Gerarchie di memoria e Cache

Caratteristiche memoria

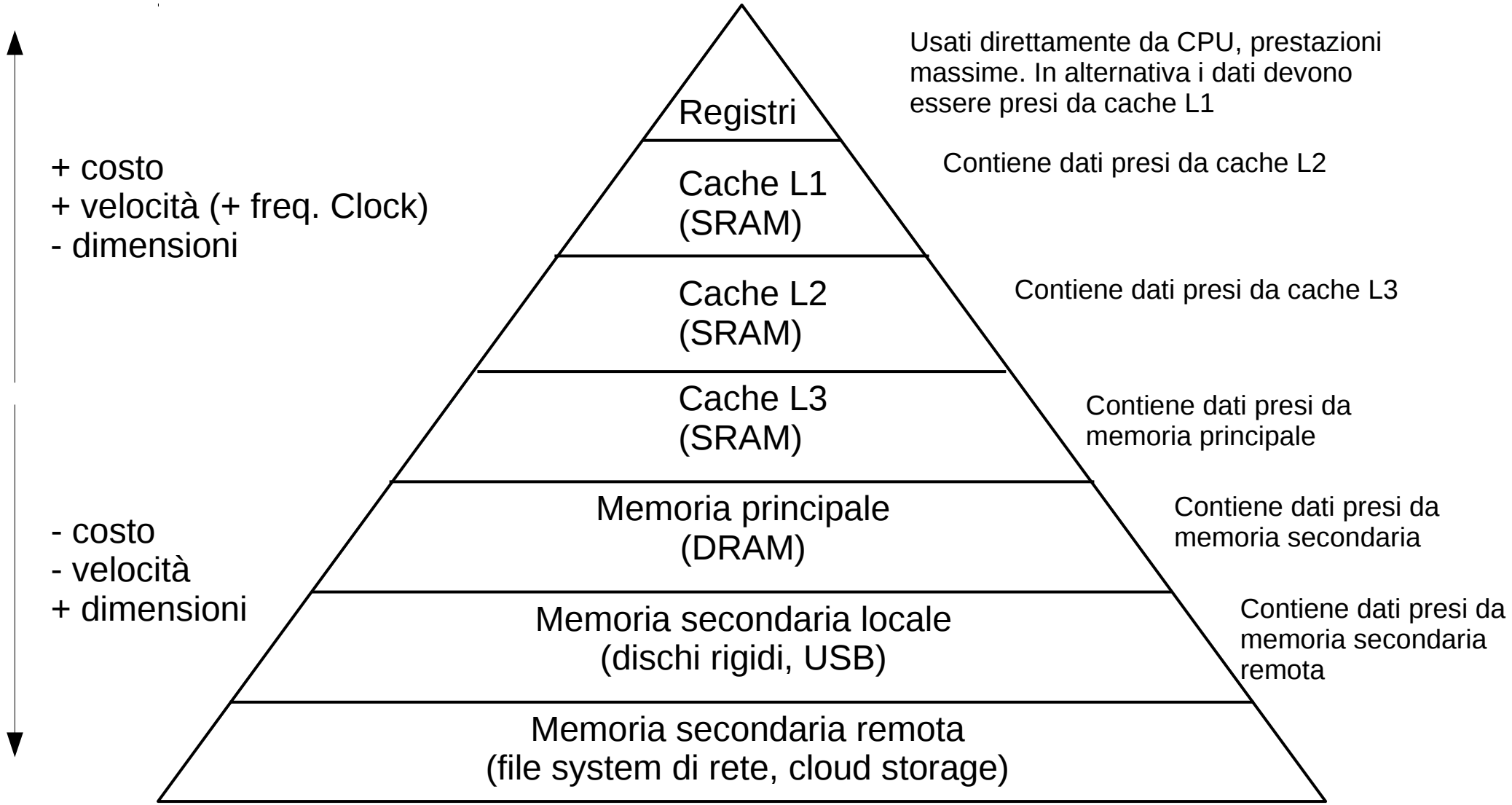
*“Ideally one would desire an **indefinitely large memory capacity** such that any particular . . . word would be **immediately available**. . . . We are . . . forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.”*

A. W. Burks, H. H. Goldstine, and J. von Neumann Preliminary Discussion of the Logical Design of an Electronic Computing Instrument, 1946

