
Gerarchie di memoria (cache)

Salvatore Orlando

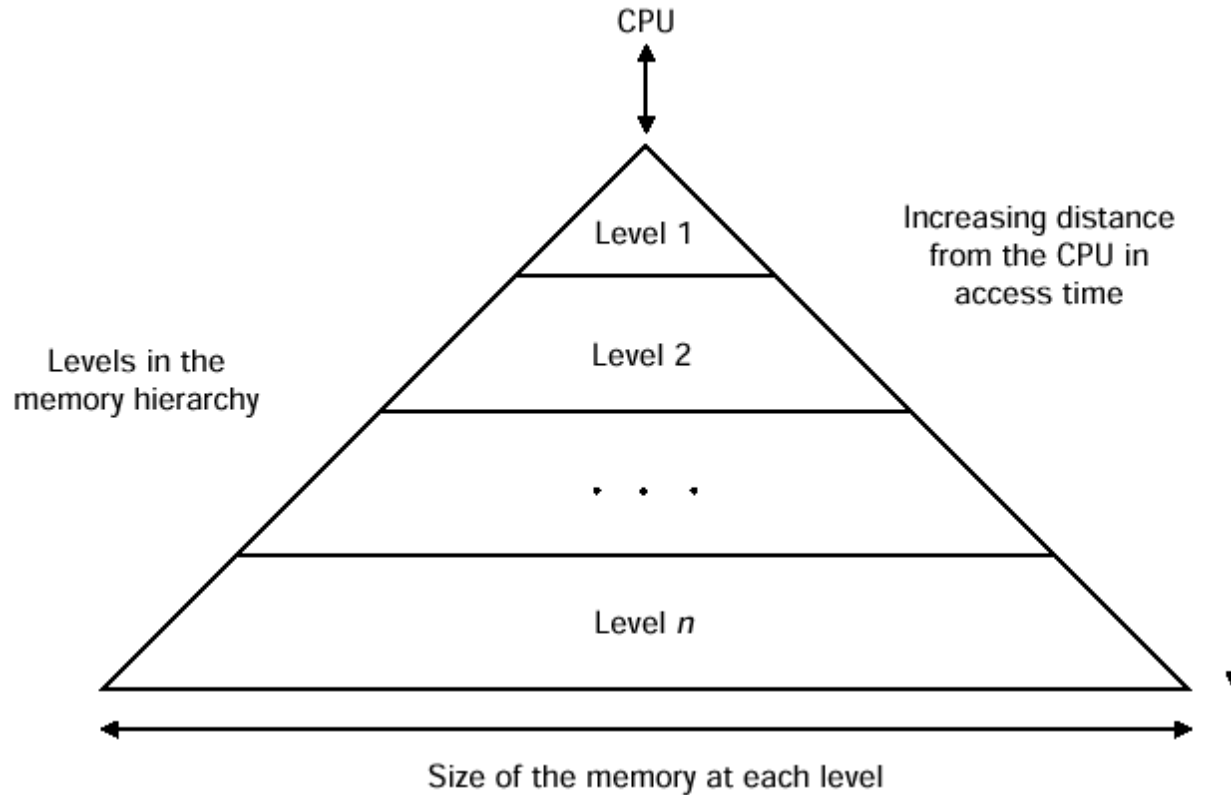
Gerarchie di memoria

- I programmatori hanno l'esigenza di avere **memorie sempre più veloci e capienti**, per poter memorizzare programmi e dati
- Purtroppo la tecnologia permette solo di costruire
 - memorie grandi e lente, ma poco costose
 - memorie piccole e veloci, ma molto costose
- Conflitto tra
 - esigenze programmatori
 - vincoli tecnologici

Gerarchie di memoria

- **Soluzione: gerarchie di memoria**
 - **piazziamo memorie veloci vicino alla CPU**
 - **per non rallentare la dinamica di accesso alla memoria**
 - **fetch delle istruzioni e load/store dei dati**
 - **man mano che ci allontaniamo dalla CPU**
 - **memorie sempre più lente e capienti**
 - **soluzione compatibile con i costi**
 - **meccanismo dinamico per spostare i dati tra i livelli della gerarchia**

Gerarchie di memoria



- Al **livello 1** poniamo la memoria più veloce (piccola e costosa)
- Al **livello n** poniamo la memoria più lenta (grande ed economica)
- Scopo gerarchia e delle politiche di gestione delle memorie
 - dare l'**illusione** di avere a disposizione una memoria
 - **grande** (come al **livello n**) e **veloce** (come al **livello 1**)

Costi e capacità delle memorie

- **Dati 2008**

- **SRAM**

- latenze di accesso di 0,5-2,5 ns
 - costo da \$2000 a \$5.000 per GB
 - tecnologia usata per i livelli più vicini all CPU (**cache**)

- **DRAM**

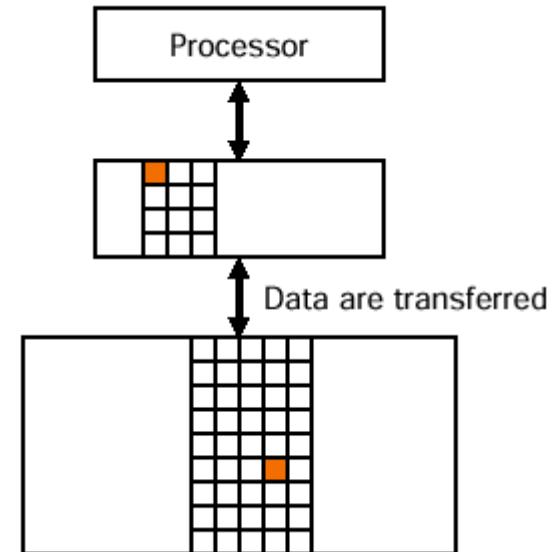
- latenze di accessi di 50-70 ns
 - costo da \$20 a \$75 per GB
 - tecnologia usata per la cosiddetta **memoria principale**

- **Dischi**

- latenze di accesso di 5-20 milioni di ns (5-20 ms)
 - costo da \$0,2 a \$2 per GB
 - memoria stabile usata per memorizzare file
 - memoria usata anche per contenere l'immagine (text/data) dei programmi in esecuzione => **memoria (principale) virtuale**

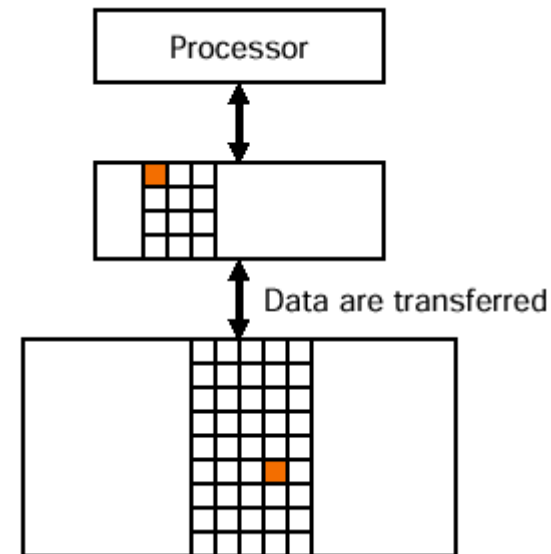
Illusione = memoria grande e veloce !?

- All'inizio i nostri dati e i nostri programmi sono memorizzati nel **livello n** (mem. più capiente e lenta)
- I **blocchi (linee) di memoria** man mano **riferiti** vengono fatti fluire verso i **livelli più alti** (memorie più piccole e veloci), più vicini alla CPU
- Se il blocco richiesto è presente nel livello più alto
 - **Hit**: l'accesso soddisfatto dal livello più alto
 - **Hit rate (%)**: $100 * n. \text{ hit} / n. \text{ accessi}$
- Se il blocco richiesto è assente nel livello più alto
 - **Miss**: il blocco è *copiato* dal livello più basso
 - **Time taken**: miss penalty
 - **Miss rate (%)**: $100 * n. \text{ miss} / n. \text{ accessi} = 100 - \text{hit ratio}$
 - Poi l'accesso è garantito dal livello più alto



Illusione = memoria grande e veloce !?

- Problema 1:
 - Cosa succede se un blocco riferito è già presente nel livello 1 (più alto) ?
 - La CPU può accedervi direttamente (**hit**), ma abbiamo bisogno di un meccanismo per individuare e indirizzare il blocco all'interno del **livello più alto** !
- Problema 2:
 - Cosa succede se il livello più alto è pieno ?
 - Dobbiamo implementare una politica di rimpiazzo dei blocchi !



Terminologia

- Anche se i trasferimenti tra i livelli avvengono sempre in blocchi, questi hanno dimensione diversa, e (per ragioni storiche) nomi diversi
 - abbiamo blocchi più piccoli ai livelli più alti (più vicini alla CPU)
 - es. di nomi: **blocco/linea di cache** e **pagina**
- **Hit (Successo)**
 - quando il blocco cercato a **livello i** è stato individuato
- **Miss (Fallimento)**
 - quando il blocco cercato non è presente al **livello i**
- **Hit rate (%)**
 - frequenza di Hit rispetto ai tentativi fatti per accedere blocchi al **livello i**
- **Miss rate (%)**
 - frequenza di Miss rispetto ai tentativi fatti per accedere blocchi al **livello i**
- **Hit Time**
 - latenza di accesso di un blocco al **livello i** in caso di Hit
- **Miss Penalty**
 - tempo per copiare il blocco dal livello inferiore

Località

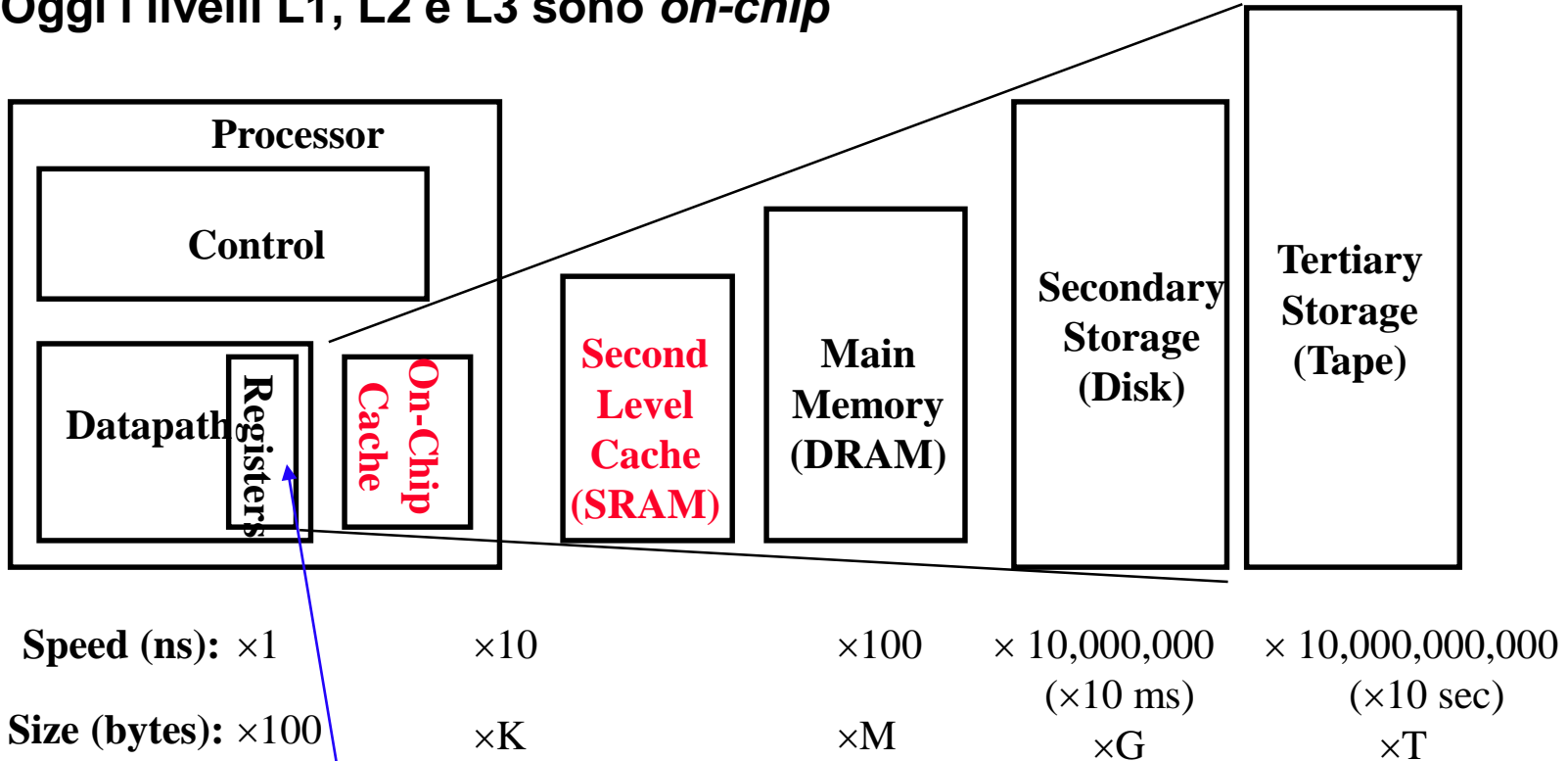
- L'illusione offerta dalla gerarchia di memoria è possibile in base al:
Principio di località
- Se un elemento (es. word di memoria) è riferito dal programma
 - esso tenderà ad essere riferito ancora, e presto \Leftarrow **Località temporale**
 - gli elementi ad esse vicini tenderanno ad essere riferiti presto \Leftarrow **Località spaziale**
- In altri termini, in un dato intervallo di tempo, i programmi accedono una porzione (relativamente piccola) dello spazio di indirizzamento totale



