CPU vs memoria DRAM

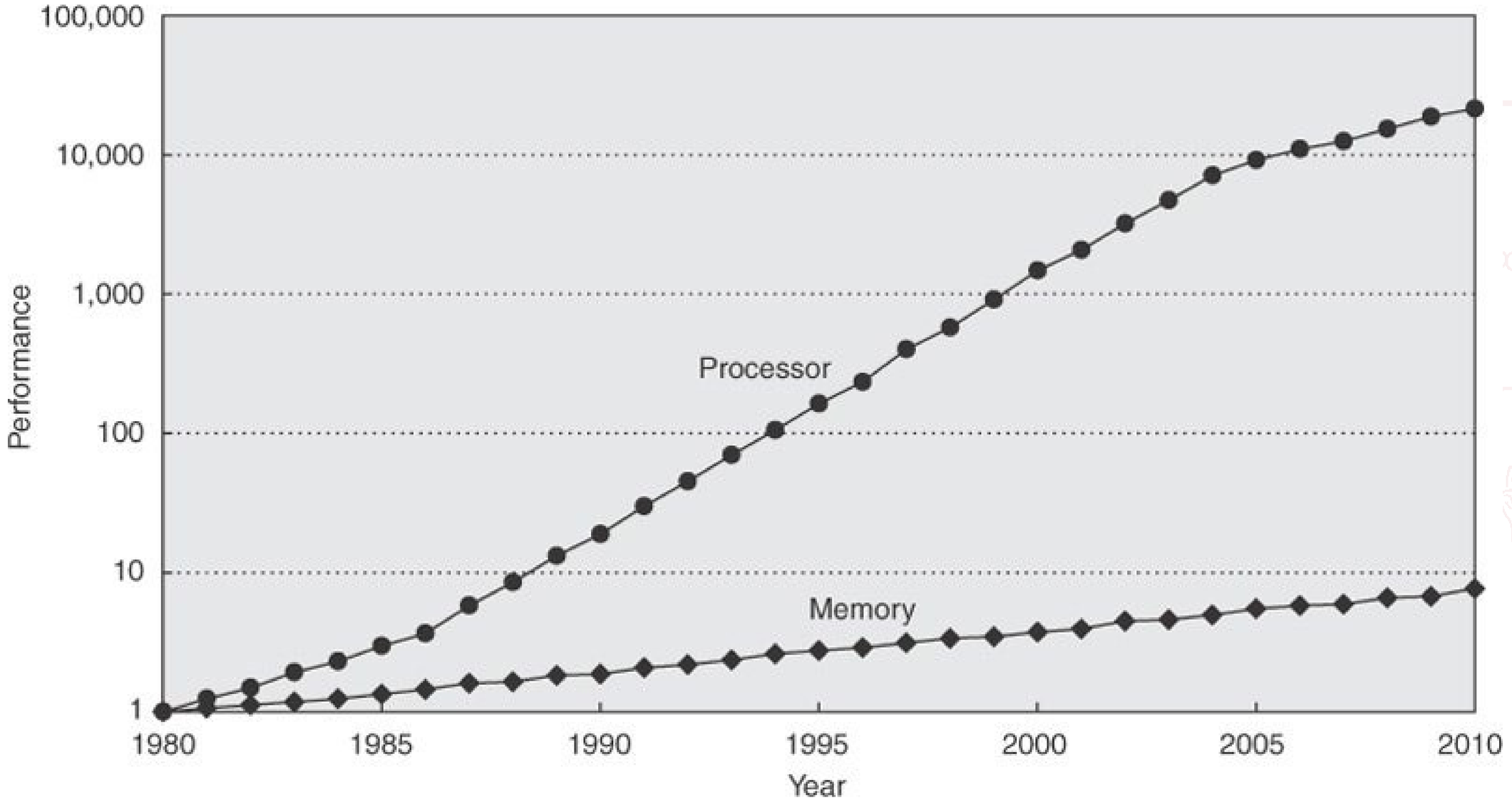


Immagine da Hennessy and Patterson, Computer Architecture, 2012

Memory Systems

Gerarchie di memoria, memoria cache, memoria virtuale

Sandro Savino (sandro.savino@dei.unipd.it)
Department of Information Engineering, University of Padova

Argomenti:

- Gerarchia della memoria
- Località e memoria cache
- Tipi di cache e utilizzo
- Memoria virtuale

Materiale:

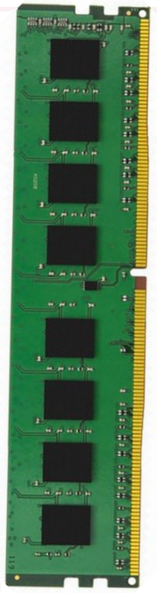
- Capitolo 12

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Gerarchia di memoria

- Le principali caratteristiche di una memoria sono:
  - Dimensione
  - Tempo di accesso
  - Costo per bit

- Esistono dei compromessi:
  - Minor tempo di accesso, maggior costo per bit (SRAM)
  - Maggior capacità, minor costo per bit (DRAM, SDRAM)
  - Maggior capacità, maggior tempo di accesso (DRAM, SDRAM)



Gerarchia di memoria

- Le memorie DRAM garantiscono grandi dimensioni e basso costo per bit, ma sono lente!

- La velocità di trasferimento dati dalla memoria al processore è molto inferiore rispetto alla velocità di elaborazione dati della CPU.

- Molte computazioni sono rallentate dall'accesso alla memoria e non dalla CPU
  - La CPU spesso deve aspettare i dati in arrivo dalla memoria!

- E' improbabile che il gap venga colmato tecnologicamente

Gerarchia di memoria

Come è possibile ridurre questo gap?

- Esistono memorie più veloci della RAM (ad es. 7 ns, invece di 70 ns) ma sono costose e di estensione ridotta (ad es. 512 Kbyte).

- Non è possibile realizzare una memoria che sia grande, veloce ed economica ma è possibile sviluppare una **gerarchia di memoria** che soddisfa tutti i requisiti.

Una memoria più VELOCE

Gerarchia di memoria

- **CPU**: alta velocità di elaborazione
- **Cache**: fra CPU e RAM
  - È molto veloce rispetto alla memoria RAM
  - Ha estensione limitata ed è costosa (in € e in area)
- **Main memory**: RAM, alta capacità, velocità ridotta
- **Hard drive**: modulo I/O

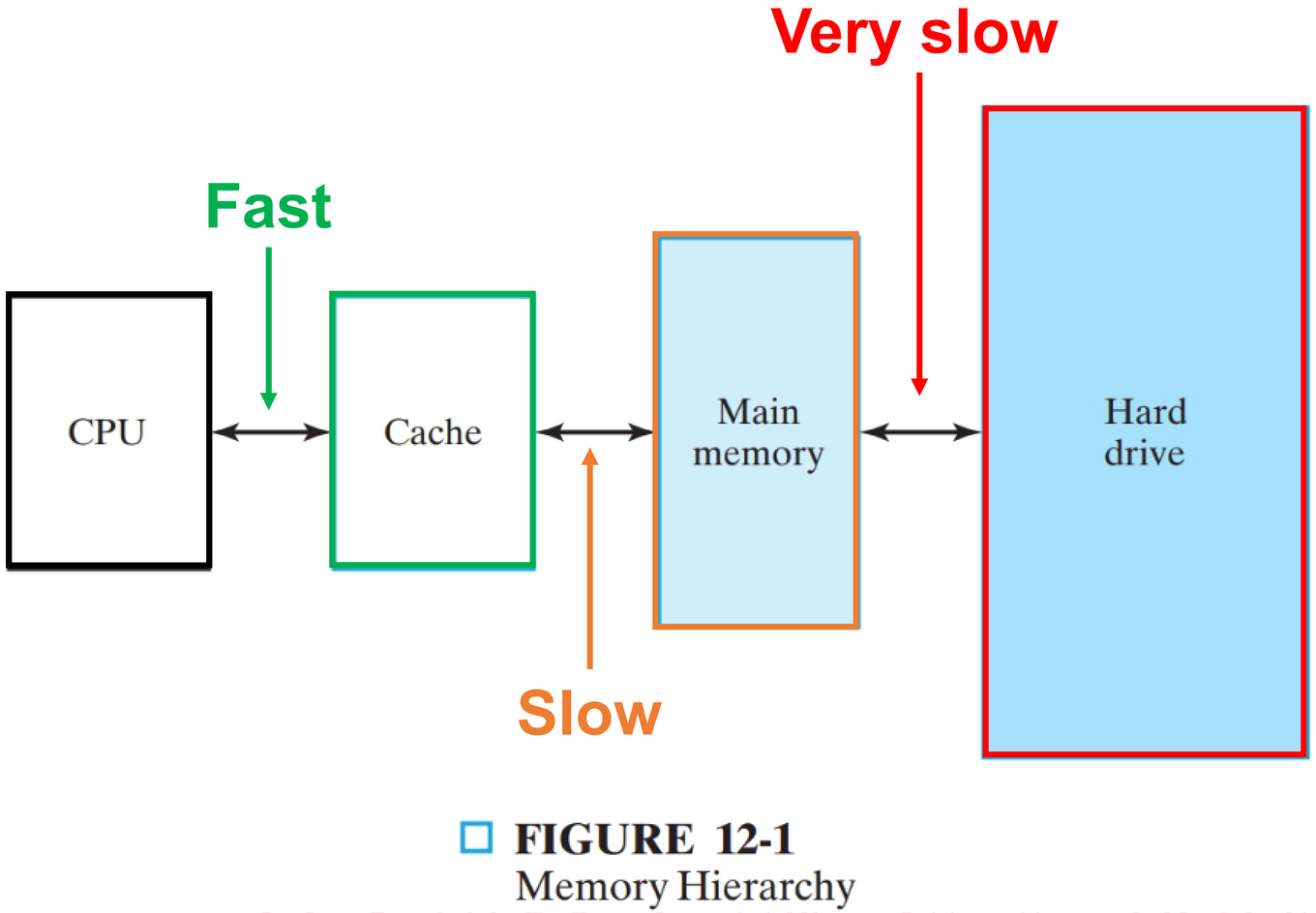


FIGURE 12-1
Memory Hierarchy

Gerarchia di memoria

Funzionamento della gerarchia:

- Ci si aspetta che la maggior parte di istruzioni e operandi per la **CPU** sia presente in **Cache**
- Quelle istruzioni non presenti in **Cache**, sono recuperate dalla **Main Memory**
- Se raramente le istruzioni non sono presenti nella **Main Memory**, la CPU accede all'**Hard Drive**

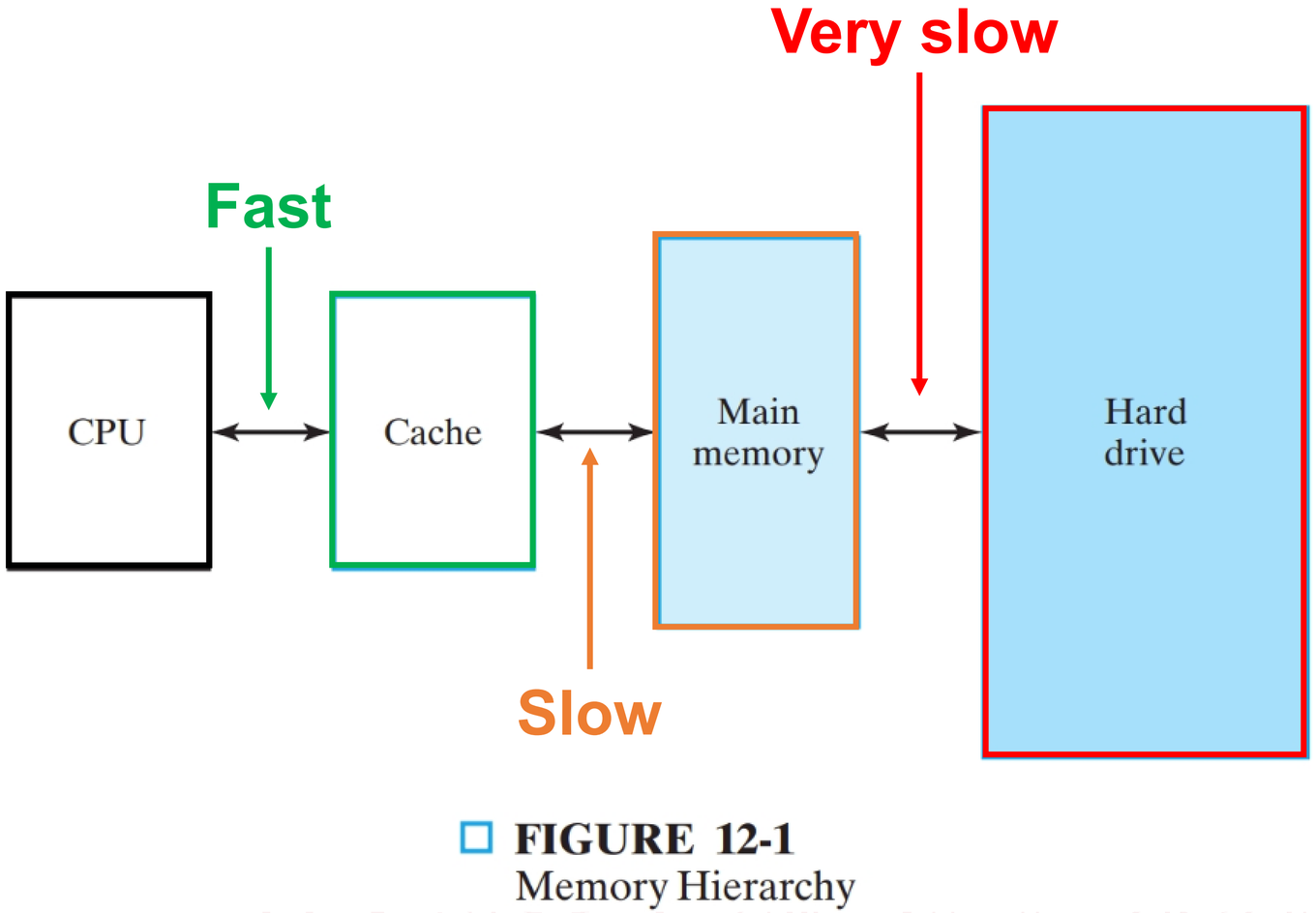
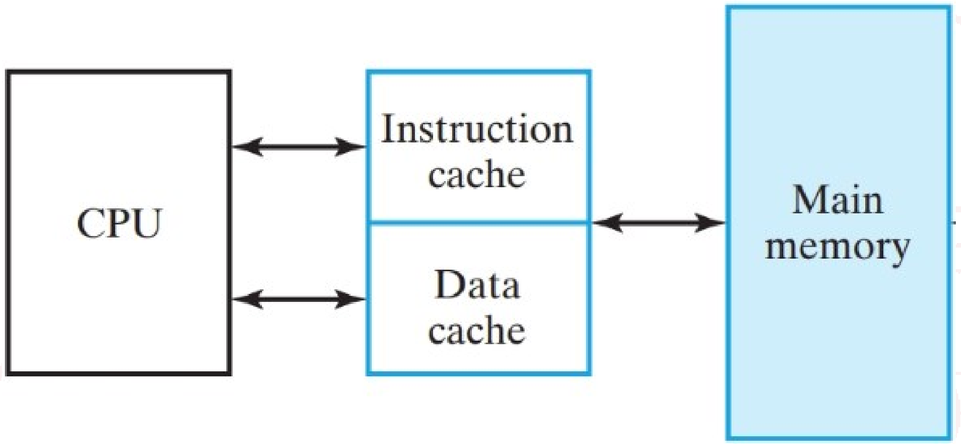


FIGURE 12-1
Memory Hierarchy

- Quindi la maggior parte delle volte la CPU accede solo alla cache (memoria veloce)
- Tempo medio di recupero dati ≈ tempo di accesso alla cache

Costo medio di accesso alla memoria

- T_1: tempo per accedere ad un dato in cache
- T_2: tempo per accedere ad un dato in memoria
- $T_1 << T_2$
- p = percentuale degli accessi a dati in cache (**hit rate**)



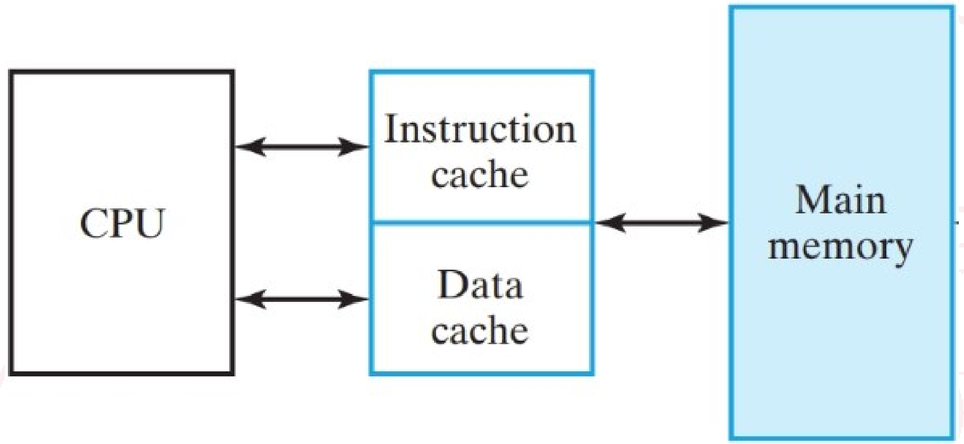
- Costo medio T di un accesso:

$$T = T_1\, p + T_2\,(1\text{-}p)$$

Costo medio di accesso alla memoria

Esercizio:

