

# Gerarchia di Memoria e Cache

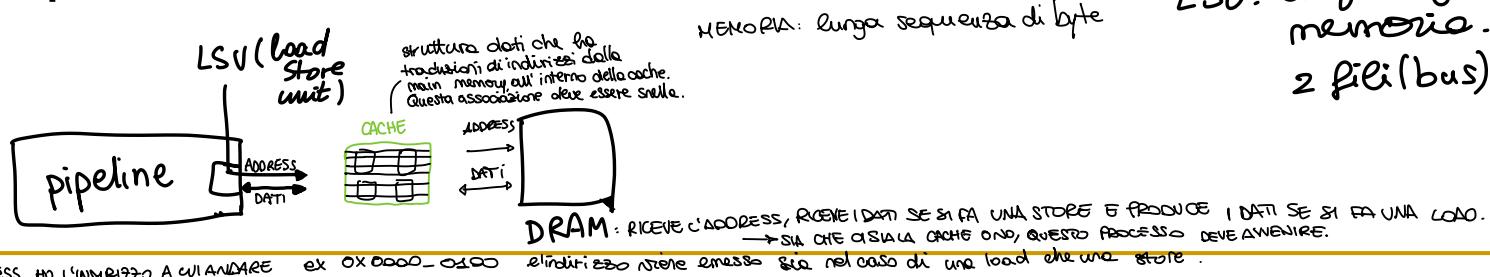
Andrea Bartolini – [a.bartolini@unibo.it](mailto:a.bartolini@unibo.it)

# Nozioni di base e funzionamento della cache

# Cache

memoria veloce basata su SRAM tra le DRAM e le Load/Store unit al fine di ridurre il tempo di accesso alla mem.

- Genericamente, qualsiasi struttura che «ricorda» i risultati utilizzati di frequente per evitare di ripetere le operazioni a lunga latenza necessarie per riprodurre i risultati da zero, ad esempio una cache Web
- Più comune nel contesto della progettazione del processore: una struttura di memoria SRAM gestita automaticamente
- Memorizza in SRAM le locazioni di memoria DRAM a cui si accede più di frequente per evitare di pagare ripetutamente per la latenza di accesso alla DRAM



LSU: esegue gli accessi in memoria.  
2 fili (bus)

PRENDO UNA PARTE DI INDIRIZZI CHE MI IDENTIFICANO DUE "FETTINI" DEL MIO SPAZIO DI INDIRIZZAMENTO.

POI DICO CHE NELLA CACHE HO SPAZIO X 4 FETTINI.

QUANDO FACCO UN ACCESSO UNA PARTE DELL'INDIRIZZO MI IDENTIFICA<sup>RA</sup> A QUALE FETTINA IN MEMORIA STO CERCANDO DI ACCEDERE

LA CACHE PRENDERÀ LA FETTINA CORRISPONDENTE ALL'INDIRIZZO, LA PORTERÀ IN UNO DEI SUOI BUCCCHI, E GLI ASSOCIERÀ UN'ETICHETTA IDENTIFICATIVA DELLA FETTINA.

Prenderò alcuni bit dell'indirizzo, che funziona da etichetta con la quale controllo se quella <sup>punta di memoria</sup> e' già nella cache nel caso accedo lì.

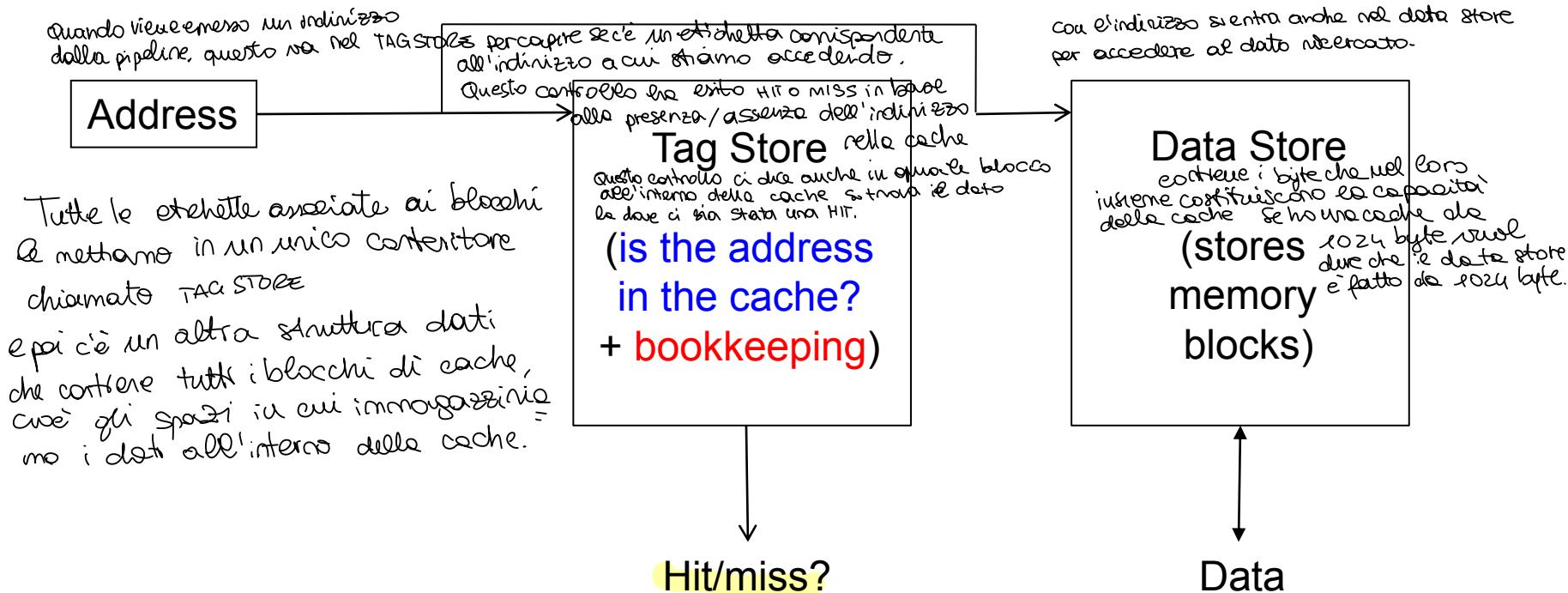
# Caching Basics

- **Blocco/Linea (Block/line):** Unità di archiviazione nella cache
  - La memoria è suddivisa logicamente in blocchi di cache che eseguono il mapping alle posizioni nella cache
- A seguito di un accesso in memoria:
  - **HIT:** Se indirizzo presente nella cache => utilizzo il dato memorizzato nella cache anziché prelevarlo dalla memoria
  - **MISS:** Se indirizzo non presente nella cache => porto il blocco relativo nella cache
    - Forse devo portar fuori qualcos'altro per inserire il nuovo blocco
- Parametri di progetto della cache
  - **Placement:** where and how to place/find a block in cache?
  - **Replacement:** what data to remove to make room in cache?
  - **Granularity of management:** large or small blocks?
  - **Write policy:** what do we do about writes?
  - **Instructions/data:** do we treat them separately?

Il suddivisione delle memoria  
dovrà avere sezioni di pari dimensioni  
dei blocchi/delle linee della cache.

TAG STORE: struttura dati HW che contiene le etichette associate ai blocchi delle cache.  
+ DATA STORE: contiene i blocchi di dati all'interno delle cache.

# Cache Abstraction and Metrics

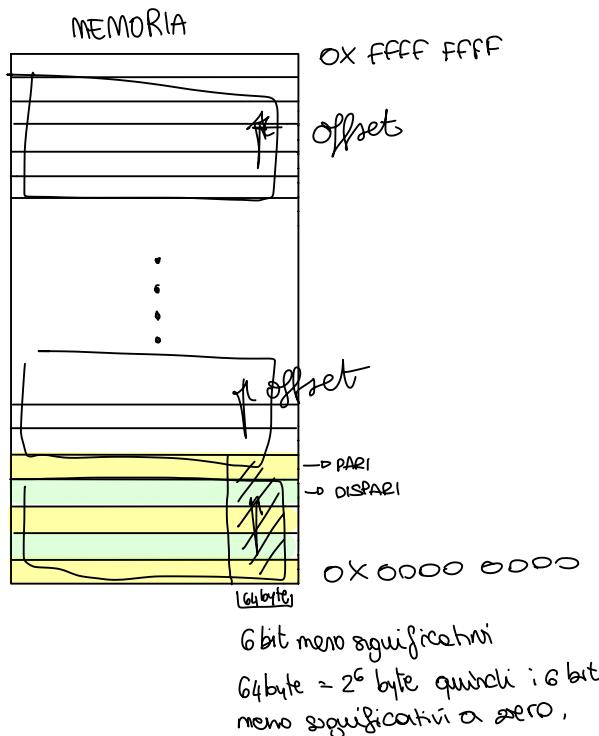


% di accessi che trovano un hit nelle cache.

- Cache hit rate =  $(\# \text{ hits}) / (\# \text{ hits} + \# \text{ misses}) = (\# \text{ hits}) / (\# \text{ accesses})$
- Average memory access time (AMAT)  
 $= (\text{hit-rate} * \text{hit-latency}) + (\text{miss-rate} * \text{miss-latency})$
- \* miss-latency = include anche la hit-latency.

# A Basic Hardware Cache Design

- Inizieremo con una progettazione della cache hardware di base
- Poi, esamineremo diverse idee per migliorarla



Se voglio dividere gli indirizzi poi de quelli disponibili.  
Quelli pari hanno il LSB a zero e sono a posti alternati

Passo a dividere logicamente la memoria in blocchi di dimensioni variabili.  
es. blocco da 64 byte → 64 byte corrispondono a  $2^6$  byte  
e quindi corrispondono agli indirizzi con i 6 bit meno significativi tutti pari a zero.

Avrò blocchi multipli di 64 byte e sarà identificato da i 32-6 bit di indirizzamento più significativi.  
Se definisco blocchi da 64 byte sto dicendo che

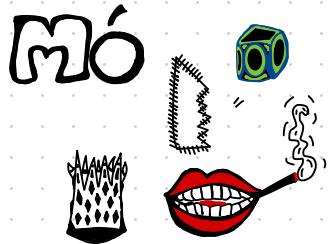
dividere lo spazio di indirizzamento in blocchi equivale a guardare un sottogruppo di bit + significativi del mio spazio di indirizzamento. I bit meno signif. identificano dei byte contigui all'interno dello spazio d'indirizz., i bit + significativi identificano regioni di memoria una contigue all'altra.

La parte meno significativa dei bit mi identifica un offset nel blocco  
quella più significativa identifica un blocco di memoria.

I rimanenti bit .32-6 e otengo un indice di quale blocco considero con quell'indirizzo

divido l'indirizzo espresso dalla load/store unit in due parti : offset all'interno del blocco  
e indice del blocco in memoria.

L'identificativo del blocco di memoria a cui sto cercando di fare accesso devo rimpicciolirlo dentro  
agli spazi di cui dispiego all'interno delle cache.



Suppongo di avere una cache che può immagazzinare 8 blocchi, di dimensione 64 byte.

→  $8 \times 64$  byte mi serve sapere dove può finire un blocco di memoria tra queste 8 posizioni.

Se la cache non mette restrizioni, quei blocchi li posso immagazzinare in qualsiasi di questi blocchi.

I bit più significativi costituiscono l'etichetta.

oppure restriuisco il mio campo etichetta ai bit 32-6-3, che saranno l'indice che identifica il blocco di memoria.

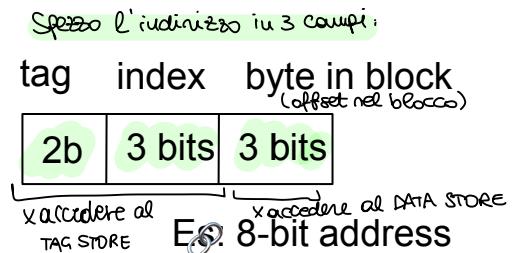
Siamo passati da una cache directly mapped dove un multiplo di 8 di ciascun blocco di memoria può stare solo dentro a un blocco di cache, a una cache associative dove non c'è questo vincolo.

Cache set associative: restringo un blocco di memoria a stare in una delle linee di cache.

→ Abbiamo diviso lo spazio logico in blocchi di dimensione fissa, e ogni blocco dello spazio di memoria è mappato in una determinata posizione all'interno della cache tramite i bit di indice. I bit di indice possono essere pari a  $\log_2 n$  blocchi di cache o possono essere pari a zero se un blocco di memoria può andare in un qualunque blocco di cache.

# Blocks and Addressing the Cache

- La memoria è logicamente suddivisa in blocchi di dimensione fissa
- Ogni blocco è mappato in una posizione nella cache tramite i bit di indice/index bits dell'indirizzo
  - Usati per indicizzare all'interno dei campi di tag e data



- Accesso alla Cache :
  - 1) Accedo ai campi di tag e dati con bit di indice dell'indirizzo

→ dice se quel blocco di cache contiene o meno un dato. Se non lo contiene sappiamo già che abbiamo una miss.
  - 2) controllo il bit valido (valid bit) all'interno del campo tag
  - 3) confronto i bit di tag dell'indirizzo con i bit di tag memorizzati nel campo tag del blocco.