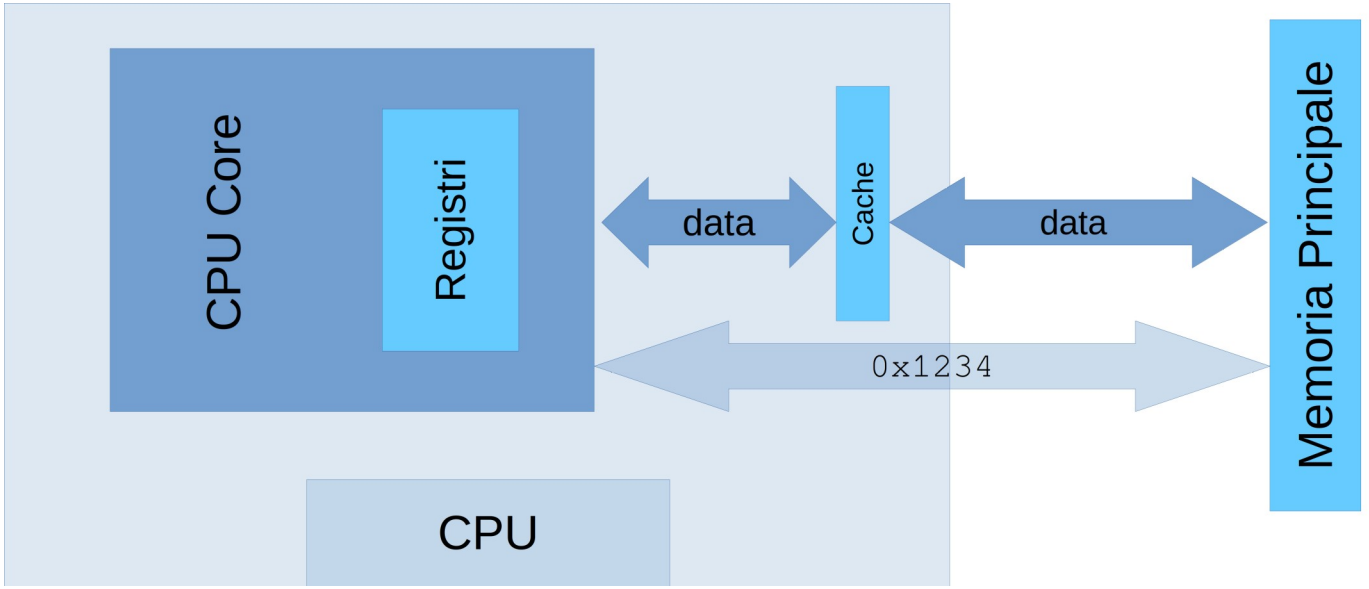
- Nella realtà si deve cercare un compromesso tra queste caratteristiche

Tecnologia	Tempo di accesso	Costo per GB (2012)
SRAM	0.5-2.5 ns	500-1000 $
DRAM	10-100 ns	10-20 $
Flash	5-50 µs	0.75-1 $
Disco rigido	5-10 ms	0.05-0.10 $

Gerarchia di memoria



Gerarchie di memoria



- Dal punto di vista della CPU (e quindi del programmatore) ogni accesso in memoria fa riferimento alla memoria principale
- Eventuali cache intermedie sono trasparenti, ovvero non vengono mai indirizzate esplicitamente
- In generale durante l'esecuzione non è possibile controllare in maniera semplice se un dato è già presente in cache
  - Normalmente esistono istruzioni speciali per forzare caricamento/svuotamento

Principio di località

- L'esecuzione di un qualunque programma *tende* a seguire il principio di località:
  - **Principio di località spaziale**
    - Se accedo ad un indirizzo X di memoria, probabilmente a breve dovrò accedere anche agli indirizzi adiacenti