- La località permette il funzionamento ottimale delle gerarchie di memoria
 - aumenta la probabilità di *riusare* i blocchi, precedentemente spostati ai livelli superiori, riducendo il *miss rate*

Cache

- E' il livello di memoria (SRAM) più vicino alla CPU (oltre ai Registri)
- Oggi i livelli L1, L2 e L3 sono *on-chip*



Registri: livello di memoria più vicino alla CPU

Movimenti tra **Cache** ↔ **Registri** gestiti a sw
dal compilatore / programmatore assembler

Cache e Trend tecnologici delle memorie

Logica digitale:

DRAM:

Dischi:

Capacità

2x in 3 anni

4x in 3 anni

4x in 3 anni

Velocità (riduz. latenza)

2x in 3 anni

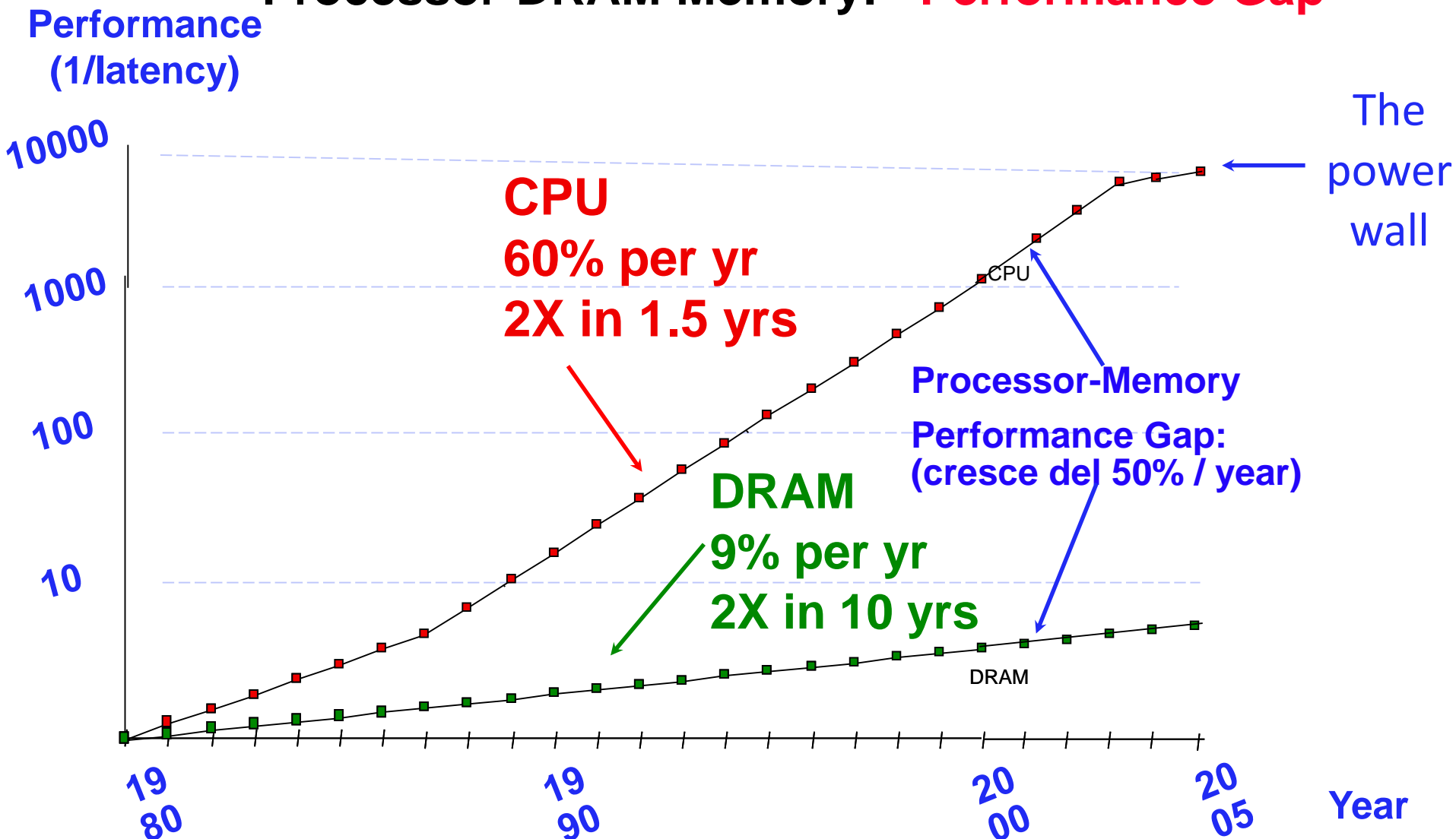
2x in 10 anni

2x in 10 anni

Anno	Size	\$ per MB	Latenza accesso
1980	64 Kb	1.500	250 ns
1983	256 Kb	500	185 ns
1985	1 Mb	200	135 ns
1989	4 Mb	50	110 ns
1992	16 Mb	15	90 ns
1996	64 Mb	10	60 ns
1998	128 Mb	4	60 ns
2000	256 Mb	1	55 ns
2002	512 Mb	0,25	50 ns
2004	1024 Mb	0,10	45 ns

Accesso alla memoria = Von Neumann bottleneck

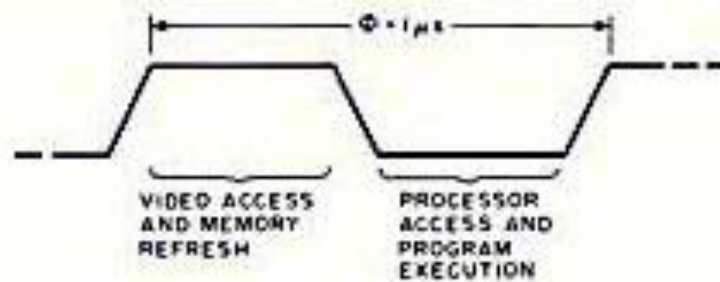
Processor-DRAM Memory: Performance Gap



1977: DRAM più veloce del microprocessore

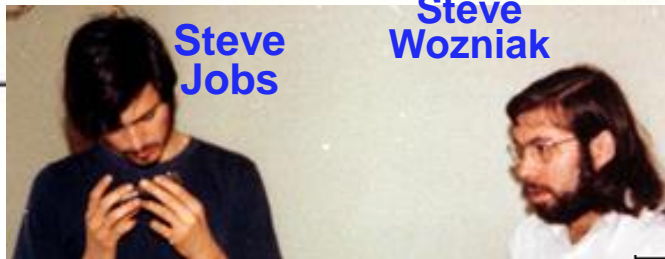
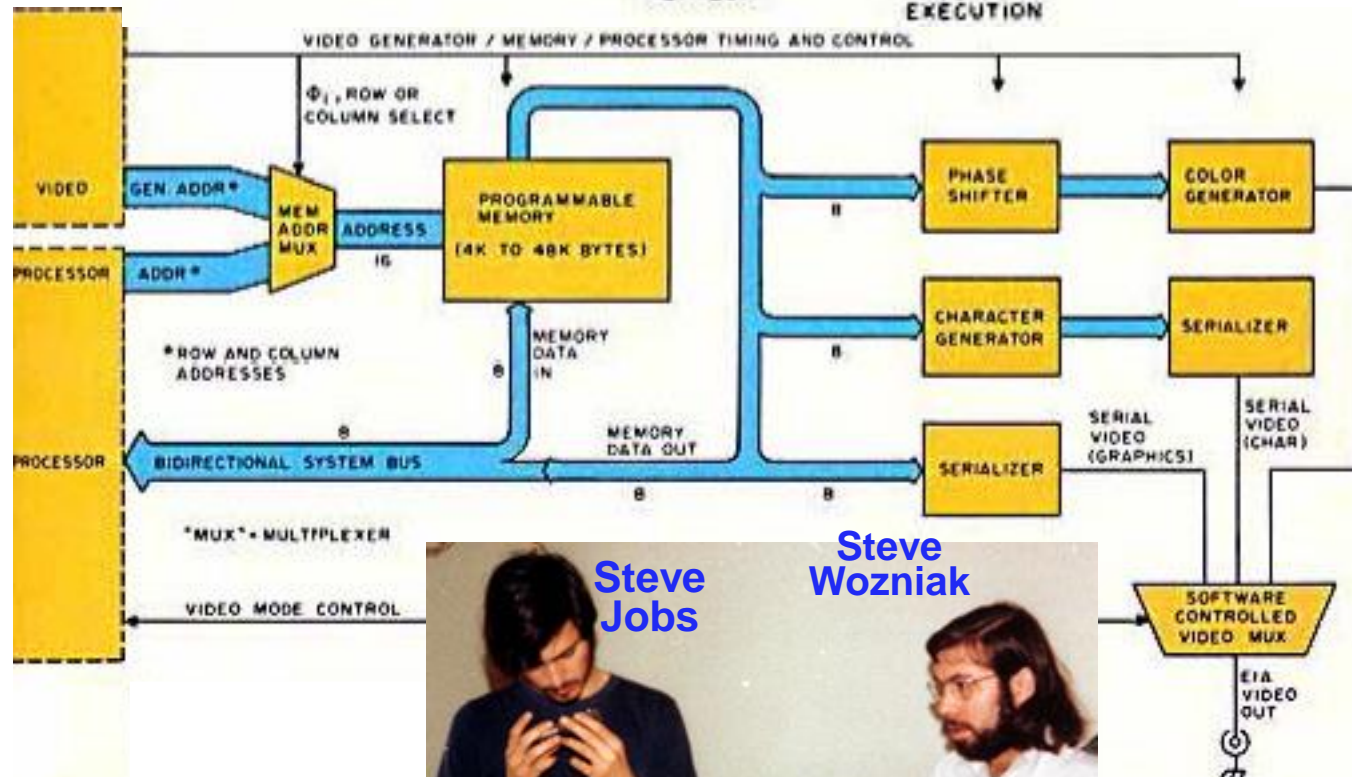
TIMING:

6502 PROCESSOR'S
 Φ_1 CLOCK SHOWING
 WHEN AND BY WHOM
 MEMORY IS ACCESSED



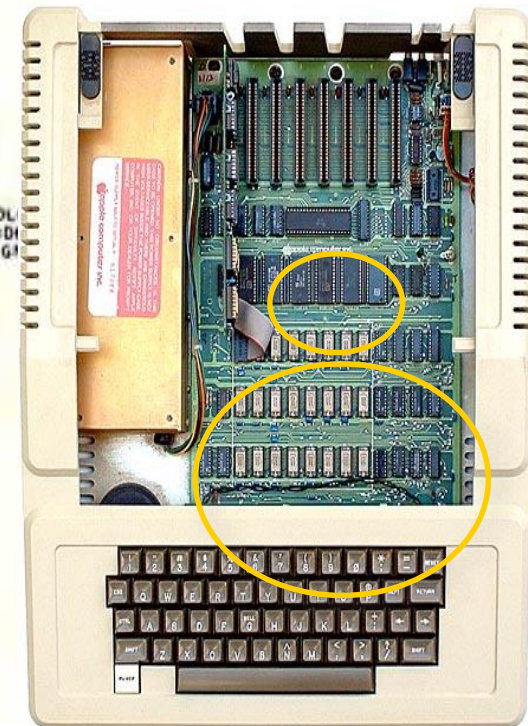
Apple II (1977)

CPU: 1000 ns
 DRAM: 400 ns



Steve Jobs

Steve Wozniak



RAM Complement	Apple II System
4K	\$ 1,298.00
48K	2,638.00

Gerarchie di Memoria nel 2005: Apple iMac G5

Managed
by compiler

Managed
by hardware

Managed by OS,
hardware,
application

	Reg	L1 Inst	L1 Data	L2	DRAM	Disk
Size	1K	64K	32K	512K	256M	80G
Latency (cycles)	1	3	3	11	160	1e7