A seguito del confronto si ha una HIT o una MISS.  
In caso di HIT, sappiamo a quale blocco fare accesso e useremo i rimanenti bit di indice per effettuare l'accesso.
- Se un blocco si trova nella cache (cache hit), il tag memorizzato deve essere valido e corrispondere al tag del blocco per il quale ho fatto l'accesso.

Data la dimensione di un blocco: 64 byte, faccio log<sub>2</sub> (dimensione del blocco) e ottengo il n° di bit necessari per identificare un byte all'interno del blocco stesso. I rimanenti bit + tag identificano un indirizzo del blocco di memoria.

# Direct-Mapped Cache: Placement and Access

Block: 00000
Block: 00001
Block: 00010
Block: 00011
Block: 00100
Block: 00101
Block: 00110
Block: 00111
Block: 01000
Block: 01001
Block: 01010
Block: 01011
Block: 01100
Block: 01101
Block: 01110
Block: 01111
Block: 10000
Block: 10001
Block: 10010
Block: 10011
Block: 10100
Block: 10101
Block: 10110
Block: 10111
Block: 11000
Block: 11001
Block: 11010
Block: 11011
Block: 11100
Block: 11101
Block: 11110
Block: 11111

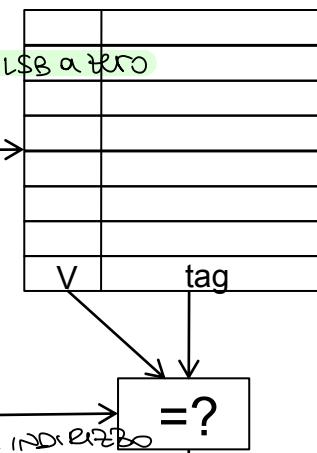
- Assumiamo byte-addressable memory:  
256 bytes, 8-byte blocks  $\rightarrow$  32 blocks
- Assumiamo cache: 64 bytes, 8 blocks
  - Direct-mapped: A block can go to only one location

tag index (Set id) byte in block

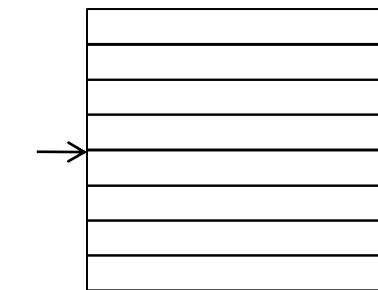


Tutti i blocchi di Address  
che hanno i 3 LSB a zero  
memoria multipli di 8 potranno  
essere contenuti solo nel  
blocco zero delle cache.

Tag store



Data store



byte in block

MUX

Data

13 BIT MENO SIGNIFICATIVI DELL'INDIRIZZO  
DI CIASCUN BLOCCO IDENTIFICANO QUALE  
POSSIBILE SET PUÒ ESSERE CONTENUTO IN  
BLOCCO DI MEMORIA.

- Addresses with same index contend for the same location
  - Cause conflict misses

## CACHE DIRECTLY MAPPED

Con i bit di set io accedo a un determinato set della cache, poi all'interno del set verifico se il tag contenuto nel tag store è corrispondente al tag a cui sto cercando di fare accesso. Se corrisponde è una HIT, senno no.

Validity bit: identifica se quel tag all'interno di quel set della cache è o meno valido, cioè se al momento quel set è occupato o meno da un tag valido.

Se è occupato e non corrispondente vuol dire che ho uno MISS e dovrò portare il blocco di memoria corrispondente dalla memoria a quella linea di cache e rimuovere il dato precedente e aggiornare il tag.

• Ho bisogno di un solo comparatore per tutta la cache. Ho un solo possibile tag per ciascun indice, quindi ogni volta dovrò prendere i bit di tag dell'indirizzo, confrontarli con i bit di tag contenuti esattamente in quel set.

L'esito della comparazione può essere zero o fatto e viene multiplexato.

# Direct-Mapped Caches

---

- Direct-mapped cache: Due blocchi in memoria mappati allo stesso indice della cache (con gli stessi bit di indice) non possono essere presenti contemporaneamente nella cache
  - One index → one entry
- 
- Può portare a un hit-rate dello 0% se si accede in modo alternato a più di un blocco mappato allo stesso indice
    - Assume addresses A and B have the same index bits but different tag bits
    - A, B, A, B, A, B, A, B, ... → conflict in the cache index
    - All accesses are conflict misses

# Set Associativity

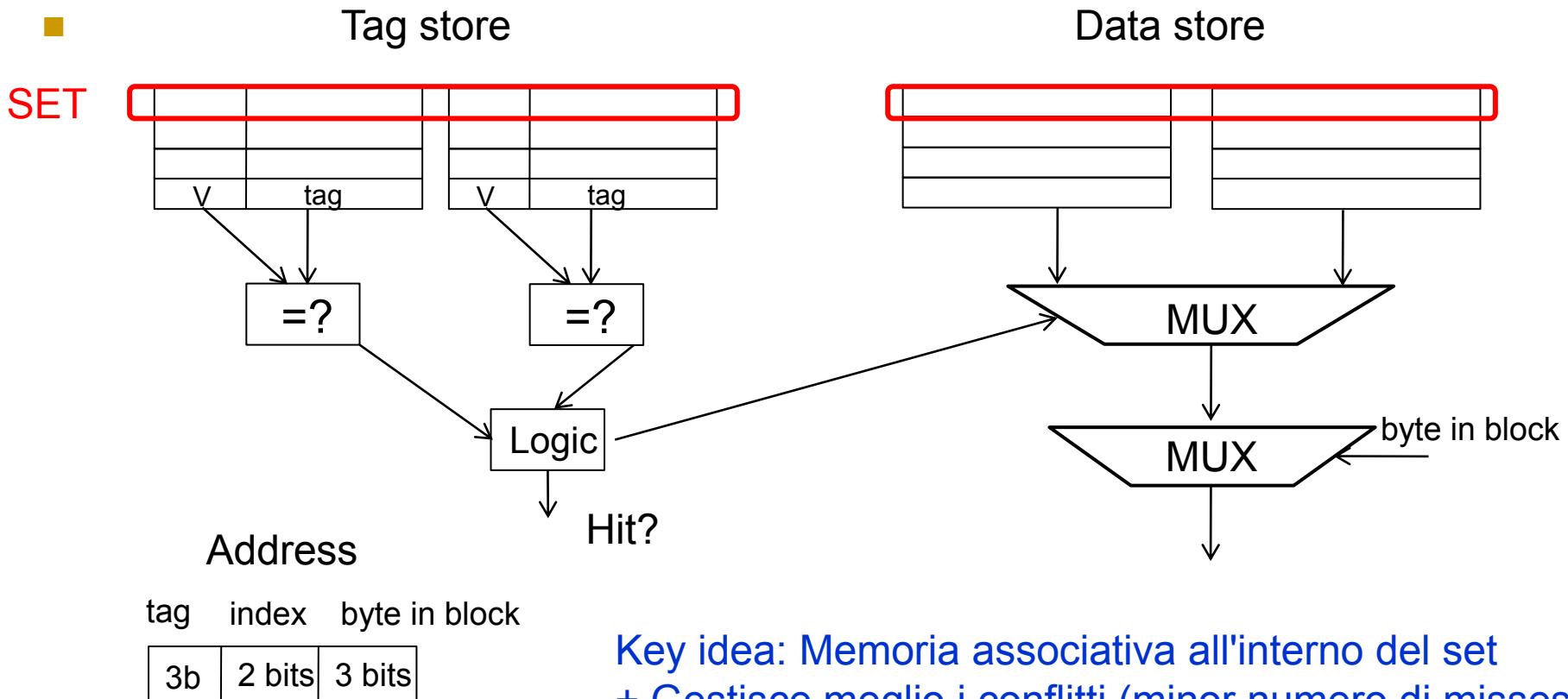
Nell'esempio di prima:

- Indirizzi 0 e 8 sempre in conflitto in direct mapped cache
- Invece di avere una colonna di 8, hanno 2 colonne di 4 blocchi
- 

prima : SET = siugela noga del data store e del tag store.  
(identificare due bit di indice).

Ora: anziché vedere gli 8 blocchi che costituiscono una cache strutturati in una colonna, ora li vedo in due colonne pari alla metà.

dato un indice, i tag può trovarsi in due possibili blocchi di cache; per cui ho 2 comparatori e posso fare hit o in una via o nell'altra. ho anche 2 MUX.



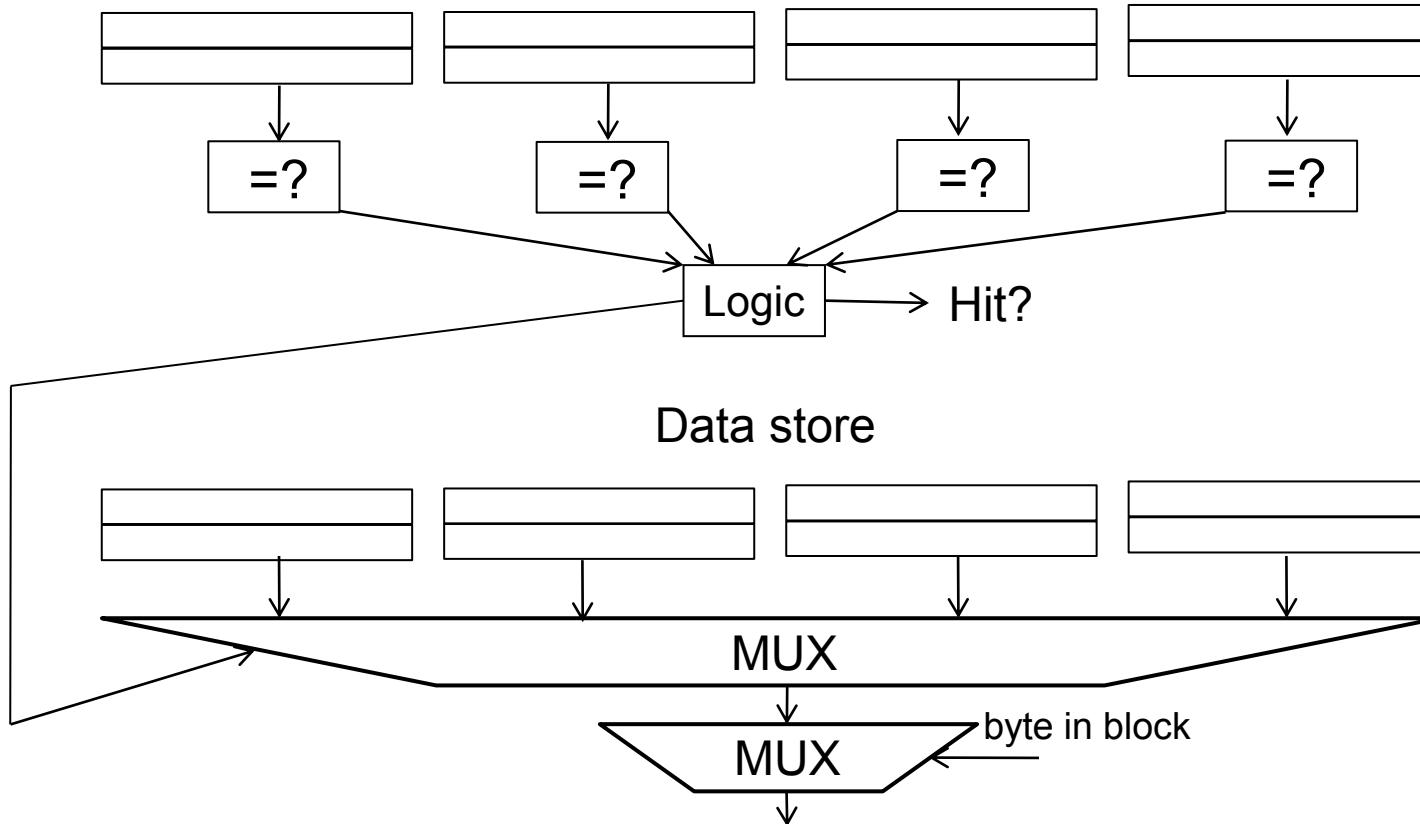
Key idea: Memoria associativa all'interno del set  
+ Gestisce meglio i conflitti (minor numero di misses)  
-- Maggiore complessità, più lenta, tag store più largo

# Higher Associativity

LE 8 LINEE DI CACHE LE DIVIDO IN 4 VIE

Tag store

## ■ 4-way



+ Minor probabilità di miss dovuta a conflitti

-- Maggior numero di comparatori per i tag, data mux più  
largo; tags con dimensione maggiore