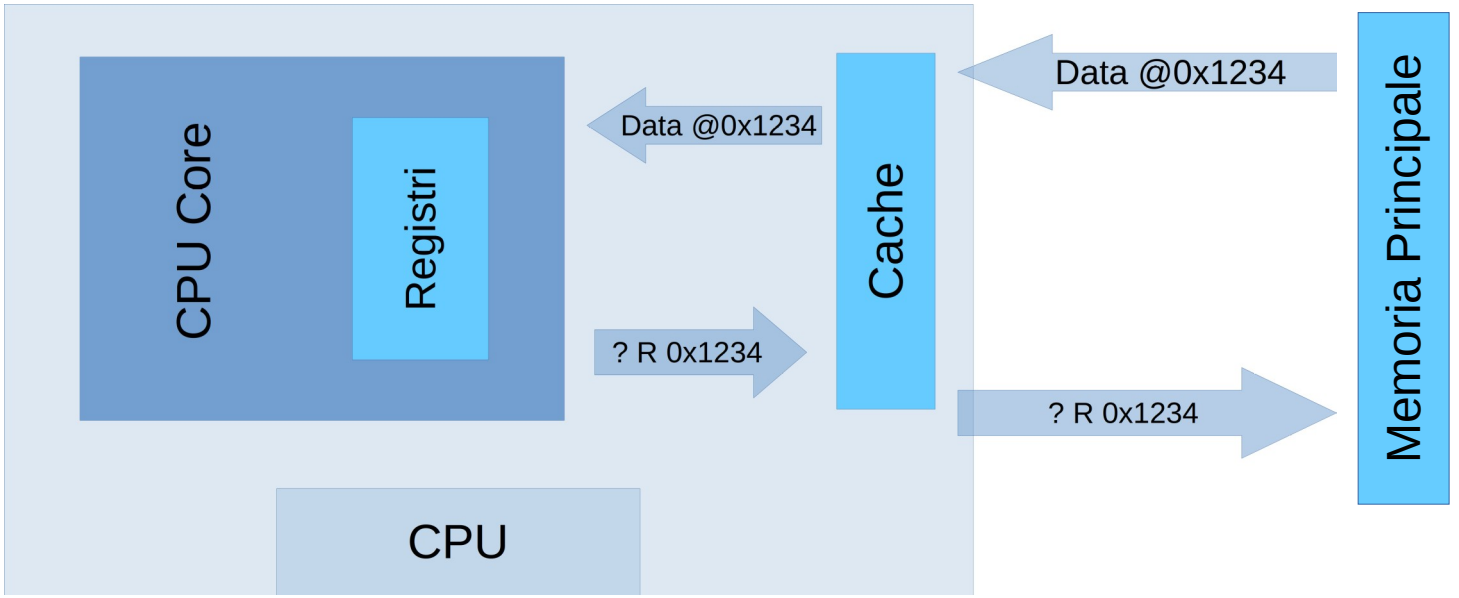
      Es. (dati) se uso `arr[i]` probabilmente userò anche `arr[i+1]`

      Es. (istruzioni) esecuzione sequenziale, dopo istruzione `I[j]` eseguo `I[j+1]` a meno che non ci sia un salto
  - **Principio di località temporale**
    - Se accedo ad un indirizzo X di memoria, probabilmente a breve dovrò accederci di nuovo

      Es. (dati) in un ciclo `for (i=0; i<10; i++)` la variabile `i` è usata ad ogni iterazione