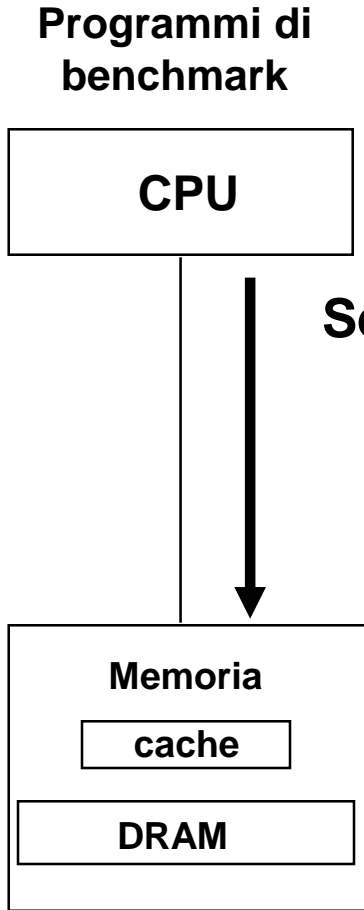


iMac G5
1.6 GHz
\$1299.00

Cache

- **L'uso di cache grandi e multivello è necessario per**
 - **tentare di risolvere il *von Neumann bottleneck*, il problema costituito dalle memorie DRAM**
 - **sempre più capienti**
 - **ma sempre meno veloci, rispetto agli incrementi di prestazione delle CPU (microprocessori)**
- **Gestione movimenti di dati tra livello cache e livelli sottostanti (Main memory)**
 - **realizzata dall'hardware**

Progetto di un sistema di cache



Sequenza di riferimenti alla memoria:

$\langle \text{op}, \text{addr} \rangle, \langle \text{op}, \text{addr} \rangle, \langle \text{op}, \text{addr} \rangle, \langle \text{op}, \text{addr} \rangle, \dots$

op: i-fetch, read (load), write (store)

Obiettivo del progettista:

Ottimizzare l'organizzazione delle gerarchie di memoria in modo da minimizzare il tempo medio di accesso alla memoria per carichi tipici (per sequenze di accesso tipiche)

Ovvero, aumentare il *cache hit rate*

Problemi di progetto di una cache

- **Dimensionamenti**
 - **size del blocco** e **numero di blocchi** nella cache
 - **Indirizzamento e mapping:**
 - Come faccio a sapere se un blocco è presente in cache, e come faccio a individuarlo ?
 - Se un blocco non è presente e devo recuperarlo dalla memoria a livello inferiore, dove lo scrivo in cache?
- **Funzione di *mapping*** tra
- Indirizzo Memoria → Identificatore blocco**
- dipende dall'organizzazione della cache:
diretta o associativa

Caso semplice: cache ad accesso diretto

- Mapping tramite **funzione hash (modulo)** dell'indirizzo (**Address**):
 - Cache block INDEX** = $\text{Address} \bmod \# \text{ cache blocks}$
resto divisione: $\text{Address} / \# \text{ cache blocks}$

$$\# \text{ cache blocks} = 8 = 2^3$$

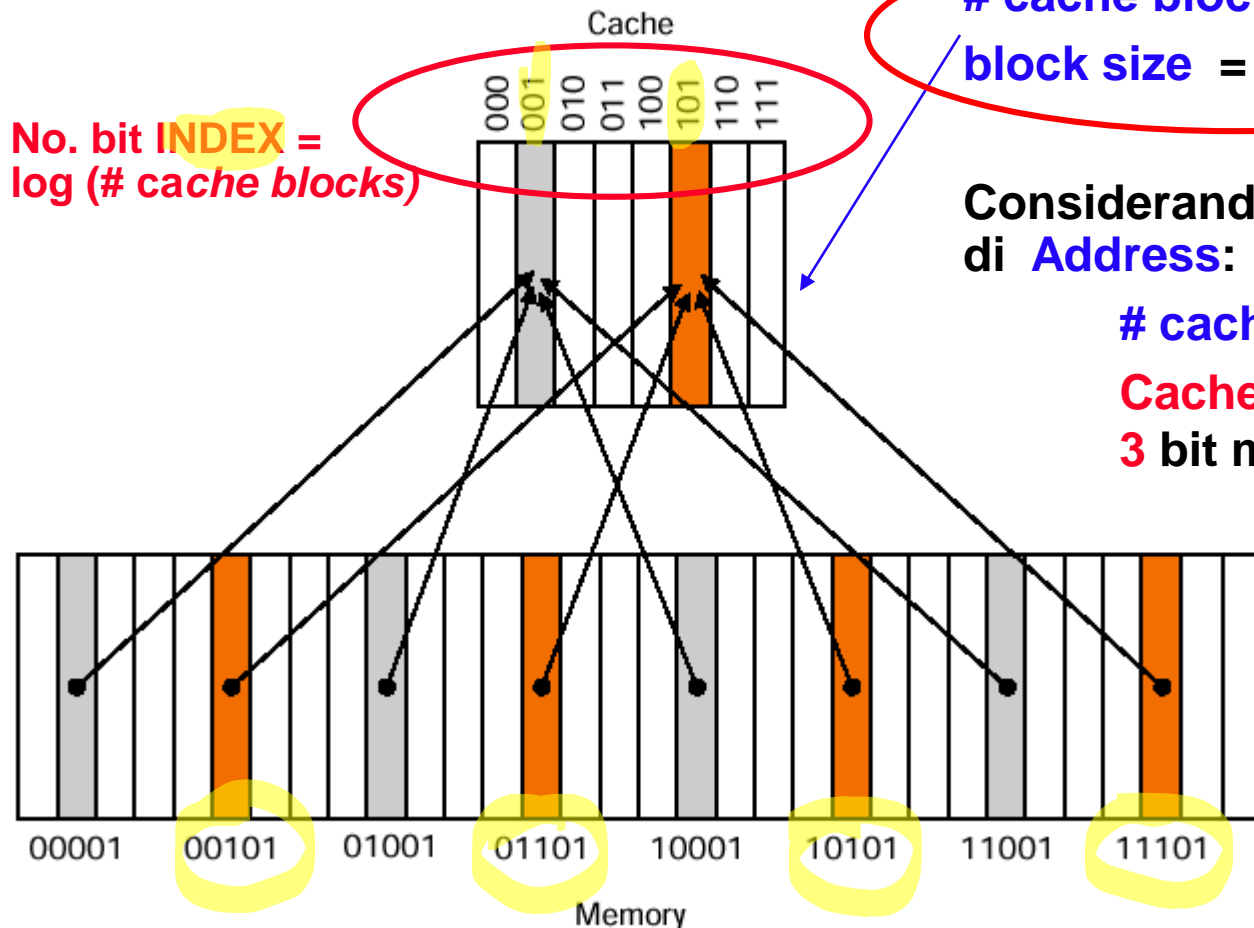
$$\text{block size} = 1 \text{ B}$$

No. bit **INDEX** =
 $\log(\# \text{ cache blocks})$

Considerando rappresentazione binaria di **Address**:

cache blocks è potenza di 2 (2^3)

Cache block Index corrisponde ai **3** bit meno significativi di **Address**



OPERAZIONI BITWISE

&	and
	or
>>	sr (SHIFT right log.)
<<	sl (SHIFT left log.)

OPERAZIONI
DISPONIBILI
IN C

$$\begin{array}{r} A = 00111011 \\ B = 11110000 \end{array}$$

$$A \& B = 00110000$$

$$\begin{array}{r} A = 00111011 \\ B = 11110000 \end{array} =$$

$$A | B = 11111011$$

shift left

$$\begin{array}{r} A = 01010110 \\ B = A \ll 3 \\ \downarrow \\ 10110000 \end{array}$$

shift right

$$\begin{array}{r} A = 01010110 \\ B = A \gg 3 \\ \downarrow \\ 00001010 \end{array}$$

MOLTIPLICAZIONE / DIVISIONE PER 2^i

$N = 11010011 \leftarrow$ rappresentazione su 8 bit

$$N * 2^3 \rightarrow \cancel{1 \cdot 2^{10} + 1 \cdot 2^9} + 1 \cdot 2^7 + 1 \cdot 2^4 + 1 \cdot 2^3$$

NON POSSONO ESSERE RAPPRESENTATI

sll

10011000

$$N / 2^3 \rightarrow 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + \cancel{1 \cdot 2^{-2} + 1 \cdot 2^{-3}}$$

NON RAPPRESENTABILI
(CIFRE DOPO LA VIRGOLA)

00011010

sr.l

$$\begin{aligned} M = N * 2^i &\Rightarrow M = N \ll i \quad (\text{shift left logical}) \\ M = N / 2^i &\Rightarrow M = N \gg i \quad (\text{" right "}) \end{aligned}$$

Caso semplice: cache ad accesso diretto

- Funzione hash:

- Cache block INDEX** = $\text{Address} \% \# \text{cache blocks} =$
resto divisione: $\text{Address} / \# \text{cache blocks}$

$\# \text{cache blocks} = 2^i$

$\text{block size} = 1 \text{ B}$

$\%$: OPERAZIONE DI MODULO
IN C/JAVA/ecc.

Considerando rappresentazione binaria di **Address**:

Quoziente = $\text{Address} / \# \text{cache blocks} = \text{Address} / 2^i = \text{Address} \gg i$

\Rightarrow $n-i$ bit più significativi di Address

Resto = $\text{Address} \% \# \text{cache blocks} = \text{Address} \& \underline{111..111}$
 i bit

\Rightarrow i bit meno significativi di Address

OPER. DI $\&$
PER IMPLEMENTARE
L'OPER. DI $\%$

Prova:

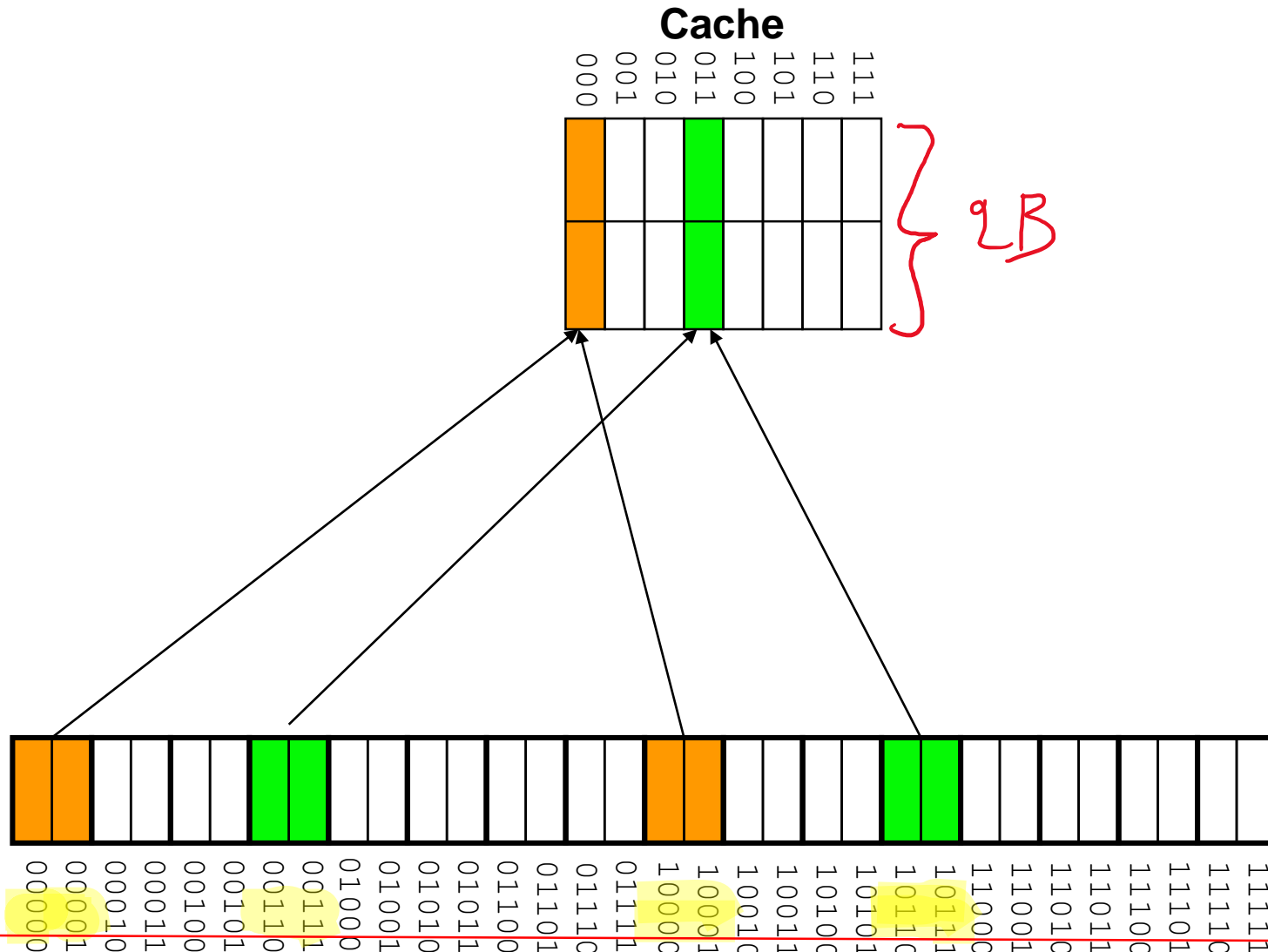
$\text{Address} = \text{Quoziente} * 2^i + \text{Resto} = \text{Quoziente} \ll i + \text{Resto}$



