Memoria:
caches y memoria virtual

Arquitectura de Computadores – IIC2343

2021-2 Yadran Eterovic

Las CPUs siempre han sido más rápidas que las memorias

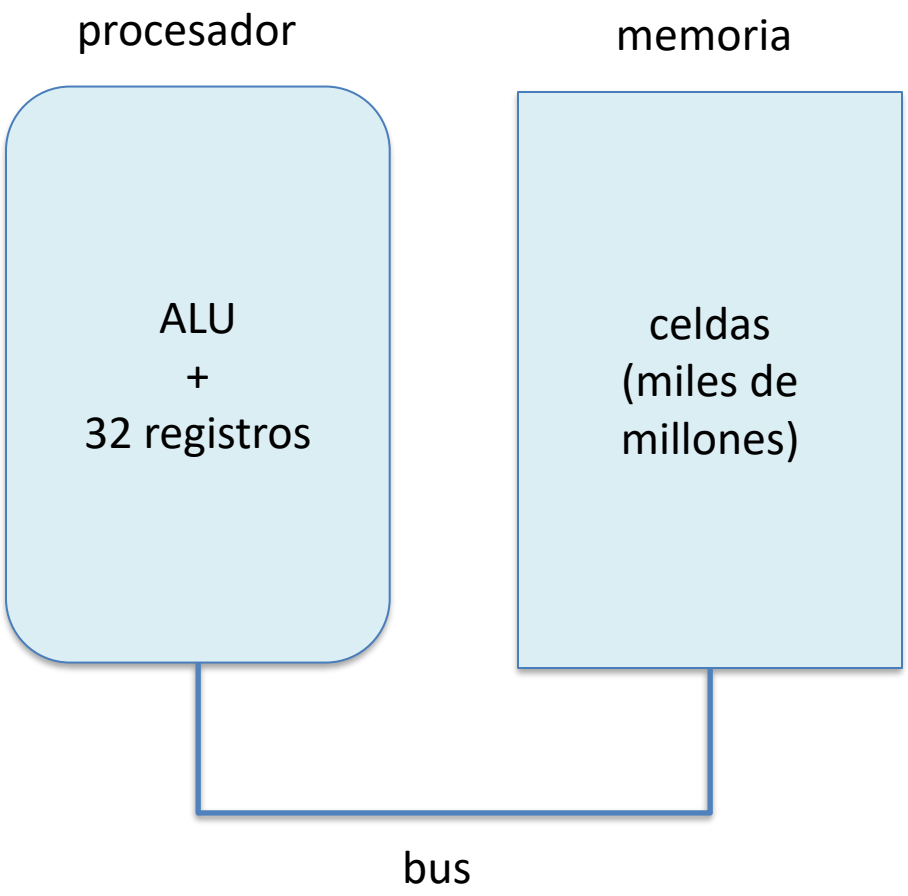
… y la diferencia sólo ha ido aumentando con el tiempo

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

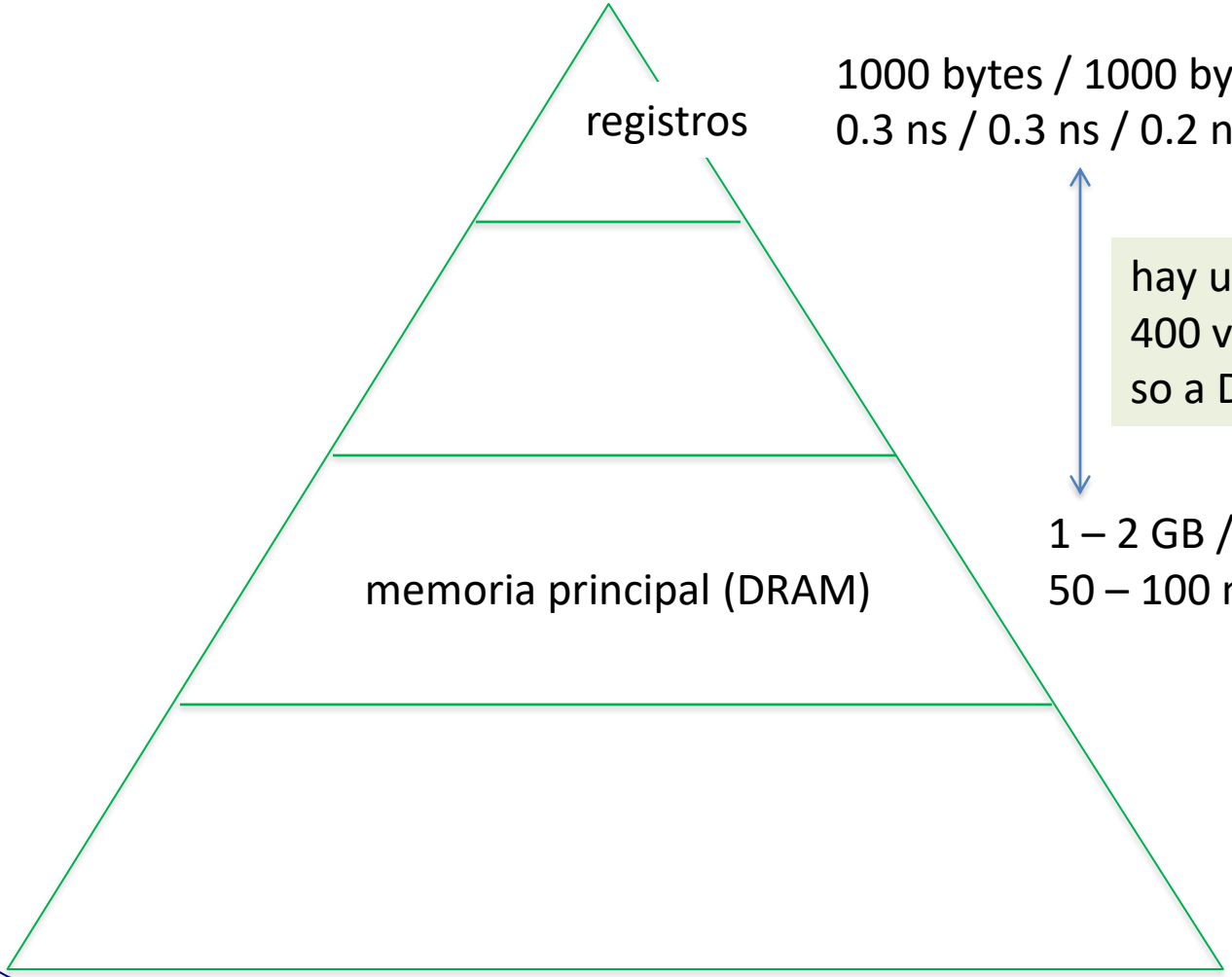
… pero esto significa CPUs más grandes:

- 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 entre 200 y 400 veces en tiempos de acceso a DRAM frente a registros

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

Los programadores siempre hemos querido tener cantidades ilimitadas de memoria rápida

Pero la elección pareciera ser entre poca memoria rápida

… o mucha memoria lenta

Excepto que hay técnicas para combinarlas de modo de obtener **simultáneamente** la velocidad de una memoria rápida

… y la capacidad de una memoria grande

… a un precio razonable:

- estudiaremos cómo crear la ilusión de tener cantidades ilimitadas de memoria rápida

El **principio de localidad** subyace a la forma en que 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

**espacio de direcciones*: 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

Esta localidad se da naturalmente en los programas:

- *loops*, cuyas instrucciones y datos son usados repetidamente —localidad temporal
- ejecución secuencial de instrucciones —localidad espacial
- accesos secuenciales a los elementos de un arreglo —localidad espacial
- ver ejs. en las próximas diaps.

Como vamos a ver, esto hace posible que la mayoría de los accesos a memoria sean rápidos

… y al mismo tiempo tengamos una memoria grande

Localidad temporal

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

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

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

Aprovechamos este principio de localidad para implementar una **jerarquía de memorias**:

- múltiples niveles de memoria con diferentes velocidades y tamaños

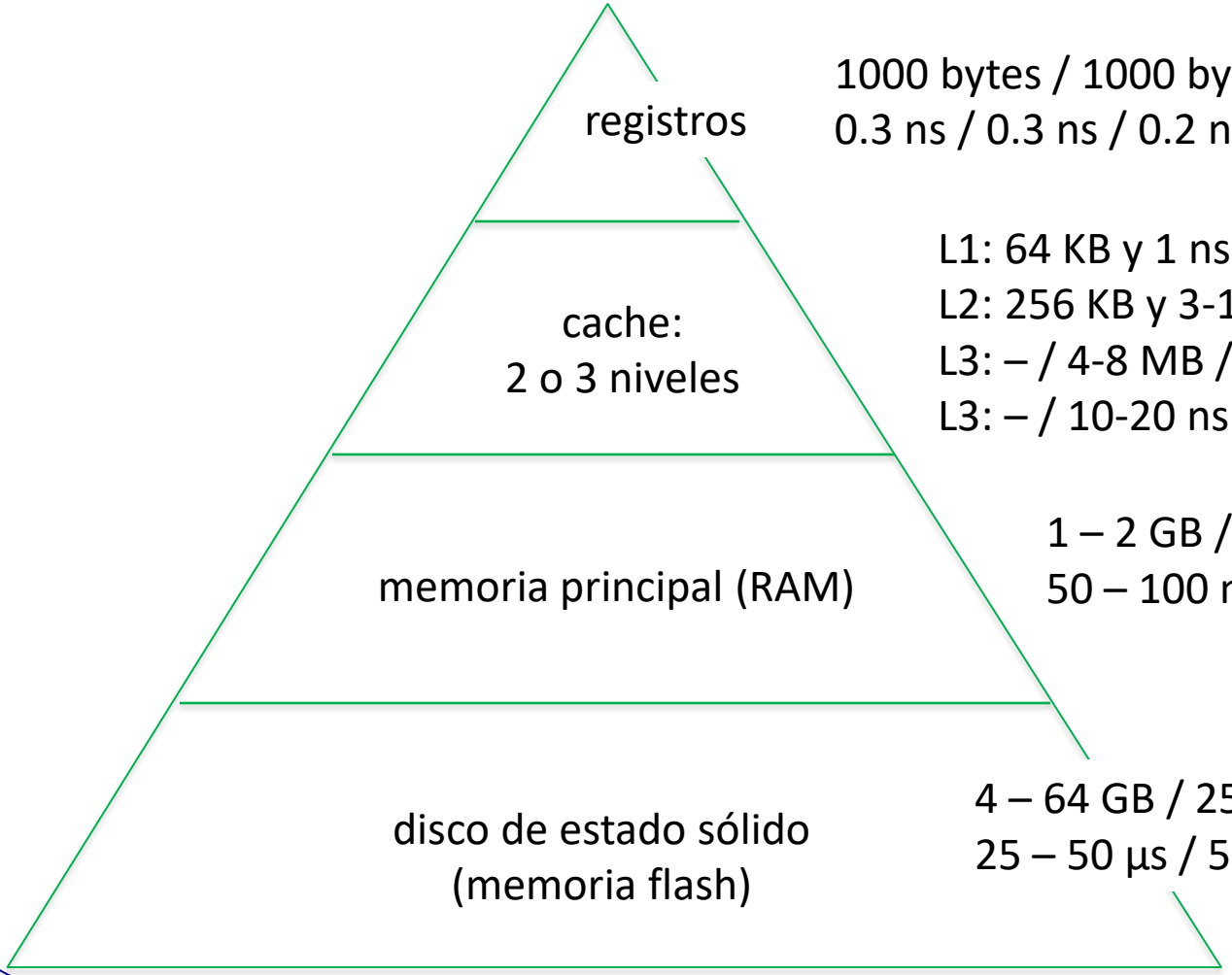
Las memorias más rápidas están más cerca del procesador, son más caras por bit que las memorias más lentas, y por lo tanto son más pequeñas

El propósito es ofrecer al programador 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 cuatro niveles de la jerarquía en tres tipos de computadores:

teléfono celular / laptop / servidor



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

L1: 64 KB y 1 ns (tres casos)
L2: 256 KB y 3-10 ns (tres casos)
L3: – / 4-8 MB / 16-64 MB
L3: – / 10-20 ns / 10-20 ns

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

4 – 64 GB / 256 GB – 1 TB / 1 – 16 TB
25 – 50 µs / 50 – 100 µs / 100 – 200 µs

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

Como vemos, 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
- para caches
- 6 a 8 transistores por bit, tiempo fijo de acceso a cualquier dato

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

- volátil
- para memoria principal (RAM), debe ser refrescada períodicamente
- más barata (un transistor + un capacitor) por bit, pero más lenta que SRAM

Flash (tecnología de semiconductores):

- no volátil
- para memoria secundaria en dispositivos móviles

Disco magnético:

- no volátil
- para memoria secundaria en computadores de escritorio y servidores

Los datos son copiados sólo entre dos niveles adyacentes cada vez:

- p.ej., entre cache y memoria principal, o entre memoria principal y disco
- el nivel más cercano al procesador es más pequeño y más rápido que el nivel más lejano

  … ya que usa tecnología más cara

La unidad mínima de información que puede estar presente o no en una jerarquía de dos niveles

… y que se copia entre niveles adyacentes es

… un **bloque** o una **línea** —entre cache y memoria principal

… y una **página** —entre memoria principal y disco

Líneas (o bloques) y páginas normalmente consisten en múltiples bytes

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

… de lo contrario → ***miss***:

- en caso de *miss*, 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 —una medida de desempeño de la jerarquía de memoria