# Full Associativity

NON HO BIT DI INDICE  
HO UN UNICO SET COMPOSTO DA 8 BLOCCI DI CACHE  
E A CIASCUN BLOCCO E' ASSOCIAUTO UN COMPARATORE

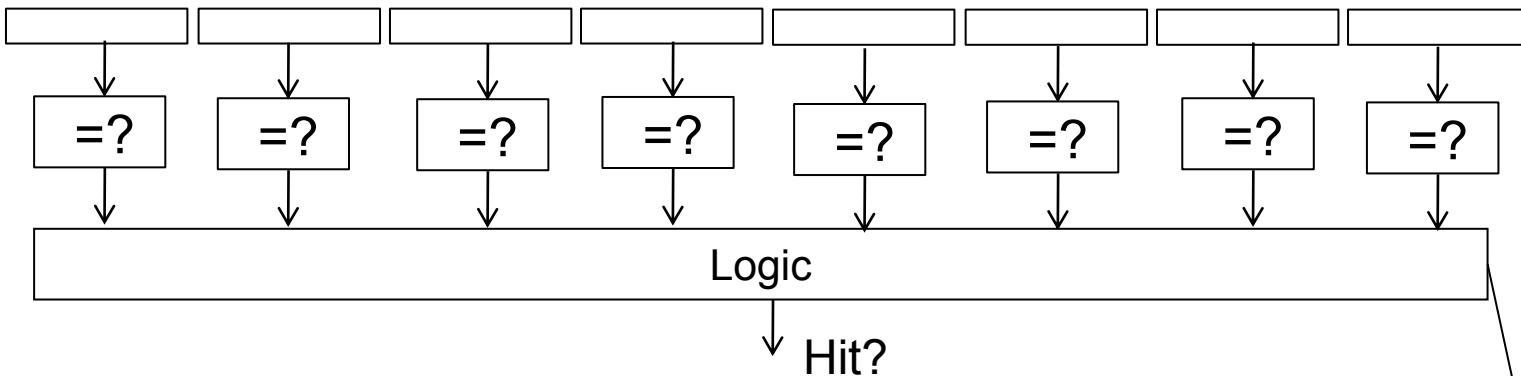
AVRÒ UN MUX A 8 INGRESSI E 1 USCITA

(Nel caso di una cache a 2 vie avrei un MUX 2 : 1)  
higher: 4:1 massima 8:1.

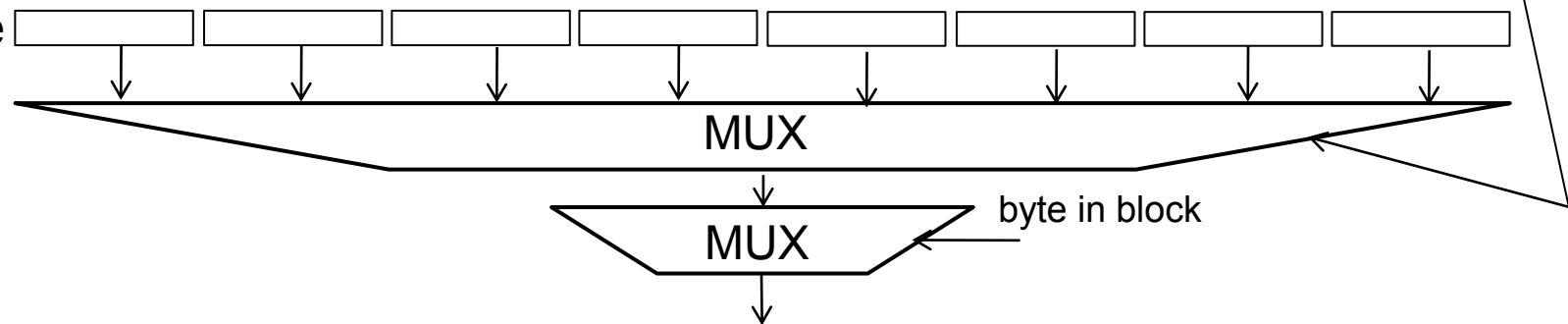
## Fully associative cache

- Un blocco può essere posizionato in **qualsiasi** posizione della cache

Tag store



Data store



# Associativity (and Tradeoffs)

CACHE

→ TAG STORE: Ci dice se quel dato è presente o no nel data store.

Lo DATA STORE: Contiene i bit di informazione, copie della memoria.

- Degree of associativity: How many blocks can map to the same index (or set)?

- Higher associativity

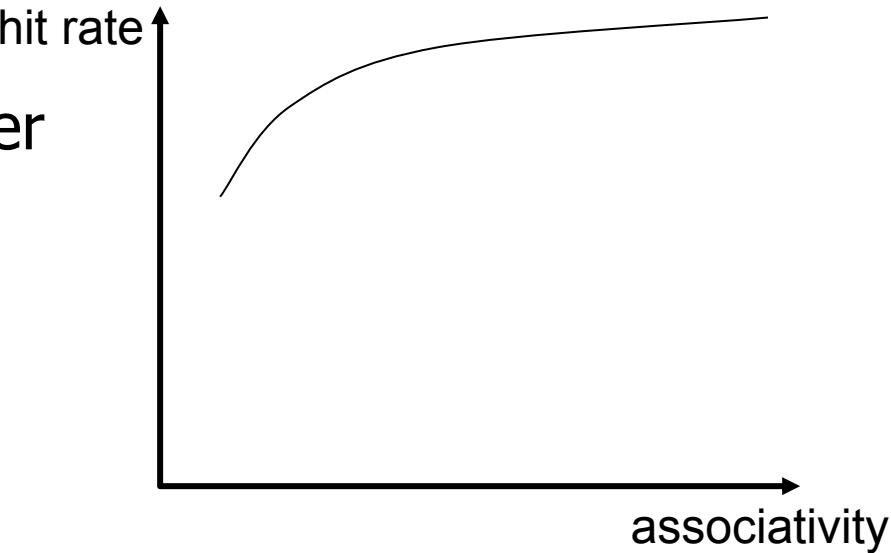
++ Higher hit rate

Tag e blocco di memoria  
↓  
Sezione + significativa  
dell'indirizzo      ↓  
porzione di indirizzi  
contigui.

- Slower cache access time (hit latency and data access latency)
- More expensive hardware (more comparators)

- Diminishing returns from higher associativity

località spaziale  
località temporale



## Tag e blocco di memoria

↓                      ↓  
Sezione + significativa      porzione di indirizzi  
dell'indirizzo                contigui.

a ciascun blocco è associato un tag, con questo tag  
controlliamo se quel blocco di memoria è presente o meno  
nel data store.

SET: una riga nel tag store.

NOTA: "blocco" fa riferimento a due cose:

- unità minime con cui si partitiona logicamente la memoria
- unità minime con cui si partitiona la capacità del data store delle cache.

SET = raggruppamento logico dei blocchi all'interno del data store.

Se ho un set composto da più blocchi, potrò trovare in detto blocco di memoria in diversi blocchi dello stesso set delle cache.

Mandiamo un indirizzo alle memorie con associato un set id. Se il set e' composto da un solo blocco si parla di cache directly mapped, se si hanno + blocchi ma Cmp. meno della totalità di blocchi che compongono il data store allora avremo una cache set associative. Se abbiamo un unico set composto da tutti i blocchi che compongono il data store si tratta di una cache fully associative.

Il n° di blocchi all'interno di ciascun set li chiamiamo VIE.

Ese. cache set associative a 2 vie  $\rightarrow$  cache con due blocchi per set.

# Issues in Set-Associative Caches

---

- Pensate a ogni blocco in un set con una «priorità»
  - Indica l'importanza di mantenere il blocco nella cache
- Key issue: Come determinare/regolare le priorità dei blocchi?
- Ci sono tre decisioni chiave in un set:
  - Insertion, promotion, eviction (replacement)
- **Insertion: What happens to priorities on a cache fill?**
  - Where to insert the incoming block, whether or not to insert the block
- **Promotion: What happens to priorities on a cache hit?**
  - Whether and how to change block priority
- **Eviction/replacement: What happens to priorities on a cache miss?**
  - Which block to evict and how to adjust priorities

# Eviction/Replacement Policy

---

- Which block in the set to replace on a cache miss?
  - Any invalid block first
  - If all are valid, consult the replacement policy
    - Random
    - FIFO
    - Least recently used (how to implement?)
    - Not most recently used
    - Least frequently used?
    - Least costly to re-fetch?
      - Why would memory accesses have different cost?
    - Hybrid replacement policies
    - Optimal replacement policy?

# Implementing LRU

---

- Idea: Evict the least recently accessed block
- Problem: Need to keep track of access ordering of blocks
- Question: 2-way set associative cache:
  - What do you need to implement LRU perfectly?
- Question: 4-way set associative cache:
  - What do you need to implement LRU perfectly?
  - How many different orderings possible for the 4 blocks in the set?
  - How many bits needed to encode the LRU order of a block?
  - What is the logic needed to determine the LRU victim?

# Approximations of LRU

---

*Last recently used*

- Most modern processors do not implement “true LRU” (also called “perfect LRU”) in highly-associative caches
- Why?
  - True LRU is complex
  - LRU is an approximation to predict locality anyway (i.e., not the best possible cache management policy)
- Examples:
  - Not MRU (not most recently used)
  - **Victim-NextVictim Replacement**: Only keep track of the victim and the next victim

# Cache Replacement Policy: LRU or Random

---

- LRU vs. Random: Which one is better?
  - Example: 4-way cache, cyclic references to A, B, C, D, E
    - 0% hit rate with LRU policy
- Set thrashing: When the “program working set” in a set is larger than set associativity
  - Random replacement policy is better when thrashing occurs
- In practice:
  - Depends on workload
  - Average hit rate of LRU and Random are similar

# What's In A Tag Store Entry?

- Valid bit : dice se il tag è valido
  - Tag
  - Replacement policy bits
- Dirty bit?

- Write back vs. write through caches

scrivere nelle cache vuol

dire scrivere nella memoria?

Non per forza, se scrivo in cache e basta,  
non sto scrivendo in memoria, quindi non  
sto consumando bande dalla cache  
alla memoria.

Se volessi scrivere in memoria bisogna battere.

Cosa c'è dentro la cache

Se ho solo letto dalla memoria?

di mem.

Dirty bit ci dice se il blocco è stato scritto  
e in questo caso vuol dire che la  
cache ha una versione aggiornata

# Handling Writes (I)

gestione delle scritture  
nelle cache.

- Quando scrivere i dati modificati nel livello successivo della cache?
  - ▶ I livelli successivi della memoria vengono modificati subito, direttamente alla scrittura → + semplice.
  - Write through: Nel momento in cui avviene la scrittura
  - Write back: Quando il blocco viene rimosso
- Write-back
  - ▶ Scrivendo nello cache e aggiorna le memorie solo quando rimuovo il blocco dalle cache. → Risparmio energetico e di banda.
  - + Può combinare più scritture allo stesso blocco prima della eviction
    - Potenziale riduzione della banda tra i livelli della cache + risparmio di energia
  - Richiede un bit nel tag store per indicare se il blocco è sporco/modificato ("dirty/modified")
- Write-through
  - + Semplice
  - + Tutti i livelli sono sempre aggiornati. Consistency: Coerenza delle cache più semplice, non c'è bisogno di controllare se il dato è presente in un tag store di una cache più vicina al processore.
  - Maggior richiesta di banda; non combina le writes

# Handling Writes (II)

- Allociamo un blocco di cache in una write miss (miss di scrittura)?
  - Alloca in caso di write miss (Write Allocate): Yes
  - Non-alloca in caso di write miss (Write Around): No
- Write Allocate - Allocate on write miss
  - + Può combinare le scritture invece di scriverle singolarmente al livello successivo
  - + Più semplice perché le write miss possono essere trattati allo stesso modo delle read miss
  - Richiede il trasferimento dell'intero blocco della cache

tratta le miss in scrittura come le miss in lettura, quindi alloca un blocco di memoria in cache per miss di scrittura.
- Write Around - No-allocate
  - + Risparmia spazio nella cache se la località delle scritture è bassa (potenzialmente miglior cache hit rate)

WRITE DIRETTAMENTE IN MEMORIA.  
conviene che ho un basso niviso del dato che ho scritto.

# Instruction vs. Data Caches

---

- Separate or Unified?
  - Pros & Cons of Unified:
    - + Condivisione dinamica dello spazio cache: no overprovisioning che potrebbe accadere con il partizionamento statico (i.e., separate I & D caches)
    - Le istruzioni e i dati possono invalidarsi (thrash) a vicenda (i.e., nessuna garazia di spazio)
    - I & D accedute in diversi stadi della pipeline.  
Dove posizionare la cache unificata per minimizzare la latenza di accesso?
  - Le cache di primo livello sono quasi sempre divise
    - ▣ Principalmente per l'ultimo motivo di cui sopra
  - Le cache di livello superiore sono quasi sempre unificate
-

# Multi-level Caching in a Pipelined Design

---

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity; latency is critical
  - Tag store and data store accessed in parallel
- Second-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency not as important
  - Tag store and data store accessed serially
- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter (filters some temporal and spatial locality)
    - Management policies are therefore different

# Cache Performance

# Cache Parameters vs. Miss/Hit Rate

---

- Cache size
- Block size
- Associativity
- Replacement policy
- Insertion/Placement policy

# Cache Examples:

.