Cache diretta e blocchi più grandi

- Per **block size > 1B**:
 - **Address** diversi e consecutivi (che differiscono per i bit meno significativi) cadono all'interno dello stesso **Cache block**
- Le dimensioni dei blocchi sono solitamente potenze di 2
 - **Block size** = 4, 8, 16, o 32 B
- **Block Address** : indirizzamento al blocco (invece che al Byte)
 - **Block Address** = **Address** / **Block size**
 - In binario, se **Block Size** è una potenza di 2, **Address** $\gg n$, dove $n = \log_2(\text{Block size})$
 - Questi n bit **meno significativi** dell'Address costituiscono il **byte offset del blocco**
- Nuova funzione di Mapping :
 - Block Address** = **Address** / **Block size**
 - Cache block INDEX** = **Block Address** % # cache blocks

Esempio di cache diretta con blocchi di 2 B



Blocco = 2 B

$\log_2 2 = 1 \text{ b}$

**Byte offset
del blocco**

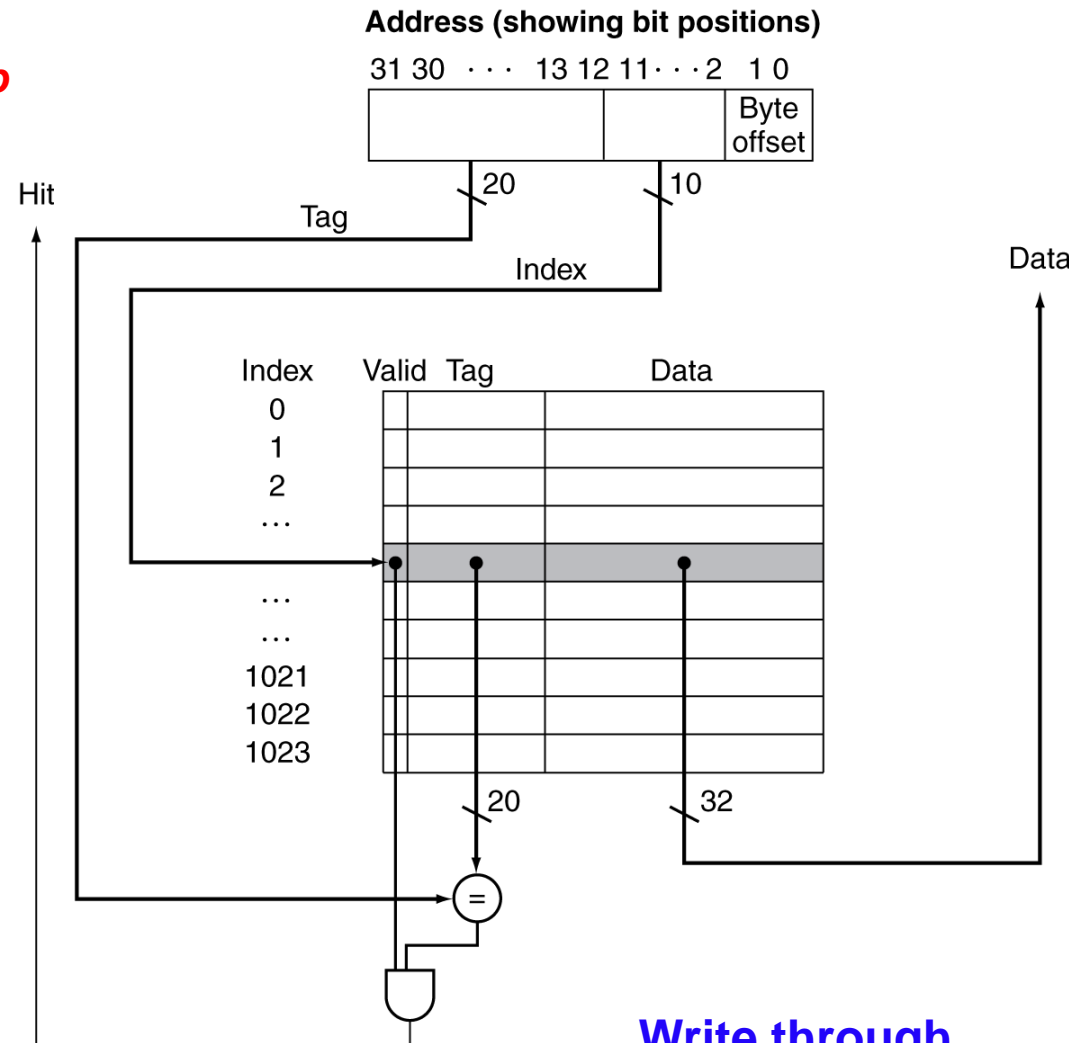
**Da non
considerare
per ottenere il
block address**

Tag e Valid Bit

- Come facciamo a conoscere quale particolare blocco di memoria è memorizzato in un certa locazione della cache?
 - Memorizziamo il **block address** assieme al blocco dei dati
 - In realtà, ci bastano solo i **bit high-order dell'address**
 - **Risparmiamo bit**
 - Chiamiamo questi bit **Tag**
 - In realtà **Tag** = Quoziente = **block address / block size**
- Come faccio a sapere se una certa locazione della cache non contiene dati, cioè è logicamente vuota?
 - **Valid bit**:
 - 1 = present
 - 0 = not present
 - Inizialmente uguale a 0

Cache ad accesso diretto (vecchio MIPS)

- **Byte OFFSET**
 - $n = \log_2(\text{Block size}) = \log_2(4) = 2 \text{ b}$
- **INDEX**
 - corrisponde a $\log_2(\# \text{ blocchi cache}) = \log_2(1024) = 10 \text{ b}$
 - 10 bit meno significativi del Block Address
 - Block Address ottenuto da Address rimuovendo gli $n=2$ bit del byte offset
- **TAG**
 - parte alta dell'Address, da memorizzare in cache assieme al blocco
 - $\text{TAG} = N \text{ bit addr.} - \text{INDEX} - \text{OFFSET} = 32 - 10 - 2 = 20 \text{ b}$
 - permette di risalire all'Address originale del blocco memorizzato
- **Valid**
 - il blocco è presente

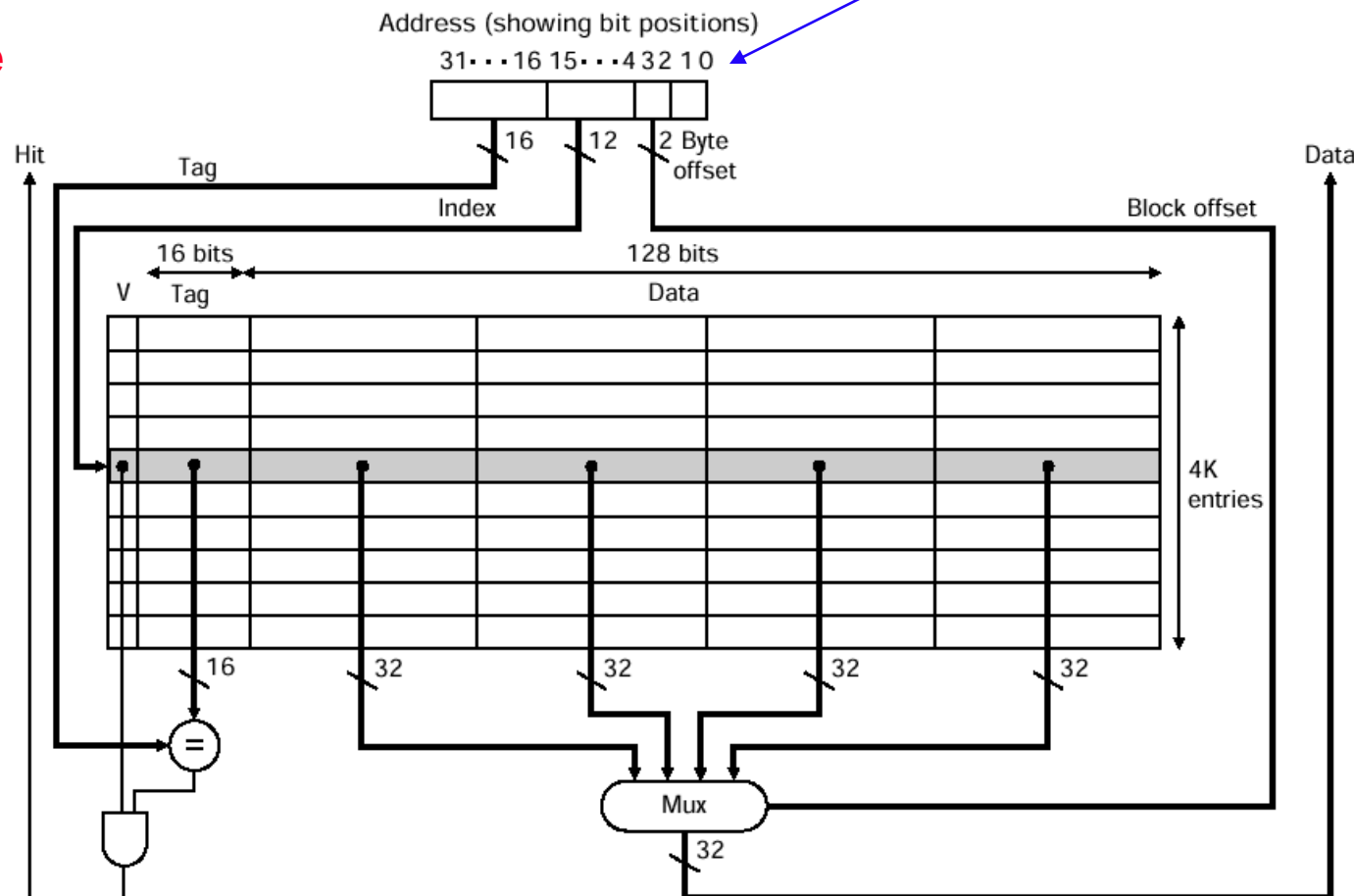


Blocco più grande di una word

- Approccio vantaggioso per ridurre il **Miss rate** se abbiamo
 - **località spaziale**
- Infatti, se si verifica un **Miss**
 - si carica un blocco grosso
 - se sono probabili accessi spazialmente vicini, questi cadono nello stesso blocco

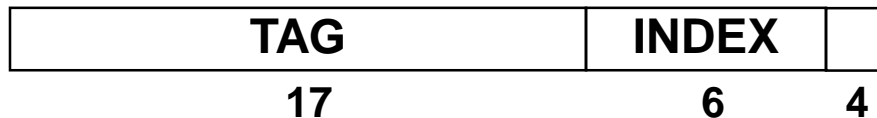
⇒ **Hit**

Offset ($n = \log_2 16 = 4$) suddiviso in due parti
Block offset (2 b) : seleziona word
Byte offset (2 b) : seleziona byte



Esempio

- Cache con 64 blocchi
- Blocchi di 16B
- Se l'indirizzo è di 27 bit, com'è composto?
 - INDEX deve essere in grado di indirizzare 64 blocchi: $SIZE_{INDEX} = \log_2(64)=6$
 - BLOCK OFFSET (non distinguiamo tra byte e block offset) deve essere in grado di indirizzare 16B: $SIZE_{BLOCK\ OFFSET} = \log_2(16)=4$
 - TAG corrisponde ai rimanenti bit (alti) dell'indirizzo:
 $SIZE_{TAG} = 27 - SIZE_{INDEX} - SIZE_{OFFSET} = 17$



- Qual è il blocco (INDEX) che contiene il byte all'indirizzo 1201 ?
 - Trasformo l'indirizzo al Byte nell'indirizzo al blocco: $1201/16 = 75$
 - L'offset all'interno del blocco è: $1201 \% 16 = 1$
 - L'index viene determinato con l'operazione di modulo: $75 \% 64 = 11$
 \Rightarrow il blocco è il 12° (INDEX=11), il byte del blocco è il 2° (OFFSET=1)
 - TAG: $75 / 64 = 1$