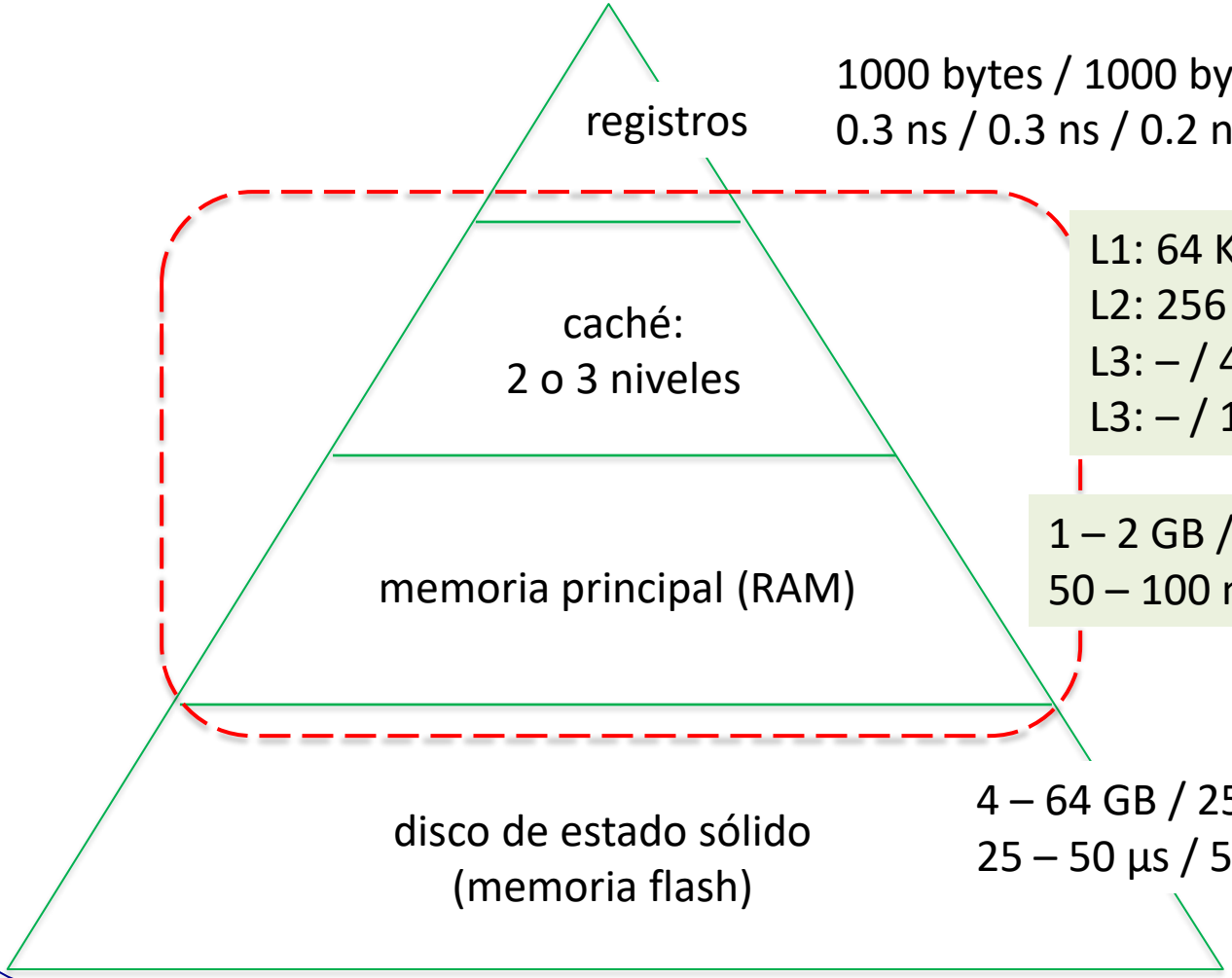
miss rate: 1 – *hit rate*

hit time: tiempo que toma tener 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/página en el nivel más cercano por la línea/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

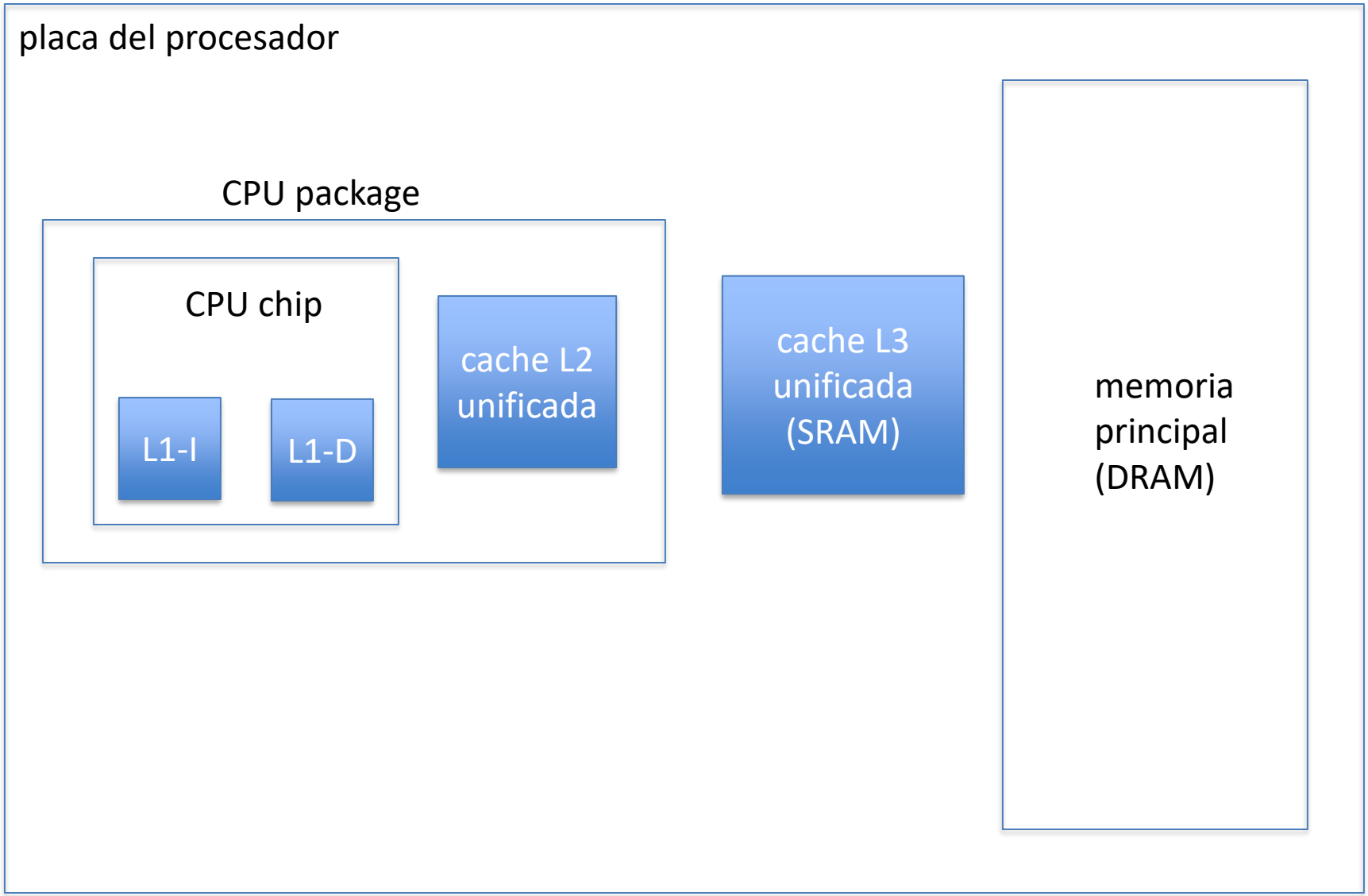


1000 bytes / 1000 bytes / 4000 bytes
0.3 ns / 0.3 ns / 0.2 ns

L1: 64 KB y 1 ns (tres casos)
L2: 256 KB y 3-10 ns (tres casos)
L3: – / 4-8 MB / 16-64 MB
L3: – / 10-20 ns / 10-20 ns

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

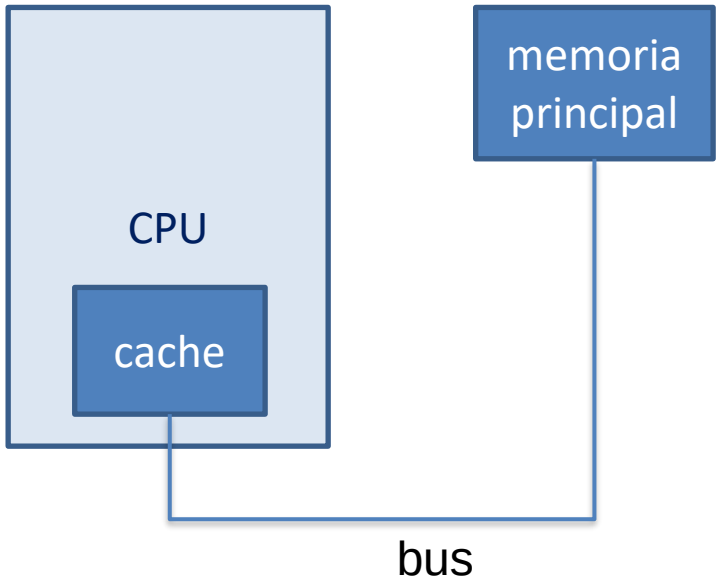
4 – 64 GB / 256 GB – 1 TB / 1 – 16 TB
25 – 50 µs / 50 – 100 µs / 100 – 200 µs


placa del procesador
CPU package
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 primer nivel de la jerarquía

La cache se encuentra en la CPU

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



La ubicación lógica de la cache: entre la CPU y la memoria

La memoria y la cache se dividen en líneas (o bloques) —aprovechando la localidad espacial:

- **las líneas de la memoria y las líneas de la cache tienen el mismo tamaño**
- cuando ocurre un *cache miss*, toda la línea de la memoria es cargada (copiada) desde la memoria a la cache, no solo 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

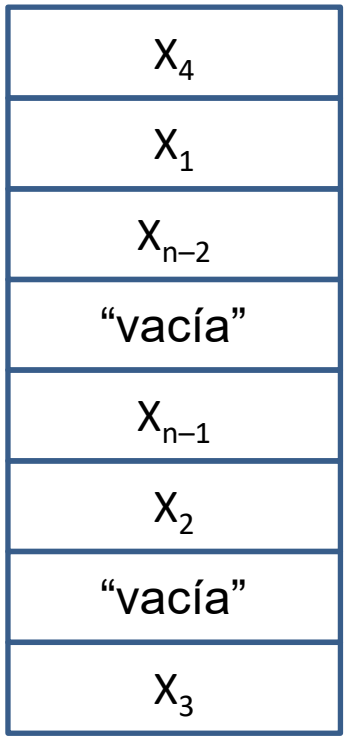
Supongamos, inicialmente, que las solicitudes del procesador son de una palabra (o de un byte)

… que las líneas contienen tam-bién una palabra (o un byte)

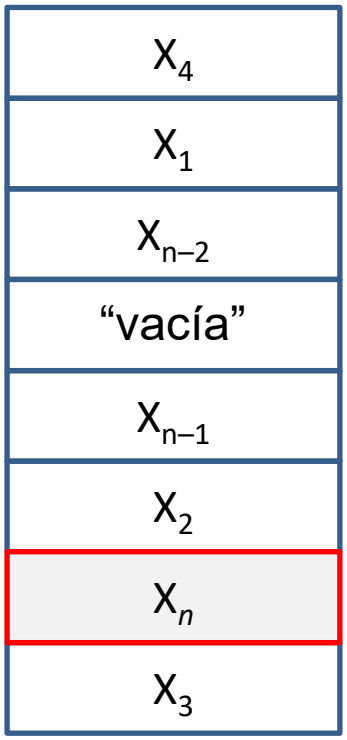
… y que la cache tiene 8 líneas

Dos preguntas:

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



Cache antes de la referencia a X_n

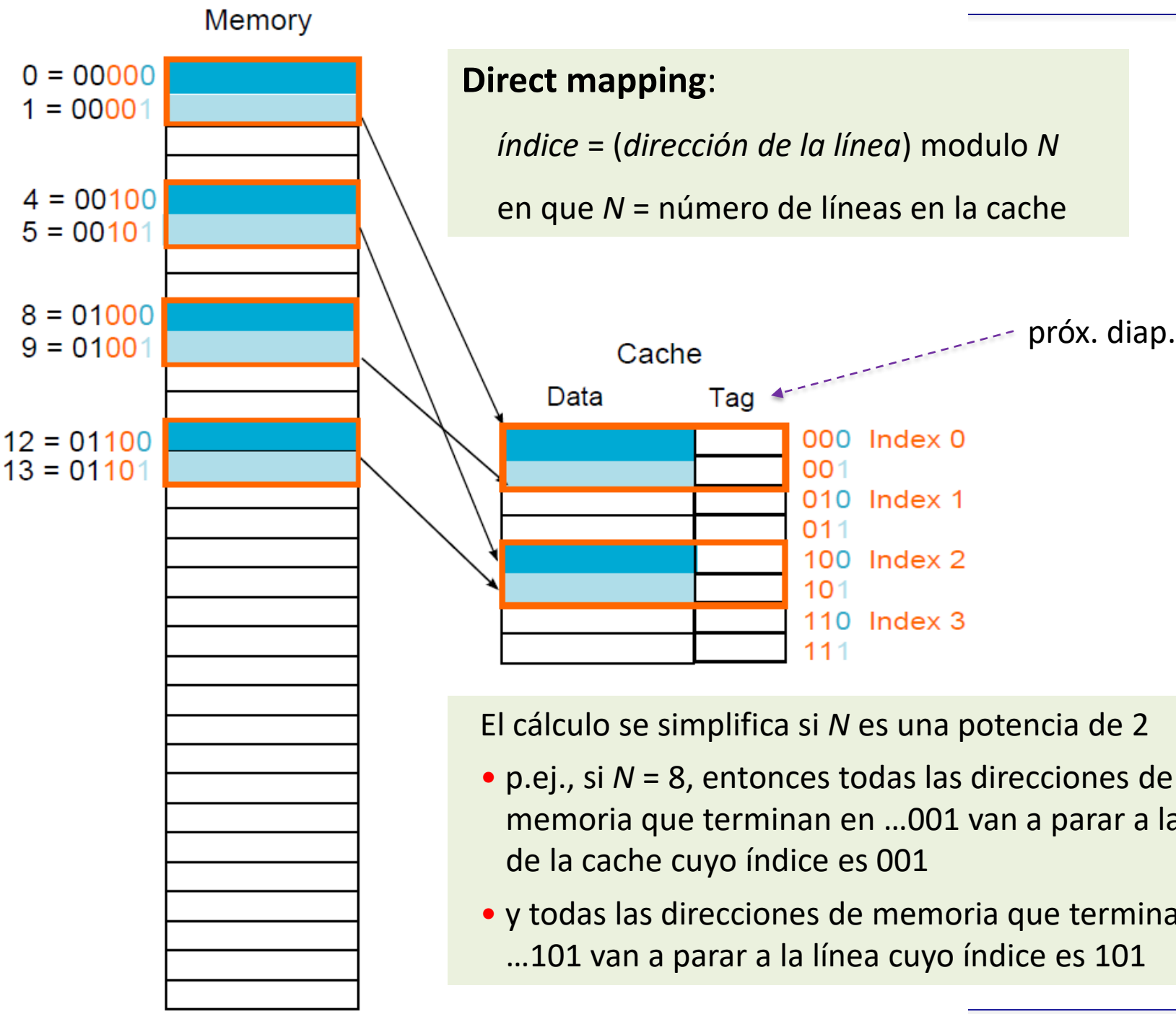


Cache después de la referencia a X_n

Si cada palabra —según su dirección de memoria— puede ir exacta-mente a un lugar en la cache