# Cache Terminology

- **Capacità ( $C$ ):**
  - Numero di bytes che la cache può immagazzinare

→ fa riferimento alla dimensione del data store della cache.
- **Dimensione Blocco/Linea ( $b$ ):** → fa riferimento a quanti bytes vengono copiati dalle memorie rispetto alla cache in seguito a una miss.
  - bytes portati nella cache in una sola volta
- **Numero di blocchi ( $B = C/b$ ):**
  - numero di blocchi nella cache:  $B = C/b$
- **Grado di associatività / # di Vie ( $M$ ):**
  - numero di blocchi in un set → se voglio vedere quanti set è composta la cache, prendere il numero di blocchi di cui è composta la cache e lo divido per il numero di vie.
- **Numero di sets ( $S = B/M$ ):**
  - ogni indirizzo di memoria viene mappato esattamente a un set nella cache

*n° di bit che servono per identificare un singolo set è  $\log_2(\text{n° di set})$ .*

*es. se ho 128 set, ho bisogno di  $\log_2(128) = 7$  bit.*

# How is data found?

- Cache organizzata in  $S$  sets
- Ogni indirizzo si mappa esattamente ad un solo set  
*generalmente un indirizzo in memoria si mappa su un set, poi all'interno del set quel determinato blocco di memoria può essere contenuto in uno o più posizioni all'interno delle cache.*
- Cache categorizzate per numero di blocchi in un set:
  - Direct mapped: 1 blocco per set
  - N-way set associative:  $N$  blocchi per set
  - Fully associative: Tutti i blocchi della cache sono in un solo set
- Esaminiamo ciascuna organizzazione per una cache con:
  - Capacity ( $C = 8$  words)  $= 8 \cdot 4 \text{ byte} = 32 \text{ byte}$
  - Block size ( $b = 1$  word)  $\leftarrow 4 \text{ byte}$
  - So, number of blocks ( $B = 8$ )  $= \frac{2^5 \text{ byte}}{2^2 \text{ byte}} = 2^3 \text{ blocchi} = 8 \text{ blocchi}$

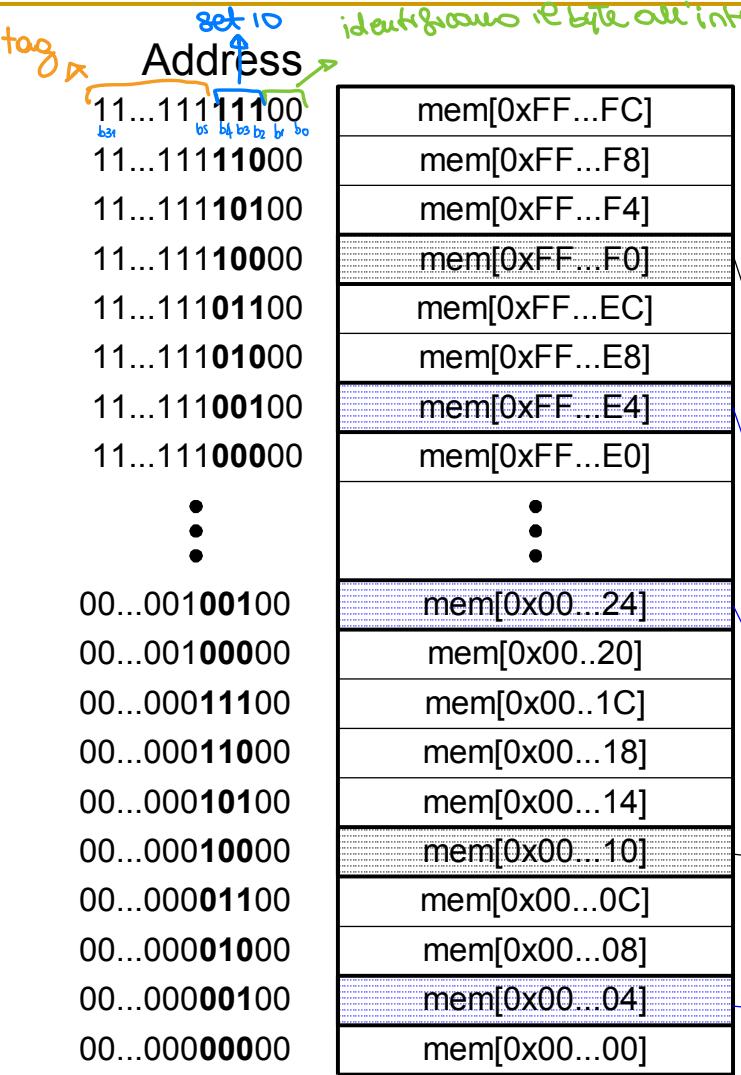
# Direct Mapped Cache

tutti e 8 i blocchi che ho a disposizione nelle cache appartengono a 8 set

1 set x blocco.

$\log_2 8$  bit x codificare i set  
3 bit

Mi servono 2 bit x identificare ciascun byte all'interno del blocco.  
Mi servono 3 bit per identificare il set-10, poi mi servono tutti gli altri  
bit: 32-3-2 per identificare il campo di tag.



$2^{30}$  Word Main Memory

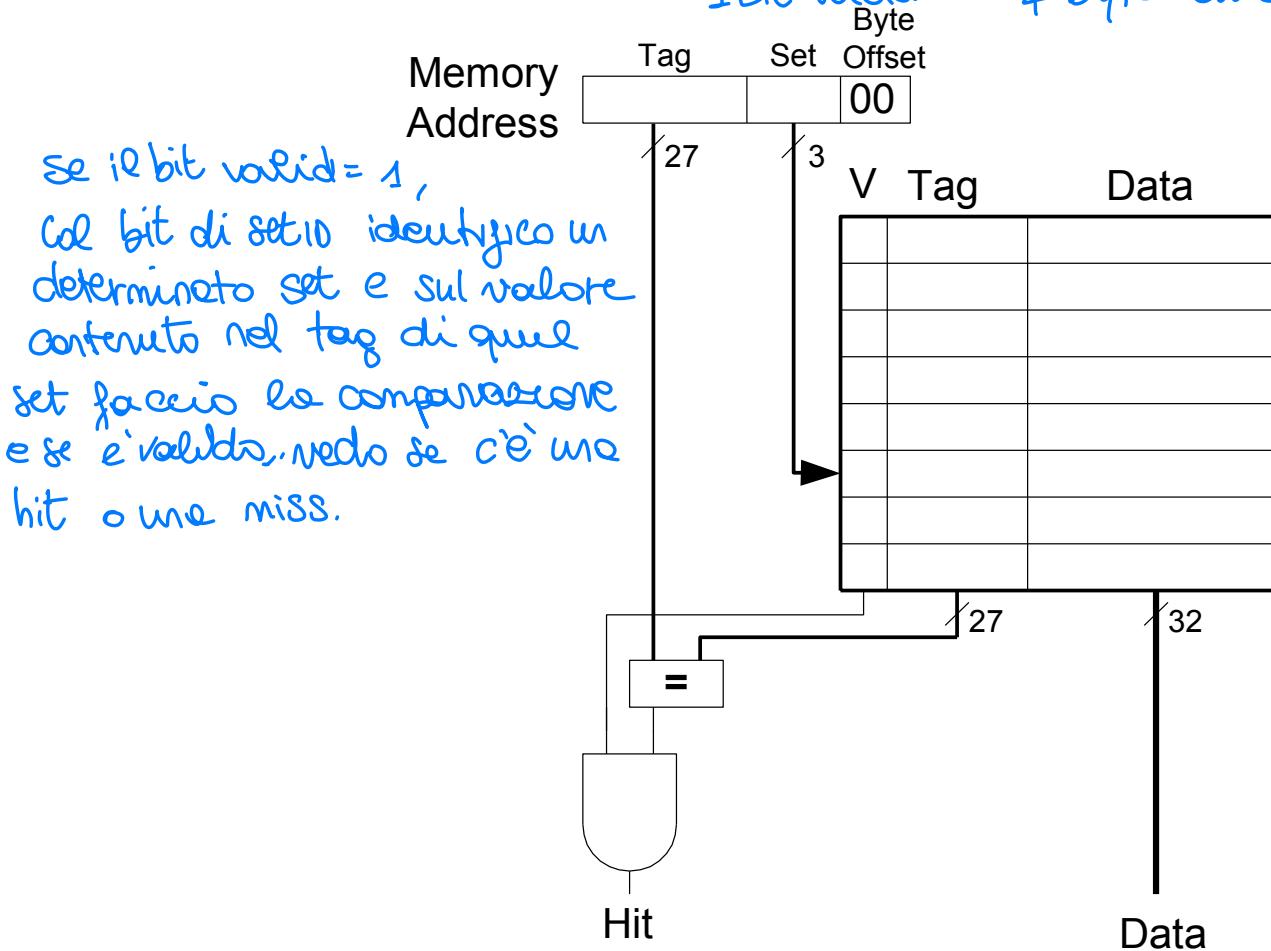
$2^3$  Word Cache

Set Number
7 (111)
6 (110)
5 (101)
4 (100)
3 (011)
2 (010)
1 (001)
0 (000)

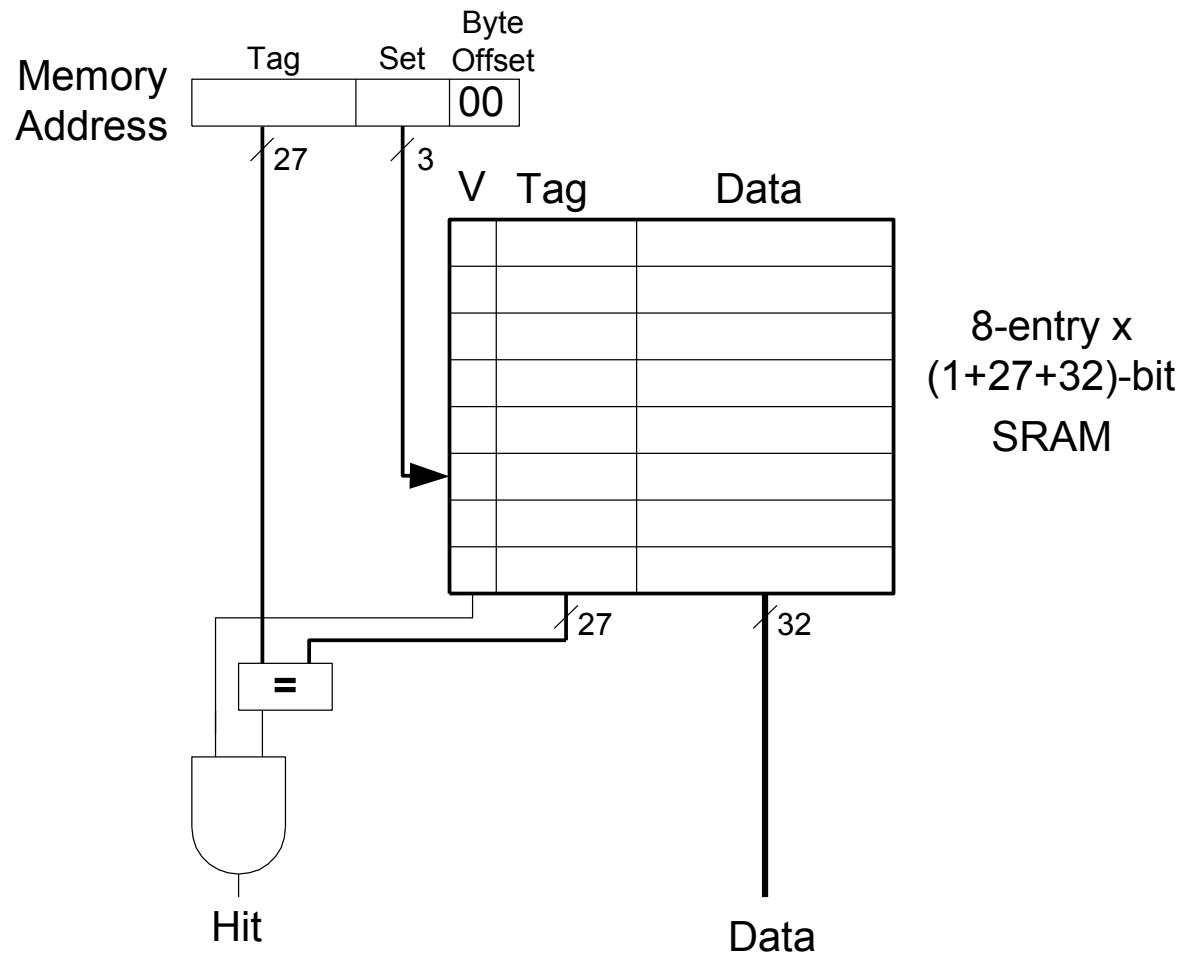
0, 1, 2, 3 accedono allo stesso blocco di memoria  
0, 4 accedono a due blocchi diversi.