Problemi di progetto di una cache

- **Conflitti** nell'uso della cache
 - Se il blocco da portare in cache deve essere **sovrascritto** (sulla base della funzione di mapping) su un altro blocco di dati già presente in cache, cosa ne faccio del vecchio blocco?
 - Se il vecchio blocco è stato solo acceduto in lettura (**Read**), possiamo semplicemente rimpiazzarlo
 - Se il vecchio blocco è stato modificato (**Read/Write**), dipende dalle **politiche di coerenza** tra i livelli di memoria
 - **Write through** (scrivo sia in cache che in memoria)
 - **Write back** (scrivo solo in cache, e ritardo la scrittura in memoria)
 - Con politica **Write through** il blocco in conflitto può essere rimpiazzato senza problemi
 - Con politica **Write back** prima di rimpiazzare il blocco in conflitto, questo deve essere scritto in memoria

Hits vs. Miss

- **Read Hit** (come conseguenza di *i-fetch* e *load*)
 - accesso alla memoria con il massimo della velocità
- **Read Miss** (come conseguenza di *i-fetch* e *load*)
 - il controllo deve mettere in stallo la CPU (cicli di attesa, con registri interni immutati), finché la lettura del blocco (dalla memoria in cache) viene completata
 - Successivamente al completamento della lettura del blocco:
 - **Instruction cache miss**
 - Ripeti il fetch dell'istruzione
 - **Data cache miss**
 - Completa l'accesso al dato dell'istruzione (load)

Hits vs. Miss

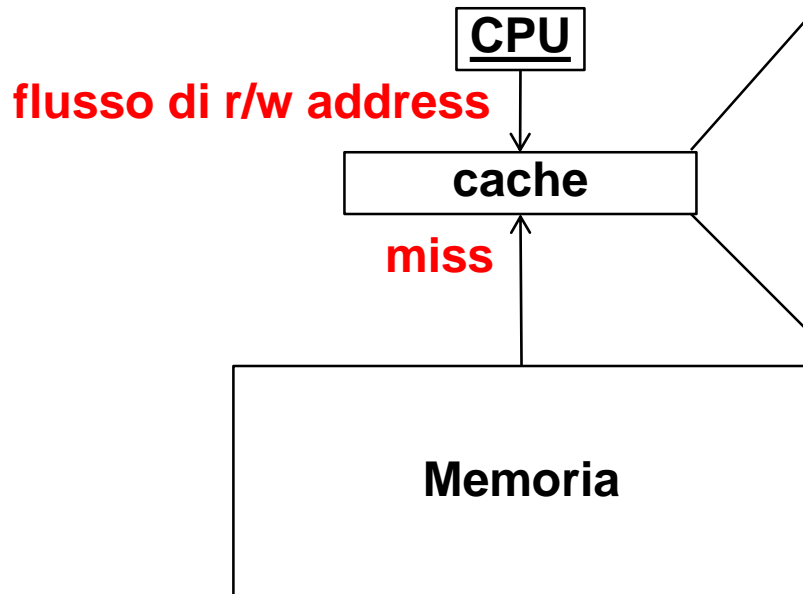
- **Write Hit** (solo come conseguenza di *store*)
 - write through: scrive sulla cache e in memoria
 - write back: scrive solo sulla cache, e segnala che il blocco è stato modificato (**setting di un bit di Dirty associato al blocco**)
- **Write Miss** (solo come conseguenza di *store*)
 - con politica *write-back*, stallo della CPU (cicli di attesa), **lettura** del blocco dalla memoria in cache (**write allocate**), completamento dell'istruzione di store in cache
 - con politica *write-through*, solitamente non si ricopia il blocco in cache prima di effettuare la scrittura (**no write allocate**) che avviene direttamente in memoria

Ottimizzazione Write-Through

- Le write pongono problemi con politica *write-through*
 - Le write diventano **più lunghe**, anche in presenza di **Write Hit**
 - Esempio:
 - di base abbiamo che: $CPI = 1$
 - se il 10% delle istruzioni fossero *store*, e ogni accesso alla memoria costasse 100 cicli:
$$CPI = 1 + 0.1 \times 100 = 11$$
- Soluzione:
 - **Write buffer** come memoria tampone «veloce» tra cache e memoria, per nascondere la latenza di accesso alla memoria
 - i blocchi sono scritti temporaneamente nel *write buffer*, in attesa della scrittura asincrona in memoria
 - il processore può proseguire senza attendere, a meno che il *write buffer* sia pieno

Esempio di funzionamento

- Si consideri una *cache diretta*, e si assuma che l'indirizzo sia di **24 bit**. La dimensione del blocco è di **16 B**, mentre la cache ha **16 ingressi**.
 - **OFFSET: $\log 16 = 4$ b**
 - **INDEX: $\log 16 = 4$ b**
 - **TAG: $24 - \text{INDEX} - \text{OFFSET} = 16$ b**
- Si supponga che i 16 ingressi della cache siano tutti non validi (**VALID=0**)



VALID	TAG	DATA
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		

Esempio di funzionamento

- Flusso di accessi r/w: indirizzi di memoria a 24 b

0x 1AB090: TAG: 1AB0 IND: 9 OFF: 0

→ miss

	VALID	TAG	DATA
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		
8	0		
9	1	1AB0	Mem[1AB090]
10	0		
11	0		
12	0		
13	0		
14	0		
15	0		

Esempio di funzionamento

- Flusso di accessi r/w: indirizzi di memoria a 24 b

0x 1AB090: TAG: 1AB0 IND: 9 OFF: 0

→ miss

0x 2AB090: TAG: 2AB0 IND: 9 OFF: 0

→ conflitto e miss (rimpiazzo)

conflitto

	VALID	TAG	DATA
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		
8	0		
9	1	1AB0	Mem[1AB090]
10	0		
11	0		
12	0		
13	0		
14	0		
15	0		

Esempio di funzionamento

- Flusso di accessi r/w: indirizzi di memoria a 24 b

0x 1AB090: TAG: 1AB0 IND: 9 OFF: 0

→ miss

0x 2AB090: TAG: 2AB0 IND: 9 OFF: 0

→ conflitto e miss

	VALID	TAG	DATA
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		
8	0		
9	1	2AB0	Mem[2AB090]
10	0		
11	0		
12	0		
13	0		
14	0		
15	0		

Esempio di funzionamento

- Flusso di accessi r/w: indirizzi di memoria a 24 b

0x 1AB090: TAG: 1AB0 IND: 9 OFF: 0

→ miss

0x 2AB090: TAG: 2AB0 IND: 9 OFF: 0

→ conflitto e miss

0x 1AB094: TAG: 1AB0 IND: 9 OFF: 4

→ conflitto e miss

	VALID	TAG	DATA
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		
8	0		
9	1	1AB0	Mem[1AB090]
10	0		
11	0		
12	0		
13	0		
14	0		
15	0		

Esempio di funzionamento

- Flusso di accessi r/w: indirizzi di memoria a 24 b

0x 1AB090: TAG: 1AB0 IND: 9 OFF: 0

→ miss

0x 2AB090: TAG: 2AB0 IND: 9 OFF: 0

→ conflitto e miss

0x 1AB094: TAG: 1AB0 IND: 9 OFF: 4

→ conflitto e miss

0x 1AB090: TAG: 1AB0 IND: 9 OFF: 0

→ hit

	VALID	TAG	DATA
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		
8	0		
9	1	1AB0	Mem[1AB090]
10	0		
11	0		
12	0		
13	0		
14	0		
15	0		

Esempio di funzionamento

- Flusso di accessi r/w: indirizzi di memoria a 24 b

0x 1AB090: TAG: 1AB0 IND: 9 OFF: 0

→ miss

0x 2AB090: TAG: 2AB0 IND: 9 OFF: 0

→ conflitto e miss

0x 1AB094: TAG: 1AB0 IND: 9 OFF: 4

→ conflitto e miss

0x 1AB090: TAG: 1AB0 IND: 9 OFF: 0

→ hit

0x 122010: TAG: 1220 IND: 1 OFF: 0

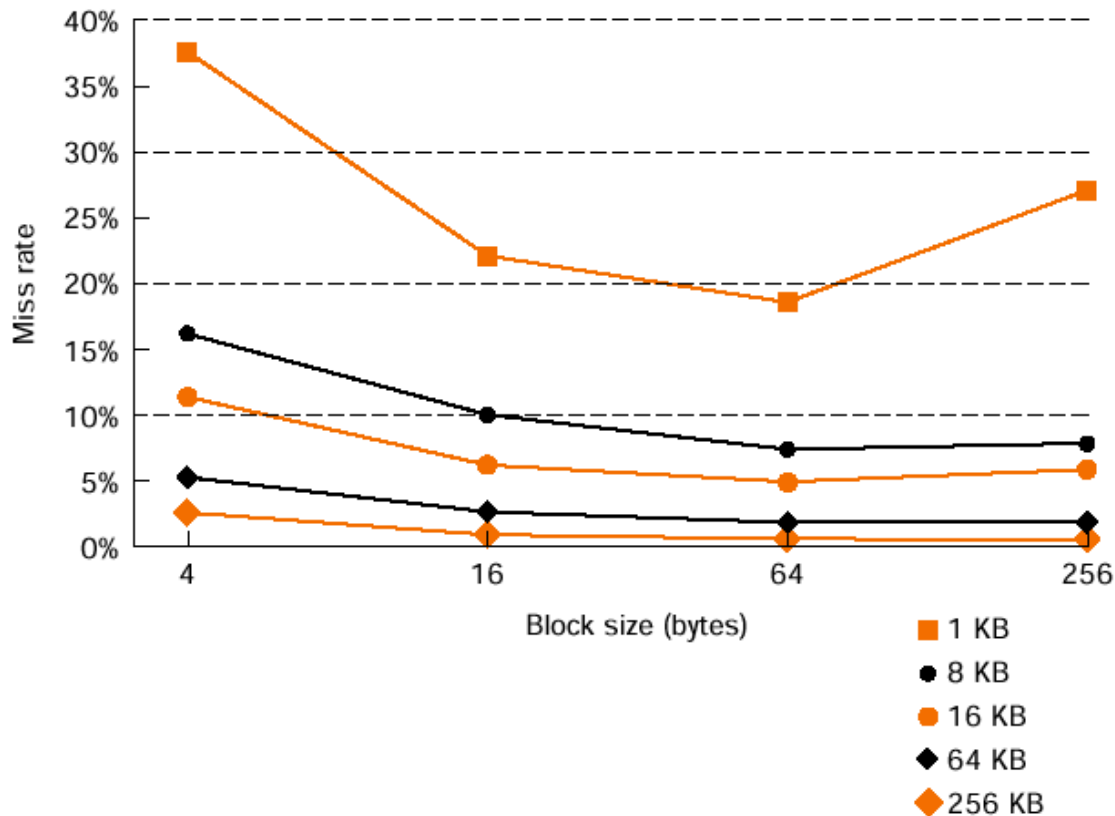
→ miss

	VALID	TAG	DATA
0	0		
1	1	1220	Mem[122010]
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		
8	0		
9	1	1AB0	Mem[1AB090]
10	0		
11	0		
12	0		
13	0		
14	0		
15	0		

Costo dei miss

- Aumentare la dimensione dei blocchi
 - può diminuire il *miss rate*, in presenza di località spaziale
 - aumenta il *miss penalty*
- Quanto costa il miss ?
 - dipende (parzialmente) dalla dimensione del blocco:
 - **Costo miss** = **Costante** + **Costo proporzionale al block size**
 - La **Costante** modella i cicli spesi per inviare l'indirizzo e attivare la DRAM
 - Ci sono varie organizzazioni della memoria per diminuire il costo di trasferimento di grandi blocchi di byte
- In conclusione
 - Raddoppiando il **block size** non si raddoppia il *miss penalty*
- Allora, perché non si usano comunque blocchi grandi invece che piccoli ?
 - **esiste un tradeoff !!!** Portare in memoria un blocco grande, senza usarlo completamente è uno spreco per l'utilizzo della cache (conflitti con altri blocchi utilizzati)

Aumento del block size



- Frequenza di miss in genere diminuisce all'aumentare della dimensione del blocco ⇐ **vantaggio dovuto alla località spaziale !!**
- Se il blocco diventa troppo grande rispetto al size della cache, i vantaggi della località spaziale diminuiscono. Per cache piccole aumenta la frequenza di miss a causa di conflitti (blocchi diversi caratterizzati dallo stesso INDEX)
⇐ **aumenta la competizione nell'uso della cache !!**