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

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



Direct mapping:

índice = (*dirección de la línea*) modulo *N*

en que *N* = número de líneas en la cache

El cálculo 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 001
- y todas las direcciones de memoria que terminan en …101 van a parar a la línea cuyo índice es 101

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

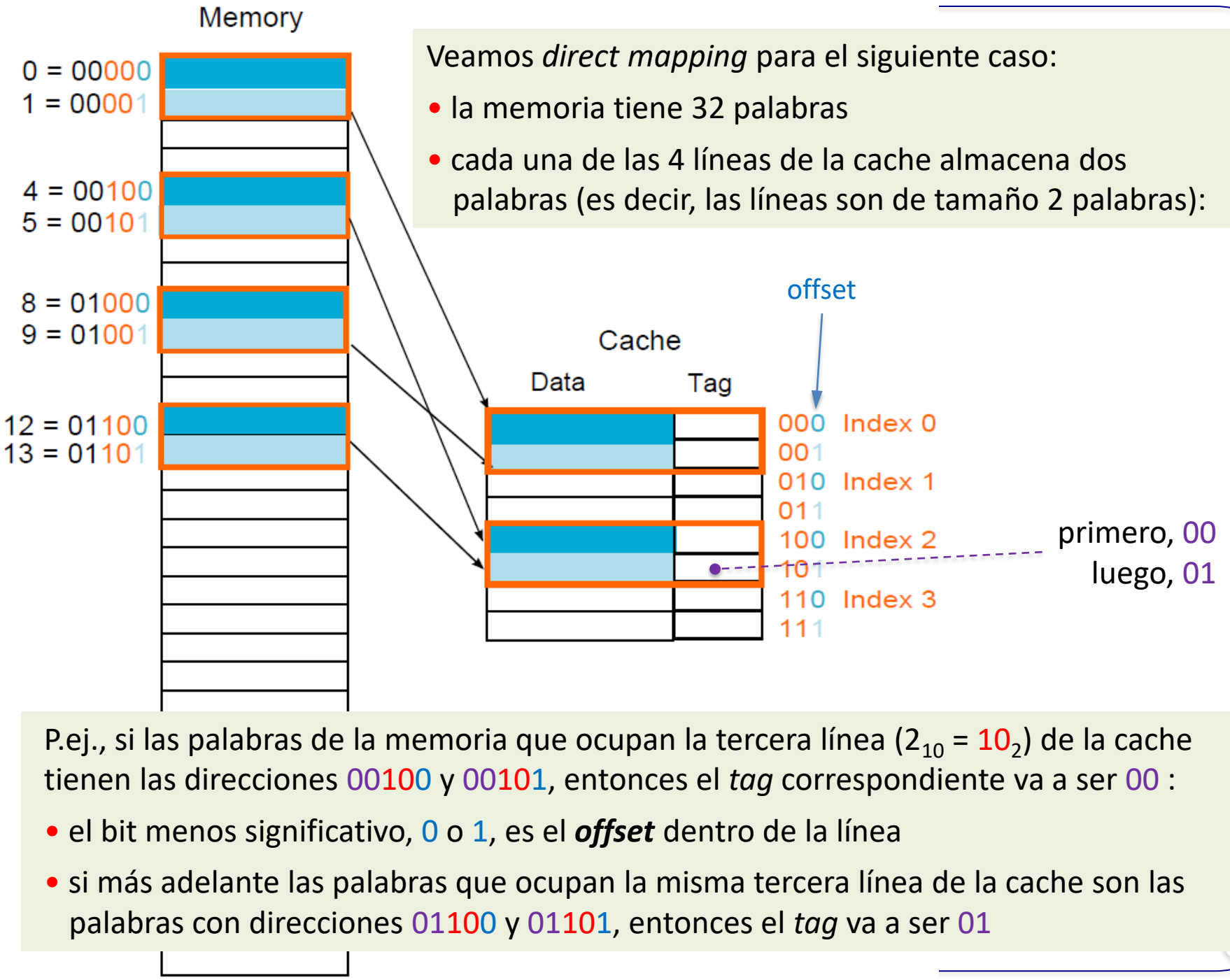
- todas aquellas líneas para las cuales el valor de

  … (*dirección de la línea*) modulo *N*

  … es el mismo

… → necesitamos 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



Veamos *direct mapping* para el siguiente caso:

- la memoria tiene 32 palabras
- cada una de las 4 líneas de la cache almacena dos palabras (es decir, las líneas son de tamaño 2 palabras):

P.ej., si las palabras 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*** dentro de la línea
- si más adelante las palabras que ocupan la misma tercera línea de la cache son las palabras con direcciones 01100 y 01101, entonces el *tag* va a ser 01

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 próximas diaps., vemos el comportamiento de nuestra cache de 4 líneas y dos palabras/línea, inicialmente "vacía" (es decir, contenido inicial inválido)

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

inicial

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

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

luego 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		

luego 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		

luego 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]

luego 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]

luego 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]

luego 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]

Supongamos una cache con 64 líneas, en que cada línea tiene 16 bytes

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

respuesta: 11

tipo de cache	hit		miss	
instrucciones (L1-I)	todo sigue como si nada		la *unidad de control* de la cache debe hacer algo	
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*	en **LOADs**, la *unidad de control* de la cache debe hacer algo	en **STOREs**, hay que mantener la consistencia entre cache y memoria: política *write-allocate* o *non-write-allocate*

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

Cuando el acceso a la cache es un *miss*, esencialmente 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 (diap. 35)

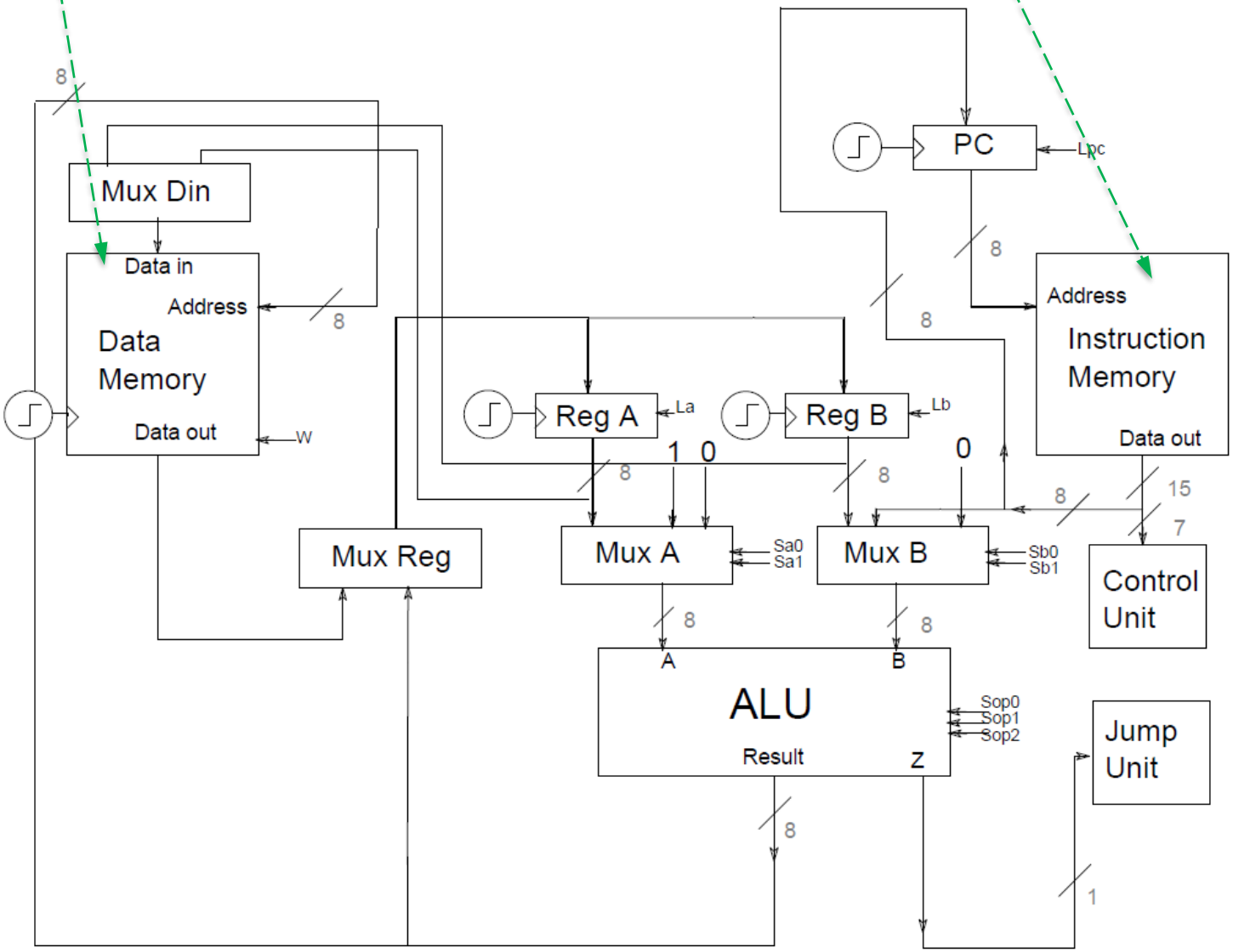
- si es un dato, que va a la cache de datos (**L1-D**), entonces la nueva línea de datos reemplaza a la línea que está en la cache

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

  … ya que pudieron haber sido modificados mientras estaban en la cache (diaps. 36 a 40)

cache de datos **L1-D**, conectada a la memoria principal

cache de instrucciones **L1-I**, conectada a la memoria principal



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 memoria
2. Ordenar a la memoria que ejecute una lectura y esperar (varios ciclos) hasta que se complete la lectura —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)

*Recordemos que al mismo tiempo que la dirección contenida en el *PC* ingresa a la cache de instrucciones, el valor en el *PC* es incrementado (en 4, ya que cada instrucción ocupa 4 bytes) para que apunte a la próxima instrucción

El manejo de la cache de datos es más complejo

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

- en cuyo caso el manejo de un *miss* es similar al manejo de un *miss* de una instrucción

… pero también pueden ser escritos —mediante operaciones *store*— durante su permanencia en la cache:

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

Manejo de *stores* (*writes*) de datos

Al ejecutar una instrucción *store*, si el dato es escrito sólo en la cache y no en la memoria

… entonces la cache y la memoria se vuelven *inconsistentes* —recordemos que el contenido de la cache es un subconjunto del contenido de la memoria

Hay que manejar las inconsistencias:

- *write-through*
- *write-back*

Una posibilidad: en el caso de un *hit*, siempre escribimos el dato tanto en la cache como en la memoria —***write-through***

Simple … sólo que el desempeño no es muy bueno:

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

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

Aternativa: (para *hits*) 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 traída desde la memoria) se actualiza la línea correspondiente en la memoria —***write-back***

Mejora el desempeño en especial si los stores ocurren frecuentemente

… pero 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*)

  … requiere dos ciclos, uno para verificar si hay un hit y otro para efectuar el write

En cualquiera de las opciones anteriores, si el intento de escritura en la cache es un *miss*, entonces hay dos políticas posibles

1) En ***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) En ***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

Vimos *direct mapping* (diaps. 23 a 31)

También hay *fully associative*

… y *n-way set associative* (para varios valores de *n*)

En cuanto a almacenamiento en la cache

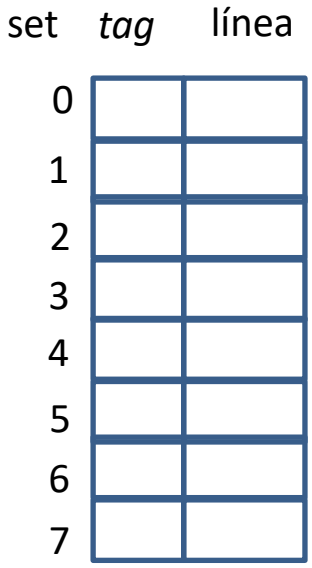
En lugar de *direct-mapping*, el otro extremo en esquemas de asignación de líneas de memoria a líneas de cache es ***fully associative***:

- una línea de la memoria puede ir a parar a cualquier línea de la cache
- → la cache puede contener simultáneamente varias líneas que, bajo el esquema de *direct-mapping*, tendrían que competir por un lugar en la cache
- para encontrar la línea en la cache, **hay que mirar todas las líneas**
- en la práctica, 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