# Direct Mapped Cache Hardware

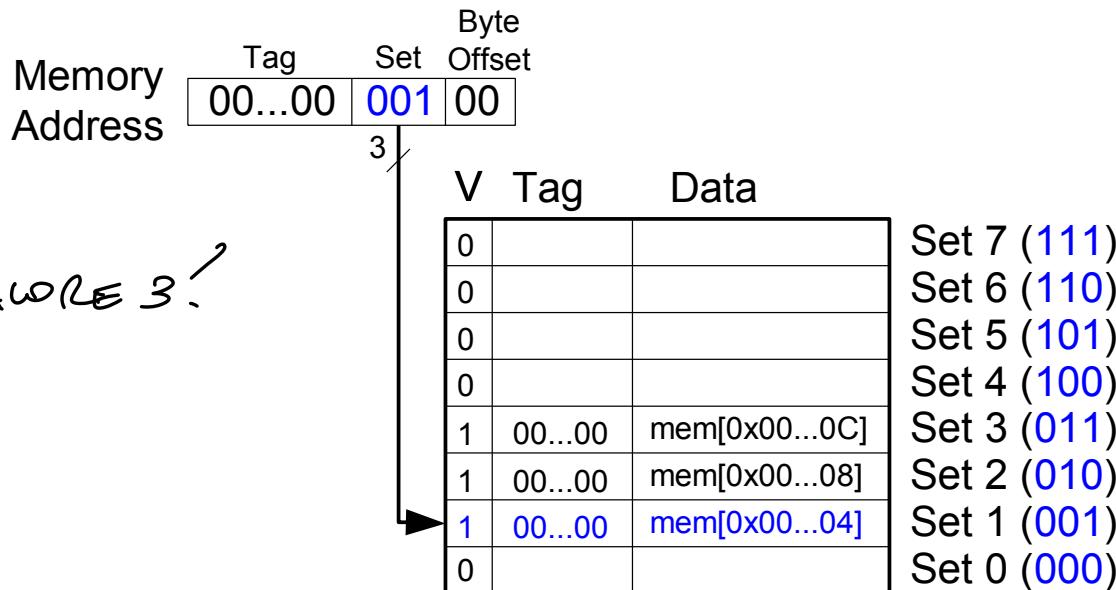
composizione della cache : 8 set che identifico con 3 bit di set id, 27 bit di tag, 1 bit validità e byte di camp offset



# Direct Mapped Cache Hardware



# Direct Mapped Cache Performance



```
# RV assembly code esegue 4 accessi in memoria con un ciclo for con i=4, su
loop:    addi   x1, x0, 4      3 array che
          beq    x0, x1, done      contengono dei
          addi   x1, x1, -1      caratteri.
          lb     x2, 0x4(x1)    si trova a 0x4
          lb     x3, 0xC(x1)    l'altro 0xC
          lb     x4, 0x8(x1)    l'altro 0x8.
          beq    x0, x0, loop
```

done:

Miss Rate =

Netto in X1 4 e poi a ogni ciclo decremento  
di 1 l'elemento dell'array a cui si sta  
accedendo.

l'indirizzo  $x_1$  contiene il valore 3, quindi con la prima riga si accede all'indirizzo 7.  $+ 0x7$

Quando si emette un indirizzo, la 1<sup>a</sup> cosa è confrontare i 32-5 bit più significativi con il valore del tag store associato al set indirizzato dal set ID.

1 singolo comparatore per tutti i set. (ma 1 per ogni set), quindi si fa le comparazioni con il tag nel tag store riferito a quel set indirizzato dai bit di set ID dell'indirizzo appena emesso.

# Direct Mapped Cache Performance



directly mapped : 1 blocco x set  
 $n^{\circ}$  set =  $n^{\circ}$  blocchi

cache set associative a 2 vie :  $n^{\circ}$  set =  $\frac{n^{\circ} \text{ blocchi}}{2}$

$\text{set} = 2$

$0x8 \rightarrow 0000\ 1000$

offset : quale dei 4 byte

Tag : identifica univocamente un blocco di memoria, così da verificare la presenza o meno di quel blocco di memoria nel dato store.

V: validity bit = ci dice se quel tag e data store contengono un blocco di memoria valido.

V Tag Data

V	Tag	Data
0		
0		
0		
0		
1	00...00	mem[0x00...0C]
1	00...00	mem[0x00...08]
1	00...00	mem[0x00...04]
0		

Set 7 (111)      dei 32 Gt dell'indirizzo, i due meno significativi identificano tre byte all'interno di un blocco.  
 Set 6 (110)  
 Set 5 (101)  
 Set 4 (100)  
 Set 3 (011)  
 Set 2 (010)  
 Set 1 (001)  
 Set 0 (000)

$\log_2(n^{\circ} \text{ di byte del blocco})$

$\log_2(n^{\circ} \text{ di set nella cache})$

↓

$n^{\circ}$  bit necessari per identificare un set nella cache (setid)

Ese: 8 set  $\rightarrow \log_2 8 = 3$  bit

```
# RV assembly code
      addi  x1, x0, 4
loop:   beq   x0, x1, done
      addi  x1, x1, -1
      lb    x2, 0x4(x1)
      lb    x3, 0xC(x1)
      lb    x4, 0x8(x1)
      beq   x0, x0, loop
```

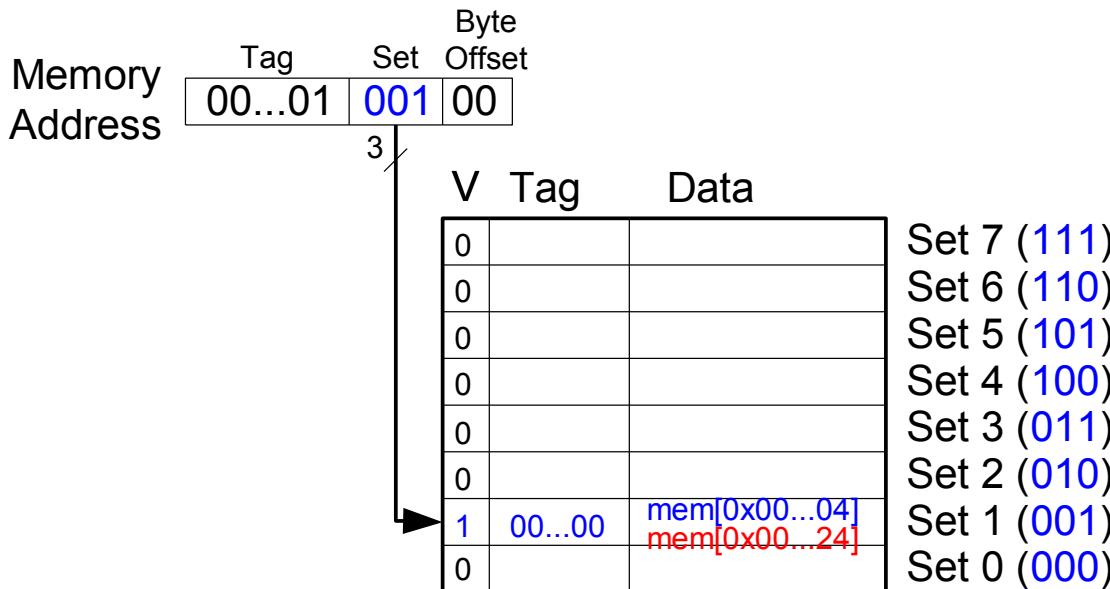
done:

**Miss Rate** =  $3/12$   
 i primi 3 accessi sono delle miss perché devo portare quei valori nella cache.  
**25%**

$$\text{n}^{\circ} \text{ di SET} = \frac{\text{n}^{\circ} \text{ blocchi del dato store}}{\text{n}^{\circ} \text{ vie delle associazioni}}$$

## Temporal Locality Compulsory Misses

# Direct Mapped Cache: Conflict



```
# RV assembly code
      addi x1, x0, 4
loop:   beq  x0, x1, done
      addi x1, x1, -1
      lb    x2, 0x4(x1)
      lb    x3, 0x24(x1)
      beq  x0, x0, loop
done:
```

*Miss Rate*

= 100 % perché sovrascrivo  
no e vicende.

0x04 ...000 001000 ]  
0x24 ...00 001000 ]

questi hanno stesso set ID, ma  
hanno tag diversi.  
Sono due blocchi di memoria  
differenti

Risolvendo con una cache associativa a 2 v. de. ( $n=2$ )  
→ gli 8 blocchi di memoria che compongono il dato  
store, ora li spolano su 4 set anziché 8.  
ogni SET contiene 2 blocchi di memoria. → Metà di n° di set.  
sono bisognati 2 bit di  
set ID (uno un meno) 36  
→ un bit in più di tag.

alla 1<sup>a</sup> iterazione  $x_1 = 3$

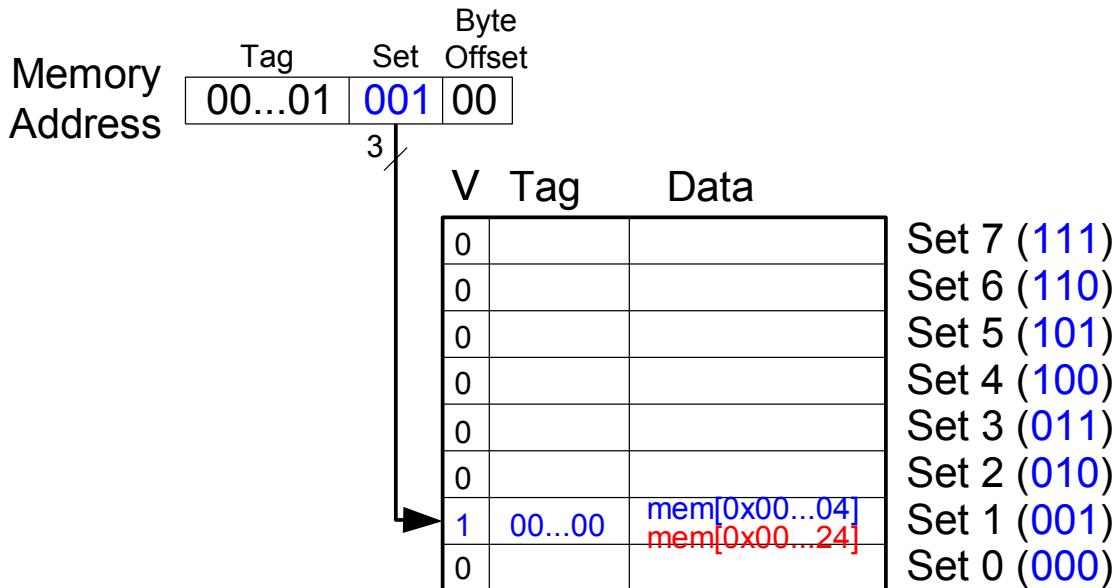
la prima lb viene fatta all'indirizzo 0x7 (4+3)

0x7 ha gli ultimi due bit pari a 1, allora viene generata una miss e l'indirizzo viene copiato nel set

l'istruzione fa una lb a 0x27  $\rightarrow$

ha l'offset set 10 001, e le come tag 0001.

# Direct Mapped Cache: Conflict

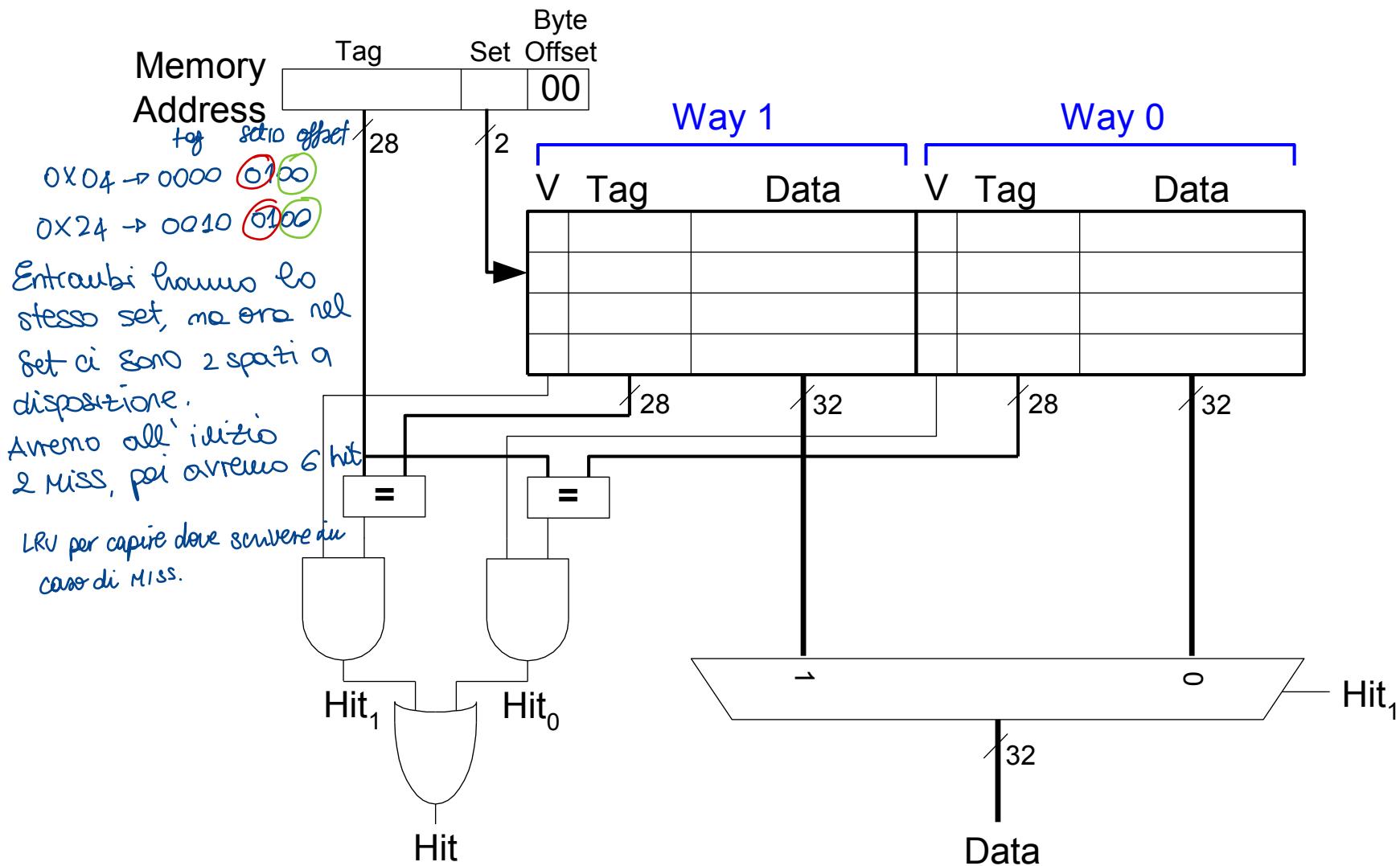


```
# RV assembly code
      addi  x1, x0, 4
loop:  beq   x0, x1, done
      addi  x1, x1, -1
      lb    x2, 0x4(x1)
      lb    x3, 0x24(x1)
      beq   x0, x0, loop
done:
```