Aumento del block size

Program	Block size in words	Instruction miss rate	Data miss rate	Effective combined miss rate
gcc	1	6.1%	2.1%	5.4%
	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
	4	0.3%	0.6%	0.4%

- Nota che aumentando la dimensione del blocco, la riduzione più marcata, soprattutto per gcc, si ha per l'**Instruction Miss Rate**
 - la località spaziale è maggiore per la lettura/fetch delle istruzioni
- Per blocchi di una sola parola
 - write miss non conteggiati

Prestazioni

- Modello semplificato:

$$\text{CPU time} = (\text{execution cycles} + \text{stall cycles}) \times \text{cycle time}$$

$$\text{stall cycles} = \text{IC} \times \text{miss ratio} \times \text{miss penalty}$$

- Il **miss ratio/rate** (ed anche gli **stall cycles**) possono essere distinti in
 - instruction miss ratio (fetch istruzioni)
 - write miss ratio (store)
 - read miss ratio (load)**considerati assieme: data miss ratio**
- Per il **miss penalty** possiamo semplificare, considerando un penalty unico per scritture/letture
- Per migliorare le prestazioni, dobbiamo
 - **diminuire il miss ratio** e/o il **miss penalty**
- Cosa succede se aumentiamo il block size?
diminuisce (per cache abbastanza grandi) il miss rate, ma aumenta (di poco) il miss penalty

Esempio (1)

- Conoscendo
miss penalty, instruction miss ratio, data miss ratio,
CPI ideale (senza considerare l'effetto della cache)
è possibile calcolare di quanto rallentiamo rispetto al caso ideale
(memoria ideale)
- In altri termini, è possibile riuscire a conoscere il CPI reale:
 - $\text{CPI}_{\text{reale}} = \text{CPI}_{\text{ideale}} + \text{cicli di stallo medi per istr.}$dove i cicli di stallo sono dovuti ai miss (e alla penalty associata)
- Programma gcc:
 - instr. miss ratio = 2%
 - data miss ratio = 4%
 - numero lw/sw = 36% IC
 - $\text{CPI}_{\text{ideal}} = 2$
 - miss penalty = 40 cicli

Esempio (2)

- Cicli di stallo dovuti alle *instruction miss*
 - $(\text{instr. miss ratio} \times \text{IC}) \times \text{miss penalty} = (0.02 \times \text{IC}) \times 40 = 0.8 \times \text{IC}$
- Cicli di stallo dovuti ai *data miss*
 - $(\text{data miss ratio} \times \text{num. lw/sw}) \times \text{miss penalty} = (0.04 \times (0.36 \times \text{IC})) \times 40 = 0.58 \times \text{IC}$
- Cicli di stallo **totali** dovuti ai miss = $1.38 \times \text{IC}$
- Cicli di stallo **medi per istr.** dovuti ai miss = $1.38 \times \text{IC} / \text{IC} = 1.38$
- Numero di cicli totali:
 - $\text{CPI}_{\text{ideal}} \times \text{IC} + \text{Cicli di stallo totali} = 2 \times \text{IC} + 1.38 \times \text{IC} = 3.38 \times \text{IC}$
- $\text{CPI}_{\text{reale}} = \text{Numero di cicli totali} / \text{IC} = (3.38 \times \text{IC}) / \text{IC} = 3.38$
- $\text{CPI}_{\text{reale}} = \text{CPI}_{\text{ideale}} + \text{cicli di stallo medi per istr.} = 2 + 1.38 = 3.38$
- Per calcolare lo speedup basta confrontare i CPI, poiché IC e Frequenza del clock sono uguali:
 - $\text{Speedup} = \text{CPI}_{\text{reale}} / \text{CPI}_{\text{ideale}} = 3.38 / 2 = 1.69$

Considerazioni

- Se velocizzassi la CPU e lasciassi immutato il sottosistema di memoria?
 - il tempo assoluto di penalty per risolvere il miss sarebbe lo stesso
- Posso velocizzare la CPU in 2 modi:
 - cambio l'organizzazione interna
 - aumento la frequenza di clock
- Se cambiassi l'organizzazione interna, diminuirebbe il CPI_{ideale}
 - purtroppo miss rate e miss penalty rimarrebbero invariati, per cui rimarrebbero invariati i cicli di stallo totali dovuti ai miss
- Se aumentassi la frequenza
 - il CPI_{ideale} rimarrebbe invariato, ma aumenterebbero i cicli di stallo totali dovuti ai miss \Rightarrow aumenterebbe il CPI_{reale}
 - infatti, il penalty per risolvere i miss rimarrebbe lo stesso, ma il numero di cicli risulterebbe maggiore perché i cicli sono più corti !!

Diminuiamo i miss con l'associatività

- **Diretta**
 - ogni blocco di memoria *associato* con un solo possibile blocco della cache
 - accesso sulla base dall'indirizzo
- **Completamente associativa**
 - ogni blocco di memoria *associato* con un qualsiasi blocco della cache
 - accesso non dipende dall'indirizzo (bisogna cercare in ogni blocco)
- **Associativa su insiemi**
 - compromesso

One-way set associative

(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

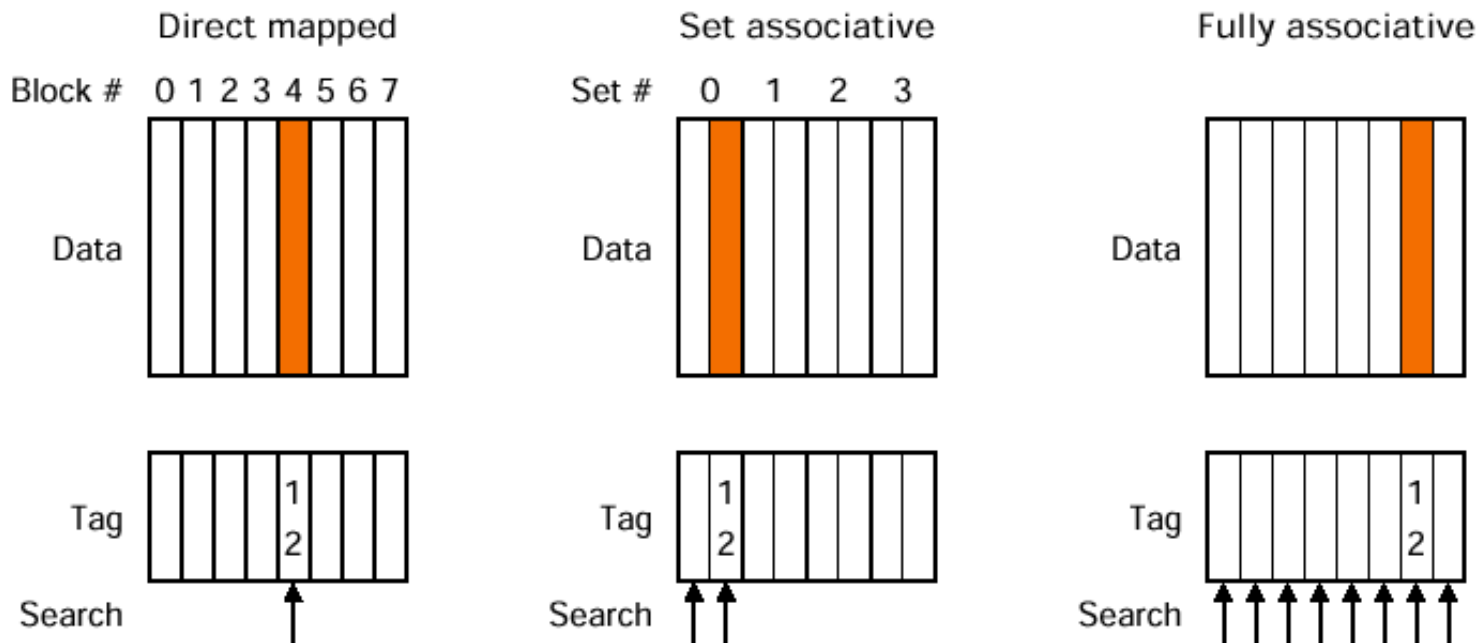
Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

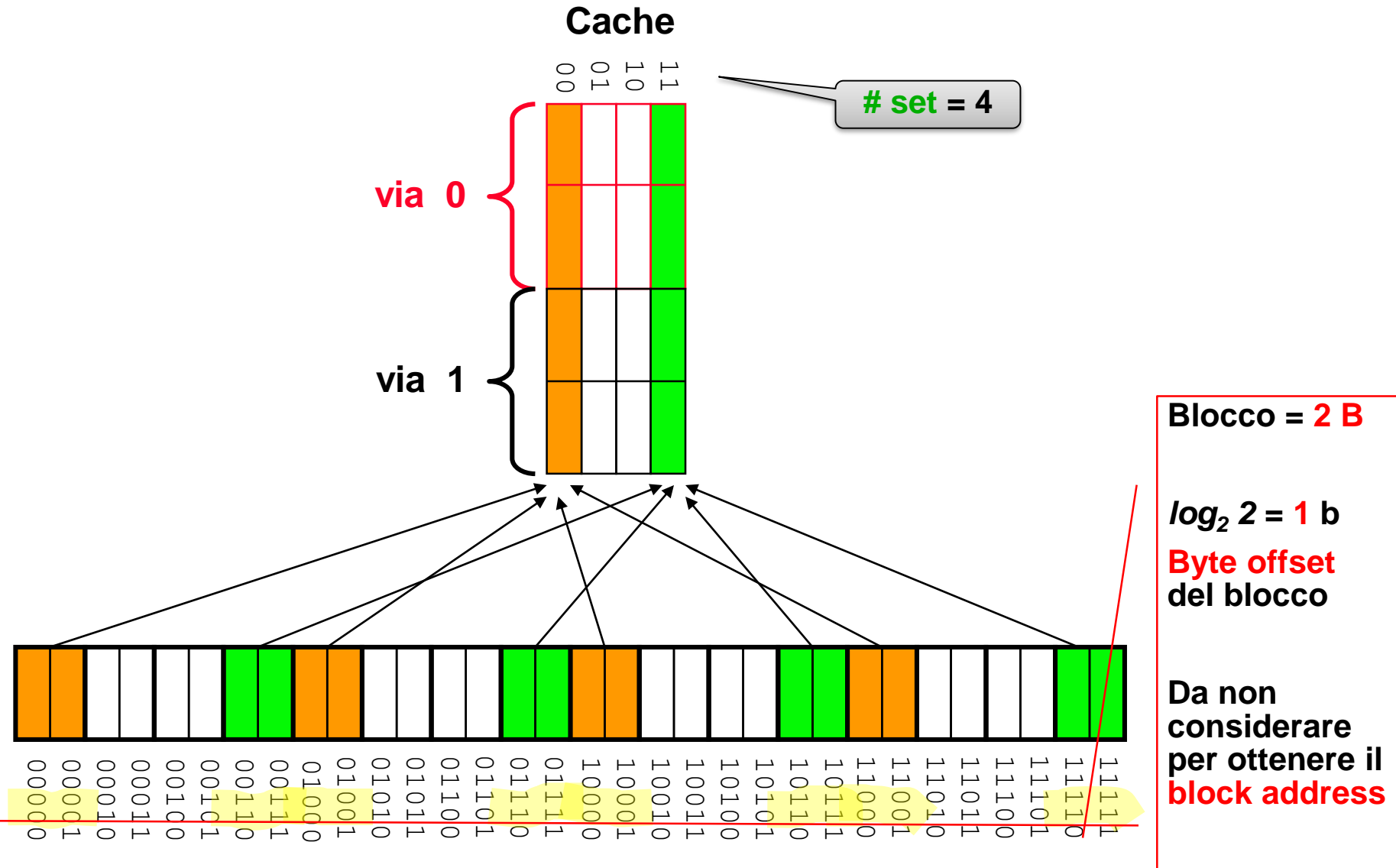
Set associativo

- Per insiemi di 2/4/8/16 ... blocchi \Rightarrow cache set-associative a 2/4/8/16 vie ...
- Cache diretta \equiv Cache set-associative a 1 via
- Nuova funzione di mapping
 - Block Address** = **Address** / **Block size**
 - Cache block INDEX** = **Block Address** % **# set**
- L'INDEX viene usato per determinare l'insieme (set)
- Dobbiamo controllare tutti i TAG associati ai vari blocchi del **set** per individuare il blocco

