numbers in order of occurrence, label this organization. Three bits of the block offset join the index to supply the RAM address to select the proper 8 bytes. Thus, the cache holds two groups of 4096 64-bit words, with each group containing half of the 512 sets. Although not exercised in this example, the line from lower-level memory to the cache is used on a miss to load the cache. The size of address leaving the processor is 40 bits because it is a physical address and not a virtual address. Figure B.24 on page B-47** explains how the Opteron maps from virtual to physical for a cache access.


Placa (*board*) del procesador **Intel Haswell**
Cache *split* de 64 KB por *core* (32 KB para L1-I y 32 KB para L1-D); líneas de 64 bytes, 8-way set associative + pseudo LRU, write-back
CPU *package*
CPU *chip*
L1-I
L1-D
cache L2 unificada
CPU *package*
CPU *chip*
L1-I
L1-D
cache L2 unificada
Cache *unified* 256 KB por *core*; líneas de 64 bytes, 8-way set asso-ciative + pseudo LRU, write-back
cache L3 unificada (SRAM)
memoria principal (DRAM)
2 MB hasta 20 MB, *unified* y compartida entre *cores*; líneas de 64 bytes, 12-way set associative + pseudo LRU, write-back

Finalmente …

Si las líneas de la cache son de mayor capacidad (almacenan más bytes), entonces se aprovecha mejor la localidad espacial y se reduce el *miss rate*

… excepto si el tamaño de las líneas llega a ser una fracción importante del tamaño de la cache:

→ va a haber pocas líneas

→ mucha competencia por esas líneas

→ cada línea va a ser reemplazada frecuentemente, antes de que haya sido posible tener acceso a varias de sus palabras

Aumentar el tamaño de las líneas aumenta el costo de un *cache miss*:

- el tiempo que toma transferir una línea de la memoria a una línea de la cache depende del tamaño de la línea, es decir, de cuántos bytes son transferidos