- Cache hit: p = 95%
- Costo accesso cache T_1 = 0.01 **µs**
- Costo accesso memoria T_2 = 0.1 **µs**



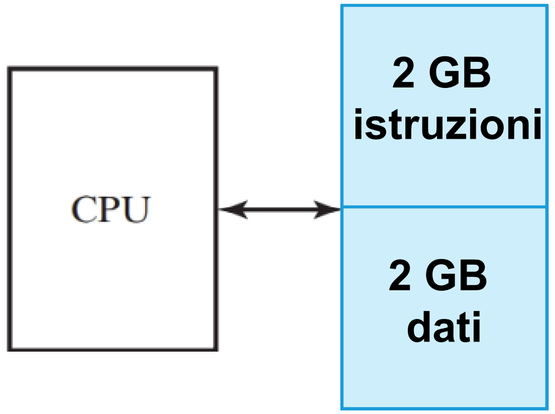
- Costo medio

$T = T_1 p + T_2 (1-p) =$

= 0.95 * 0.01 **µs** + 0.05 * 0.1 **µs** = 0.0145 **µs**

Esempio accesso memoria cache

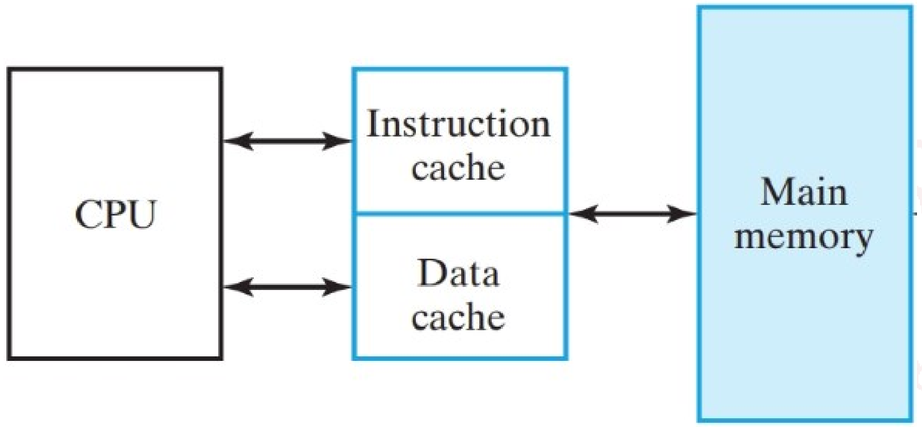
Accesso diretto alla RAM



- CPU ←→RAM di 4GB x 32
- $CLK_{CPU} = 1GHz \rightarrow T_{CPU} = 1\ ns$
- 2 moduli da 2GB: per istruzioni e dati
- Operazione di recupero e immagazzinamento: $t_{RAM} = 10\ ns$
- $t_{RAM} \gg T_{CPU}$

Troppo lenta e troppo costosa

Accesso con cache



- $t_{Cache} = 2\ ns, t_{RAM} = 10\ ns$
- 95% degli accessi fatti in cache, 5% in RAM
- $0.95 \cdot 2\ ns + 0.05 \cdot 10\ ns = 2.4\ ns \approx T_{CPU}$

- $t_{HD} = 13\ ms = 1.3 \cdot 10^7 ns$
- 95% in cache, 4.999995% RAM, 0.000005% HD
- $0.95 \cdot 2\ ns + 0.04999995 \cdot 10\ ns + 5 \cdot 10^{-8} \cdot 1.3 \cdot 10^7 ns = 3.05\ ns \approx T_{CPU}$

Quando la gerarchia di memoria funziona?

L'assunto per il successo della gerarchia di memoria è che le informazioni necessarie siano presenti in cache:

- Se la word W è presente (**cache hit**), la word viene consegnata al processore in pochi cicli di clock
- Se la word non è presente (**cache miss**), un blocco di word consecutive contenenti W viene caricato in cache e la word consegnata al processore

- Quando l'assunto viene verificato:
  - il processore riutilizza frequentemente gli stessi dati
    → **Località di riferimento temporale**
  - il processore richiede dati "vicini" in memoria in un breve intervallo di tempo
    → **Località di riferimento spaziale**

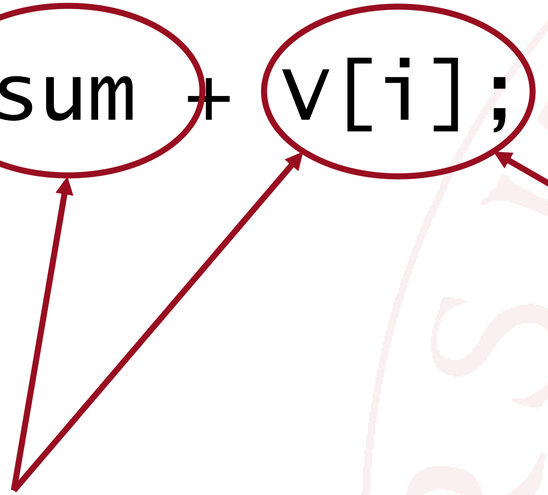
Esempi di località di riferimento

Somma degli elementi di un vettore V di n elementi:

```
int sum = 0
for(int i =0; i<n; i++) {

    sum = sum + V[i];

}
```



Località spaziale: il vettore *V* viene letto una posizione dopo l'altra.

Si assume che gli indirizzi dei dati siano contigui in memoria

Il loro indirizzo viene riutilizzato in un breve arco temporale

Località temporale: le variabili *sum* e *i* vengono usate ad ogni iterazione

Località temporale

- Se i dati più utilizzati vengono mantenuti in cache, allora l'accesso risulta più veloce.
- I dati meno utilizzati risiedono in memoria: il loro accesso costa di più, ma vengono richiesti raramente
- Problemi:
  - La cache ha una dimensione limitata
  - Non è noto quali dati verranno utilizzati più frequentemente nel futuro

Esempio for loop:

- Al primo accesso, le variabili sum e i sono caricate in cache dalla RAM
- Gli accessi seguenti, utilizzeranno solo l'accesso in cache

Località spaziale

- Dati "vicini" in memoria significa dati con indirizzi vicini
- Quando si accede in memoria per un dato all'indirizzo W, vengono caricate anche **le word "vicine"**
  (ad esempio quelle ad indirizzo W, W+1, W+2,…, W+K)
- **Block transfer**: viene trasferito un blocco di K words

Esempio for loop:

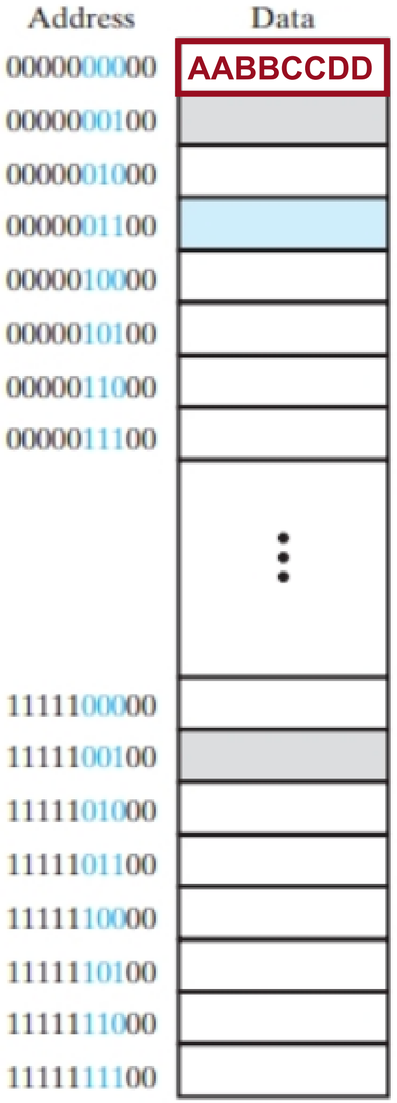
- Al primo accesso, il primo elemento del vettore V e gli elementi «vicini» sono caricati in cache
- Gli accessi seguenti, utilizzeranno solo l'accesso in cache

Tipi di memorie cache

- Una memoria cache che funziona in modo ottimale tende a:
  - Massimizzare i **cache hit**
  - Minimizzare i **cache miss**
- Questo si ottiene cercando di sfruttare al massimo i concetti di località di riferimento
- In altri termini, si cerca di avere in cache le istruzioni/dati che si stima vengano utilizzati negli intervalli di tempo successivi

- Per ottimizzare ciò, esistono diversi tipi di memorie cache:
  - Cache **a mappatura diretta**
  - Cache **completamente associativa**
  - Cache **associativa a k-vie**

Locazioni e bit di indirizzo

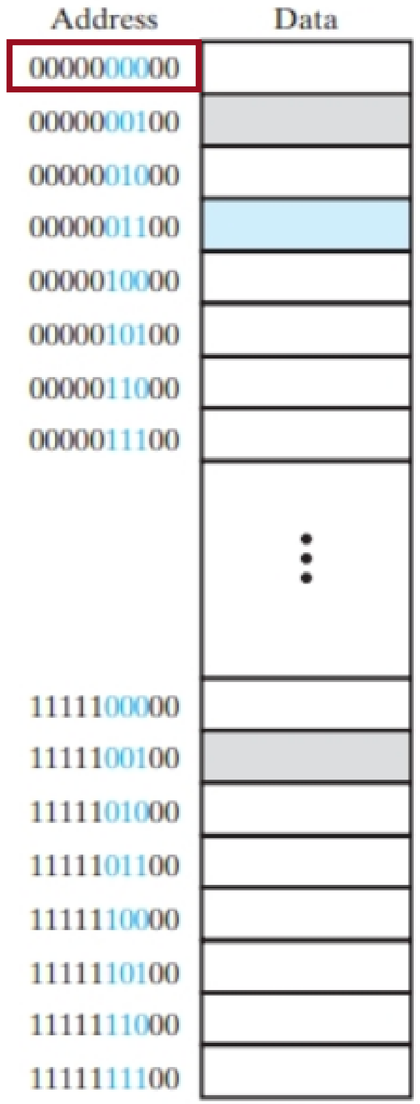


- Una memoria è divisa in **locazioni**
- Ogni locazione contiene **L** bits
  - In genere L = 8 bits = 1 byte, ma in alcuni casi L = 32 bits = 4 bytes
- Nel caso di L=32 bits, la word contenuta in memoria è di 4 bytes

Esempio:

- Memoria da 1KB con 32 bits per locazione
- 1 word = 4 bytes = 32 bits
- La memoria conterrà: $\frac{1024\ bytes}{4\ bytes} = 256\ words$

Locazioni e bit di indirizzo



- Ogni locazione della memoria è individuata da un **indirizzo**
- Con un indirizzo da **n bit** sono indirizzabili un totale di **2^n locazioni** (nota: 2^{10} = 1024 = 1K, 2^{20} = 1M, 2^{30} = 1G)

Esempio:

- Memoria da 1KB con 32 bits per locazione
- La memoria contiene $1024\ byte$
- Per indirizzare tutti i bytes in memoria ho bisogno di un indirizzo con lunghezza $\log_2 1024 = 10\ bit$
- La memoria conterrà: $\frac{1024\ bytes}{4\ bytes} = 256\ words$
- Per indirizzare i word mi servono meno bit: $\log_2 256 = 8\ bit$

Il problema di costruire una CACHE

- Una memoria cache NON può memorizzare TUTTE le righe della RAM
- È necessario trovare un modo per «ricordarsi» quali righe della RAM sono nella cache e quali no:
- Nella cache è necessario memorizzare:
  - Il dato
  - L'indirizzo di memoria di questo dato

Indirizzo occupato in memoria	Dato
0101 0101 01	**AABBCCDD**

10 bit per l'indirizzo

(1024 word)

32 bit di dato (1 word, 4 byte)

42 bit per una riga di cache

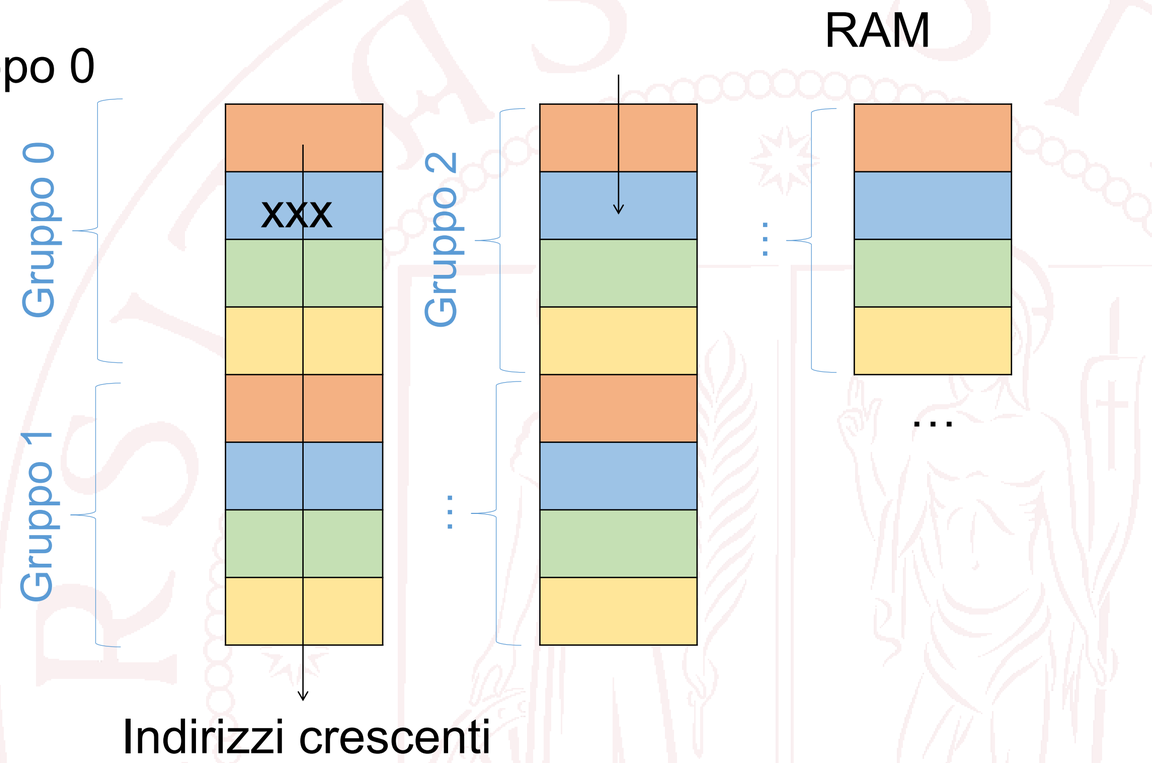
Usiamo tanti bit!
Sarebbe preferibile trovare un modo per risparmiarne scrivendo l'indirizzo in modo più compatto!

Il problema di costruire una CACHE

- Possiamo risparmiare memoria usando il «numero di riga» della cache come indirizzo implicito
- E' come se dividessimo l'intera memora in gruppi di celle

Riga 1 del gruppo 0

Riga cache	Sub indirizzo all'interno del gruppo	
0		
1	Gruppo 0	xxx
2		
3		



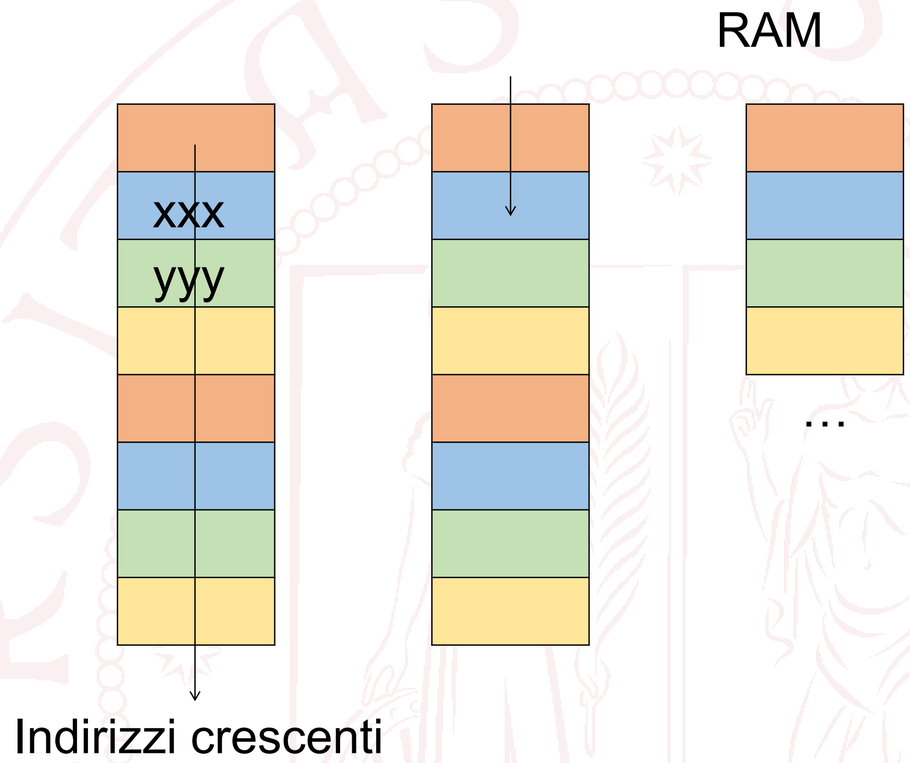
- con questa cache da con 4 righe, divido la memoria in M gruppi di 4 celle
- come indirizzo mi basta segnare l'indirizzo del gruppo (che è più corto dell'indirizzo della cella)

Il problema di costruire una CACHE

- Possiamo risparmiarne ancora usando il «numero di riga» della cache come indirizzo implicito
- E' come se dividessimo l'intera memora in gruppi di celle