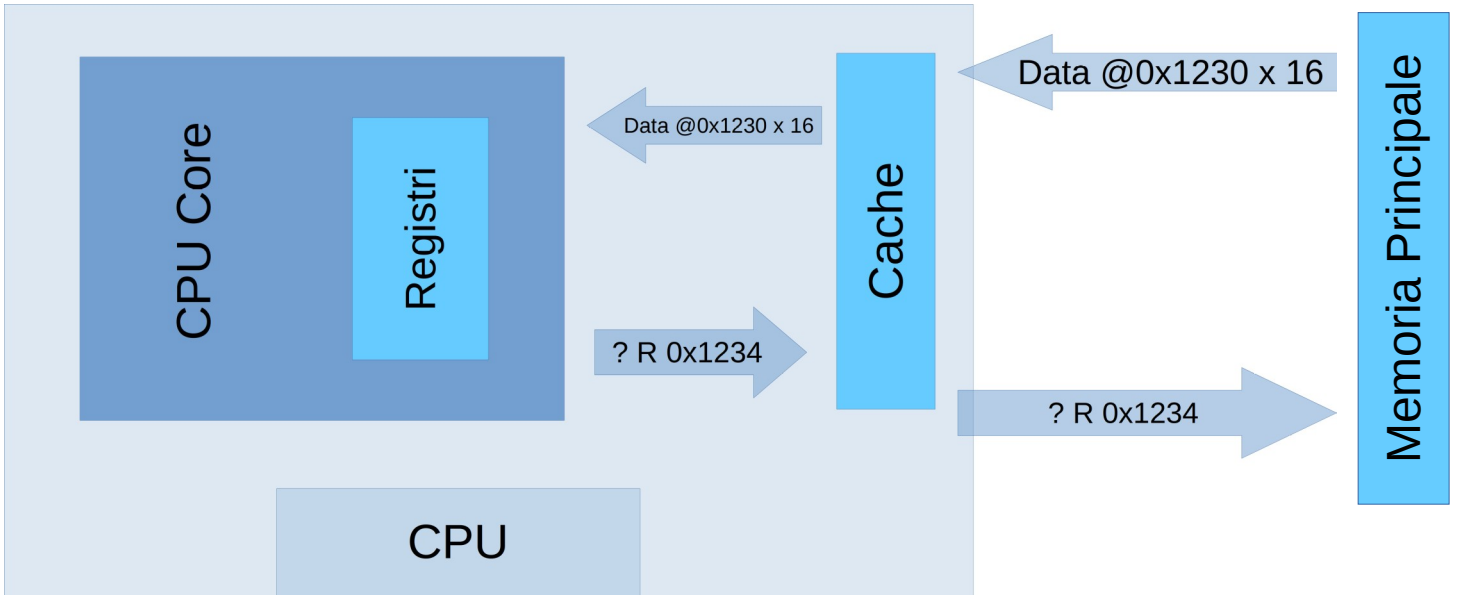
      Es. (istruzioni) In un ciclo molte istruzioni vengono eseguite ripetutamente a intervalli brevi (ad ogni iterazione)

Cache e località temporale



- Quando la CPU inizialmente richiede una lettura da memoria, il dato viene caricato dalla memoria principale e salvato nei livelli di cache intermedia
  - Accesso a DRAM lento
- Se dopo poco tempo la CPU accede allo stesso indirizzo, il dato viene fornito direttamente dalla cache
  - Accesso a SRAM molto più rapido

Cache e località spaziale



- Quando la CPU inizialmente richiede una lettura di 4 byte dall'indirizzo 0x1234, la cache ordina il trasferimento anche dei dati agli indirizzi adiacenti
  - Trasferisco l'intero blocco da 0x1230 a 0x123F (16 byte)
- Se dopo poco tempo la CPU accede a indirizzi adiacenti, il dato viene fornito direttamente dalla cache
  - Il dato all'indirizzo 0x123A è già presente in cache

Cache Hit e Miss

- **Cache Hit –** Il dato richiesto dalla CPU è già presente in cache
  - Prestazioni elevate
  - Hit-rate – percentuale di hit sul totale di accessi in memoria
- **Cache Miss –** Il dato richiesto dalla CPU non è presente in cache e deve essere caricato dalla memoria principale
  - Prestazioni ridotte
  - Miss-rate – percentuale di miss sul totale di accessi in memoria
- Obiettivo: massimizzare hit-rate

  ```
  hit_rate + miss_rate = 1
  ```

- Normalmente `hit_rate` si aggira intorno a 0.9, con un incremento di prestazioni notevole

Prestazioni di una cache

- **hit time** - tempo richiesto perché un dato venga copiato dalla cache alla CPU
  - è un'operazione veloce, solitamente richiede da 1 a 3 cicli di clock
- **miss penalty** - tempo richiesto perché un dato venga copiato dalla memoria principale alla cache
  - è un'operazione più lenta, solitamente richiede da 10 a 100 di cicli di clock
- **AMAT** – Average Memory Access Time, tempo medio di accesso alla memoria

```
AMAT = hit_time + miss_rate • miss_penalty
```

- **AMAT** valuta le prestazioni *solo* per quanto riguarda gli accessi in memoria
  - **CPU time** rimane metrica valida più in generale

Esempio prestazioni

- La hit rate è 0.9 e l'hit time è un solo ciclo di clock, ma la miss penalty è 50 cicli di clock.

```
miss_rate = 1 – 0.9 = 0.1
AMAT = 1 + 0.1 • 50 = 6 cicli di clock
```