$$\text{Miss Rate} = 8/8 \\ = 100\%$$

Conflict Misses

# N-Way Set Associative Cache



# N-way Set Associative Performance

```
# RV assembly code
    addi  x1, x0, 4
loop:   beq   x0, x1, done
        addi  x1, x1, -1
        lb    x2, 0x4(x1)
        lb    x3, 0x24(x1)
        beq   x0, x0, loop
done:
```

*Miss Rate =*

Way 1			Way 0			Set 3 Set 2 Set 1 Set 0
V	Tag	Data	V	Tag	Data	
0			0			
0			0			
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]	
0			0			

# N-way Set Associative Performance

```
# RV assembly code
    addi  x1, x0, 4
loop:   beq   x0, x1, done
        addi  x1, x1, -1
        lb    x2, 0x4(x1)
        lb    x3, 0x24(x1)
        beq   x0, x0, loop
done:
```

$$\begin{aligned} \text{Miss Rate} &= 2/8 \\ &= 25\% \end{aligned}$$

Associativity reduces conflict misses

Way 1			Way 0		
V	Tag	Data	V	Tag	Data
0			0		
0			0		
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]
0			0		

Set 3  
Set 2  
Set 1  
Set 0

# Fully Associative Cache

---

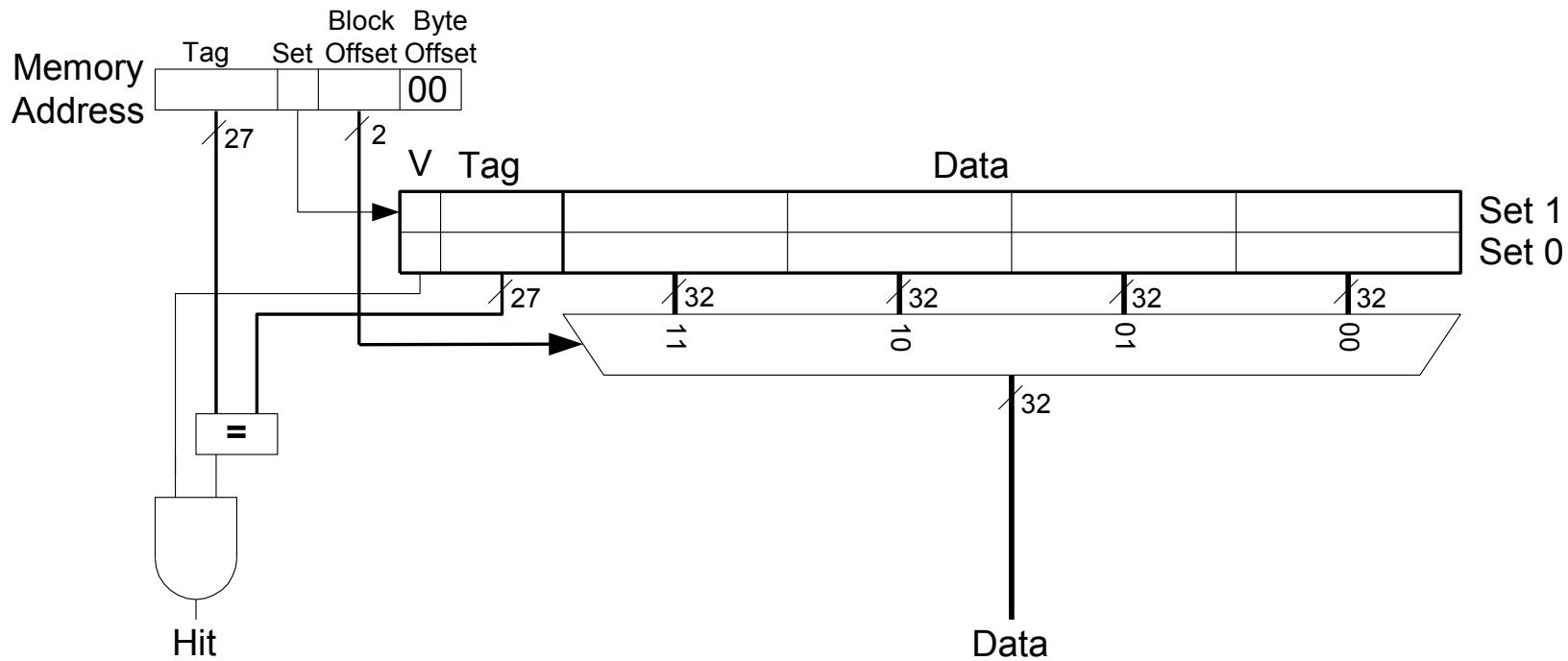
- No conflict misses
- Expensive to build

Serve un comparatore per ciascun blocco dello cache.

V	Tag	Data															

# Spatial Locality?

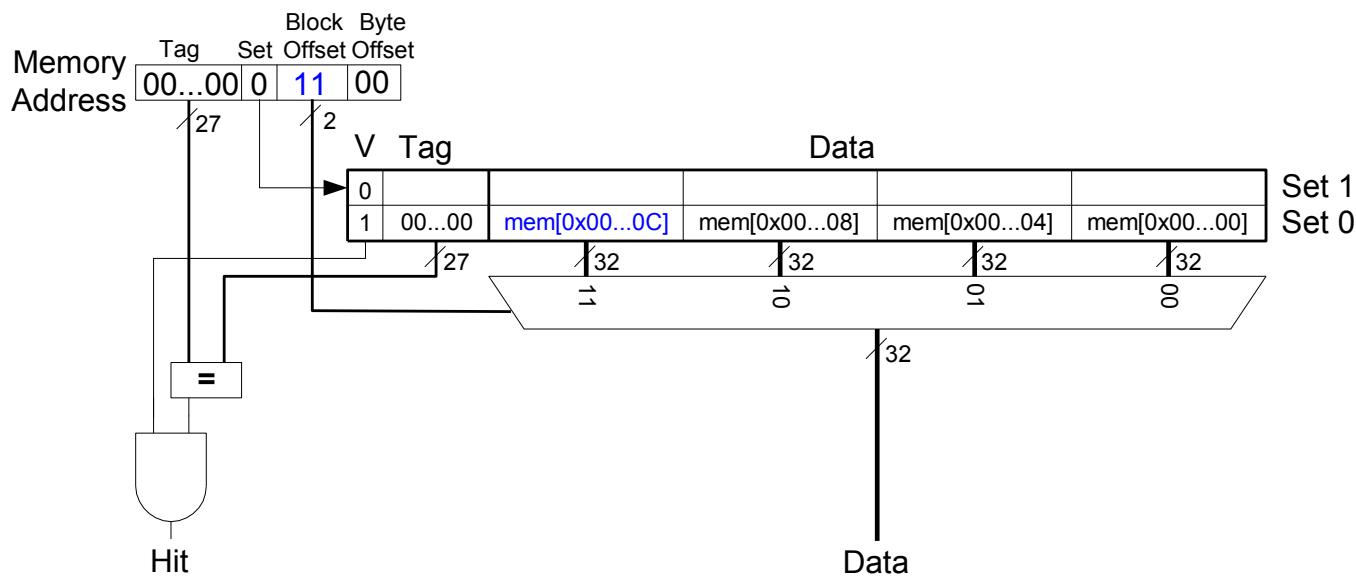
- Increase block size:
  - Block size,  $b = 4$  words
  - $C = 8$  words
  - Direct mapped (1 block per set)
  - Number of blocks,  $B = C/b = 8/4 = 2$



# Direct Mapped Cache Performance

```
# RV assembly code
    addi  x1, x0, 4
loop:   beq   x0, x1, done
        addi  x1, x1, -1
        lb    x2, 0x4(x1)
        lb    x3, 0xC(x1)
        lb    x4, 0x8(x1)
        beq   x0, x0, loop
done:
```

*Miss Rate =*



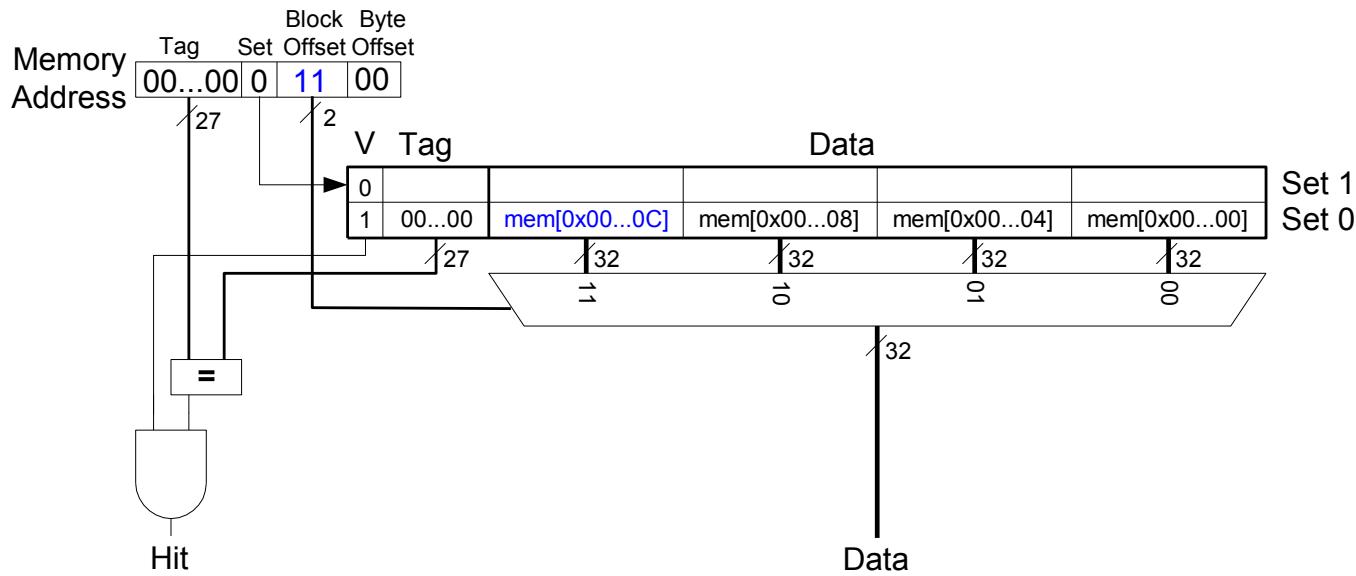
# Direct Mapped Cache Performance

```
# RV assembly code
    addi  x1, x0, 4
loop:   beq   x0, x1, done
        addi  x1, x1, -1
        lb    x2, 0x4(x1)
        lb    x3, 0xC(x1)
        lb    x4, 0x8(x1)
        beq   x0, x0, loop
done:
```

$$\text{Miss Rate} = 1/12$$

$$= 8.33\%$$

Larger blocks reduce compulsory misses through spatial locality



# Cache Organization Recap

## ■ Main Parameters

- Capacity:  $C$
- Block size:  $b$
- Number of blocks in cache:  $B = C/b$
- Number of blocks in a set:  $N$
- Number of Sets:  $S = B/N$

Organization	Number of Ways (N)	Number of Sets (S = B/N)
Direct Mapped	1	B
N-Way Set Associative	$1 < N < B$	$B / N$
Fully Associative	B	1

# Capacity Misses

---

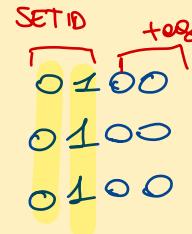
- La cache è troppo piccola per contenere tutti i dati di interesse contemporaneamente
- Se la cache è piena e il programma accede al dato X non presente in cache, la cache deve rimuovere (evict) il dato Y per fare spazio a X
- **Capacity miss** si verifica se il programma tenta di accedere a Y di nuovo
  - X sarà collocato in un set particolare in base al suo indirizzo
- In cache a **mapping diretto**, c'è solo un posto dove mettere X
- In cache **associativa**, ci sono diverse vie in cui X potrebbe essere scritto nel set.
- Come scegliere Y per ridurre al minimo la possibilità di averne bisogno di nuovo?
  - Least recently used (LRU) replacement: il blocco utilizzato meno di recente in un set viene rimosso quando la cache è piena.