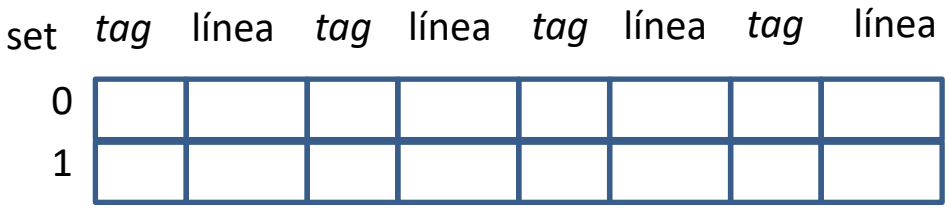
… si hay *n* ubicaciones posibles para cada dirección de memoria —un conjunto, o *set*, de *n* líneas— se habla de una **cache *n-way set associative***:

- 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 más de una línea de memoria que, bajo el esquema de *direct-mapping*, tendrían que competir por un lugar (una línea) en la cache
- 2-*way* y hasta 8-*way* funcionan bien en la práctica

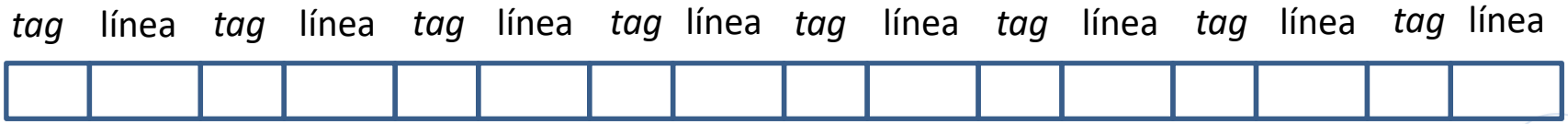


direct-mapped

2-way set associative



4-way set associative



fully associative

Reemplazo de líneas en esquemas *n-way* / *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 *timestamp*; en general el de mejor rendimiento — localidad temporal
- *random*: muy rápido y con rendimiento algo inferior a LFU y LRU

Los tres niveles de cache del procesador Intel Haswell:

- L1: cache *split* de 64 KB por núcleo, o *core* (32 KB para instrucciones y 32 KB para datos); líneas de 64 bytes, 8-*way associative* + pseudo LRU, *write-back*; en el propio chip de la CPU
- L2: cache *unified* 256 KB por núcleo; líneas de 64 bytes, 8-*way associative* + pseudo LRU, *write-back*; en el paquete de la CPU
- L3: 2 MB hasta 20 MB, *unified* y compartida entre núcleos; líneas de 64 bytes, 12-*way associative* + pseudo LRU, *write-back*; en el *board* del procesador (junto a la memoria y los controladores de i/o)

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

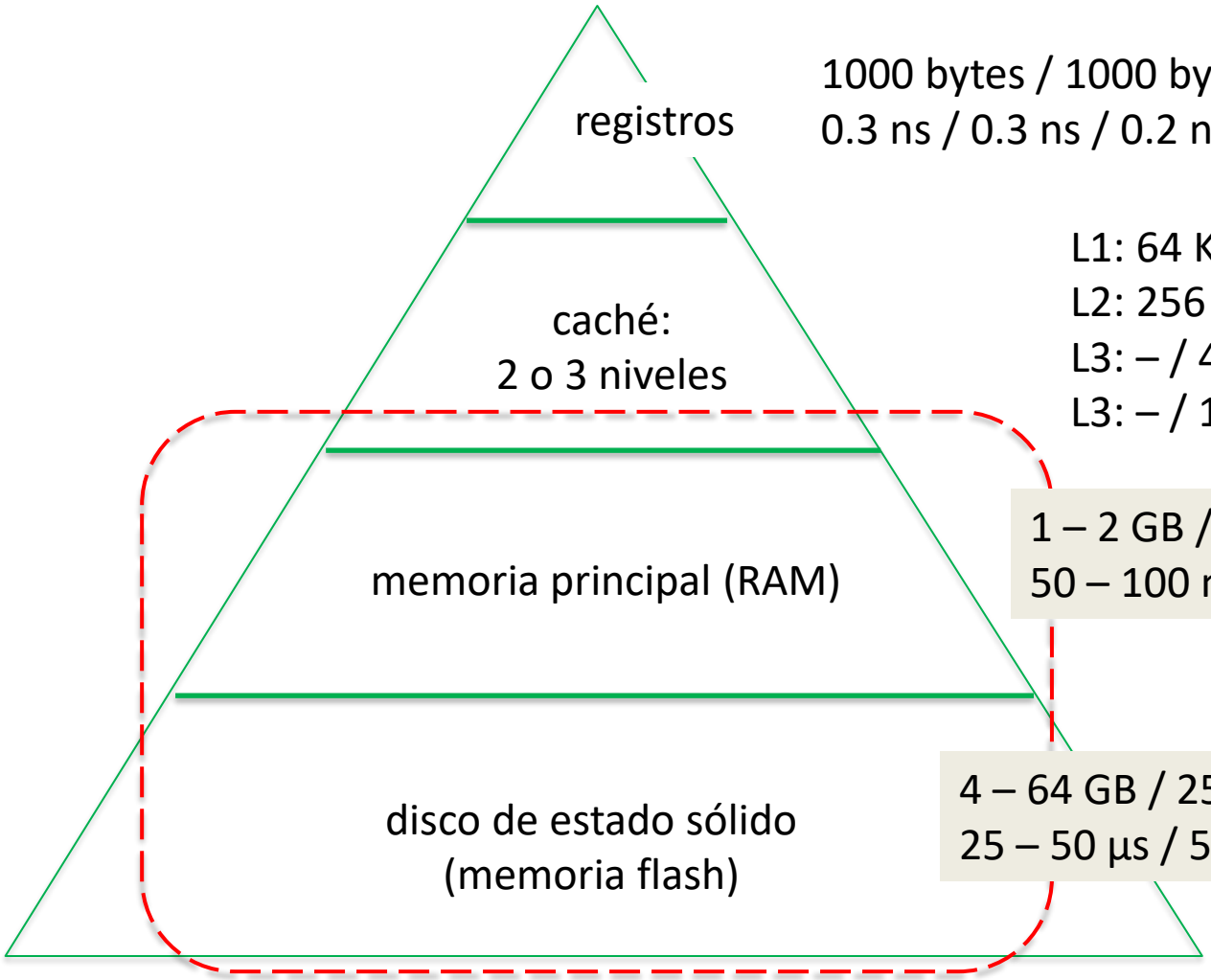
… excepto cuando el tamaño de las líneas llega a ser una fracción importante del tamaño de la cache:

- → 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 también 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 cuántas palabras o bytes se transfieren

A continuación, vamos a estudiar la interacción entre la memoria principal y el disco (memoria secundaria): **memoria virtual**



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

L1: 64 KB y 1 ns (tres casos)
L2: 256 KB y 3-10 ns (tres casos)
L3: – / 4-8 MB / 16-64 MB
L3: – / 10-20 ns / 10-20 ns

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

4 – 64 GB / 256 GB – 1 TB / 1 – 16 TB
25 – 50 µs / 50 – 100 µs / 100 – 200 µs

En los comienzos de la computación, los programadores pasaban mucho tiempo tratando de hacer caber los programas en la memoria, que era poca y cara

La solución era usar memoria secundaria (p.ej., disco) y *overlays*:

- *overlays*, o trozos de programa que contenían código y datos
- era responsabilidad del programador asegurar que el programa en ningún momento tratase de tener acceso a un *overlay* que no estuviera cargado (en memoria) y que los *overlays* cargados no excedieran el tamaño total de la memoria

1961: automatización del proceso de *overlays* –> **memoria virtual**:

- la memoria principal (o memoria física) almacena sólo una parte de todo el programa, que está en memoria secundaria (p.ej., en disco)

  … y en la práctica actúa como una “cache” para el disco

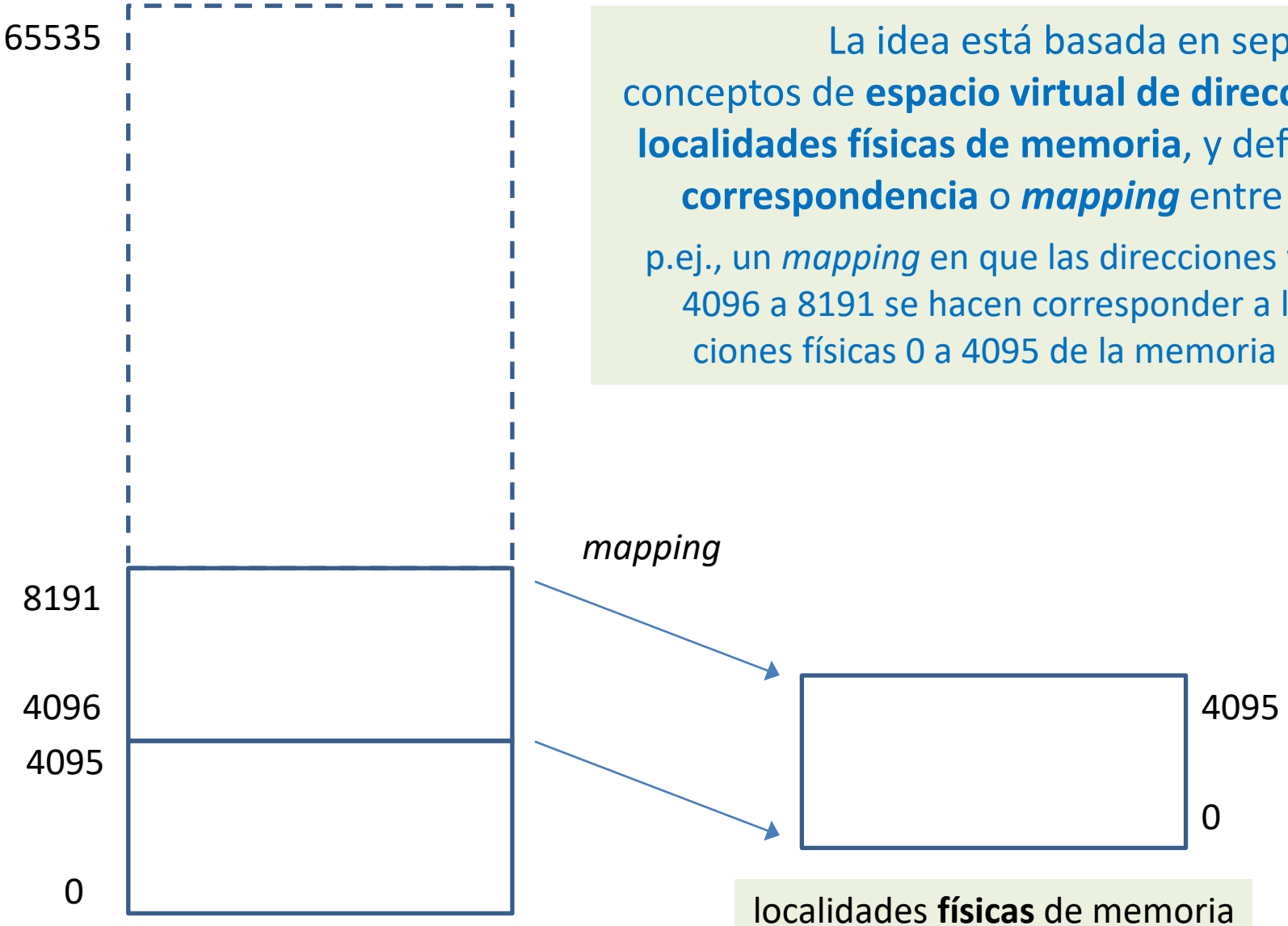
- usada hasta nuestros días

  … hoy, principalmente con el propósito de permitir que múltiples procesos compartan una única memoria principal (y el procesador)

  … proporcionando además protección de memoria entre estos procesos y el sistema operativo

La idea está basada en separar los conceptos de **espacio virtual de direcciones** y **localidades físicas de memoria**, y definir una **correspondencia** o ***mapping*** entre ambos:

p.ej., un *mapping* en que las direcciones virtuales 4096 a 8191 se hacen corresponder a las direcciones físicas 0 a 4095 de la memoria principal



espacio **virtual** de direcciones: direcciones 0 a 65535

localidades **físicas** de memoria en la memoria principal: direcciones 0 a 4095

P.ej., un computador con direcciones de 16 bits y memoria de 4096 palabras:

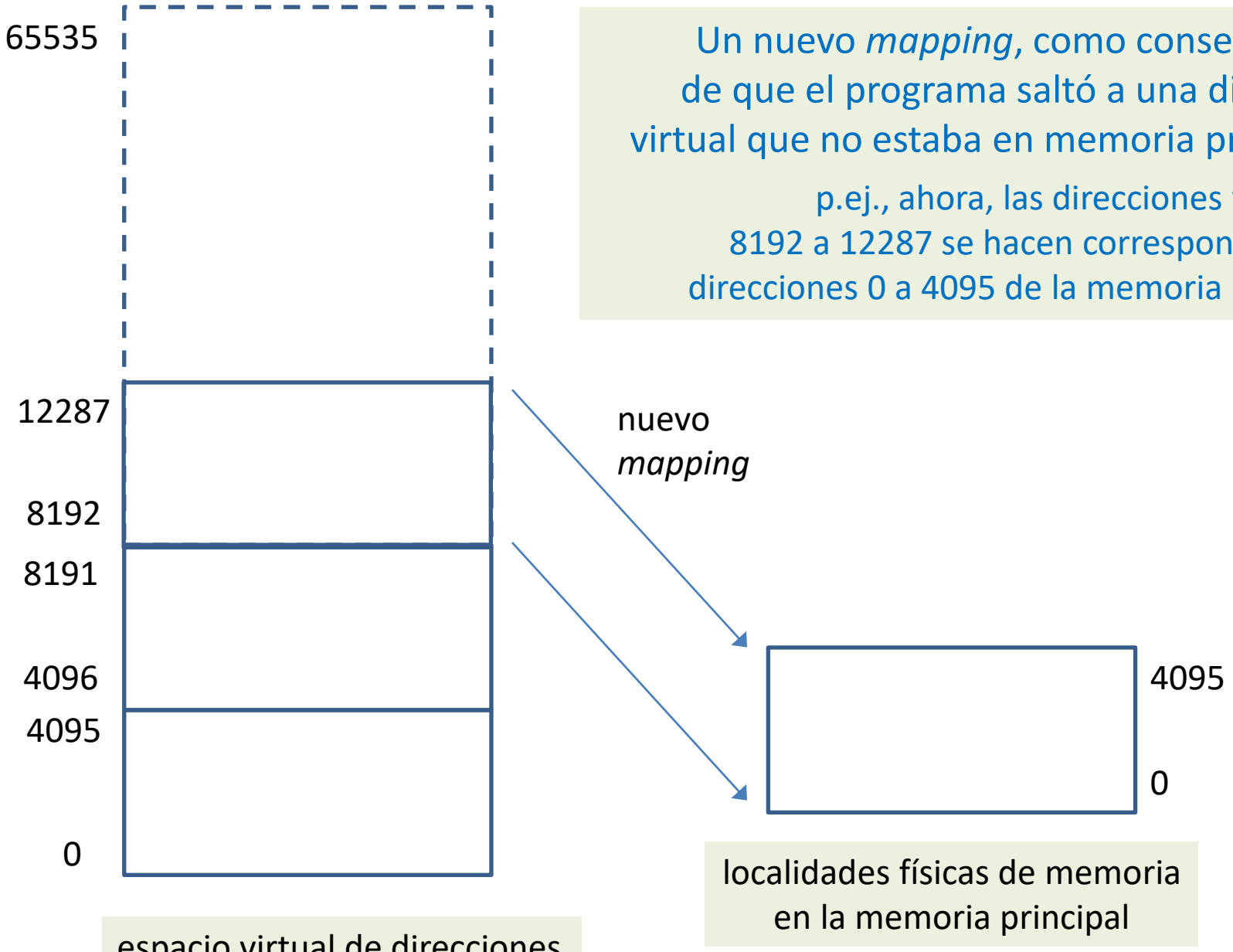
- 16 bits pueden direccionar 65536 palabras: direcciones 0 – 65535 → este es el **espacio virtual de direcciones**
- la memoria principal del computador sólo tiene 4096 palabras con direcciones 0 a 4095 → estas son las **localidades físicas de memoria**
- podemos decirle al computador que, p.ej., haga corresponder las direcciones 4096 a 8191 referenciadas en el programa con las direcciones 0 a 4095 de la memoria → esta correspondencia se llama el ***mapping***, o traducción, **de direcciones**

¿Qué pasa si el programa salta a una dirección entre 8192 y 12287?

Sin memoria virtual: “memoria referenciada inexistente”

Con memoria virtual:

1. El contenido de la memoria principal, que corresponde a las direcciones (virtuales) entre 4096 y 8191 se guarda en el disco
2. Se buscan en el disco las palabras con direcciones (virtuales) 8192 a 12287
3. Las palabras 8192 a 12287 son cargadas en la memoria
4. El mapa de direcciones se cambia: las direcciones virtuales 8192 a 12287 se hacen corresponder ahora a las localidades físicas de memoria 0 a 4095
5. La ejecución del programa continúa como si nada hubiera pasado



Un nuevo *mapping*, como consecuencia de que el programa saltó a una dirección virtual que no estaba en memoria principal:

p.ej., ahora, las direcciones virtuales 8192 a 12287 se hacen corresponder a las direcciones 0 a 4095 de la memoria principal

Los bloques de programa (o de datos) leídos desde el disco se llaman **páginas** (en lugar de “bloques” o “líneas”, como en el caso de la cache)

… y la técnica de traer páginas a la memoria principal y de guardar de vuelta páginas en el disco, según las necesidades del programa, se llama **paginación** (*paging*)

Los programas son escritos como si hubiera suficiente memoria para todo el espacio virtual de direcciones:

- un programa puede leer desde, o almacenar en, cualquier palabra en el espacio virtual

  … o saltar a cualquier instrucción ubicada en cualquier parte dentro del espacio virtual

- el programador puede escribir el programa sin ni siquiera sospechar que existe la memoria virtual
- el computador se ve como si tuviera una gran memoria
- esta simulación de una gran memoria principal mediante paginación no es detectable por el programa (excepto si corre *tests* que midan el tiempo)

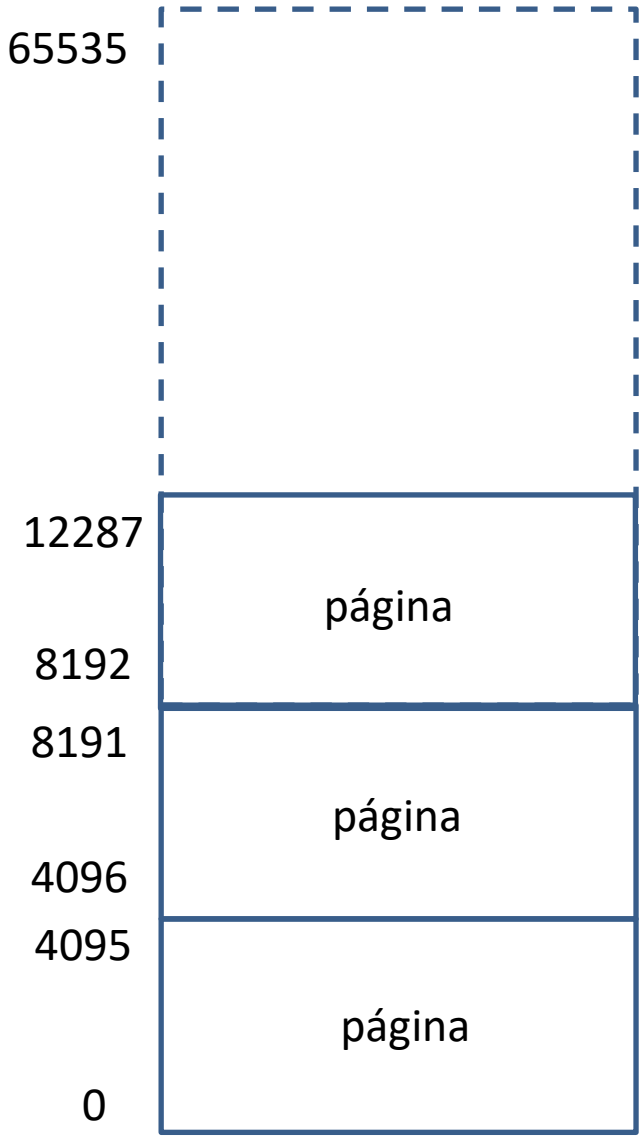
→ el mecanismo de paginación es **transparente**:

- (excepto para quienes escriben sistemas operativos)

Hablamos del ***espacio virtual de direcciones***:

- las direcciones a las que el programa puede hacer referencia
- sólo depende del número de bits que se pueden usar para especificar direcciones
- está dividido en *páginas* (*virtuales*), todas del mismo tamaño
- p.ej., si las direcciones son de 16 bits, entonces el espacio virtual de direcciones consiste en las direcciones 0 a 2^{16}–1 (= 65535)

  … y podría estar dividido en 16 páginas de 4 KB c/u



espacio virtual de direcciones

… de las ***localidades físicas de memoria***:

- las localidades de memoria reales, físicamente disponibles en la memoria principal, también llamado el *espacio físico de direcciones*
- depende del tamaño de la memoria principal,

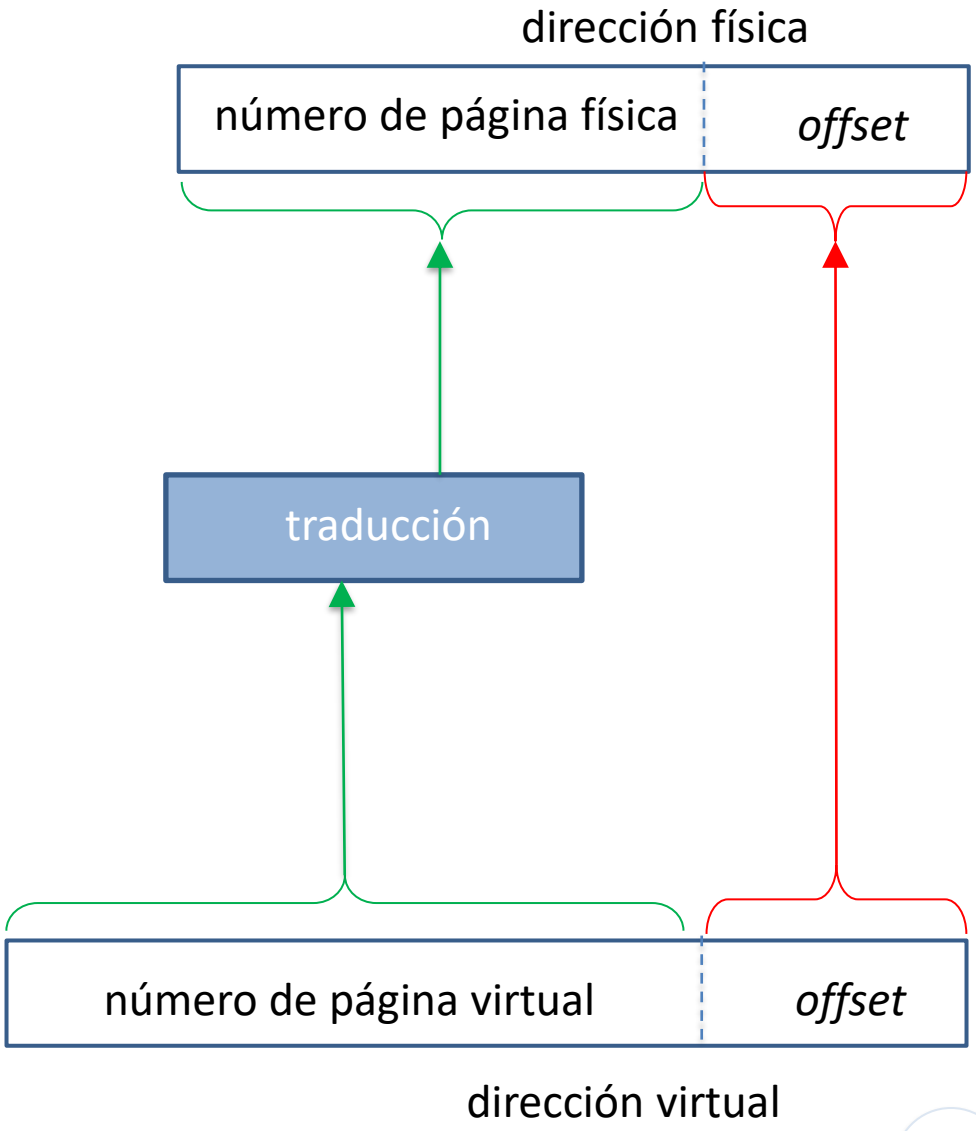
  … y del tamaño de la porción de memoria principal que el sistema operativo asigna para la ejecución del programa
- está dividido en *page frames*, c/u del mismo tamaño que una página

  … de modo que c/u puede almacenar exactamente una página

localidades físicas de memoria en la memoria principal; en este ej., la memoria principal contiene un solo *page frame*, pero en general puede contener millones

… y de la ***correspondencia***, ***mapping***, o ***traducción*** de direcciones:

- dividimos una dirección de memoria en el número de página (los bits más significativos) y el *offset* dentro de la página (los bits menos significativos)
- como las páginas virtuales y las páginas físicas (o *page frames*) son del mismo tamaño, el *offset* de una dirección dentro de la página es el mismo en ambos casos
- la traducción consiste en traducir el número de página virtual al número de página física (o de *page frame*)
- esta traducción se hace mediante una **tabla de páginas**



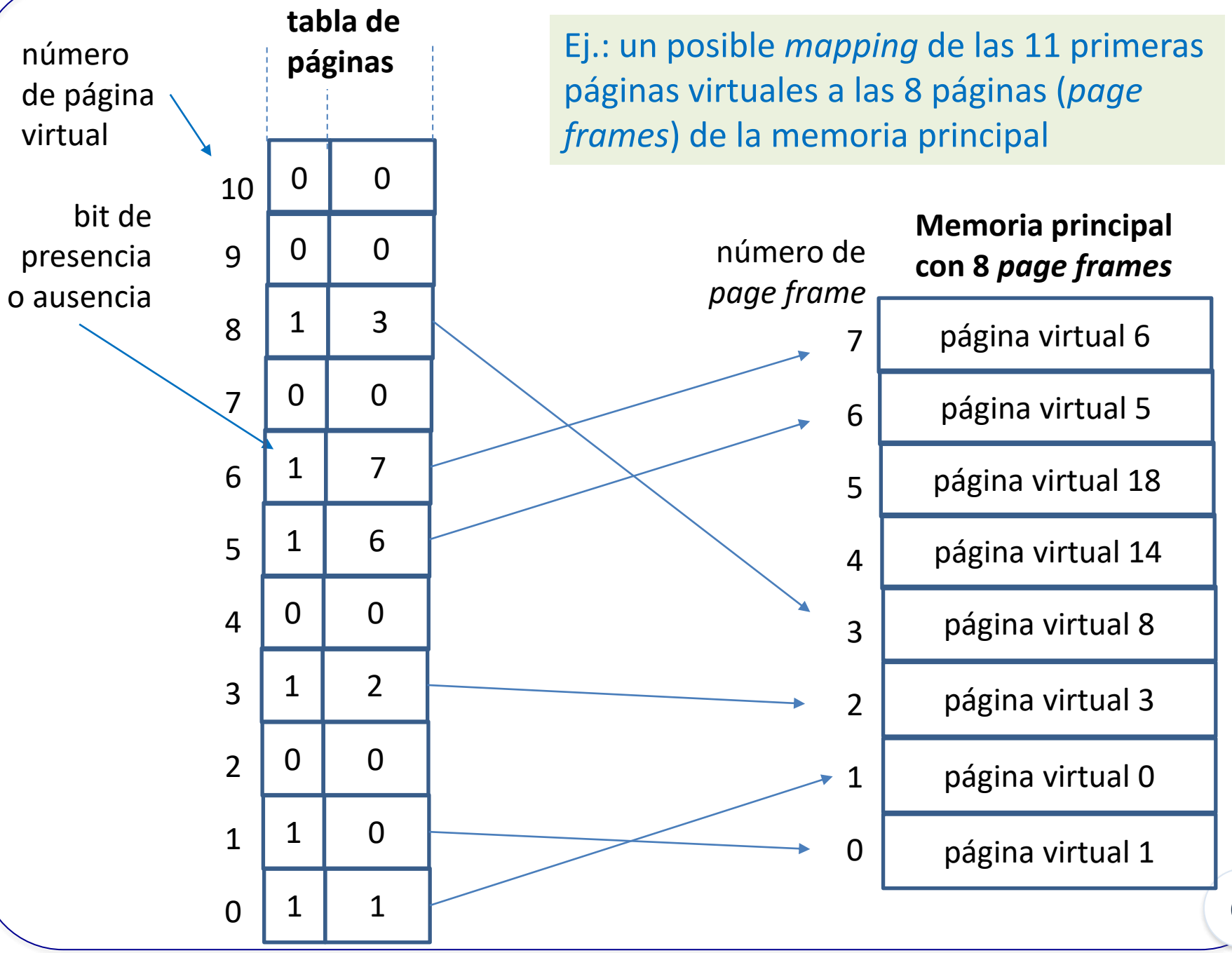
La **tabla de páginas** es esencialmente un arreglo *T* —típicamente, muy grande— con tantas filas como la cantidad máxima posible de páginas virtuales (p.ej., millones)