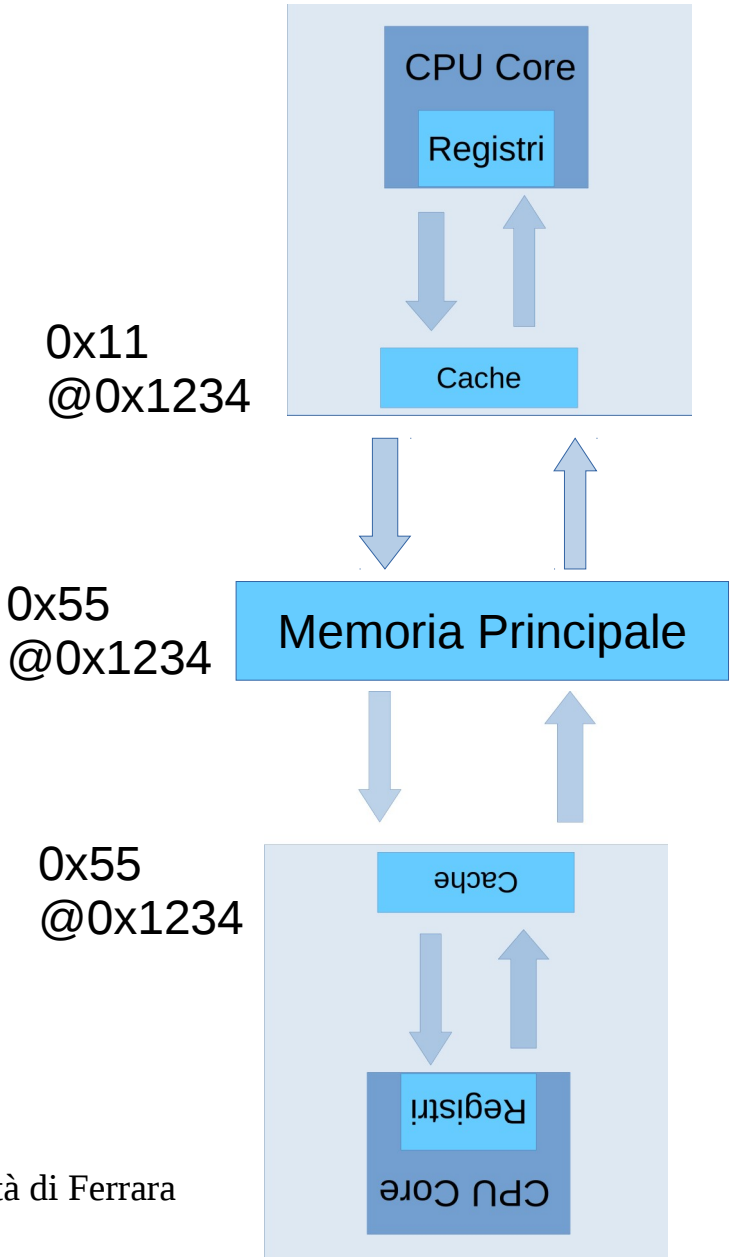
- In media un accesso in memoria impiega 6 cicli di clock, rispetto ad 1 solo ciclo di clock del caso migliore e 50 del caso peggiore!

- Possibili miglioramenti:
  - hit_time e miss_penalty legati all'hardware
    - Cache multi-livello (L1/L2/L3)
  - miss_rate dipende invece dal funzionamento della cache!

Coerenza della memoria

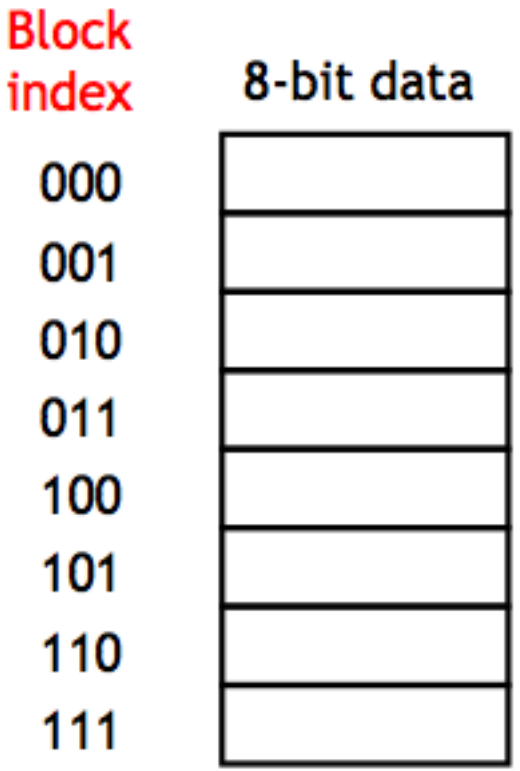
- La memoria principale e la cache contengono gli stessi dati per uno stesso indirizzo?
- Memoria non coerente può essere un problema
  - Sistemi multiprocessore
  - Memoria condivisa
  - Interfacciamento con periferiche

  Diversi core/processi/periferiche possono “vedere” dati diversi ad uno stesso indirizzo!!