Funzione indipendente dal numero di elementi (blocchi) contenuti in ogni **set**



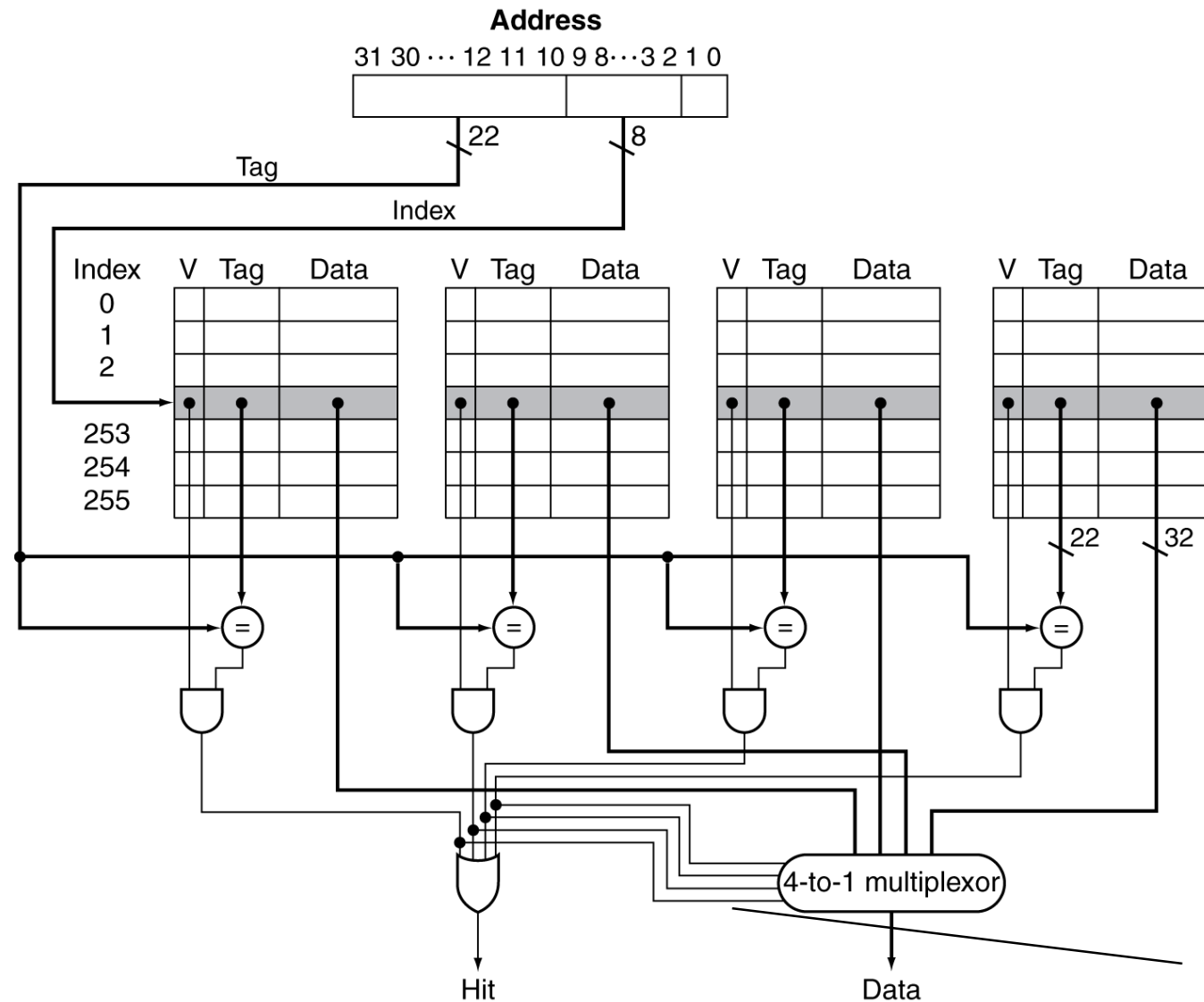
Cache associativa a 2 vie (blocchi di 2 B)



Scelta del blocco da sostituire

- In caso di **miss**, possiamo dover sostituire un blocco
- Cache diretta
 - se il blocco corrispondente ad un certo INDEX è occupato (bit V=1), allora dobbiamo rimpiazzare il blocco
 - se il vecchio blocco è stato modificato e abbiamo usato politica di *write-back*, dobbiamo aggiornare la memoria
- Cache associativa
 - INDEX individua un insieme di blocchi
 - se nell'insieme c'è un **blocco libero**, lo **usiamo per risolvere il miss**
 - se **tutti i blocchi sono occupati**, dobbiamo **scegliere il blocco da sostituire per risolvere il miss**
 - sono possibili diverse politiche per il rimpiazzamento
 - LRU (Least Recently Used) - necessari bit aggiuntivi per considerare quale blocco è stato usato recentemente
 - Casuale

Un'implementazione (4-way set-associative)




- **Nota**
 - i 4 comparatori
 - il multiplexer
- **Vantaggio**
 - minore miss rate
- **Svantaggio**
 - aumenta tempo di hit

0000: **miss**
0001, 0010, 0100,
oppure 1000: **hit**

Esempio di cache diretta vs. associativa

- Compariamo diverse cache, tutte composte da **4 blocchi**
 1. Direct mapped
 2. 2-way set associative,
 3. Fully associative
- Sequenza di block address: **0, 8, 0, 6, 8**
- **Cache Set Index** = $\text{block_address} \% \text{\#Sets}$

Direct mapped (4 set, ciascuno con un solo elemento):




Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	Mem[0]			
8	0	miss	Mem[8]			
0	0	miss	Mem[0]			
6	2	miss	Mem[0]		Mem[6]	
8	0	miss	Mem[8]		Mem[6]	

Esempio di cache diretta vs. associativa


Sequenza di block address: 0, 8, 0, 6, 8

2-way set associative (2 set, ciascuno 2 elementi):



Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		Set 1	
0	0	miss	Mem[0]			
8	0	miss	Mem[0]	Mem[8]		
0	0	hit	Mem[0]	Mem[8]		
6	0	miss	Mem[0]	Mem[6]		
8	0	miss	Mem[8]	Mem[6]		

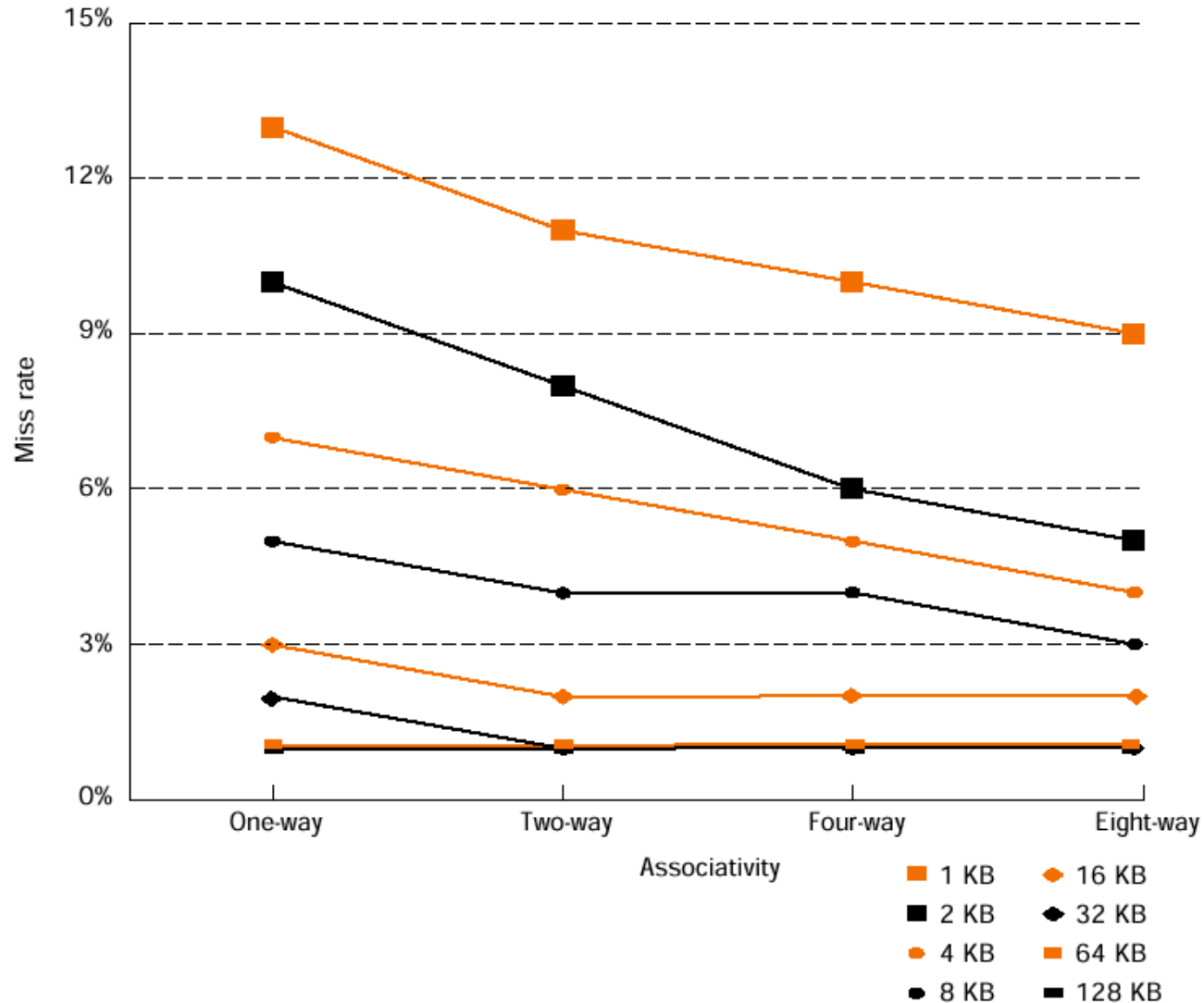
Fully associative (1 set, di 8 elementi):



Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	Mem[8]	Mem[6]	

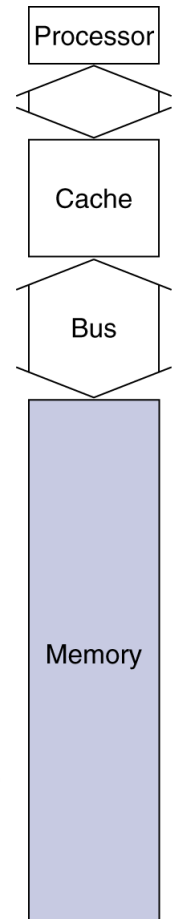
Associatività e miss rate

- Le curve a lato si riferiscono a
 - blocchi di 32 B
 - benchmark Spec92 interi
- Benefici maggiori per cache piccole
 - perché si partiva da un miss rate molto alto nel caso diretto (one-way)



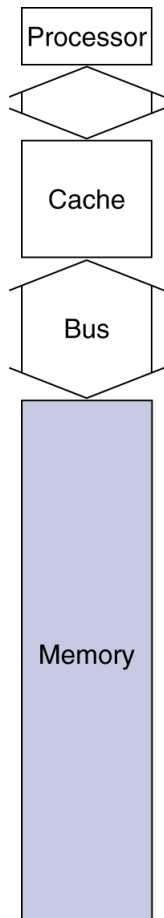
Memoria Principale e Miss penalty

- Si usa la DRAM per realizzare la main memory
 - Fixed width (es.: dato letto/scritto = 1 word)
 - DRAM connessa alla cache da un bus clock-ato, anch'esso fixed-width (numero fisso di fili)
 - Il clock del bus è tipicamente più lento di quello della CPU
- Esempio di una read di un blocco di cache da 1-word (bus width = 1 word)
 - 1 bus cycle: invio dell'address
 - 15 bus cycles: DRAM access
 - 1 bus cycle: data transfer
- Lettura di un blocco da 4-word: 1-word-wide DRAM
 - Miss penalty = $1 + 4 \times 15 + 4 \times 1 = 65$ bus cycles
 - Bandwidth = $16 \text{ bytes} / 65 \text{ cycles} = 0.25 \text{ B/cycle}$