El número de la página virtual buscada (los bits más significativos de la dirección virtual), digamos *k*, se usa como índice para seleccionar una fila de la tabla de páginas: *T*[*k*]

Cada fila *T*[*k*] de la tabla contiene un bit y número:

- el bit —llamado bit de presencia/ausencia— indica si la página virtual *k* está (bit = 1) o no (bit = 0) en memoria principal
- … en caso afirmativo (bit = 1), el número es el número del *page frame* en que se encuentra la página virtual
- … en caso negativo (bit = 0), el número representa de alguna manera la dirección de la página virtual en el disco (para que se la pueda ir a buscar, traerla y cargarla en alguno de los *page frames* de la memoria principal)

Ej.: un posible *mapping* de las 11 primeras páginas virtuales a las 8 páginas (*page frames*) de la memoria principal



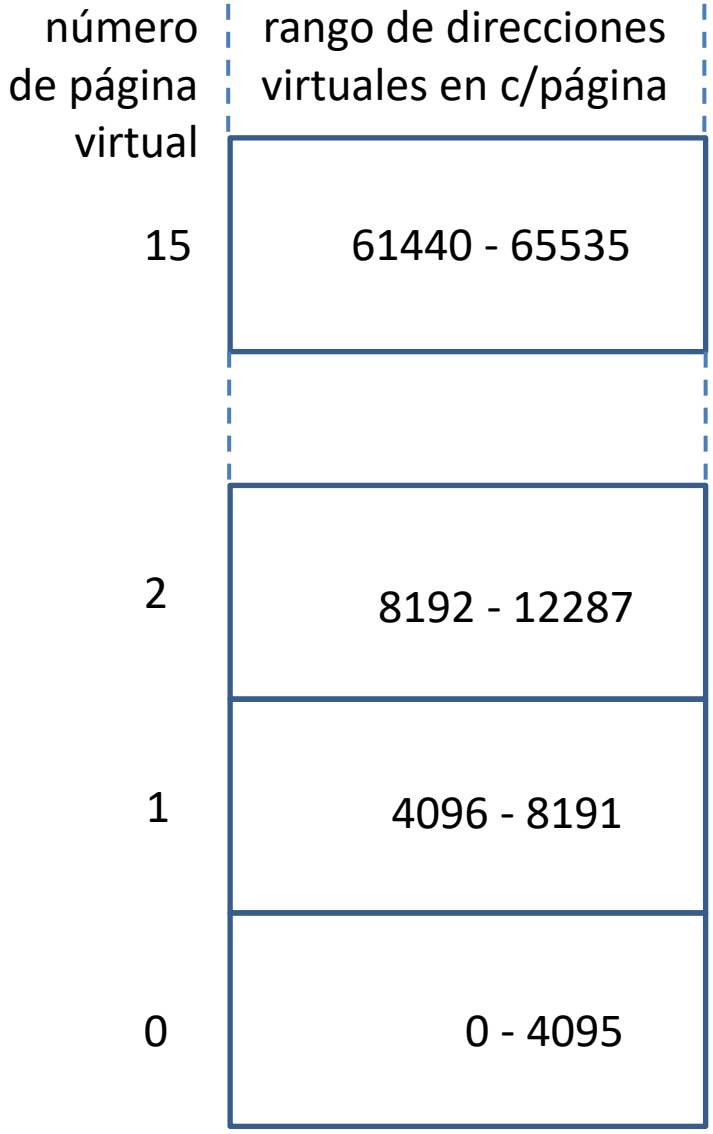
Si el computador maneja direcciones de 32 bits → puede direccionar 2^{32} bytes → 1,048,576 páginas virtuales de 4 KB c/u

… y si el computador tiene 1 GB (2^{30} bytes) de memoria principal → 262,144 páginas físicas (o *page frames*) de 4 KB c/u:

- para c/u de las 1,048,576 páginas virtuales, la tabla de páginas debe saber si la página está o no en memoria principal
- … y, en caso afirmativo, en cuál de las 262,144 páginas físicas está

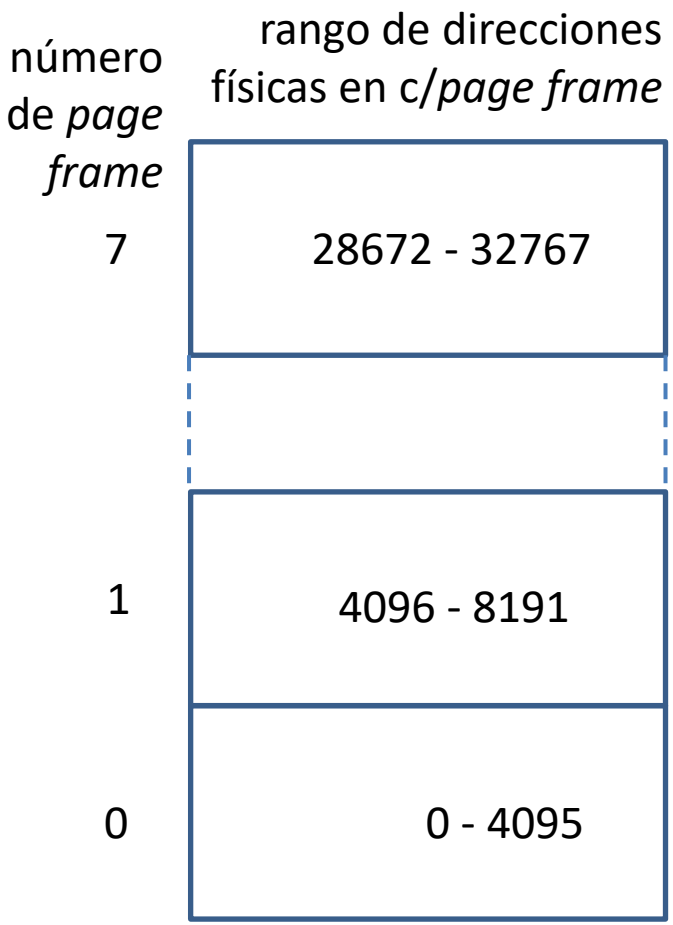
La tabla de páginas tiene 1,048,576 filas, en que la fila *k*-ésima contiene:

- un bit de presencia/ausencia, que indica si la página virtual número *k* está (bit = 1) o no (bit = 0) en la memoria física
- … y, si bit = 1, entonces también contiene el número de la página física —un número entre 0 y 262,143— en la que está la página virtual número *k*



espacio virtual de direcciones (de 32 bits) divido en páginas de 4 KB

En esta figura, en vez de 262,144 páginas físicas, o *page frames*, suponemos que sólo hay 8



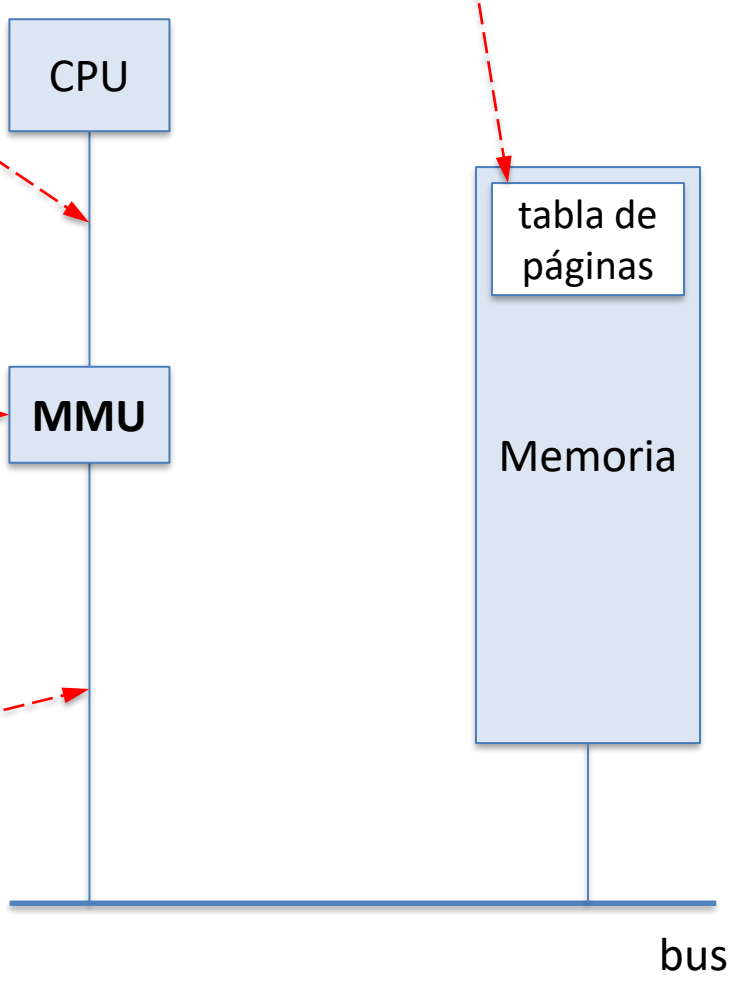
memoria principal de 32 KB dividida en 8 *page frames* de 4 KB

la tabla de páginas, debido a su tamaño, está en memoria, a partir de la dirección almacenada en el ***page table register***

la CPU produce direcciones virtuales

la **unidad de manejo de memoria** (**MMU**) convierte las direcciones virtuales en direcciones físicas:
… puede estar en el chip de la CPU o en un chip aparte
… tiene un *registro de input* y un *registro de output*
… y hace uso de la tabla de páginas

la MMU envía direcciones físicas a la Memoria



Supongamos por un momento que el computador usa base 10

… que el espacio virtual de direcciones va de 0 a 9999, divido en 100 páginas virtuales de 100 direcciones c/u

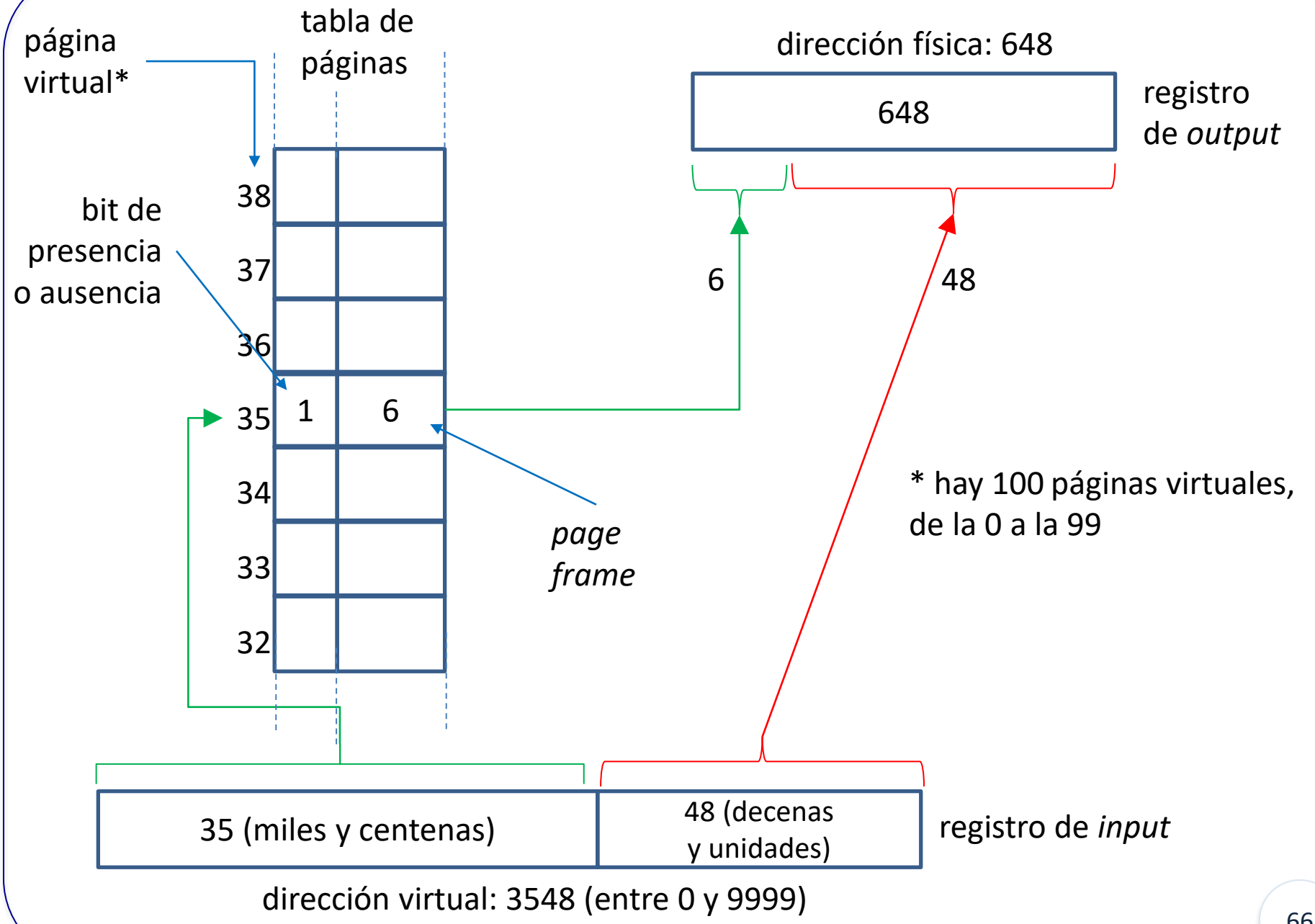
… y que la memoria contiene 10 *page frames* de 100 direcciones c/u

Una dirección virtual tal como 3548 es dividida en dos partes:

- número de página virtual, 35 (los miles y las centenas)
- *offset*, 48, dentro de la página (las decenas y unidades)

¿Está la página en memoria?