Struttura di una cache

- Una cache è un insieme di $m=2^p$ blocchi di $n=2^k$ byte ciascuno
- Ogni blocco è identificato da un indice (**block index**)
  - "indirizzo" valido esclusivamente nella cache
- Dimensione totale della cache molto inferiore a memoria principale
- Esempio: cache composta da 8 blocchi da 1 byte → 8 byte
- Sufficienti 3 bit per block index



Funzionamento generale

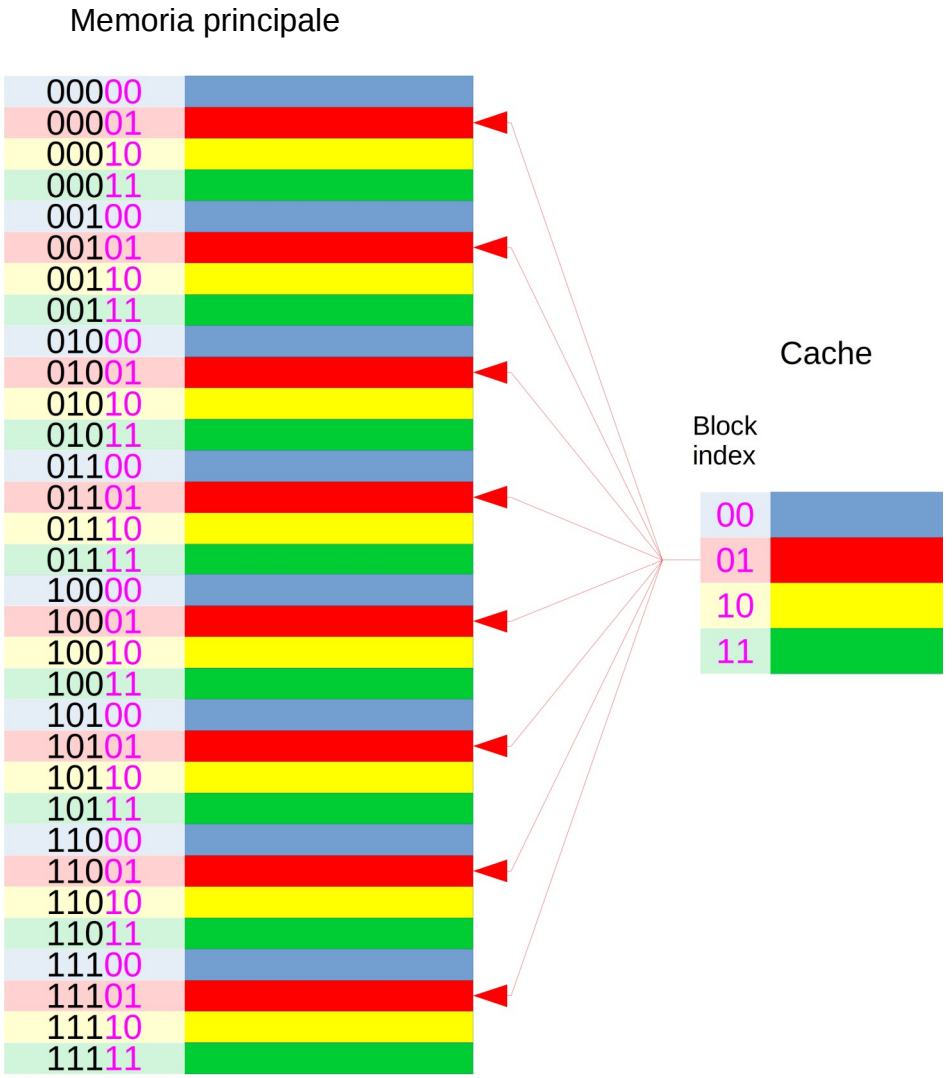
1) Quando copiamo un blocco di dato dalla memoria principale nella cache, dove lo mettiamo esattamente? **(caricamento)**
2) Come possiamo dire che un dato è già in cache o se deve essere prima recuperato dalla memoria principale? **(ricerca)**
3) Prima o poi la cache si riempirà. Per caricare un nuovo dato in cache dobbiamo eliminarne uno vecchio: come decidiamo quale eliminare?**(sostituzione)**
4) Come gestiamo le scritture in memoria? **(politica di scrittura)**