Riga 2 del gruppo 0

Riga cache	Sub indirizzo all'interno del gruppo	
0		
1	Gruppo 0	xxx
2	Gruppo 0	yyy
3		



- con questa cache da con 4 righe, divido la memoria in M gruppi di 4 celle
- come indirizzo mi basta segnare l'indirizzo del gruppo (che è più corto dell'indirizzo della cella)

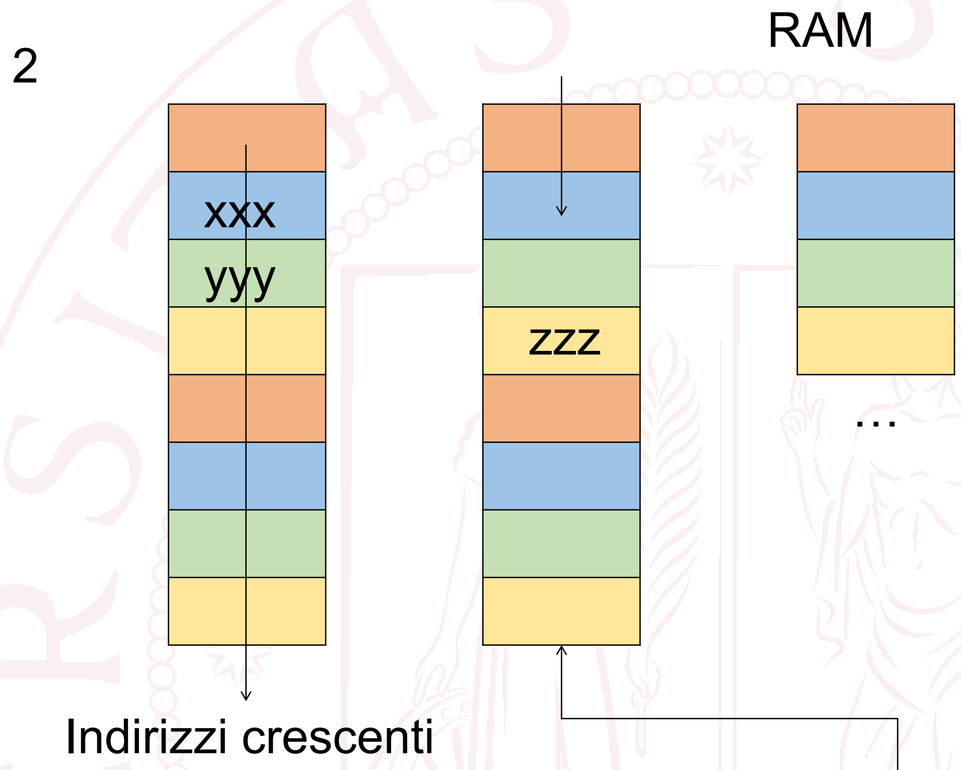
Il problema di costruire una CACHE

- Possiamo risparmiarne ancora usando il «numero di riga» della cache come indirizzo implicito
- E' come se dividessimo l'intera memora in gruppi di celle

Riga 3 del gruppo 2

Riga cache	Sub indirizzo all'interno del gruppo	
0		
1	Gruppo 0	xxx
2	Gruppo 0	yyy
3	Gruppo 2	yyy

- con questa cache da con 4 righe, divido la memoria in M gruppi di 4 celle
- come indirizzo mi basta segnare l'indirizzo del gruppo (che è più corto dell'indirizzo della cella)



Cache a mappatura diretta

Consideriamo una **memoria cache** a 32 bit con 8 words (4 bytes ciascuna) e una **memoria RAM** da 1KB (256 words)

L'idea alla base della mappatura diretta è di associare direttamente **gruppi** di locazioni della **RAM** con indirizzi consecutivi all'interno della **cache** utilizzando lo stesso **indice**

- Nel nostro caso la cache può contenere solo 8 words della memoria RAM
- I primi 5 bits dell'indirizzo identificano il **gruppo (TAG)** di appartenenza
- I seguenti 3 bits dell'indirizzo identificano la locazione (**INDEX**) all'interno della cache
- Esiste un'**associazione diretta** fra i bits 4, 3, 2 dell'indirizzo e l'indice nella cache
- Esempio: il dato all'indirizzo **00000011100** si troverà alla locazione di cache con **INDEX 011**

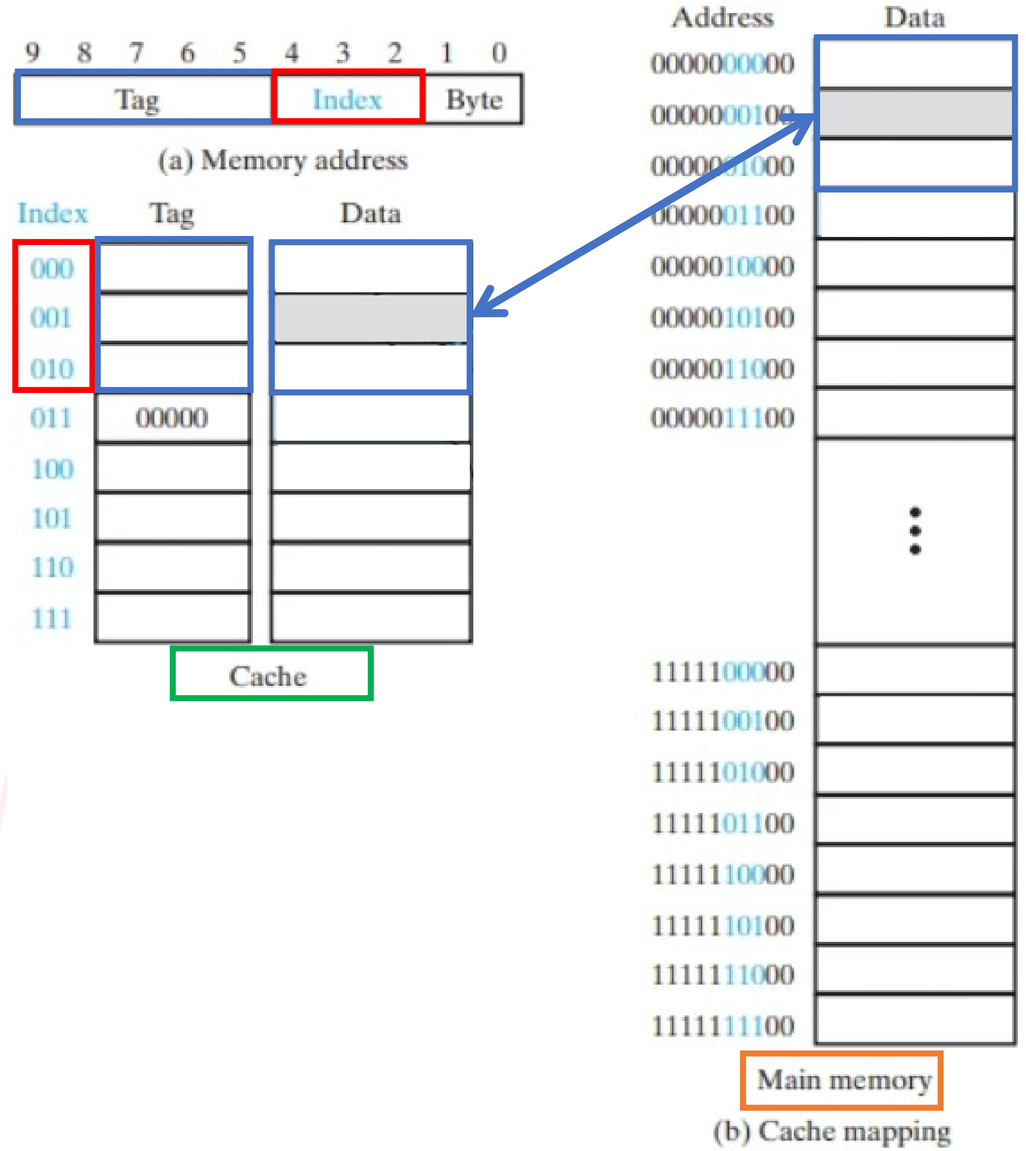


FIGURE 12-3
Direct Mapped Cache

Cache a mappatura diretta: funzionamento

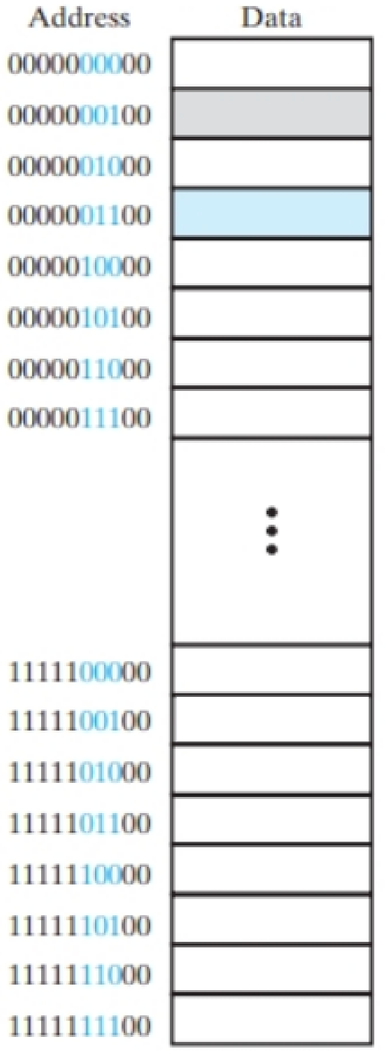
- La CPU richiede un'istruzione dall'indirizzo:

0	0	0	0	0	0	1	1	0	0

- L'istruzione può essere in cache o in memoria
- La cache separa il **TAG** e l'**INDEX** (00000 e 011)
- Recupera la linea nella cache relativa all'**INDEX**
- Compara il **TAG** recuperato con quello dell'istruzione

(a) Memory address

Cache



Main memory

(b) Cache mapping

È uguale
Il dato recuperato
è quello desiderato
Cache Hit

CPU

È diverso
Il dato non è
presente in cache
Cache Miss

Richiesta a memoria

Viene caricato un nuovo gruppo di dati dalla memoria alla cache

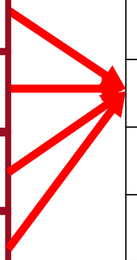
FIGURE 12-3
Direct Mapped Cache

Cache a mappatura diretta: pro e contro

- **Pro**:
  - Organizzazione semplice
- **Contro**:
  - La linea da sovrascrivere è predeterminata: un blocco non può essere inserito in una linea scelta tra quelle che presumibilmente non servono più
  - Hit rate decisamente più basso (minore efficienza).

Esempio: La CPU richiede l'accesso ai seguenti indirizzi, la cache è inizialmente vuota

Indirizzi richiesti	
1	0000001000
2	0010001000
3	0100101000
4	1001001000



INDEX	TAG	DATA
000		
001		
010	00100	
011		
100		
101		
110		
111		

1. Carica il dato all'indice **010**
2. **TAG** diverso, cancella il dato con **indice 010**, e ricarica il nuovo dato con **TAG 00100** (anche con linee libere)
3. **TAG** diverso, cancella il dato con **indice 010**, e ricarica il nuovo dato con **TAG 01001** (anche con linee libere)
4. **TAG** diverso, cancella il dato con **indice 010**, e ricarica il nuovo dato con **TAG 10010** (anche con linee libere)

Cache completamente associativa

Consideriamo una **memoria cache** a 32 bit con 8 words (4 bytes ciascuna) e una **memoria RAM** da 1KB (256 words)

L'idea alla base della cache completamente associativa è che ogni indirizzo in memoria può essere associato arbitrariamente a qualsiasi locazione della cache

- Il **TAG** viene identificato dagli 8 bits più significativi
- Se dopo una richiesta da parte della CPU il **TAG** viene trovato nella cache, il dato corrispondente viene caricato
- Se il **TAG** non viene trovato, il dato viene richiesto in memoria e viene anche salvato nella cache
- Il nuovo **TAG** viene inserito in una linea vuota
- Se non ci sono linee vuote, si sostituisce un **TAG** «vecchio» con una politica di rimpiazzo

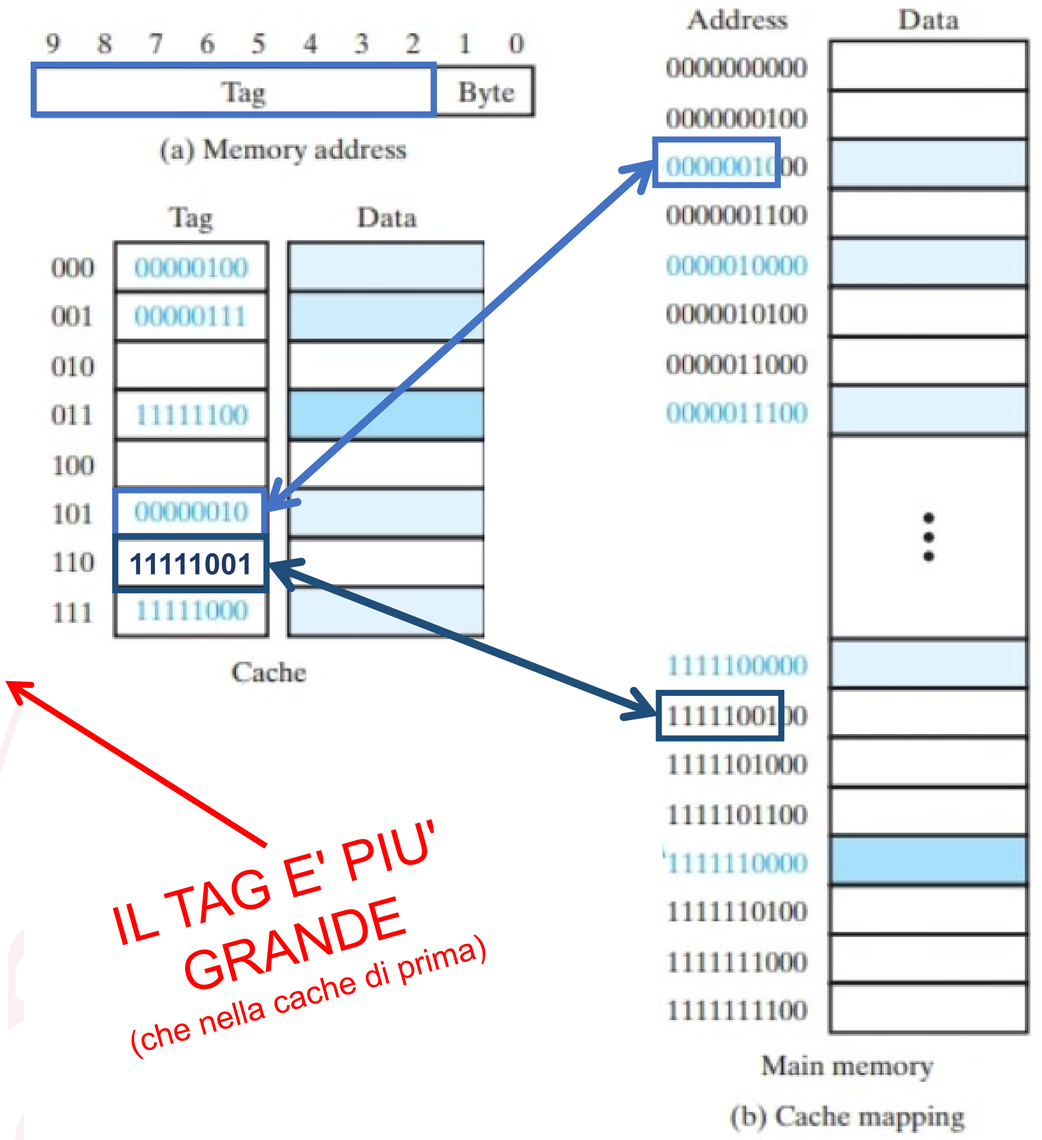


FIGURE 12-4
Fully Associative Cache

Cache completamente associativa: funzionamento

- La CPU richiede un'istruzione dall'indirizzo:

- L'istruzione può essere in cache o in memoria
- La cache separa il **TAG** e il dato
- Cerca il **TAG** all'interno della cache

Viene trovato

Cache Hit

CPU

Non viene trovato

Cache Miss

Richiesta a memoria

CPU + CACHE

(in una riga libera)

Scenario 1

TAG	DATA
01000100	
11000100	
11111100	
00000011	
00000100	

Scenario 2

TAG	DATA
01000100	
11000100	
11111100	
00000011	
00001011	
00000100	

Cache completamente associativa: metodi di rimpiazzo

Poichè nella cache completamente associativa un nuovo dato può essere inserito ovunque …

- Bisogna decidere DOVE metterlo.
- Possibilmente si sceglie una riga vuota, ma se la cache è già piena, bisogna scegliere quale riga sovrascrivere!
- Esistono vari modi di rimpiazzo:
  1. **Random replacement**: rimpiazzo casuale di un elemento attualmente in cache
  2. **FIFO replacement**: l'elemento più vecchio presente in cache
  3. **Least recently used (LRU)**: l'elemento usato meno recentemente

Cache completamente associativa: TAG search

Poichè nella cache completamente associativa un nuovo dato può essere inserito ovunque …

- Per vedere se abbiamo un hit o un miss, dobbiamo cercare tra tutti i TAG!
- La ricerca del **TAG** viene velocizzata con l'utilizzo di una memoria associativa
- **Memoria associativa** o **CAM** (Content Addressable Memory) è una memoria in grado di effettuare, in parallelo, il confronto tra un dato cercato e tutti i dati in essa contenuti.