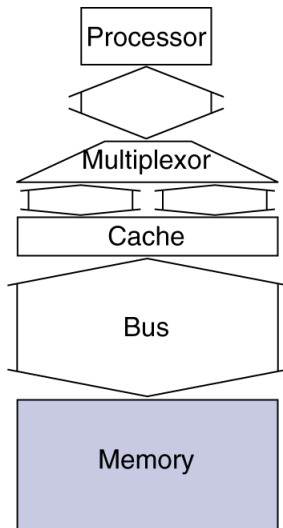


a. One-word-wide memory organization

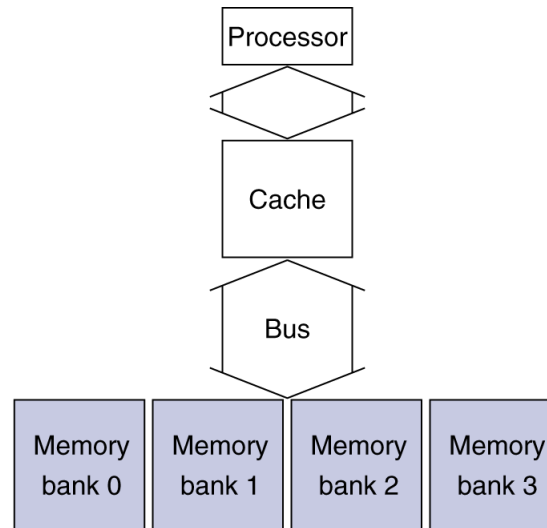
Aumentare la DRAM Bandwidth e diminuire la latenza



a. One-word-wide
memory organization



b. Wider memory organization



c. Interleaved memory organization

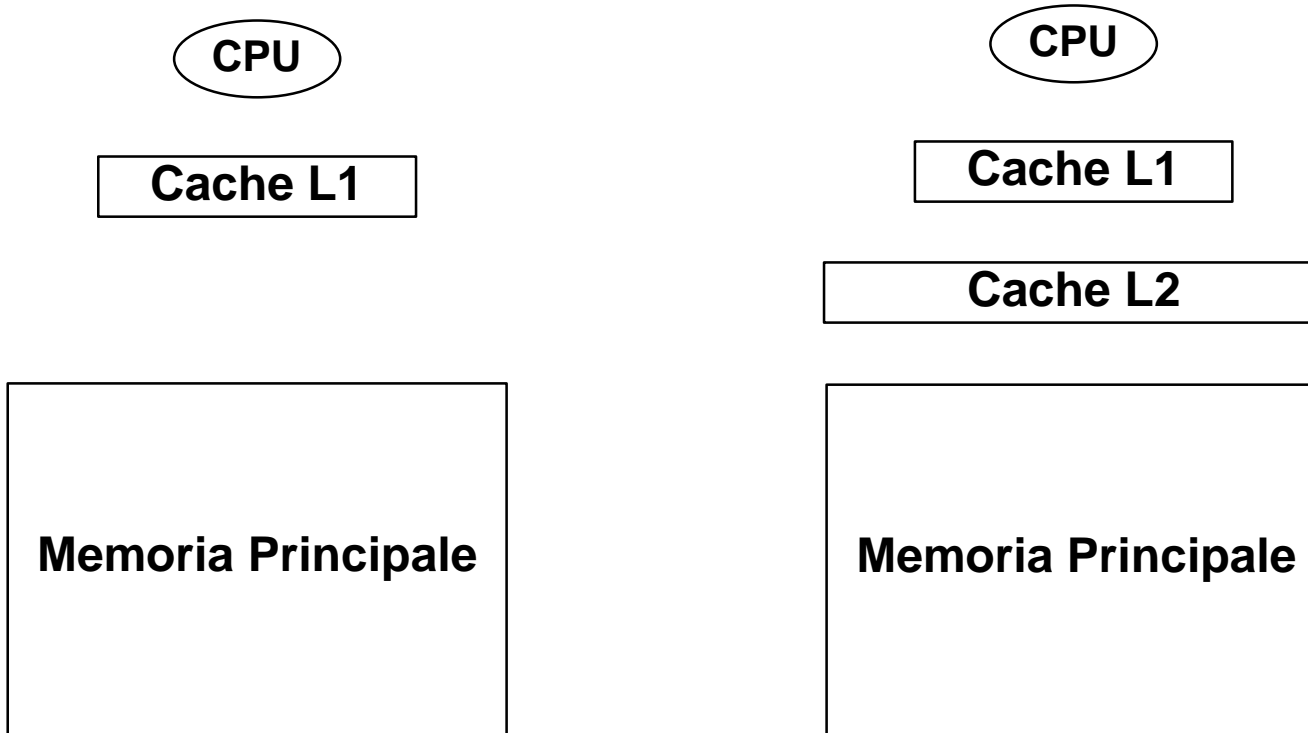
- **4-word wide memory**
 - **Miss penalty = 1 + 15 + 1 = 17 bus cycles**
 - **Bandwidth = 16 bytes / 17 cycles = 0.94 B/cycle**
- **4-bank interleaved memory (sempre con 1-word memories)**
 - **Miss penalty = 1 + 15 + 4 × 1 = 20 bus cycles**
 - **Bandwidth = 16 bytes / 20 cycles = 0.8 B/cycle**

Organizzazioni DRAM avanzate (per diminuire la latenza per blocchi grandi)

- I bits di una DRAM moderna sono organizzati come un array rettangolare/bidimensionale row-major
 - la DRAM accede un'intera riga (blocco)
 - Burst mode: word successive che compongono la riga acceduta sono prodotte in output con latenza ridotta, es., una word per cycle
- Double data rate (DDR) DRAM
 - Capaci di trasferire sia quando il segnale di clock sale e sia quando scende
- Quad data rate (QDR) DRAM
 - Gli input e output della DDR sono separati, raddoppiando ancora la capacità rispetto alla DDR

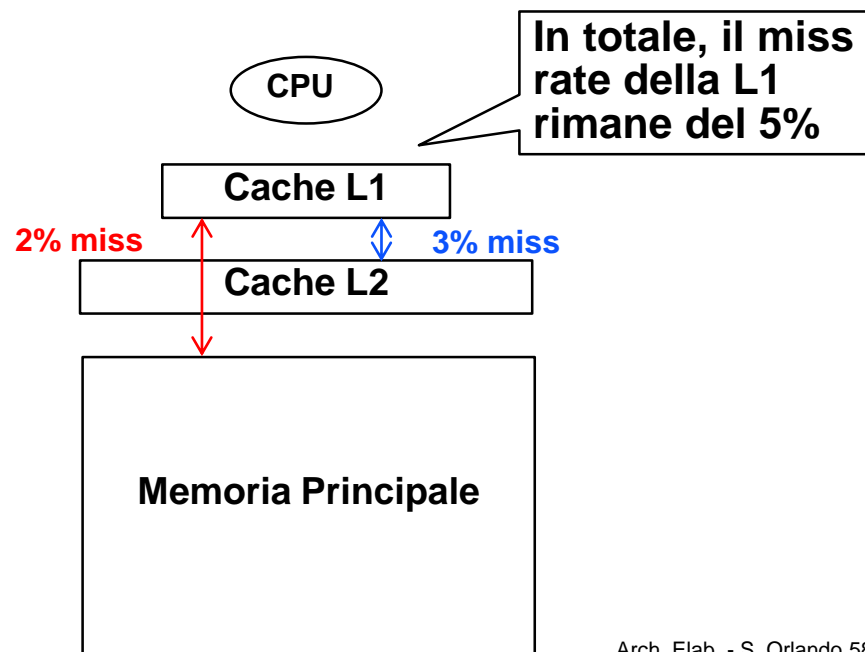
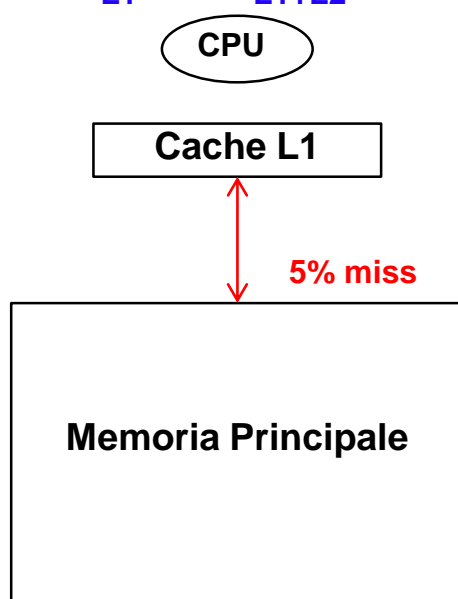
Cache a più livelli

- CPU con
 - cache di 1^a livello (L1), di solito sullo stesso chip del processore
 - cache di 2^a livello (L2), interna/esterna al chip del processore, implementata con SRAM
 - cache L2 serve a ridurre il **miss penalty** per la cache L1
 - solo se il dato è presente in cache L2



Cache a più livelli (esempio)

- $CPI=1$ su un processore a 500 MHz con cache unica (L1), con miss rate del 5%, e un tempo di accesso alla DRAM (miss penalty) di **200 ns (100 cicli)**
 - $CPI_{L1} = CPI + 5\% \cdot 100 = 1 + 5 = 6$
- Cache L2 con tempo di accesso di **20 ns (10 cicli)**, il miss rate della cache L1 rispetto alla DRAM viene ridotto al 2%
 - il miss penalty in questo caso aumenta (200 ns + 20 ns, ovvero **110 cicli**)
 - il restante 3% viene risolto dalla cache L2
 - il miss penalty in questo caso è solo di 20 ns
 - $CPI_{L1+L2} = CPI + 3\% \cdot 10 + 2\% \cdot 110 = 1 + 0.3 + 2.2 = 3.5$
- **Speedup = $CPI_{L1} / CPI_{L1+L2} = 6 / 3.5 = 1.7$**



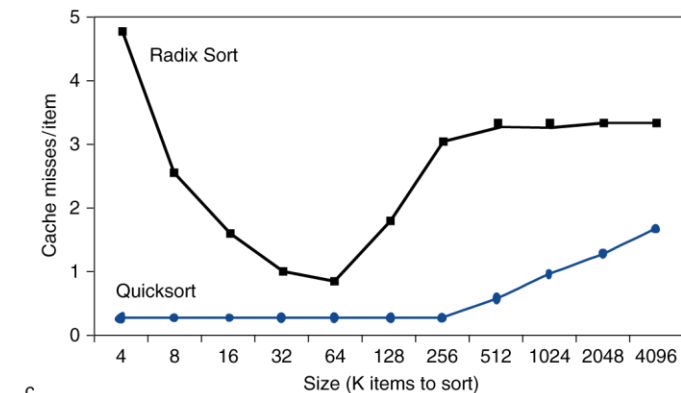
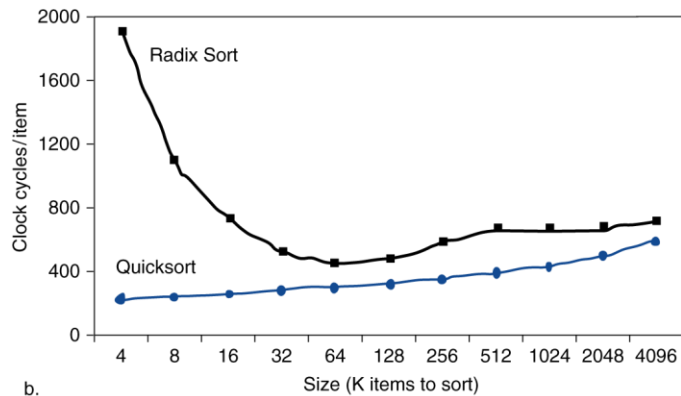
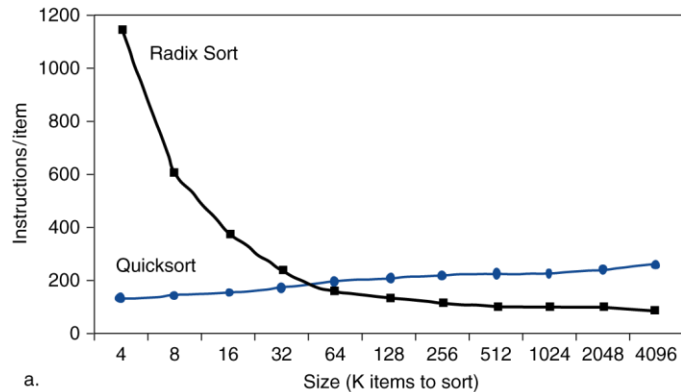
Cache a 2 livelli

- **Cache L1: piccola e con blocchi piccoli, di solito con maggior grado di associatività, il cui scopo è**
 - **ottimizzare l'hit time per diminuire il periodo del ciclo di clock**
- **Cache L2: grande e con blocchi più grandi, con minor grado di associatività, il cui scopo è**
 - **ridurre il miss rate verso la DRAM (Memoria Principale)**
 - **la maggior parte dei miss sono risolti dalla cache L2**

CPU avanzate e cache

- Le CPU out-of-order possono eseguire altre istruzioni durante i cache miss, sovrapponendo **calcolo utile** all'**attesa della memoria** (penalty)
 - le write/read **pendenti** sono allocate a speciali load/store unit
 - le istruzioni **dipendenti** attendono nelle cosiddette “reservation stations”
 - le istruzioni **indipendenti** continuano l'esecuzione
- L'effetto delle miss sulle prestazioni dipendono da molti fattori
 - E' più difficile analizzare le prestazioni
 - Non possiamo applicare formule analitiche, necessaria la simulazione

L'effetto del software sulle hit-rate della cache



- Il miss rate dipende dai pattern di accesso alla memoria dovuto all'algoritmo
- Ottimizzazioni del compilatore per migliorare i pattern di accesso alla memoria