Diverse implementazioni, con un compromesso tra performance e complessità
- **Direct Mapped**
- **N-way/Fully associative**

Diverse strategie di scrittura
- Dato in cache:
  Write-back/Write-through
- Dato non in cache
  - **Write-allocate/no-allocate**

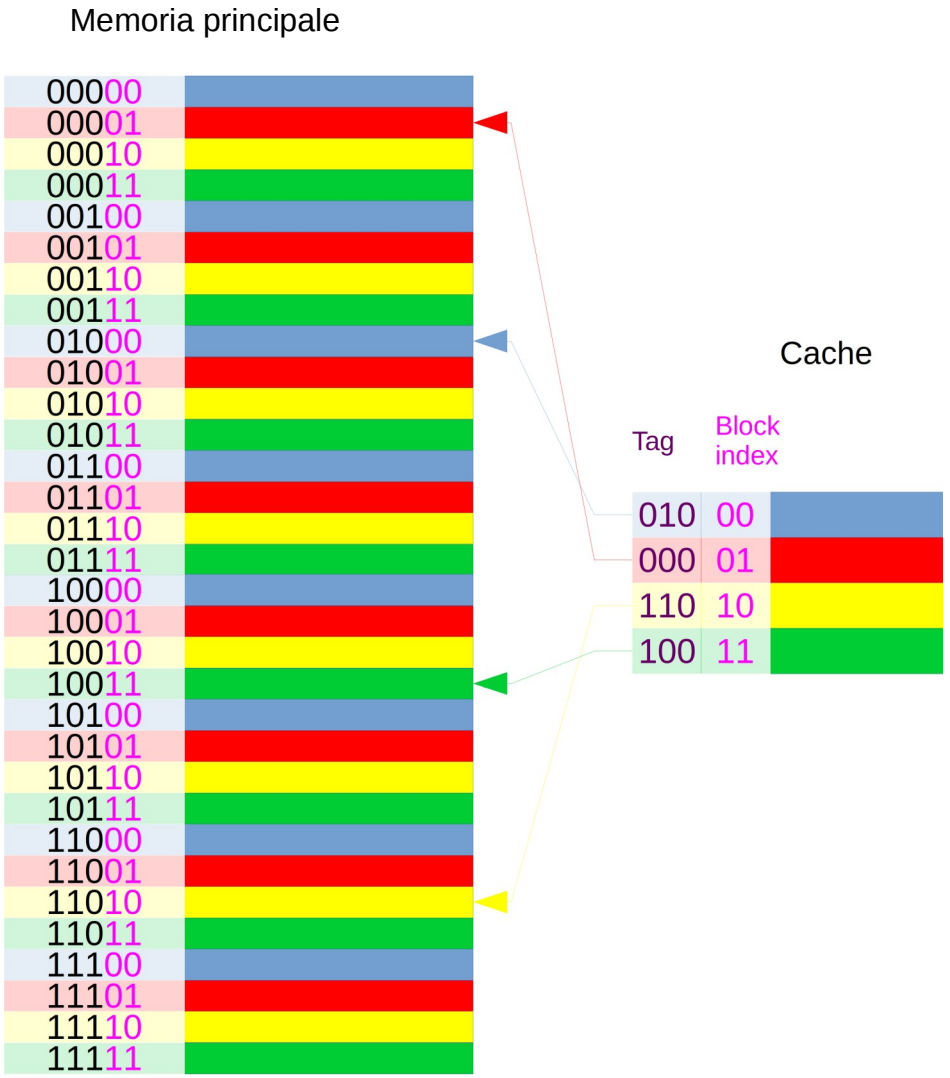
Cache Direct-Mapped

- Implementazione più semplice
  - Ogni indirizzo in memoria principale corrisponde a un solo blocco (ovviamente non vale il contrario)
- Posso trovare il blocco corrispondente ad un dato indirizzo con un'operazione di modulo
  - Se i blocchi sono da 1 byte (con $m=2^p$) posso usare i p bit meno significativi dell'indirizzo
- Esempio:
  - ho una memoria principale da 32 byte e una cache da 4 blocchi con 1 byte/blocco
  - La dimensione dell'indice è $p = \log_2 4 = 2$ bit
  - Ogni blocco può corrispondere a $32/4 = 8$ locazioni di memoria diverse!