- hay 100 páginas virtuales y sólo 10 páginas físicas (*page frames*)
- **bit de presencia**(1)**/ausencia**(0) en cada fila de la tabla


página virtual*
tabla de páginas
dirección física: 648
648
registro de *output*
bit de presencia o ausencia
38
37
36
35
34
33
32
1
6
6
48
page frame
* hay 100 páginas virtuales, de la 0 a la 99
35 (miles y centenas)
48 (decenas y unidades)
registro de *input*
dirección virtual: 3548 (entre 0 y 9999)

Volvamos a nuestro ej. de la diap. 64:

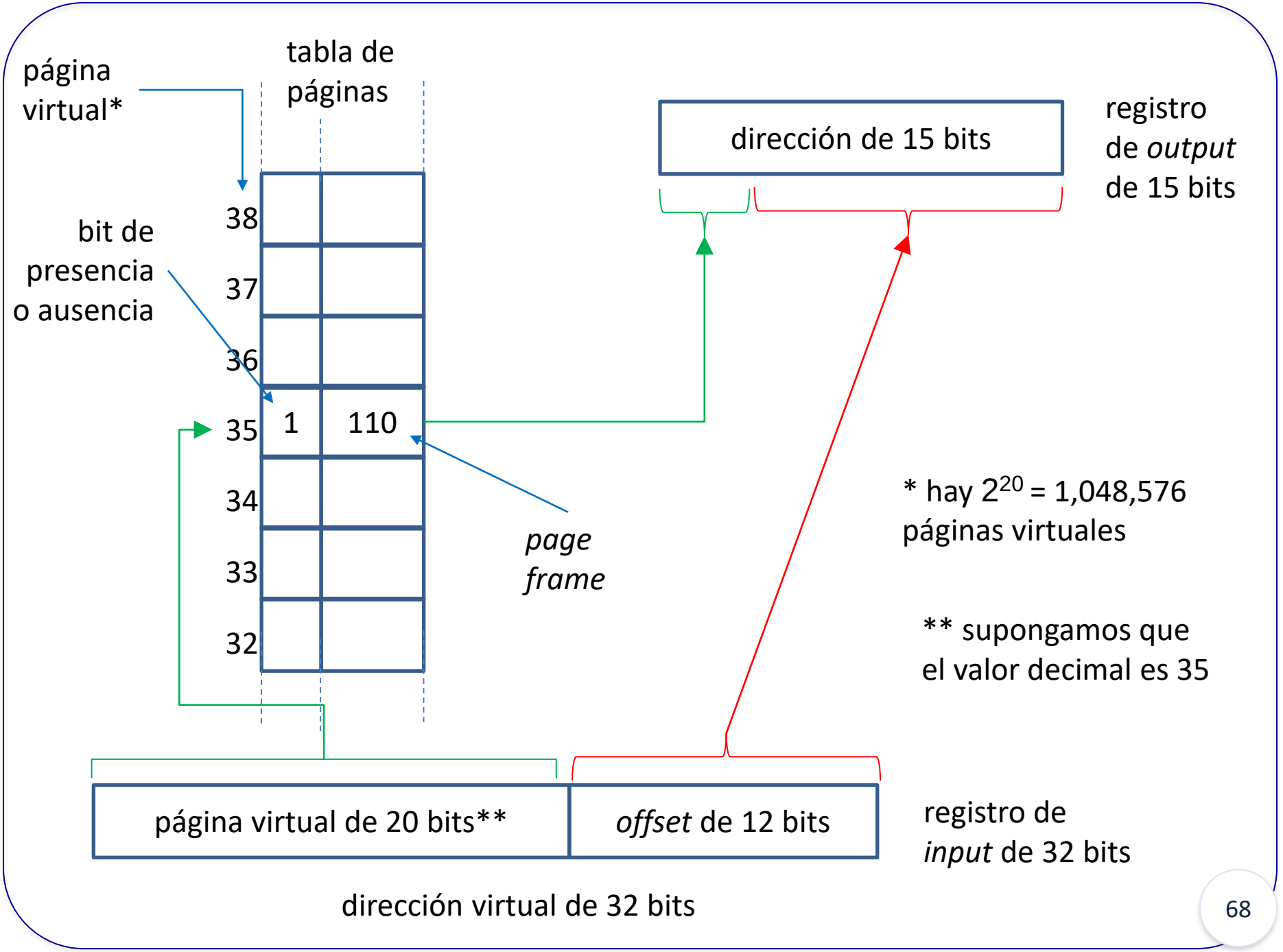
- direcciones virtuales de 32 bits (de 0 a $2^{32}–1$)
- 2^{20} páginas virtuales de 2^{12} = 4096 direcciones c/u
- memoria de sólo 8 *page frames* de 2^{12} = 4096 direcciones c/u

Una dirección virtual de 32 bits es dividida en dos partes:

- número de página de 20 bits (los 20 bits más significativos), como índice a la tabla de páginas
- *offset* de 12 bits (los 12 bits menos significativos) dentro de la página (páginas de 4K)

Pero, ¿está la página en memoria?

- hay 2^{20} páginas virtuales y solo 8 páginas físicas (*page frames*)
- **bit de presencia**(1)**/ausencia**(0) en cada fila de la tabla


página virtual*
tabla de páginas
bit de presencia o ausencia
38
37
36
35
34
33
32
1
110
page frame
dirección de 15 bits
registro de output de 15 bits
* hay 2^{20} = 1,048,576 páginas virtuales
** supongamos que el valor decimal es 35
página virtual de 20 bits**
offset de 12 bits
registro de input de 32 bits
dirección virtual de 32 bits

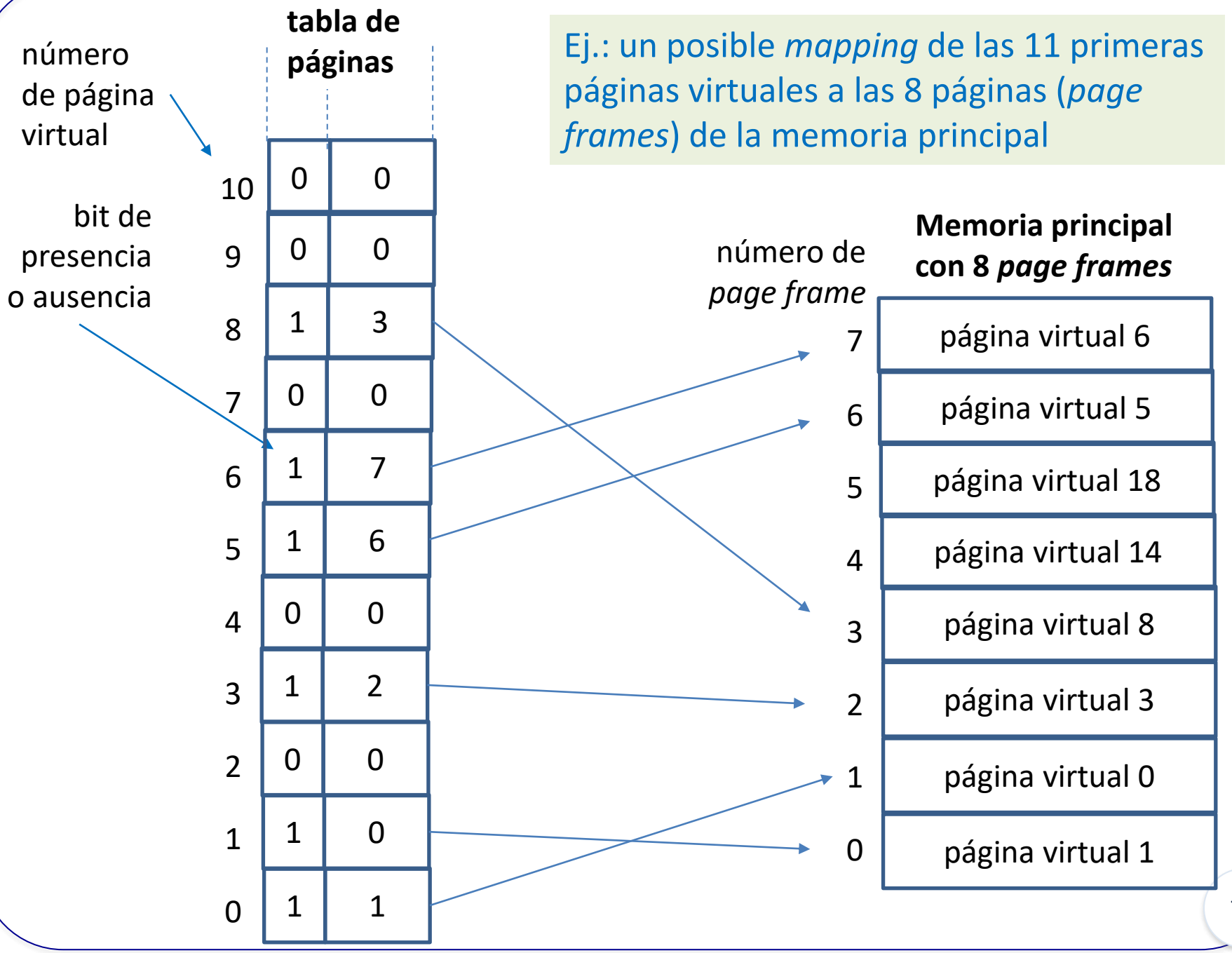
Si la página virtual está en memoria (bit de presencia/ausencia = 1 → un "*hit*"), como en el ej. de la diap. anterior,

… entonces se usa el número del *page frame* almacenado en esa fila de la tabla:

- este número (3 bits, ya que hay 8 *page frames*) se une a los 12 bits del *offset* para formar la dirección física que se envía a la memoria

Si no está → ***page fault*** (en lugar de un "*miss*")

Ej.: un posible *mapping* de las 11 primeras páginas virtuales a las 8 páginas (*page frames*) de la memoria principal



¿Qué se hace cuando ocurre un *page fault*?

- p.ej., si en la diap. anterior la dirección virtual perteneciera a alguna de las páginas 2, 4, 7, 9 o 10 (bit de presencia/ausencia = 0)

- el sistema operativo debe traer desde el disco la página requerida

  … ponerla en algún *page frame* de la memoria principal

  … ingresar su nueva dirección física (número de *page frame*) a la tabla de páginas

  … poner el bit de presencia/ausencia en 1

  … y repetir la ejecución de la instrucción que causó el *page fault*

- … pero justo antes de todo lo anterior debe decidir cuál página saca de la memoria principal y devuelve al disco (para hacer espacio para la página recién traída)

  … → se necesita un algoritmo para decidir cuál página sacar

Sacar una página al azar no es una buena idea:

- si saliera la página que produjo el *page fault*, entonces se va a producir otro *page fault* en cuanto se trate de leer la próxima instrucción

Los sistemas operativos predicen cuál de las páginas en memoria es la menos “útil” :

- predicen cuándo va a ocurrir la próxima referencia a cada página

  … y sacan la página cuya próxima referencia está más adelante en el futuro (una página que no va a ser requerida en mucho tiempo)