# LRU Replacement

per ciascun SET della cache abbiamo un bit di RW (1 perché ho 2 vie)

# R5 assembly

**lb** x2, 0x04(x1) → 0000  
**lb** x3, 0x24(x1) → 0010  
**lb** x4, 0x54(x1) → 0101



Set : 1

Il bit RW ci dice in caso di miss in quale via andare inserire il blocco di memoria corrispondente a quell'indirizzo.

VIA 1

VIA 0

Primo del 1° accesso,  
il bit RW = 0.

Leggiamo la 1° var. (a)

scopriamo del tag associato  
al nostro indirizzo e poi  
tutti i tag store associati  
a quel set.

Sembra c'è corrispondenza,  
si ha una miss. →

Si prende dalla memoria  
quel blocco associato (b)  
a quell'indirizzo e  
lo posizioniamo nella via punto-  
ta dal bit RW.

In questo caso è la via zero.

V	U	Tag	Data	V	Tag	Data
0	0			0		

Set Number

3 (11)

2 (10)

1 (01)

0 (00)

V	U	Tag	Data	V	Tag	Data

Set Number

3 (11)

2 (10)

1 (01)

0 (00)

# LRU Replacement

```
# R5 assembly
```

```
lb x2, 0x04(x1)  
lb x3, 0x24(x1)  
lb x4, 0x54(x1)
```

Way 1			Way 0			
V	U	Tag	Data	V	Tag	Data
0	0			0		
0	0			0		
1	0	00...010	mem[0x00...24]	1	00...000	mem[0x00...04]
0	0			0		

Set 3 (11)  
Set 2 (10)  
Set 1 (01)  
Set 0 (00)

(a)

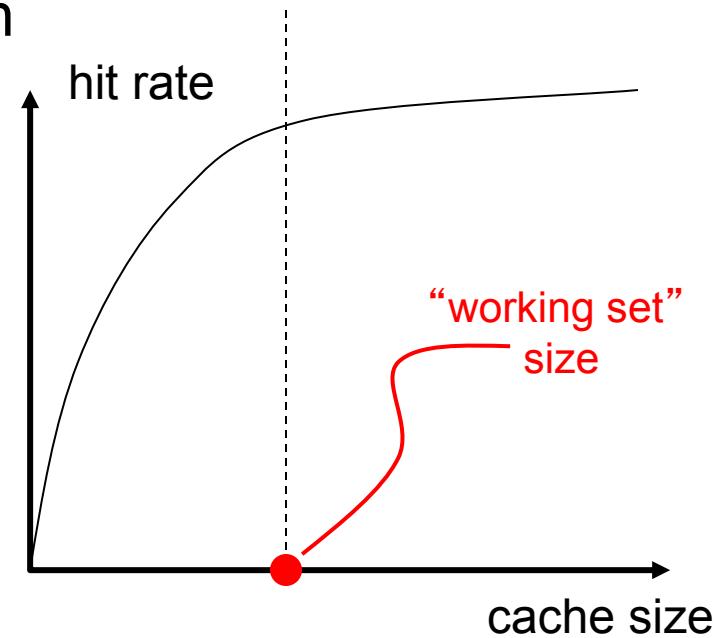
Way 1			Way 0			
V	U	Tag	Data	V	Tag	Data
0	0			0		
0	0			0		
1	1	00...010	mem[0x00...24]	1	00...101	mem[0x00...54]
0	0			0		

Set 3 (11)  
Set 2 (10)  
Set 1 (01)  
Set 0 (00)

(b)

# Cache Size

- Cache size: total data (not including tag) capacity
  - bigger can exploit temporal locality better
  - not ALWAYS better
- **Too large** a cache adversely affects hit and miss latency
  - smaller is faster => bigger is slower
  - access time may degrade critical path
- **Too small** a cache
  - doesn't exploit temporal locality well
  - useful data replaced often
- **Working set**: l'intero set di dati a cui fa riferimento l'applicazione in esecuzione
  - Entro un intervallo di tempo



# Block Size

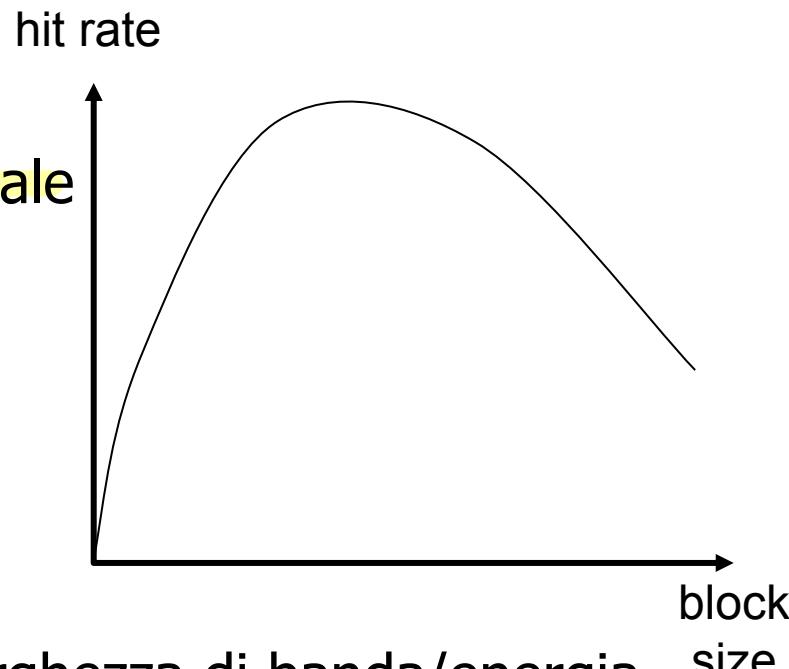
- Block size è la dimensione del dato associato a un address tag

- Blocchi troppo piccoli

- Non sfruttano bene la località spaziale
  - Maggior overhead per tag

- Blocchi troppo larghi

- Pochi blocchi → non sfrutta bene la località temporale
  - spreco di spazio nella cache e di larghezza di banda/energia
    - se località spaziale non è alta



# Associativity

- Quanti blocchi possono essere presenti allo stesso indice (i.e., set)?

dopo un certo grado  
diventa marginale.

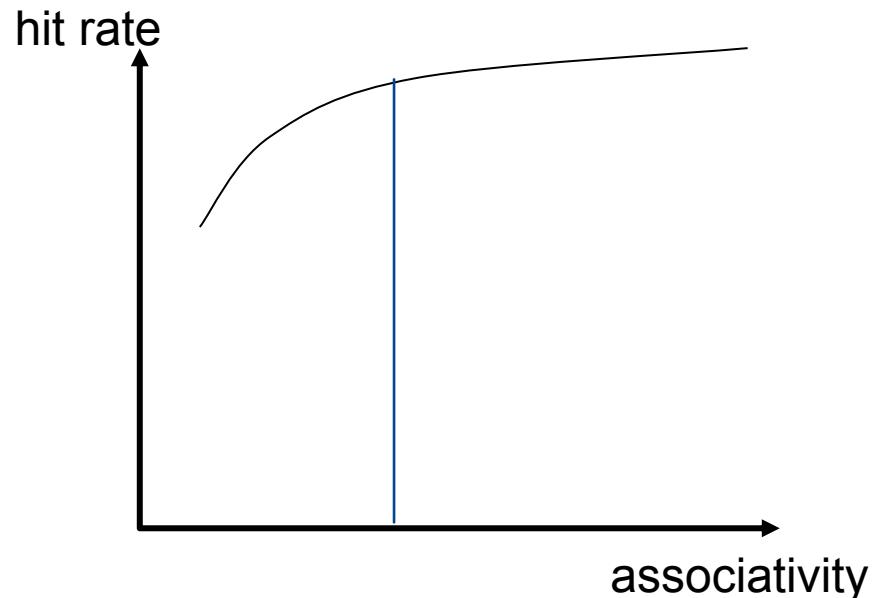
> associatività > hit rate

## Larger associativity

- minore miss rate (reduce i conflitti)
- maggiore hit latency e costo area (plus diminishing returns)

## Smaller associativity

- minor costo
- minore hit latency
  - Molto importante per L1 caches



# Classification of Cache Misses

- Miss obbligatorie (Compulsory miss) *non risolve x es. car logiche di prefetch.*
  - first reference to an address (block) always results in a miss
  - subsequent references should hit unless the cache block is displaced for the reasons below
    - ↗ i dati del programma sono > capacità della cache
- Miss di Capacità (Capacity miss) *(nel caso di fully associative)*
  - cache is too small to hold everything needed
  - defined as the misses that would occur even in a fully-associative cache (with optimal replacement) of the same capacity
- Miss di Conflitto (Conflict miss)
  - defined as any miss that is neither a compulsory nor a capacity miss
    - ↑ tutto il resto di tipi di miss.

# Types of Misses

---

- **Obbligatoria (Compulsory)**: se è la prima volta che si accede al dato (Cold Start Miss)
- **Di capacità (Capacity)**: se accessi successivi al primo falliscono perché la cache è troppo piccola per il programma eseguito (caso di cache normalmente piena)
- **Di conflitto (Conflict)**: se accessi successivi al primo falliscono perché il numero di vie è troppo piccolo (troppe linee del programma sono associate allo stesso set)

# How to Reduce Each Miss Type

---

## ■ Compulsory

- Caching cannot help
- Prefetching can: Anticipate which blocks will be needed soon

## ■ Conflict

- More associativity
- Other ways to get more associativity without making the cache associative

- Victim cache → introduce un buffer (registro) tra la cache e le memorie inferiori. Questo buffer contiene una lista di cache.  
Se l'istruzione era patologica trovava victim cache.
- Better, randomized indexing
- Software hints?

## ■ Capacity

- Utilize cache space better: keep blocks that will be referenced
- Software management: divide working set and computation such that each “computation phase” fits in cache

# Software Approaches for Higher Hit Rate

---

- Restructuring data access pattern
- Restructuring data layout
  
- Loop interchange
- Data structure separation/merging
- Blocking
- ...

# Restructuring Data Access Patterns (I)

- Idea: Cambiare il layout dei dati o i modelli di accesso ai dati
- Example: Se row-major
  - Es. C:  $\text{int } x[5000][100] = \{\{\dots\}, \{\dots\}, \dots, \{\dots\}\};$
  - $x[i][j+1]$  segue  $x[i][j]$  in memoria
  - $x[i+1][j]$  è lontano da  $x[i][j]$

Code A

```
for (i = 0; i < 5000; i = i + 1)
    for (j = 0; j < 100; j = j + 1)
        x[i][j] = 2 * x[i][j];
```

Code B

```
for (j = 0; j < 100; j = j + 1)
    for (i = 0; i < 5000; i = i + 1)
        x[i][j] = 2 * x[i][j];
```

- Which one is best?
- Moving from CodeA to CodeB is called **loop interchange**
- Other optimizations can also increase hit rate
  - Loop fusion, array merging, ...

# Restructuring Data Access Patterns (I)

## ■ Example: Se row-major

- Es. C:  $\text{int } x[5000][100] = \{\{ \dots \}, \{ \dots \}, \dots, \{ \dots \} \};$
- $x[i][j+1]$  segue  $x[i][j]$  in memoria
- $x[i+1][j]$  è lontano da  $x[i][j]$

entra sub codice faccio le stesse cose ma sarebbe  
neglig il codice A.

Iudizio che hanno le stesse  $j$ , ma hanno i diverse sono  
distanti in memoria. Iudizzi con  
stessa: mej diverse sono contigui.

Se tengo fissa  $i$  esco  $j$ , sto scorrendo in modo contiguo la memoria.  
In caso di uno ~~ness~~ compulsorie punto in cache tutto il blocco di memoria  
e avro' poi una serie di hit negli elementi dell'array consecutivi.  
se la matrice e' + piccole delle cache col e' fully ass. i codici sono indifferenti.