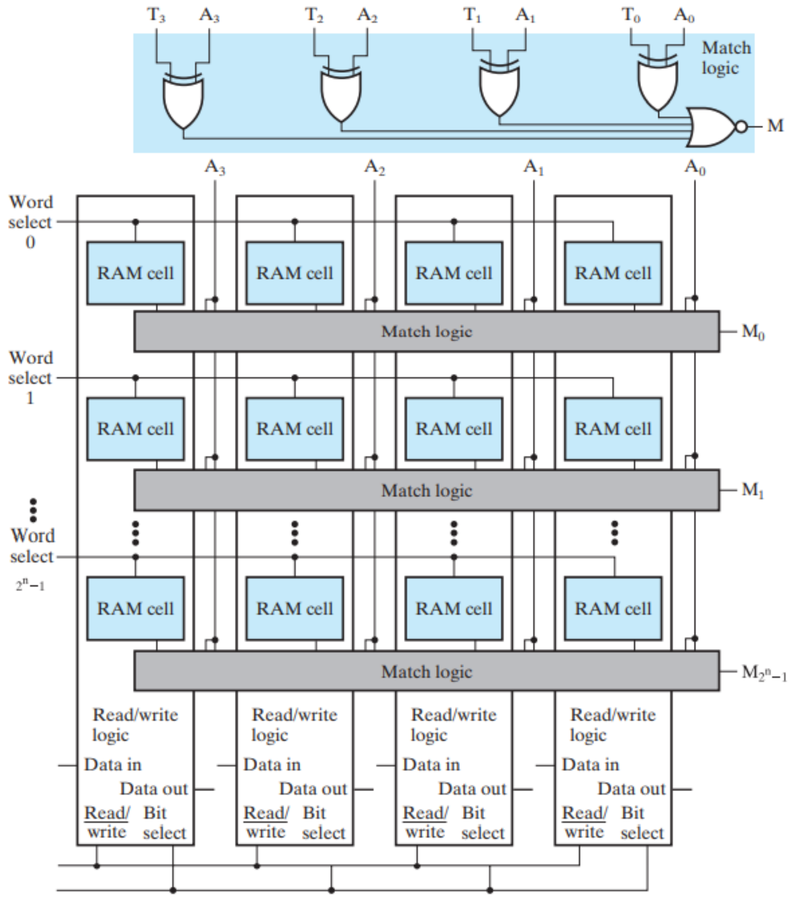


FIGURE 12-5
Associative Memory for 4-Bit Tags

CAM: Content Addressable Memory

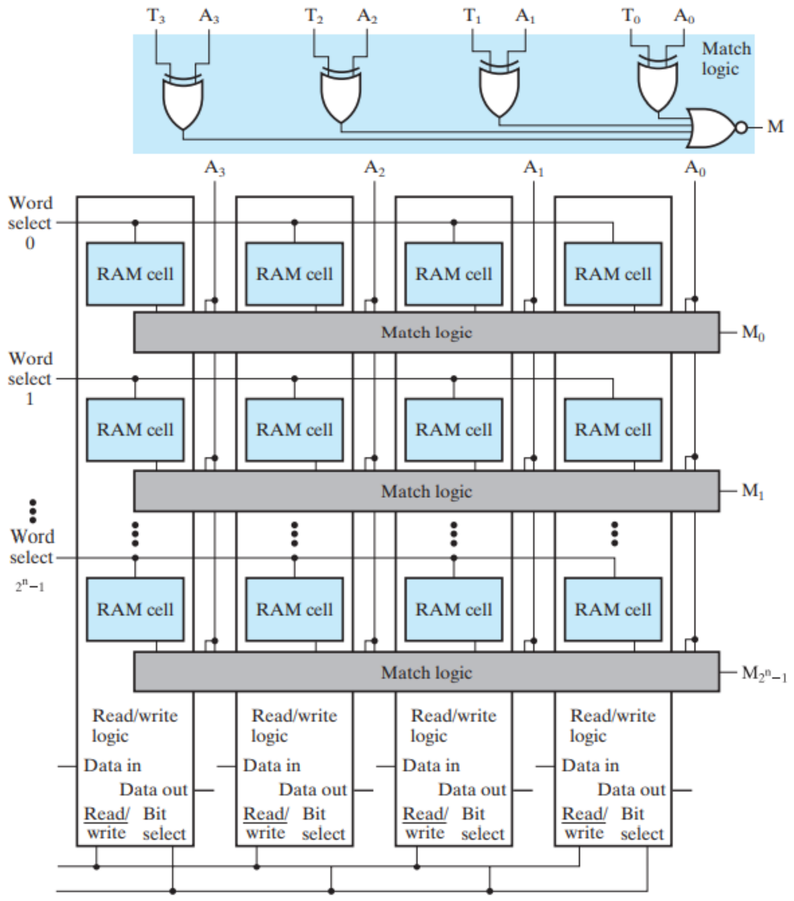
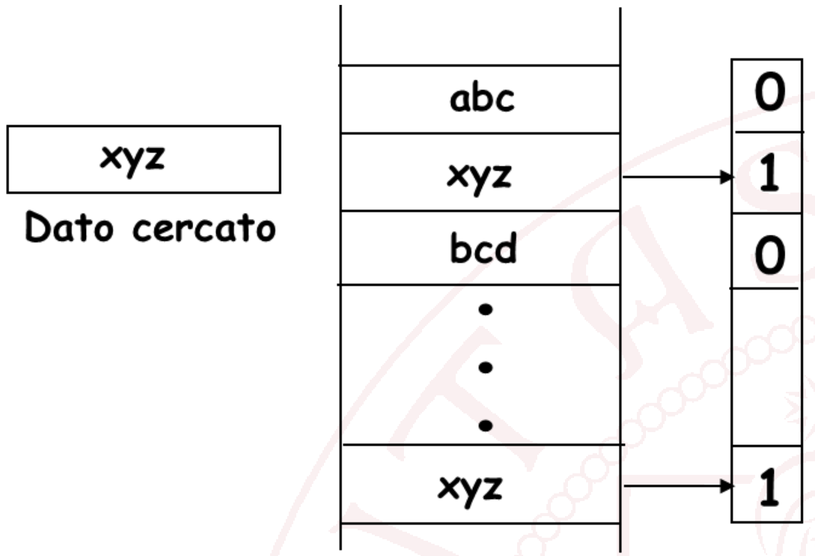


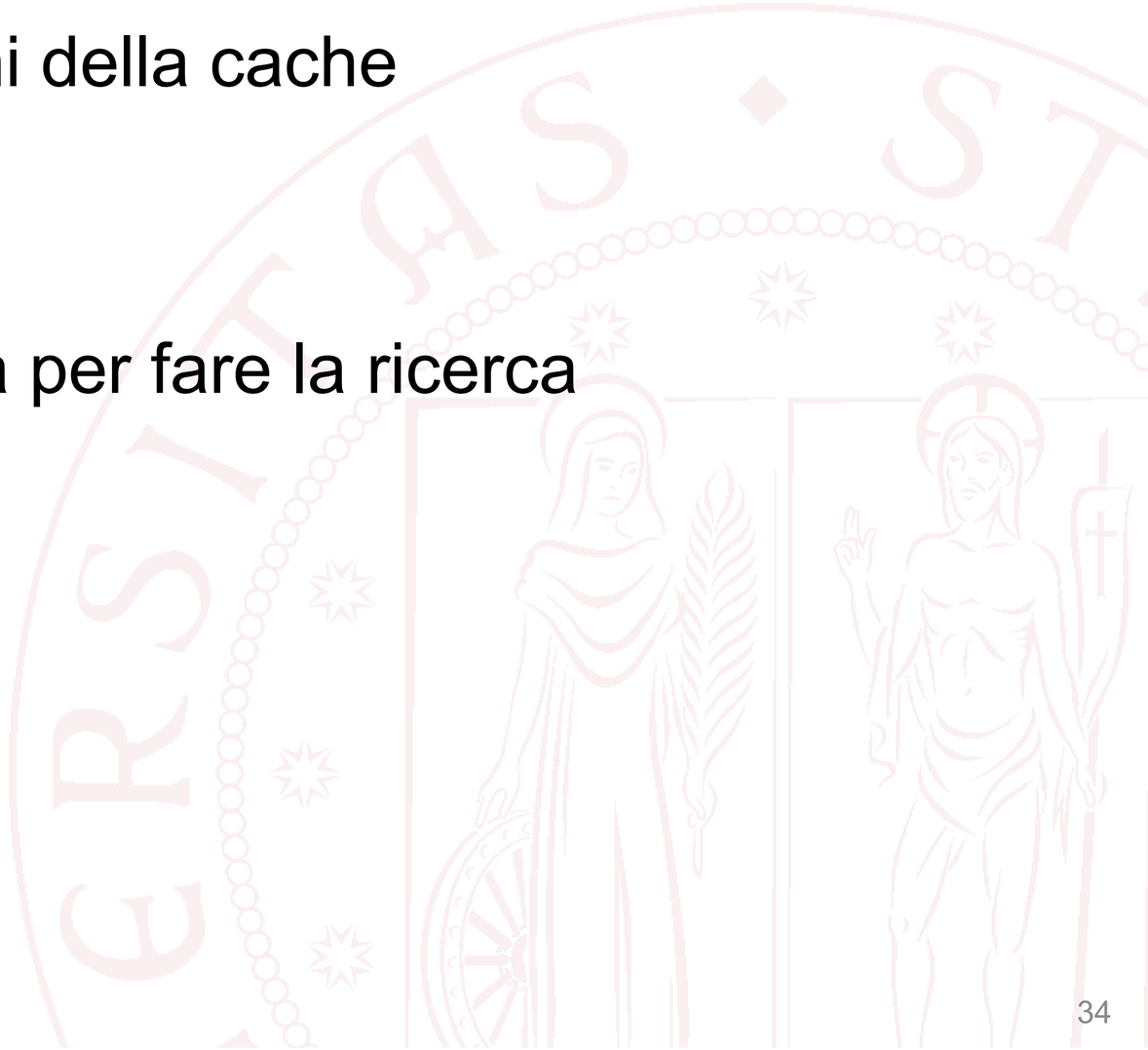
FIGURE 12-5
Associative Memory for 4-Bit Tags



La CAM ci dice se il dato che vogliamo è già presente in cache confrontando il suo TAG con tutti i TAG nella cache.
Se lo abbiamo → HIT
Altrimenti → MISS

Cache completamente associativa: pro e contro

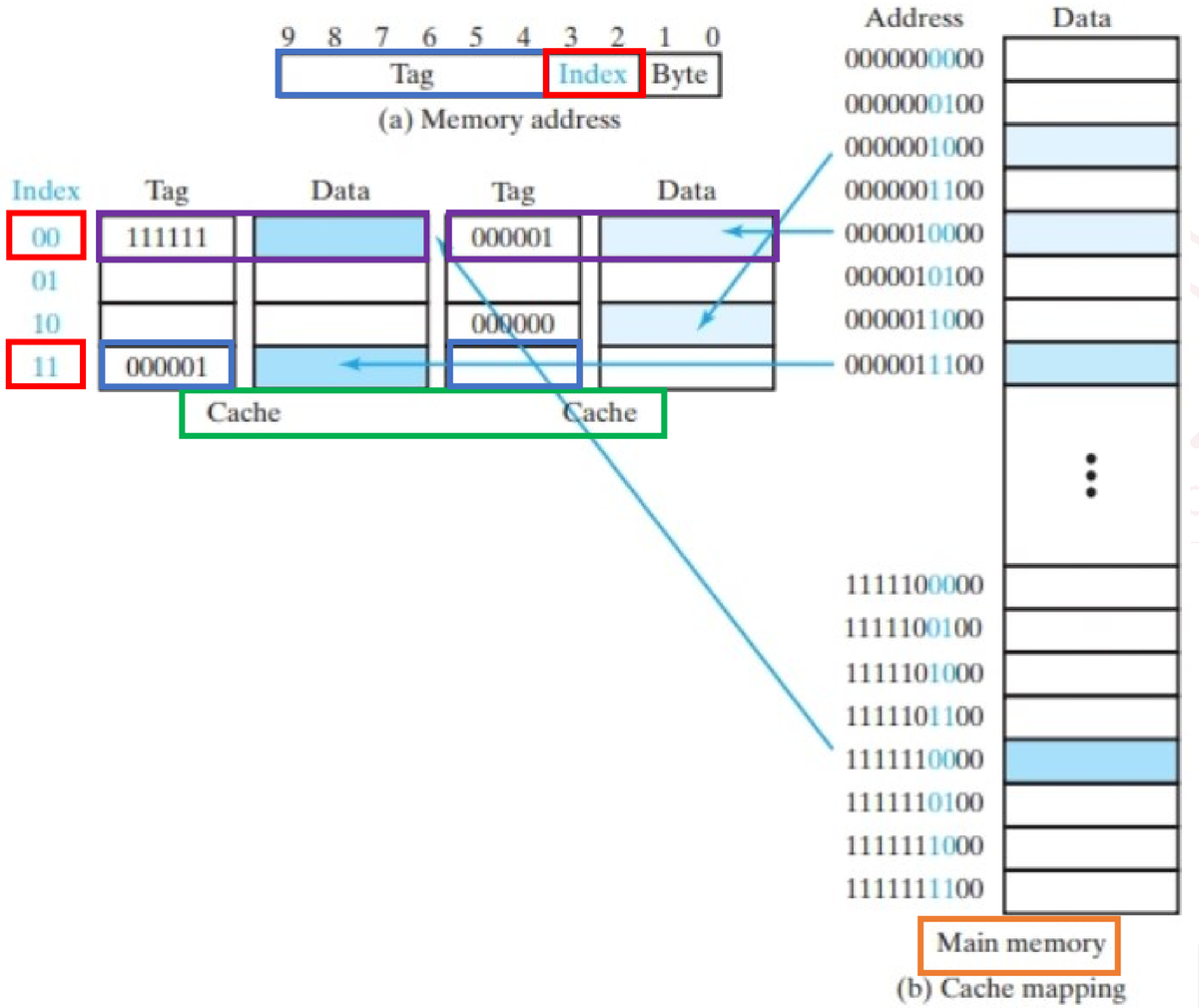
- **Pro**:
  - Organizzazione relativamente semplice
  - Possibilità di sfruttare tutte le locazioni della cache
- **Contro**:
  - Tanti bit nei tag
  - Necessita di una memoria associativa per fare la ricerca
  - Molto complessa da realizzare
  - Molto costosa



Cache associativa a k-vie

Consideriamo una **memoria cache** a 32 bit con 8 words (4 bytes ciascuna) e una **memoria RAM** da 1KB (256 words)

L'idea alla base della cache associativa a k-vie è di associare un **INDEX** a più locazioni (< delle locazioni della CACHE) in modo da velocizzare il **TAG** match. È a metà strada fra la mappatura diretta e quella completamente associativa.



- Per ogni **INDEX** c'è un **SET** s (vie) di locazioni (es. con s=2 → a 2 vie)
- Struttura della cache: 4 righe, 2 colonne
- Per ogni **INDEX** (riga), si associano 2 **TAGs** (colonne)
- Una volta identificato **l'INDEX**, la ricerca del **TAG** viene fatta sul numero di colonne

Cache associativa a k-vie: funzionamento

- La CPU richiede un'istruzione dall'indirizzo:

- La cache separa il **TAG** e l'**INDEX** (000000 e 11)
- Vengono recuperati i **TAGs** in cache relativi all'indice 11
- Si comparano i **TAGs** con l'istruzione richiesta

INDEX	TAG	DATA
00		
01		
10		
11	001000	
	000100	

2 vie

Viene trovato

Cache Hit

CPU

Non viene trovato

Cache Miss

Richiesta a memoria

Cache entry: numero di words

- È stato ipotizzato che ogni entry della cache includesse un **TAG** e una **WORD**
- In casi reali, viene sfruttata la località spaziale e più di una **WORD** viene inclusa in una cache entry
- Quindi quando succede un cache miss, viene caricato in cache un blocco di L **WORDs**, chiamato **LINE**

Questo va a braccetto con le SDRAM!