P.ej.,

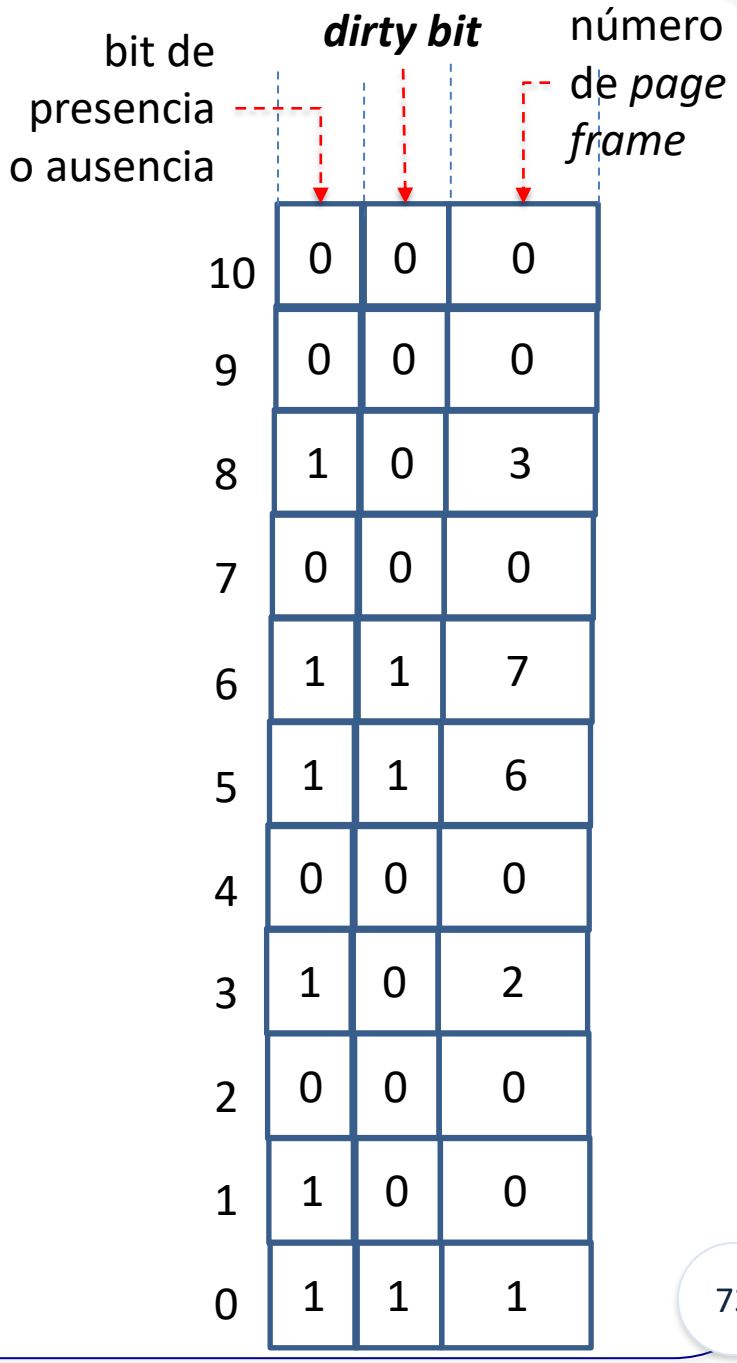
- **LRU** —la página que se usó por última vez hace más tiempo— aproximado
- FIFO —la página que se trajo a memoria hace más tiempo— aproximado

¿Qué pasa si la página elegida fue modificada mientras estaba en memoria?

Si una página que está en memoria es modificada por el programa, la página debe ser actualizada en el disco

En la práctica, la página se actualiza sólo cuando se la saca de su *page frame* para poner allí otra página (es decir, al ser reemplazada) → **protocolo *write-back***:

- se agrega un ***dirty bit*** a cada fila de la tabla de páginas → se pone en 1 cuando (cual-quier palabra de) la página es escrita
- cuando el sistema operativo saca una página cuyo *dirty bit* está en 1, la página es copiada en el disco antes de que su *page frame* en memoria pueda ser ocupado por otra página
- **(** el protocolo *write-through* no es práctico en memoria virtual, ya que la escritura en disco puede tomar millones de ciclos **)**



La tabla de páginas, debido a su tamaño (fácilmente, más de un millón de líneas), se mantiene en memoria

→ cada acceso a la memoria que hace un programa puede llegar a significar dos accesos a memoria:

- un acceso a la memoria (a la tabla de páginas) para obtener la dirección física —la traducción
- un segundo acceso para obtener el dato (este acceso puede ahorrarse muchas veces gracias a un buen funcionamiento de la cache)

Para mejorar, aprovechamos la localidad temporal/espacial de las referencias a la tabla:

- la mayoría de los programas tienden a hacer un gran número de referencias a una cantidad pequeña de páginas (y no vice versa)
- cuando se hace una traducción de un número de página virtual, probablemente la misma traducción se va a necesitar de nuevo pronto
- → …

→ Incluimos en la MMU una cache especial que almacena las traduc-ciones usadas recientemente —**TLB** (*translation-lookaside buffer*):

- almacena un subconjunto de la tabla de páginas → las traducciones más recientemente consultadas
- cada entrada:

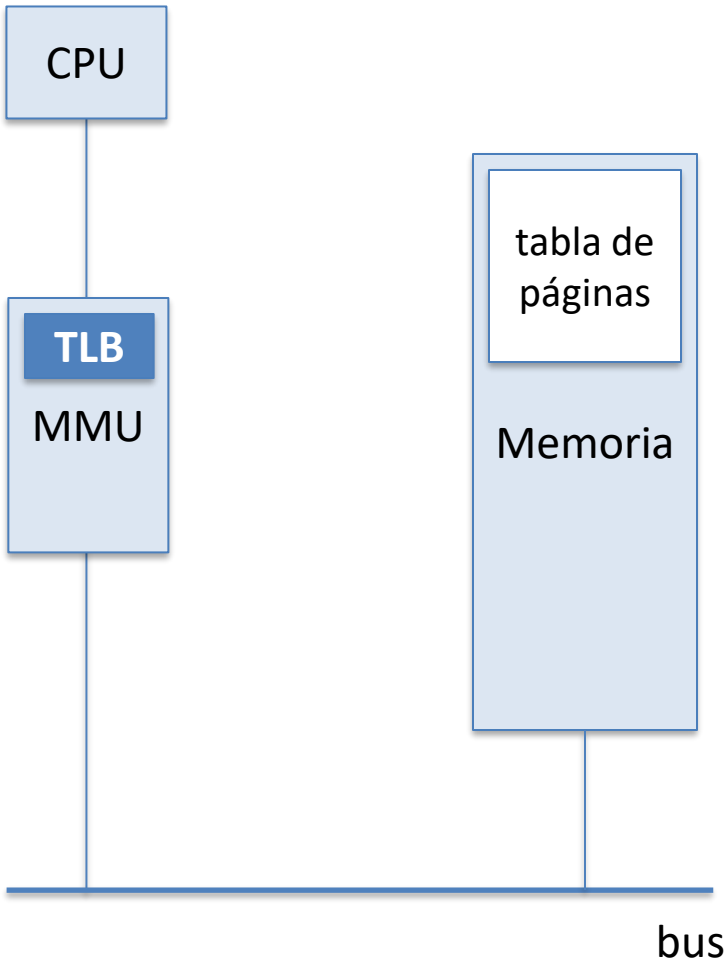
  - *tag* con el número de la página virtual

  - número de la página física

  - bit de validez

  - *dirty bit*

- p.ej., 16 – 512 entradas, con líneas que contienen una o dos filas de la tabla de páginas



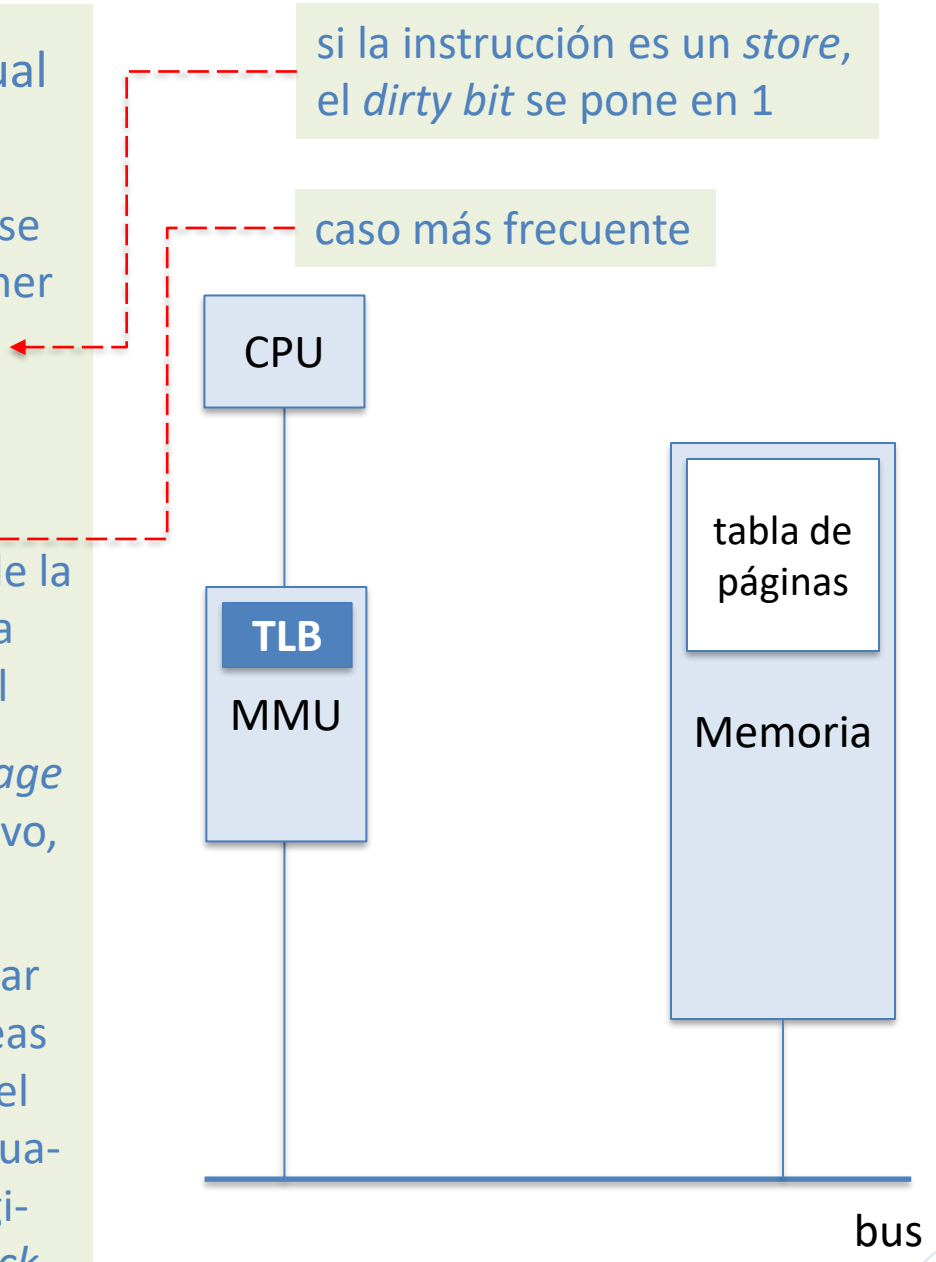
La MMU busca cada dirección virtual primero en la TLB:

- *hit* → el número de la página física se usa para formar la dirección (sin tener que ir a la tabla de páginas)
- *miss* → dos posibilidades

  … la **página está en memoria** → se copia la línea correspondiente desde la tabla de páginas al TLB y se vuelve a procesar la dirección virtual original

  … la **página no está en memoria** (*page fault*) → se llama al sistema operativo, usando una excepción…

  … en ambos casos, hay que actualizar el TLB, remplazando una de sus líneas → como la línea que sale contiene el *dirty bit* de una página, hay que actualizar este *dirty bit* en la tabla de páginas, normalmente usando *write-back*



La memoria virtual y la cache funcionan juntas como una jerarquía:

- un dato no puede estar en la cache a menos que esté presente en la memoria principal

El sistema operativo mantiene esta jerarquía

- eliminando el contenido de la página que corresponda de la cache cuando decide migrar esa página al disco

  … y actualizando la tabla de páginas y el TLB → cualquier intento de tener acceso a algún dato de la página migrada producirá un *page fault*

…

…

En el mejor caso:

- una dirección virtual es traducida por el TLB y enviada a la cache, donde el dato que se busca es encontrado, leído y finalmente enviado al procesador

En el peor caso:

- una dirección virtual no está en el TLB, ni en la tabla de páginas, ni en la cache

→ es posible iniciar la ejecución de un programa aun cuando nada del programa está en la memoria principal —**paginación por demanda:**

- hay que inicializar apropiadamente la tabla de páginas
- la CPU trata de leer la primera instrucción → *page fault* → la página que contiene esa instrucción es cargada en la memoria e ingresada a la tabla de páginas
- si la instrucción contiene dos direcciones en páginas diferentes, y diferentes a la página de la instrucción → dos *page faults* adicionales → dos nuevas páginas son traídas desde el disco antes de que la instrucción pueda finalmente ser ejecutada
- la próxima instrucción puede producir nuevos *page faults*, etc.

La paginación por demanda —se trae una página a la memoria sólo cuando la página es requerida (no por adelantado)— presenta un problema:

- si la memoria del computador es compartida por múltiples procesos (como ya veremos), y los procesos son sacados de la CPU después de ejecutar durante un cierto tiempo (p.ej., 100 ms), entonces cada programa va a ser reanudado muchas veces
- como la tabla de páginas es única para cada programa, la pregunta de si usar o no este sistema de paginación se vuelve crítica

Principio de localidad (misma idea que en el caso de caches):

- los programas no referencian su espacio de direcciones uniformemente
- las referencias tienden a ser a un grupo pequeño de páginas

Working set:

- en todo instante de tiempo, existe un conjunto de todas las páginas usadas por las referencias de memoria más recientes
- este conjunto cambia de a poco a lo largo del tiempo
- → se puede adivinar cuáles páginas van a a ser requeridas cuando el programa sea reanudado
- … a partir de su *working set* cuando fue suspendido la última vez

Necesidad de reemplazo de páginas:

- idealmente, el *working set* se mantiene en memoria principal para reducir el número de *page faults*
- el *working set* debe ser "descubierto" por el sistema operativo a medida que el programa es ejecutado
- si el programa hace referencia a una página que no está en memoria, hay que ir a buscarla al disco

  … y cambiarla por alguna otra página que es enviada de vuelta al disco
- → se necesita un algoritmo para decidir cuál página sacar

TLB	Tabla de páginas	Cache	¿Es posible? ¿Bajo qué circunstancia?
hit	hit	miss	Posible, pero …
miss	hit	hit	
miss	hit	miss	
miss	miss	miss	
hit	miss	miss	Imposible: …
hit	miss	hit	Imposible: …
miss	miss	hit	Imposible: …