**Code A** qui prima scorso le righe poi le colonne      **Code B** prima scorso le colonne poi le righe

```
for (i = 0; i < 5000; i = i + 1)  
    for (j = 0; j < 100; j = j + 1)  
        x[i][j] = 2 * x[i][j];
```

```
for (j = 0; j < 100; j = j + 1)  
    for (i = 0; i < 5000; i = i + 1)  
        x[i][j] = 2 * x[i][j];
```

# Restructuring Data Access Patterns (II)

## ■ Blocking

Dati i due loop di prima, crea cicli aggiuntivi in cui limita lo scope dei loop interni. → es. scorre da 0 a  $N/4$  alla volta, così che i blocchi a cui accede siano contenuti nella cache. **DINDO IL WORKING SET IN PEZZETTI CHE SINGOLARMENTE STANNO NELLA CACHE.** L'esempio è nella slide successiva.

- Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the cache
- Avoids cache conflicts between different chunks of computation
- Essentially: **Divide the working set so that each piece fits in the cache**
- Also called Tiling

# Restructuring Data Access Patterns (II)

tutte le matrici sono immagazzinate per righe e sono di pari dimensioni.

## ■ Es: Matrix Multiplication

- $int Y[N][N] = \{\{\ldots\}, \{\ldots\}, \dots, \{\ldots\}\};$
  - $int Z[N][N] = \{\{\ldots\}, \{\ldots\}, \dots, \{\ldots\}\};$
  - $int X[N][N] = \{\{\ldots\}, \{\ldots\}, \dots, \{\ldots\}\};$

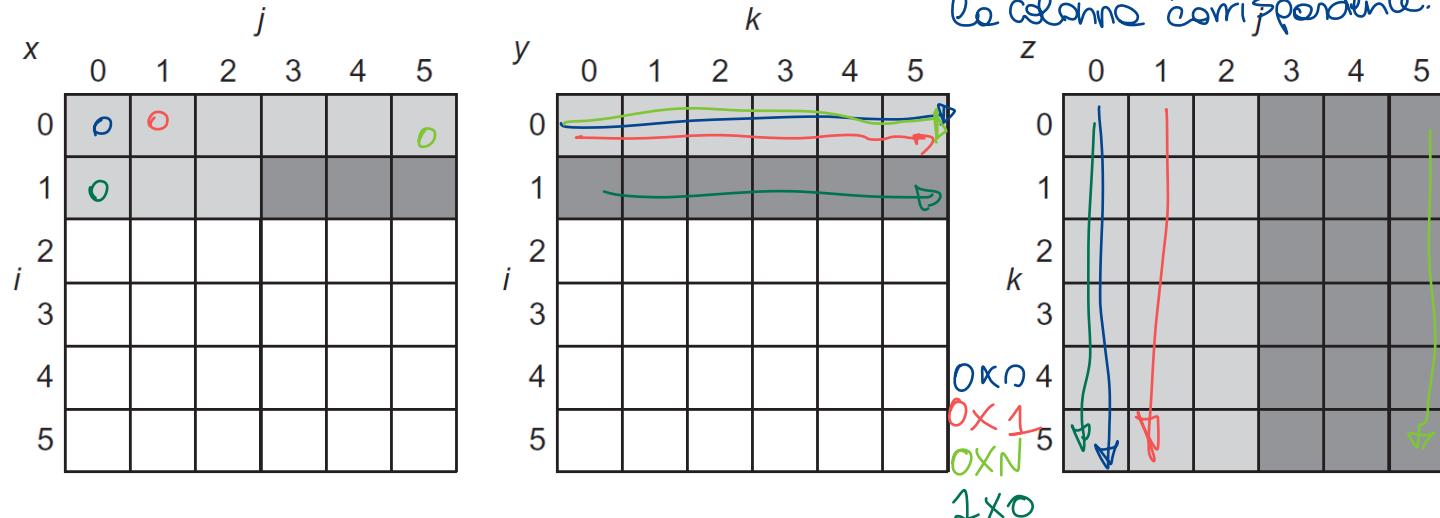
# RISULTATI

$$X = Y * Z$$

```

for (i = 0; i < N; i = i + 1)
    ↓   for (j = 0; j < N; j = j + 1)
        ↓   {r = 0; → azzerare la var. accumulatore
        ↓   Scorre le righe della matrice x.
        ↓   for (k = 0; k < N; k = k + 1)
            ↓   r = r + y[i][k]*z[k][j];
            ↓   x[i][j] = r;
            ↓   };
            ↓   Scorre le colonne
            ↓   k
            ↓   scorre contemporaneamente le righe e le colonne corrispondenti.

```



Sulla  
matrice  $\Sigma$   
ho altissima  
località  
temporale.  
Se le metto  
tutta in cache

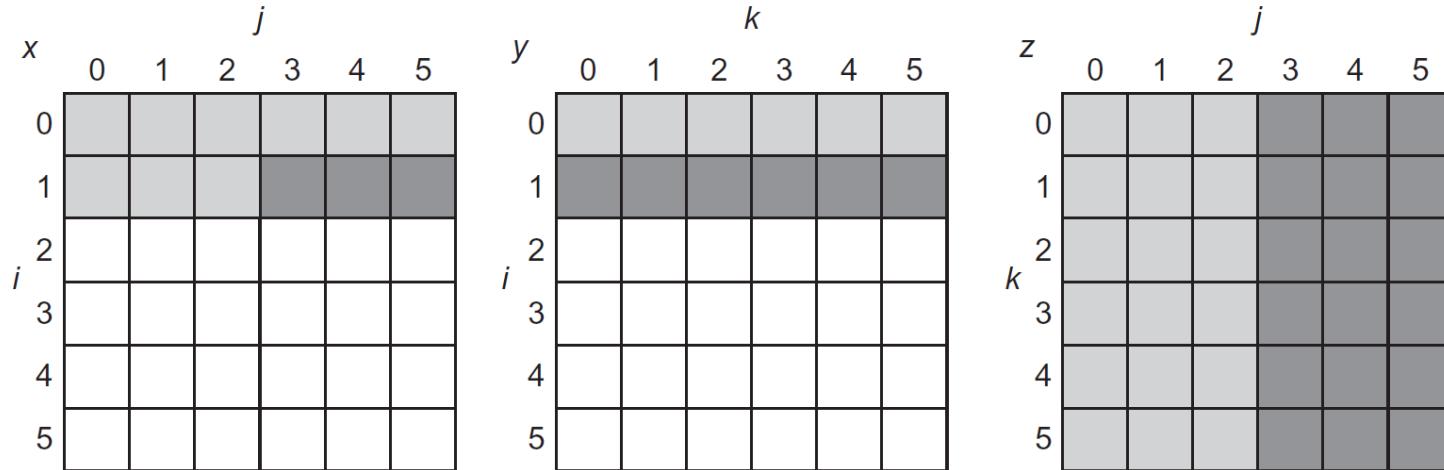
**Figure 2.13** A snapshot of the three arrays  $x$ ,  $y$ , and  $z$  when  $N=6$  and  $i = 1$ . The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. The elements of  $y$  and  $z$  are read repeatedly to calculate new elements of  $x$ . The variables  $i$ ,  $j$ , and  $k$  are shown along the rows or columns used to access the arrays.

# Restructuring Data Access Patterns (II)

I due cicli più interni leggono tutti gli elementi di Z ( $N \times N$ ), leggono ripetutamente gli  $N$  elementi di una riga di Y per scrivere gli elementi in una riga di X.

$$X = Y * Z$$

```
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        {r = 0;
         for (k = 0; k < N; k = k + 1)
             r = r + y[i][k]*z[k][j];
         x[i][j] = r;
        };
```



**Figure 2.13** A snapshot of the three arrays  $x$ ,  $y$ , and  $z$  when  $N=6$  and  $i=1$ . The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. The elements of  $y$  and  $z$  are read repeatedly to calculate new elements of  $x$ . The variables  $i$ ,  $j$ , and  $k$  are shown along the rows or columns used to access the arrays.

# Restructuring Data Access Patterns (II)

Se Cache Size > sizeof(X) + sizeof(Y) + sizeof(Z)

- No capacity misses, but possible conflict misses

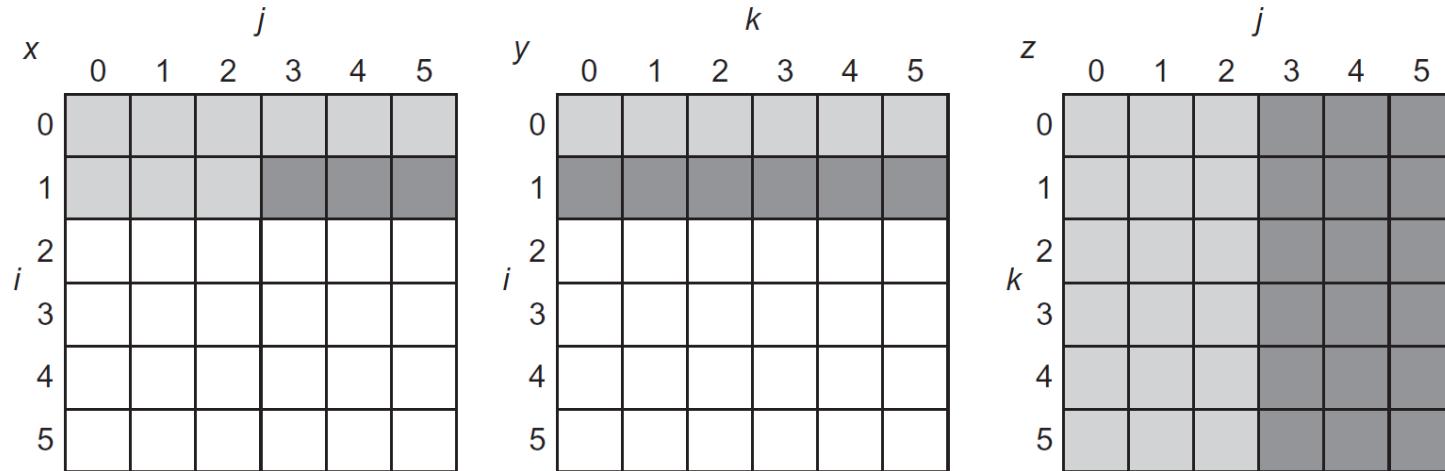
Se Cache Size > sizeof(Z) + sizeof(Y)/N [#una riga]

- Possiamo mantenere in cache Z e una riga di Y

Se Cache Size inferiore

- Molte miss per Z e Y.

} ;



**Figure 2.13** A snapshot of the three arrays  $x$ ,  $y$ , and  $z$  when  $N=6$  and  $i=1$ . The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. The elements of  $y$  and  $z$  are read repeatedly to calculate new elements of  $x$ . The variables  $i$ ,  $j$ , and  $k$  are shown along the rows or columns used to access the arrays.

# Restructuring Data Access Patterns (II)

## ■ Es: Matrix Multiplication w. Blocking/Tiling

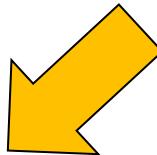
- $\text{int } Y[N][N] = \{\{\dots\}, \{\dots\}, \dots, \{\dots\}\};$
- $\text{int } Z[N][N] = \{\{\dots\}, \{\dots\}, \dots, \{\dots\}\};$
- $\text{int } X[N][N] = \{\{\dots\}, \{\dots\}, \dots, \{\dots\}\};$

```
for (jj = 0; jj < N; jj = jj + B)
for (kk = 0; kk < N; kk = kk + B)
for (i = 0; i < N; i = i + 1)
    for (j = jj; j < min(jj + B, N); j = j + 1)
        {r = 0;
         for (k = kk; k < min(kk + B, N); k = k + 1)
             r = r + y[i][k]*z[k][j];
         x[i][j] = x[i][j] + r;
        };
    }
```

Divido il prodotto scalare di righe e colonne con due sottoprodotto scalari di una parte di righe con una parte di colonna.

→ facciamo più operazioni, ma con meno località maggiore.

x	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						



```
for (i = 0; i < N; i = i + 1)
    for (j = 0; j < N; j = j + 1)
        {r = 0;
         for (k = 0; k < N; k = k + 1)
             r = r + y[i][k]*z[k][j];
         x[i][j] = r;
        };
    }
```

y	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						

z	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						

Figure 2.14 The age of accesses to the arrays x, y, and z when  $B = 3$ . Note that, in contrast to Figure 2.13, a smaller number of elements is accessed.

# Restructuring Data Layout (I)

---

```
struct Node {  
    struct Node* next;  
    int key;  
    char [256] name;  
    char [256] school;  
}  
  
while (node) {  
    if (node→key == input-key) {  
        // access other fields of node  
    }  
    node = node→next;  
}
```

520 byte

- Pointer based traversal (e.g., of a linked list)
- Assume a huge linked list (1B nodes) and unique keys
  
- Why does the code on the left have poor cache hit rate?
  - “Other fields” occupy most of the cache line even though rarely accessed!

# Restructuring Data Layout (II)

```
struct Node {  
    struct Node* next;  
    int key;  
    struct Node-data* node-data;  
}  
  
struct Node-data {  
    char [256] name;  
    char [256] school;  
}  
  
while (node) {  
    if (node->key == input-key) {  
        // access node->node-data  
    }  
    node = node->next;  
}
```

- Idea: separate frequently-used fields of a data structure and pack them into a separate data structure
- Who should do this?
  - Programmer
  - Compiler
    - Profiling vs. dynamic
  - Hardware?
  - Who can determine what is frequently used?