Esempio:

- Una **LINE** è composta da 4 **WORDs**
- Cache miss per l'indirizzo **0000 01** 0000
- Tutte le **WORDs** da «000001 0000» a «000001 1100» vengono caricate simultaneamente nel blocco della cache identificato dall'**INDEX 01** e dal **TAG 0000**

Cache associativa a 2-vie con 4-word lines

0	0	0	0	0	0	1	1	0	0

- L'indirizzo a 10 bits viene diviso in **TAG**, **INDEX**, **WORD**, Data
- Per ogni **TAG**, c'è un blocco (**LINE**) di 4 words identificate dai bits 3 e 2
- L'**INDEX** viene applicato per identificare 2 **TAGs**
- L'**INDEX** e **WORD** vengono applicate per leggere le 2 words associate dai **TAGs**
- I 2 **TAGs** vengono confrontati con l'indirizzo richiesto
- Se c'è un **MATCH**, la word viene fornita alla CPU
- Se non c'è un **MATCH**, si usa il **TAG** e la **WORD** per caricare tutta la **LINE** (4 words) nella cache

↓

Località spaziale

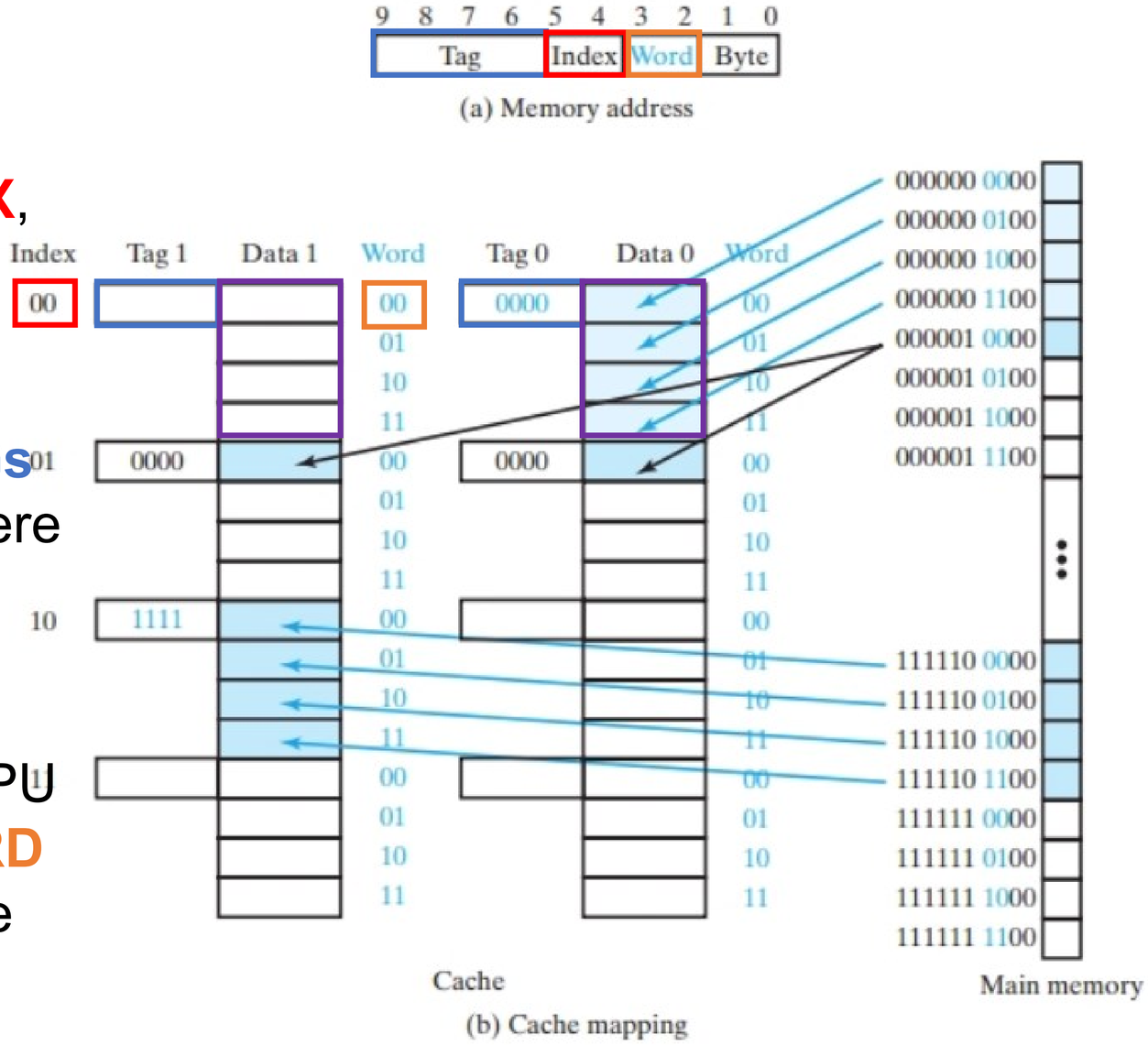


FIGURE 12-8
Set-Associative Cache with 4-Word Lines

Metodi di scrittura nella cache

- Finora abbiamo considerato le words nella cache come copie di quelle nella memoria centrale utilizzate per velocizzare l'accesso in lettura
- Tuttavia il discorso cambia se vogliamo anche scrivere una word che ovviamente non può essere scritta solo nella cache

- **Metodo write-through:**
  - Il risultato viene sempre scritto sia nella cache che nella memoria centrale
  - Simmetria fra cache e memoria centrale
  - Può rallentare le operazioni

- **Metodo copy-back:**
  - Il risultato viene scritto in cache in caso di Cache Hit
  - Il risultato viene scritto in memoria in caso di Cache Miss (write miss)
  - (e anche in cache (write-allocate) con la speranza di avere hit successivi)
  - Mancanza simmetria Cache ←→ Memoria: i dati in cache vanno ricopiati in memoria prima di liberare la cache!

Bit di controllo nella cache: controllo scrittura

- Nel caso di metodo di scrittura copy-back, è necessario tenere traccia se un elemento in cache è stato scritto oppure no
- Perché?
  - In modo da scrivere in memoria solo quando è necessario e non ogni volta che c'è un Cache Miss
- Si usa un bit di controllo chiamato **dirty bit**:

Bit di controllo nella cache: controllo scrittura

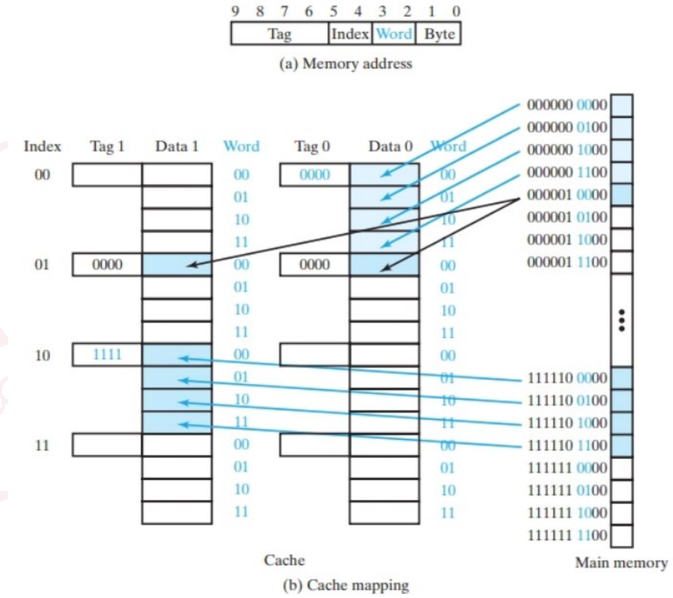
- Nel caso di metodo di scrittura copy-back, è necessario tenere traccia se un elemento in cache è stato scritto oppure no
- Perché?
  - In modo da scrivere in memoria solo quando è necessario e non ogni volta che c'è un Cache Miss
- Si usa un bit di controllo chiamato **dirty bit**:
  - Se è uguale a 1 significa che la LINE in cache è stata scritta (e deve essere scritta anche in memoria)
  - Se è uguale a 0 significa che non serve la scrittura in memoria

Bit di controllo nella cache: Bit di validità

- Ipotizziamo che la cache sia vuota, i valori al suo interno sono indefiniti
- A seguito di una richiesta della CPU, può comunque succedere di avere un **Cache Hit**
- Tuttavia in questo caso, la word fornita alla CPU non sarebbe valida
- Viene quindi introdotto un **bit di validità** che indica che la LINE della cache è valida
- Il **bit di validità** viene letto insieme al TAG

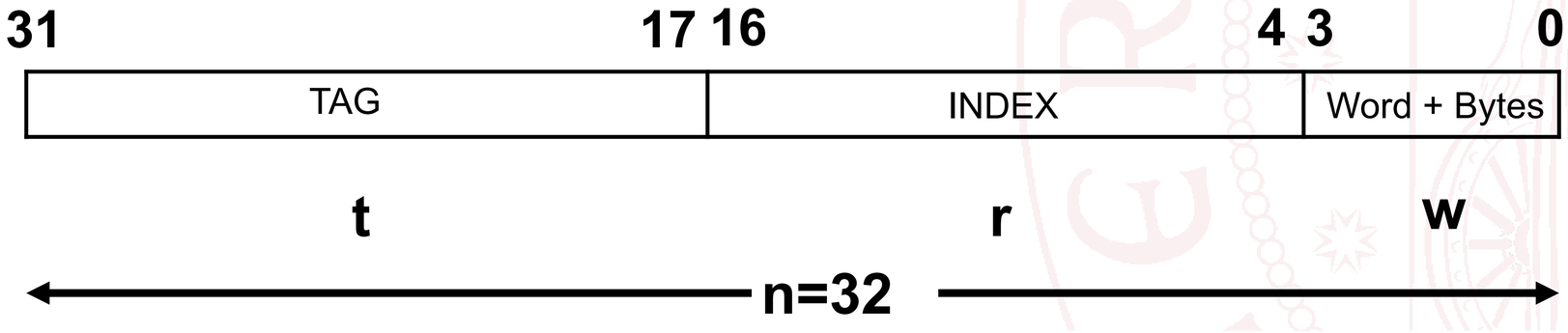
Esempio di cache associativa a 2-vie con 4-words con write-through

- Memoria cache con 256 KB e 4 words per line e indirizzo a 32 bits, word di 32 bit
- Quanto grande è il TAG?
- $N_{bytes} = 256 \cdot 1024 = 262144\ bytes$ totali nella cache
- $N_{bytes \times word} = 4\ bytes$ ogni word contiene 4 bytes
- $N_{words} = \frac{N_{bytes}}{N_{bytes \times word}} = \frac{252144}{4} = 65536\ words$
- $N_{bytes \times line} = N_{words \times line} * N_{bytes \times word} = 16$
- $N_{lines} = \frac{N_{words}}{N_{words \times line}} = \frac{65536}{4} = 16384$
- $N_{vie} = 2 \rightarrow N_{lines \times via} = 8192$
- $n - w - r = t = 32 - 4 - 13 = 15$



Per identificare 8192 line ho bisogno di:
$\log_2 8192 = 13\ bits$

Per identificare 16 bytes ho bisogno di:
$\log_2 16 = 4\ bits$



Esempio

Descrizione:

ESEMPIO
PRESO DA UNA
PROVA D'ESAME

Un computer a ciclo singolo ha le seguenti caratteristiche:
- Clock di 2 MHz;
- 16 bits address lines;
- Memoria istruzioni di 16 KB SDRAM
- Memoria dati di 32 KB SDRAM
- Memoria indirizzata a word di 16-bit
- C'è una memoria cache a mappatura diretta con 2 linee collegata alla memoria dati. Ogni linea contiene 8 Byte.

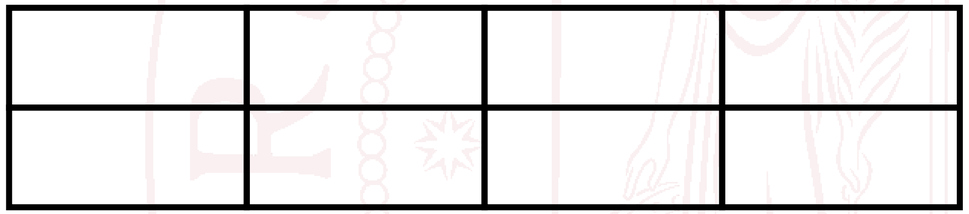
DISEGNARE LO SCHEMA DELLA CACHE COSI DESCRITTA

Esempio

Descrizione:

Un computer a ciclo singolo ha le seguenti caratteristiche:
• Clock di 2 MHz;
• 16 bits address lines;
• Memoria istruzioni di 16 KB SDRAM
• Memoria dati di 32 KB SDRAM
• Memoria indirizzata a word di **16**-bit
• C'è una memoria cache a mappatura diretta con 2 linee collegata alla memoria dati. Ogni linea contiene **8** Byte.

- $\frac{8*8\ bit}{16\ bit} = 4\ words\ per\ linea$
- $2\ lines$



Esempio

Descrizione:

Un computer a ciclo singolo ha le seguenti caratteristiche:
- Clock di 2 MHz;
- 16 bits address lines;
- Memoria istruzioni di 16 KB SDRAM
- Memoria dati di 32 KB SDRAM
- Memoria indirizzata a word di 16-bit
- C'è una memoria cache a mappatura diretta con 2 linee collegata alla memoria dati. Ogni linea contiene 8 Byte.

C)

RAM 16 bits data	
0x0000	4
0x0001	3
0x0002	8
0x0003	2
0x0004	1
0x0005	3
0x0006	2
0x0007	8
0x0008	10
0x0009	2
0x000A	2
0x000B	8
0x000C	11
0x000D	18

VIENE DATO IL CONTENUTO DELLA MEMORIA E DELLE ISTRUZIONI DA ESEGUIRE: BISOGNA CALCOLARE IL NUMERO DI HIT, MISS, REPLACEMENT

Esempio

Descrizione:

Un computer a ciclo singolo ha le seguenti caratteristiche:
- Clock di 2 MHz;
- 16 bits address lines;
- Memoria istruzioni di 16 KB SDRAM
- Memoria dati di 32 KB SDRAM
- Memoria indirizzata a word di 16-bit
- C'è una memoria cache a mappatura diretta con 2 linee collegata alla memoria dati. Ogni linea contiene 8 Byte.

C)

RAM 16 bits data	
0x0000	4
0x0001	3
0x0002	8
0x0003	2
0x0004	1
0x0005	3
0x0006	2
0x0007	8
0x0008	10
0x0009	2
0x000A	2
0x000B	8
0x000C	11
0x000D	18

La cache è composta da **2 linee**.
Ogni linea contiene **4 words**, possiamo divider la memoria in blocchi

Esempio

Descrizione:

Un computer a ciclo singolo ha le seguenti caratteristiche:
- Clock di 2 MHz;
- 16 bits address lines;
- Memoria istruzioni di 16 KB SDRAM
- Memoria dati di 32 KB SDRAM
- Memoria indirizzata a word di 16-bit
- C'è una memoria cache a mappatura diretta con 2 linee collegata alla memoria dati. Ogni linea contiene 8 Byte.

C)

RAM 16 bits data	
0x0000	4
0x0001	3
0x0002	8
0x0003	2
0x0004	1
0x0005	3
0x0006	2
0x0007	8
0x0008	10
0x0009	2
0x000A	2
0x000B	8
0x000C	11
0x000D	18

Operazione di LOAD da M[3]

Esempio

Descrizione:

Un computer a ciclo singolo ha le seguenti caratteristiche:
- Clock di 2 MHz;
- 16 bits address lines;
- Memoria istruzioni di 16 KB SDRAM
- Memoria dati di 32 KB SDRAM
- Memoria indirizzata a word di 16-bit
- C'è una memoria cache a mappatura diretta con 2 linee collegata alla memoria dati. Ogni linea contiene 8 Byte.

C)

RAM 16 bits data	
0x0000	4
0x0001	3
0x0002	8
0x0003	2
0x0004	1
0x0005	3
0x0006	2
0x0007	8
0x0008	10
0x0009	2
0x000A	2
0x000B	8
0x000C	11
0x000D	18

La cache è vuota (invalid): abbiamo un MISS e dobbiamo copiare il valore in cache. Riempiamo tutta la line, copiando i 4 word del blocco

	4	3	8	**2**

Esempio

Descrizione:

Un computer a ciclo singolo ha le seguenti caratteristiche:
- Clock di 2 MHz;
- 16 bits address lines;
- Memoria istruzioni di 16 KB SDRAM
- Memoria dati di 32 KB SDRAM
- Memoria indirizzata a word di 16-bit
- C'è una memoria cache a mappatura diretta con 2 linee collegata alla memoria dati. Ogni linea contiene 8 Byte.

LOAD dei dati in M[1]

C)

RAM 16 bits data	
0x0000	4
0x0001	3
0x0002	8
0x0003	2
0x0004	1
0x0005	3
0x0006	2
0x0007	8
0x0008	10
0x0009	2
0x000A	2
0x000B	8
0x000C	11
0x000D	18

Esempio

Descrizione:

Un computer a ciclo singolo ha le seguenti caratteristiche:
- Clock di 2 MHz;
- 16 bits address lines;
- Memoria istruzioni di 16 KB SDRAM
- Memoria dati di 32 KB SDRAM
- Memoria indirizzata a word di 16-bit
- C'è una memoria cache a mappatura diretta con 2 linee collegata alla memoria dati. Ogni linea contiene 8 Byte.

C)

RAM 16 bits data	
0x0000	4
0x0001	3
0x0002	8
0x0003	2
0x0004	1
0x0005	3
0x0006	2
0x0007	8
0x0008	10
0x0009	2
0x000A	2
0x000B	8
0x000C	11
0x000D	18

La cache contiene già il valore che stiamo cercando: abbiamo un **HIT**

4	**3**	8	2

Esempio

Descrizione:

Un computer a ciclo singolo ha le seguenti caratteristiche:
- Clock di 2 MHz;
- 16 bits address lines;
- Memoria istruzioni di 16 KB SDRAM
- Memoria dati di 32 KB SDRAM
- Memoria indirizzata a word di 16-bit
- C'è una memoria cache a mappatura diretta con 2 linee collegata alla memoria dati. Ogni linea contiene 8 Byte.

LOAD dei dati in M[4]

C)

RAM 16 bits data	
0x0000	4
0x0001	3
0x0002	8
0x0003	2
0x0004	1
0x0005	3
0x0006	2
0x0007	8
0x0008	10
0x0009	2
0x000A	2
0x000B	8
0x000C	11
0x000D	18

Esempio

Descrizione:

Un computer a ciclo singolo ha le seguenti caratteristiche:
- Clock di 2 MHz;
- 16 bits address lines;
- Memoria istruzioni di 16 KB SDRAM
- Memoria dati di 32 KB SDRAM
- Memoria indirizzata a word di 16-bit
- C'è una memoria cache a mappatura diretta con 2 linee collegata alla memoria dati. Ogni linea contiene 8 Byte.

C)

RAM 16 bits data	
0x0000	4
0x0001	3
0x0002	8
0x0003	2
0x0004	1
0x0005	3
0x0006	2
0x0007	8
0x0008	10
0x0009	2
0x000A	2
0x000B	8
0x000C	11
0x000D	18

La riga che cerchiamo non è in cache: abbiamo un MISS. Come prima copiamo non solo tutto il valore ma tutto il blocco di 4 word

	4	3	8	2
	1	3	2	8

Esempio

Descrizione:

Un computer a ciclo singolo ha le seguenti caratteristiche:
- Clock di 2 MHz;
- 16 bits address lines;
- Memoria istruzioni di 16 KB SDRAM
- Memoria dati di 32 KB SDRAM
- Memoria indirizzata a word di 16-bit
- C'è una memoria cache a mappatura diretta con 2 linee collegata alla memoria dati. Ogni linea contiene 8 Byte.

LOAD dei dati in M[9]

C)

RAM 16 bits data	
0x0000	4
0x0001	3
0x0002	8
0x0003	2
0x0004	1
0x0005	3
0x0006	2
0x0007	8
0x0008	10
0x0009	2
0x000A	2
0x000B	8
0x000C	11
0x000D	18

Esempio

Descrizione:

Un computer a ciclo singolo ha le seguenti caratteristiche:
- Clock di 2 MHz;
- 16 bits address lines;
- Memoria istruzioni di 16 KB SDRAM
- Memoria dati di 32 KB SDRAM
- Memoria indirizzata a word di 16-bit
- C'è una memoria cache a mappatura diretta con 2 linee collegata alla memoria dati. Ogni linea contiene 8 Byte.

Il dato che non cerchiamo non è in cache, abbiamo un MISS. I dati devono essere scritti nella prima line, che però non è vuota: abbiamo un **REPLACEMENT**

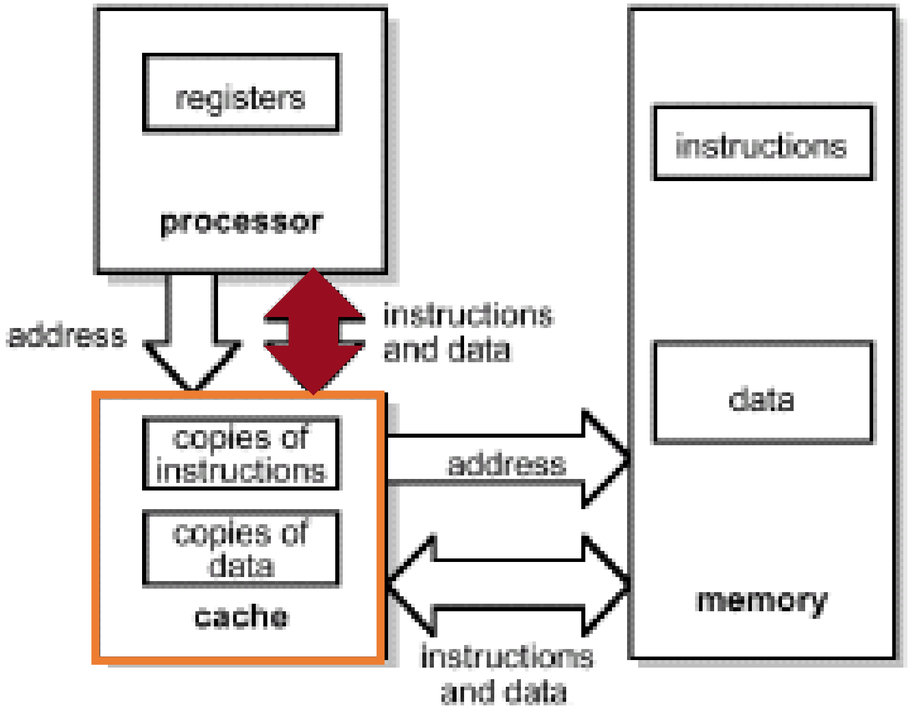
	10	**2**	2	8
	1	3	2	8

C)

RAM 16 bits data	
0x0000	4
0x0001	3
0x0002	8
0x0003	2
0x0004	1
0x0005	3
0x0006	2
0x0007	8
0x0008	10
0x0009	2
0x000A	2
0x000B	8
0x000C	11
0x000D	18

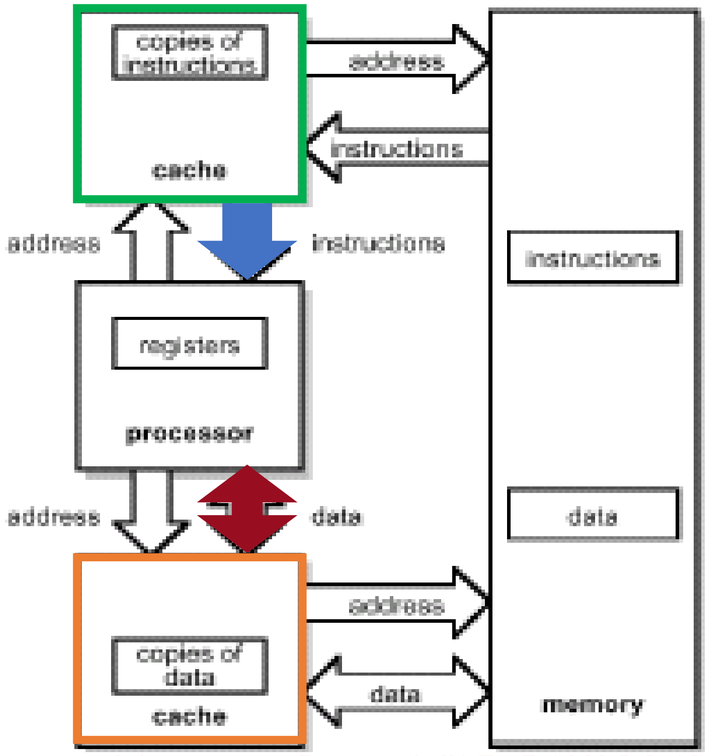
Cache unica o distinta per istruzioni e dati

Cache unica



- Dati e istruzioni nella stessa cache
- Flessibilità e maggiore cache hit
- 1 accesso per ciclo di clock

Cache distinte

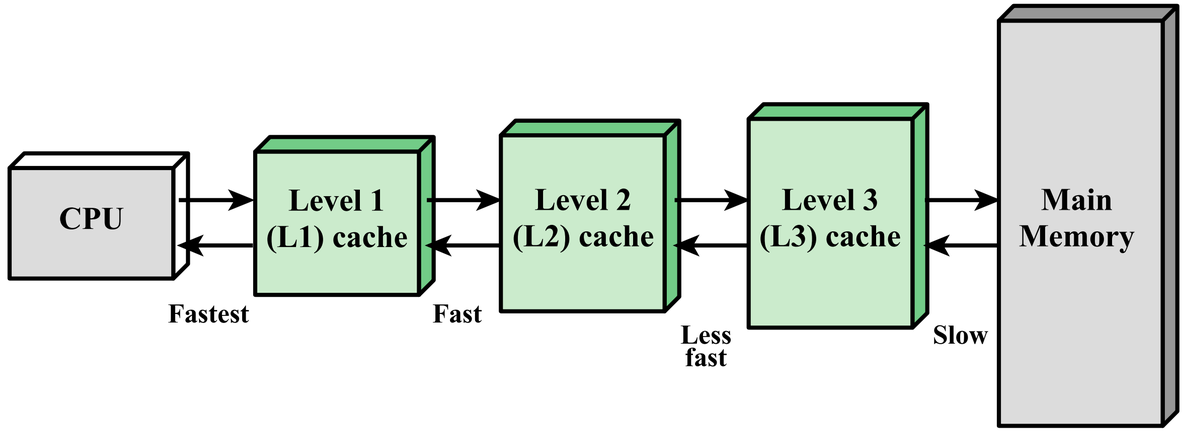


- Dati e istruzioni in due cache diverse
- Design diversi per ogni cache
- Ogni cache è più semplice
- 2 accessi per ciclo di clock (istruzioni e dati)

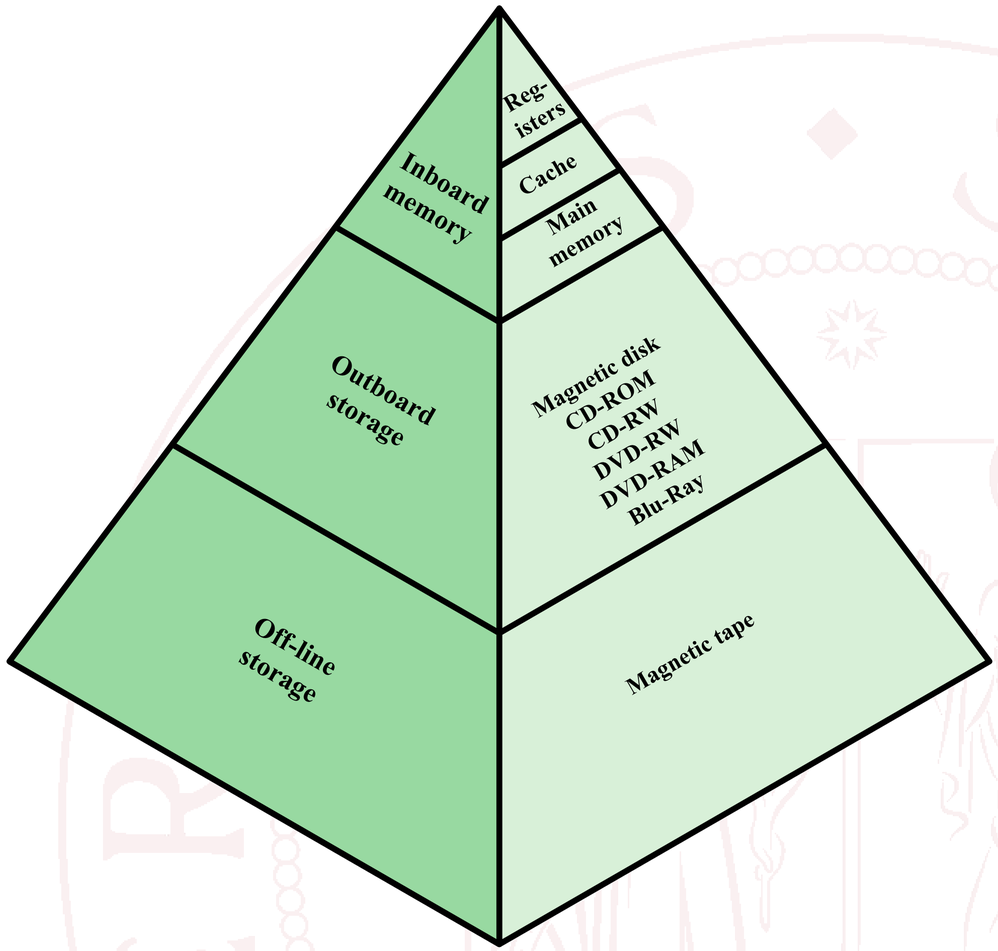
Gerarchia multilivello

- Possono essere presenti **più livelli di cache**:
  - una cache di 1° livello, quasi sempre integrata nello stesso chip del processore, ad accesso rapidissimo;
  - una cache di 2° livello, talvolta esterna al chip del processore, ad accesso rapido;
  - a volte anche una cache di 3° livello.
- Il **livello più vicino** al processore conterrà i dati che serviranno al processore nell'immediato futuro;
- I dati che serviranno più avanti vengono contenuti nei **livelli più lontani** che sono più lenti ma più capienti.

Gerarchia multilivello



(b) Three-level cache organization



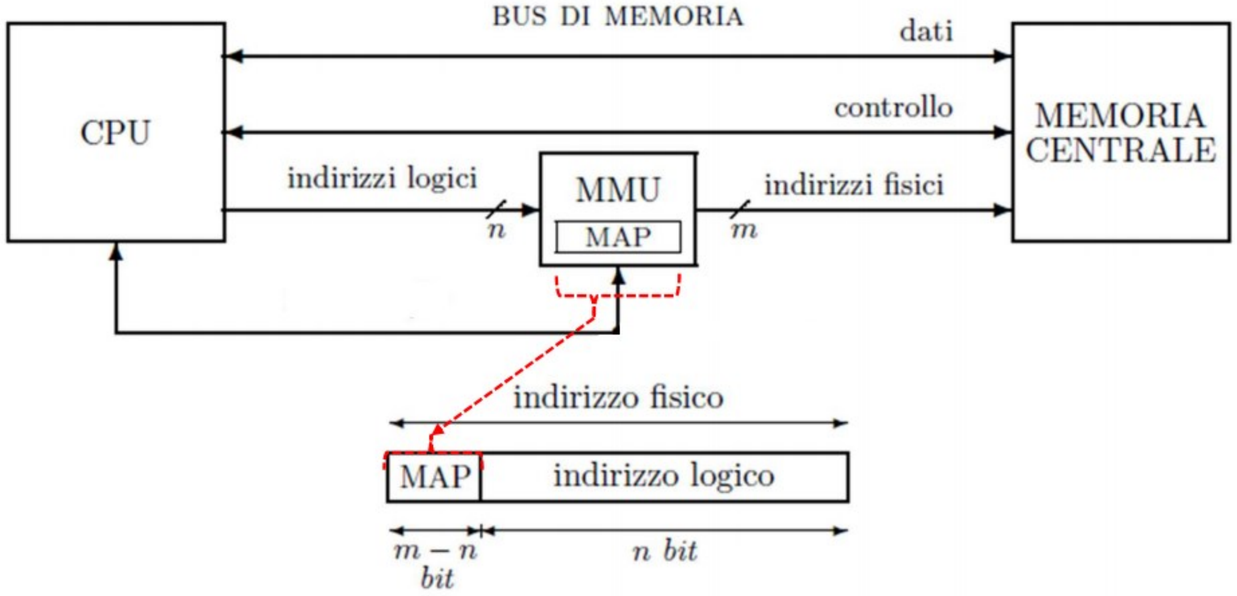
Una memoria più GRANDE

Dimensioni diverse

- Sappiamo che un processore a n bit può gestire indirizzi validi per una memoria di 2^n byte
- Un processore «antico» a 8 bit può quindi al massimo indirizzare una memoria da 256 byte …e se la memoria a disposizione fosse più grande (ad esempio 1Kbyte) ?
  - La memoria ha un indirizzo a 10 bit (indirizzo fisico)
  - Il processore lavora con un indirizzo a 8 bit (indirizzo logico)

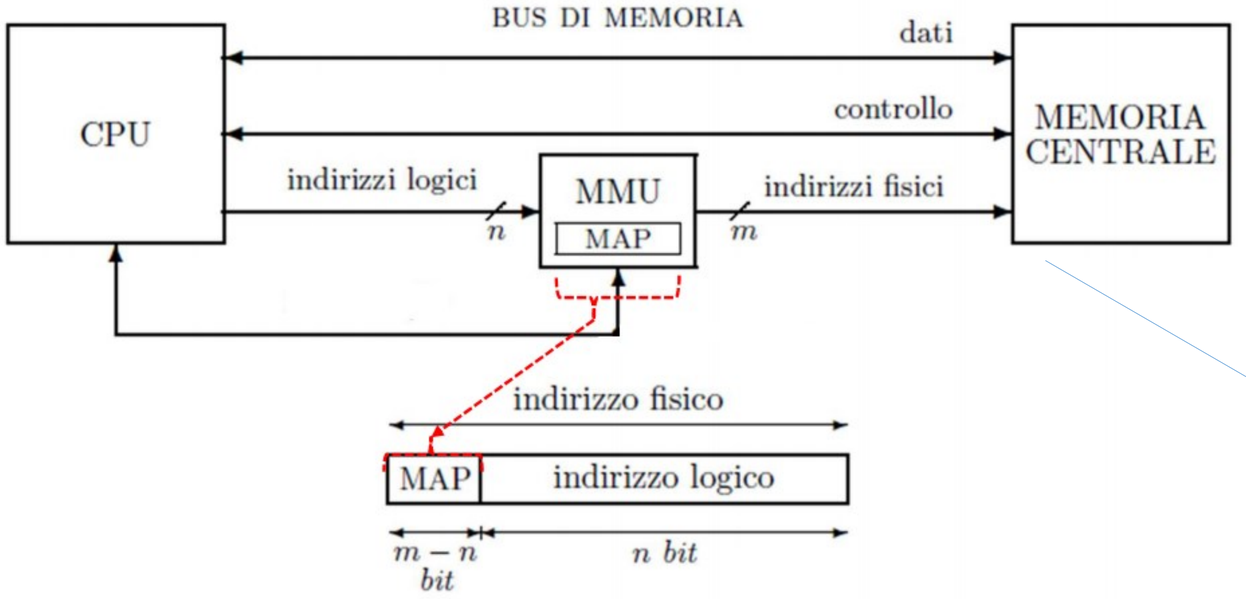
Nasce la MMU:

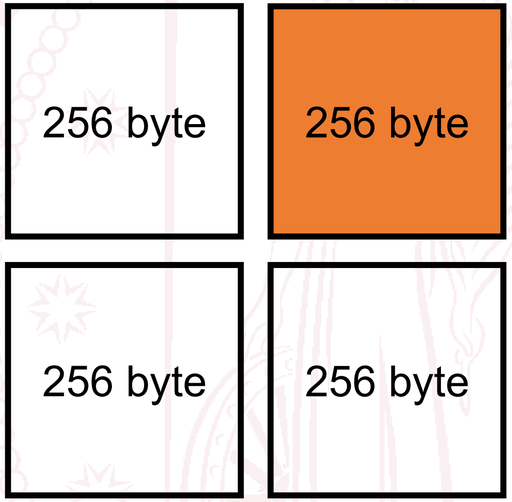
Memory Management Unit



La MMU per memorie grandi

- Possiamo usare tutti i nostri 1 KB di RAM per ospitare contemporaneamente 4 programmi da 256 byte in memoria (programma A, B, C, D)
- Dividiamo la memoria in 4 sezioni da 256 byte: le chiamiamo PAGINE
- Ad ogni programma associamo una pagina (pagina 0, 1, 2, 3)
- Quando stiamo eseguendo ad esempio il programma B, la MMU traduce gli indirizzi logici del processore in indirizzi fisici aggiungendo i 2 bit associati al programma B: di fatto stiamo «mappando» la memoria del programma B su un determinato spazio nella RAM
- Per lavorare su un altro programma, cambio il prefisso (registro MAP della MMU) e uso l'indice della pagina di un altro programma



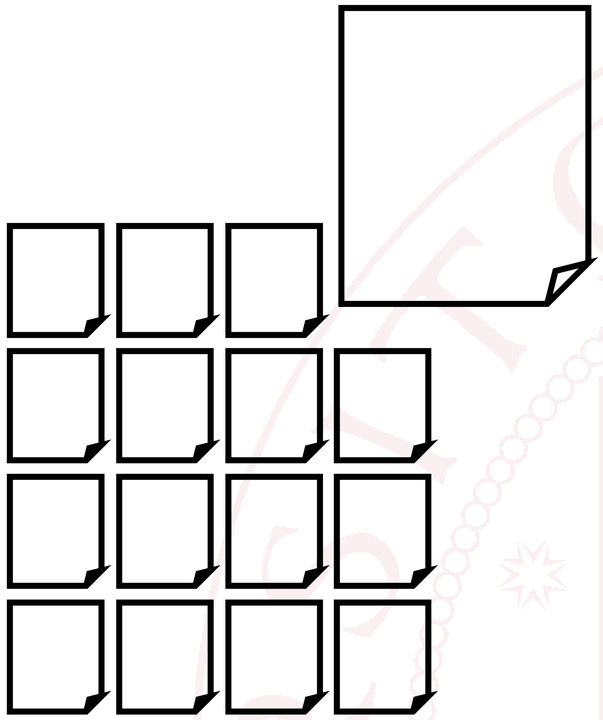


La MMU per memorie piccole

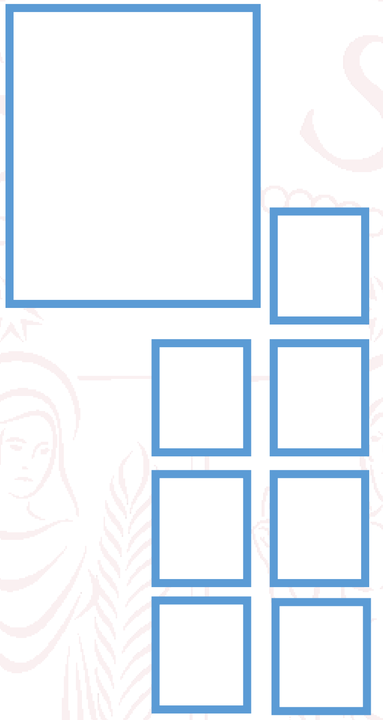
- Sappiamo che un processore a n bit può gestire indirizzi validi per una memoria di 2^n byte
- Un processore «moderno» a 32 bit può indirizzare 4GB di dati, ma potremmo non avere così tanta memoria
- Come facciamo ?
- Il concetto di MMU e di pagine ci aiuta anche qui!
- Dividiamo la memoria in parti e associamo ad un programma una o più di queste parti
  - La memoria fisica (indirizzo fisico, piccolo) è divisa in **PAGE FRAMES**
  - La memoria logica (indirizzo virtuale, grande) è divisa in **PAGES**

La MMU per memorie piccole

- I blocchi in cui sono divisi la memoria fisica (RAM) e la memoria virtuale sono della stessa dimensione
- Quando carichiamo i dati in memoria, associamo un page ad un page frame
- Possiamo massimizzare l'uso della RAM tenendo in memoria le pages che ci stanno, e usando un'altra memoria (HDD) per rilocare quelle che non ci stanno



PAGES
Pagine di memoria virtuale



PAGE FRAMES
Pagine nella memoria fisica

Tecniche di gestione della memoria

Programma A

```
int main(int argc, char** argv) {
   int a = 0;
   const int NIter = 100;
   for (unsigned int i=0; i++; i<NIter) {
      a = a+1;
   }
   return 0;
}
```

8 MB

in: /home/ltonin/mainA.c

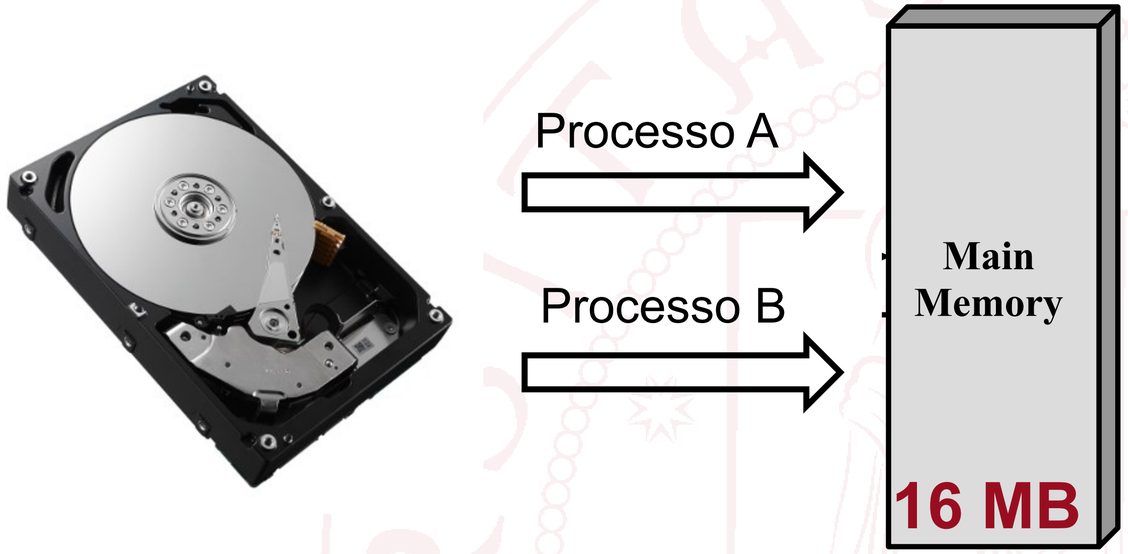
Programma B

```
int main(int argc, char** argv) {
   const int NElem = 100;
   std::vector v(NElem);
   for (unsigned int i=0; i++; i<NElem) {
      v[i] = i;
   }
   return 0;
}
```

16 MB

in: /home/ltonin/mainB.c

- **Ipotizziamo** che il primo processo abbia bisogno di 8MB di spazio in memoria
- **Ipotizziamo** che il secondo processo abbia bisogno di 16MB di spazio in memoria
- **Ipotizziamo** di avere una memora da solo 16MB



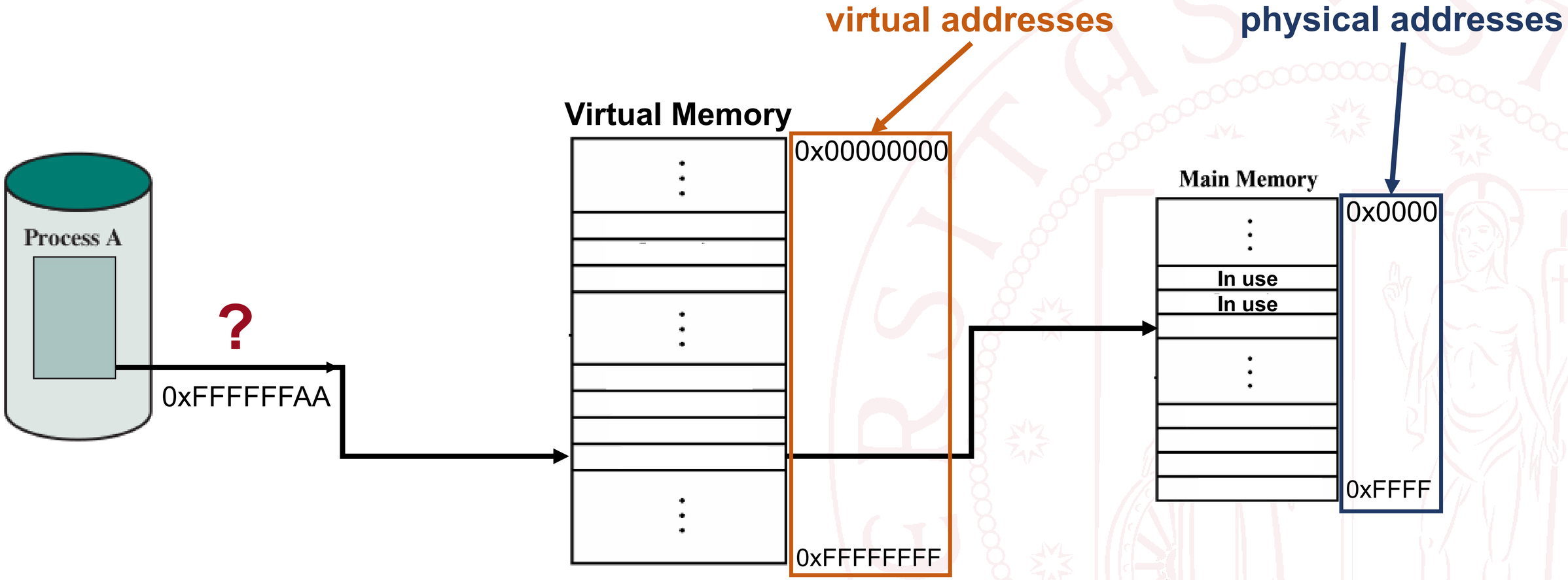
Soluzioni:

- Estendere memoria → costoso
- **Ottimizzare la gestione della memoria → memoria virtuale**

- I due processi non possono essere caricati in memoria contemporaneamente
- Il sistema operativo dovrebbe aspettare la fine dell'esecuzione di A per caricare B

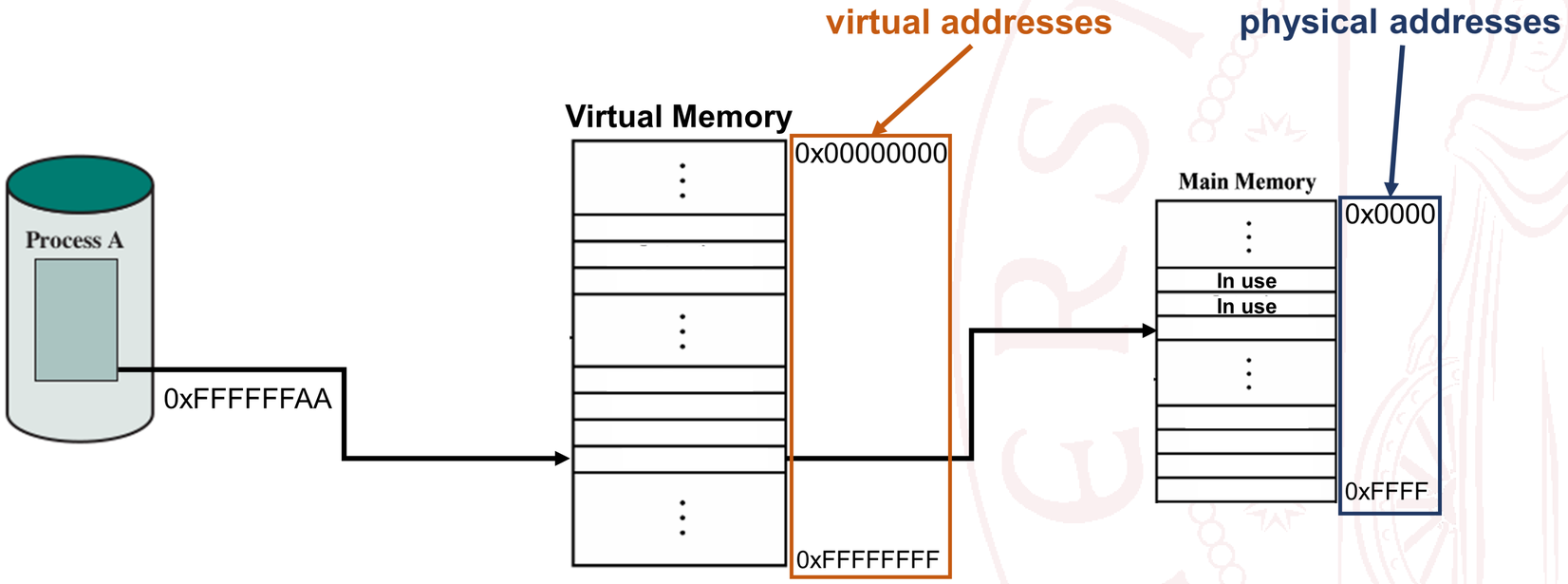
Memoria virtuale

- Immaginiamo che il processo A abbia bisogno degli indirizzi di memoria da 0xFFFFFFAA
- Tuttavia abbiamo una memoria limitata con locazioni che vanno da 0x0000 a 0xFFFF
- La nostra memoria è per la maggior parte «free»
- Possiamo quindi **rimappare** gli indirizzi richiesti dal processo in locazioni libere della memoria



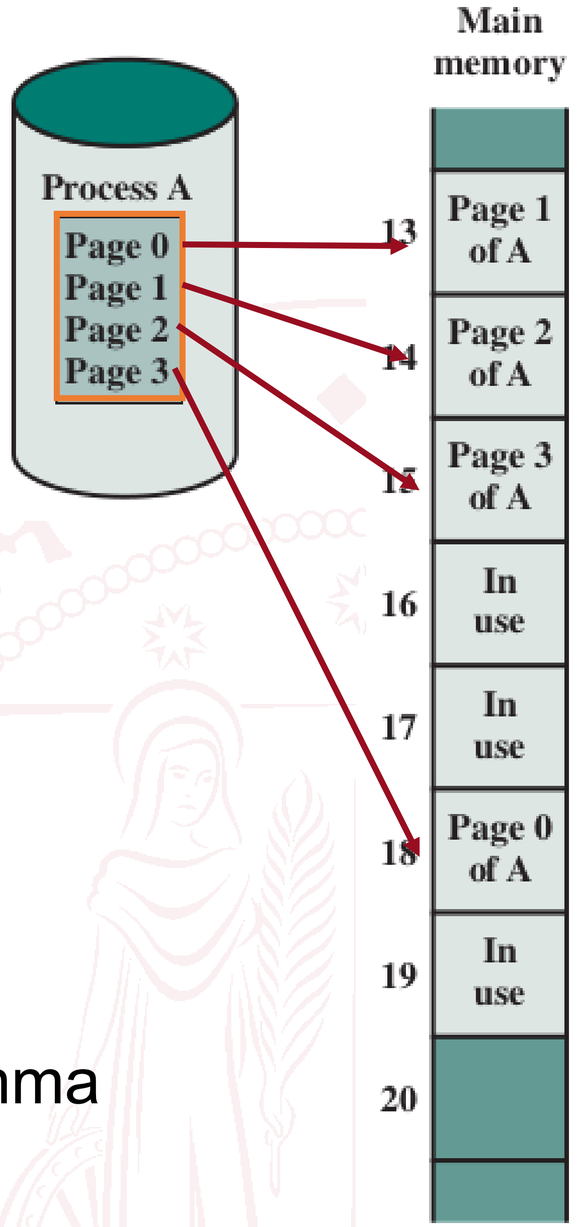
Memoria virtuale

- Il programmatore «vede» sempre e solo gli **indirizzi virtuali**
- Gli **indirizzi fisici** sono calcolati automaticamente da hardware/software
- Gli indirizzi virtuali vanno a comporre la **memoria virtuale**
- La memoria virtuale può essere molto più grande di quella fisica, grazie all'ottimizzazione della sua gestione (es. tramite paging)
- La memoria virtuale permette di:
    1. Utilizzare processi che richiedono più memoria di quella fisicamente disponibile
    2. Utilizzare aree di memoria condivise da più programmi



Paginazione (paging)

- Abbiamo un processo da eseguire A (un programma)
- La memoria è divisa in piccoli blocchi (**page frames)**
- Il processo A viene scomposto in blocchi delle stesse dimensioni (**pages)**
- Quando il sistema operativo carica il processo in memoria, assegna le diverse pages ai page frames liberi
- Alcuni page frames della memoria possono essere già in uso da altri processi



- Un **indirizzo fisico** è il reale indirizzo in memoria
  - Per il processo A → 13, 14, 15, 18
- Un **indirizzo virtuale** è l'indirizzo rispetto all'inizio del programma
  - Per il processo A→ 1, 2, 3, 4

Paginazione (paging)

- Lo spazio degli indirizzi fisici in memoria è diviso in **page frames** di piccola dimensione (es. 4 KB)
- Anche lo spazio di indirizzi virtuali è diviso in blocchi della stessa dimensione, dette **pages**

- Assumiamo che l'indirizzo virtuale sia a 32 bits
- Per indirizzare i singoli byte della pagina di 4KB devo usare $\log_2(4 \cdot 1024) = \boldsymbol{12\ bits}$
- I rimanenti 32-12=**20 bits** posso usarli per indirizzare $2^{\mathbf{20}}\ pages$ nella memoria virtuale
- Assumendo di avere una memoria fisica di 16 MB, ho $2^{\mathbf{12}}\ page\ frames$
- Il **page offset** indica la word all'interno della page
- **Virtual page number**
- **Physical page frame number**

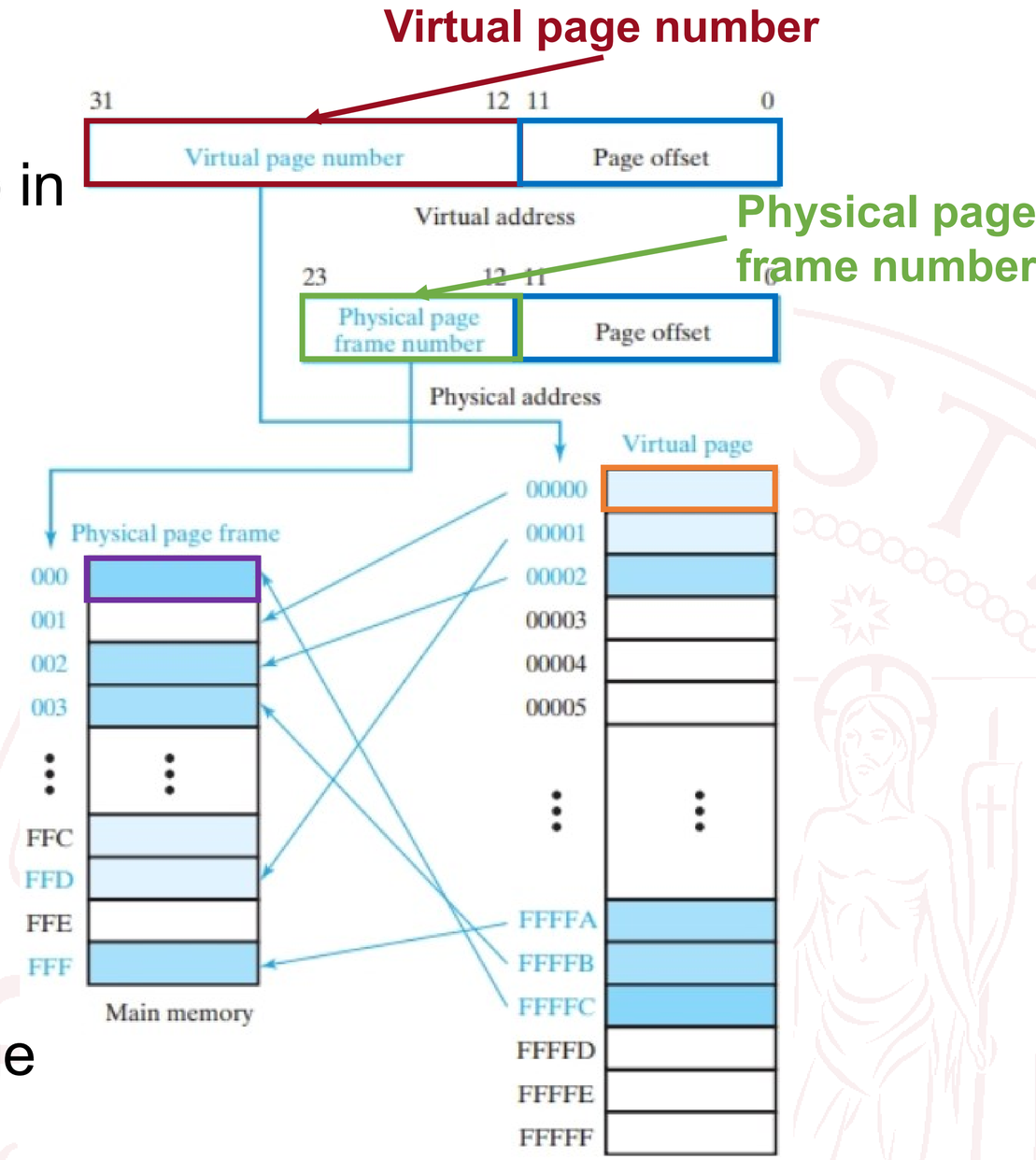


FIGURE 12-11
Virtual and Physical Address Fields and Mapping

Paginazione (paging)

- Ovviamente dobbiamo tenere traccia della corrispondenza tra le pagine
- La struttura potrebbe essere questa:

20 bit	12 bit

- In aggiunta potremmo voler memorizzare anche:
  - un validity bit
  - un dirty bit
  - un used bit
  - …altri bit (es. di protezione)

  - Come possiamo salvare tutti questi dati senza occupare la nostra preziosa memoria?

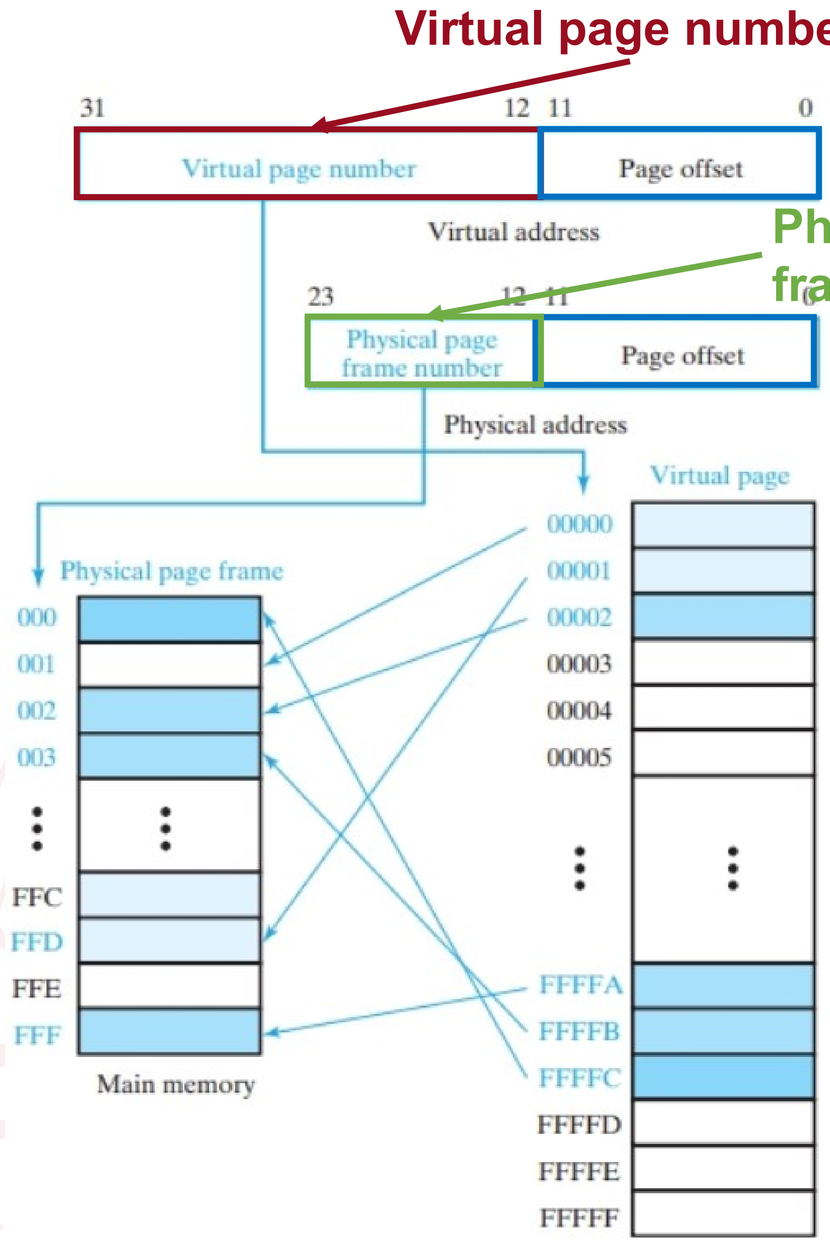


FIGURE 12-11
Virtual and Physical Address Fields and Mapping

Tabelle di pagine (page tables)

- Le mappature fra indirizzi virtuali e indirizzi fisici vengono immagazzinate nelle **page tables**
- Le page tables possono essere raggruppate a loro volta in altre tabelle (**page directory**)
  - L'indirizzo virtuale viene scomposto in **virtual page number** e **page offset**
  - Il virtual page number viene a sua volta diviso in **directory offset** e **page table offset**
  - Il puntatore all'inizio del programma viene sommato con il directory offset per identificare la **page table relativa**
  - Il **page table offset** viene sommato alla **page table relativa** per ricavare il **physical page frame number**
  - A quest'ultimo viene sommato il **page offset** per ricavare la locazione desiderata
- Non tutte le page table sono in memoria RAM, alcune sono su hard disk

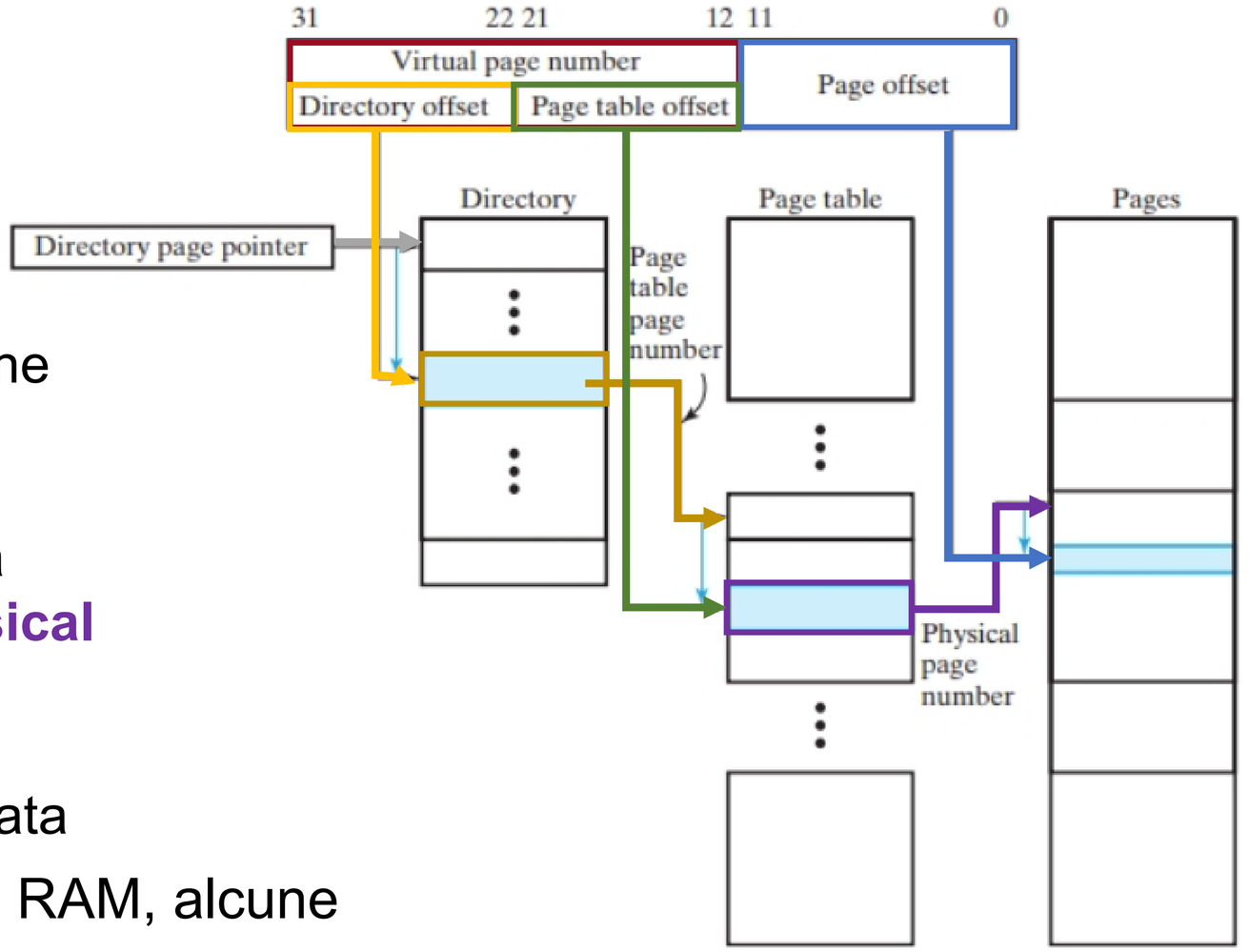


FIGURE 12-13
Example of Page Table Structure

Accesso alle page tables

- Una gestione della memoria basata su page tables soffre comunque di latenze per accedere ai dati
- Ad esempio, nel caso precedente, per recuperare un'istruzione è necessario:
  - Accedere alla **pages directory**
  - Accedere alla **page table**
  - Accedere all'**istruzione**

Tutte queste operazioni richiedono comunque accessi alla memoria

- Per ridurre gli accessi alla memoria, ipotizziamo di avere una cache in grado di tradurre direttamente i virtual addresses in physical addresses
- Questa cache viene chiamata **Translation Lookaside Buffer (TLB)**

Translation Lookaside Buffer

- È una cache tipicamente fully associative o set associative
- Compara il **virtual page number** richiesto con quelli presenti in cache

- Usa come input il **virtual page number** richiesto dalla CPU
- Viene comparato con tutti i virtual page number presenti nella cache
- Se c'è un match, e il bit di validità è 1 (TLB Hit), l'output è il **page frame number** corrispondente
- Se non c'è (TLB Miss), la pagina viene cercata in memoria e caricata nella TLB
- Se la pagina non viene trovata si ha un **page fault** →la pagina viene caricata dall'hard disk alla memoria
- L'efficacia dipende dalla località spaziale e temporale delle pagine presenti nella TLB

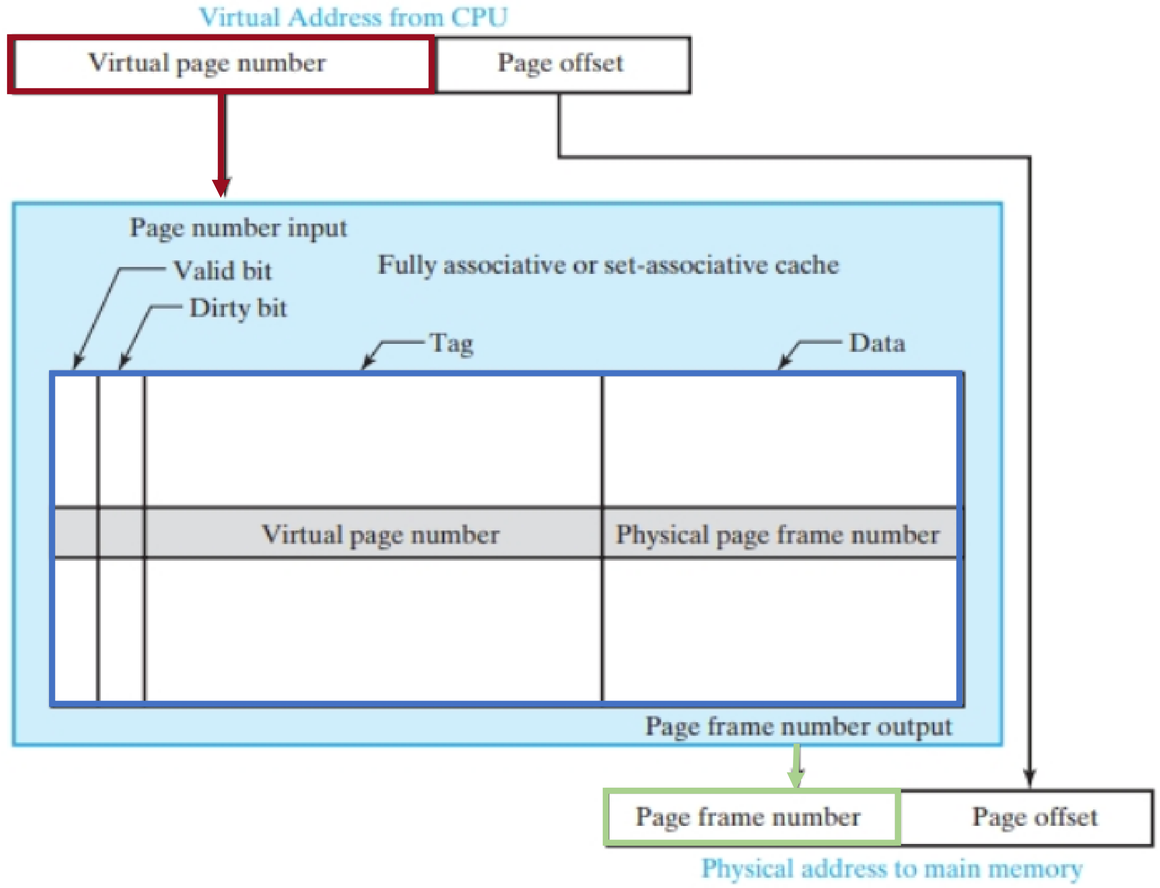


FIGURE 12-14
Example of Translation Lookaside Buffer

Vantaggi della memoria virtuale

- Rilocamento dei dati: posso liberare memoria spostando i dati di un programma non in esecuzione su una memoria secondaria (HDD) e ri-caricarli quando quel programma torna in funzione (swap)
- Protezione di interferenza tra i dati: dividendo la memoria in «zone» possiamo impedire ad un programma di leggere dati di una zona che non gli compete (es. dati di un altro programma)
- Comunicazione intra processo: oppure possiamo fare in modo che due programmi possano condividere uno stesso frame, permettendo cosi che comunichino
- Risoluzione della frammentazione: quando un programma termina e libera memoria, posso usare i blocchi lasciati liberi per scriverci alcuni blocchi di un programma più grosso (non è necessario che le pages siano veramente contigue nella RAM)
- Protezione dei dati: posso marcare alcune pages come read-only oppure permettervi l'accesso in base a dei privilegi del programma/utente che lo esegue