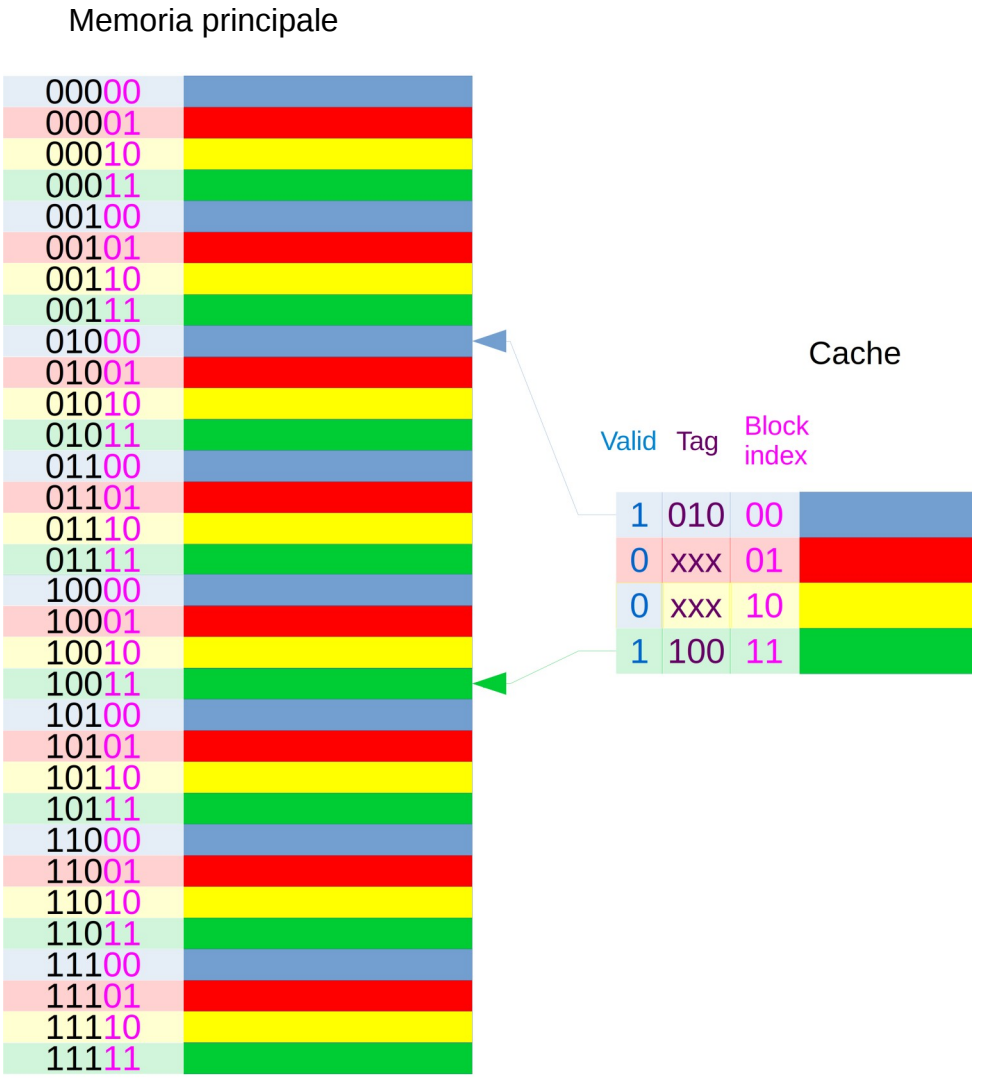
Cache Direct-Mapped

- Come tengo traccia di quale indirizzo è effettivamente caricato in ogni blocco di cache?
  - Aggiungo campo `Tag` con parte restante dell'indirizzo
- Concatenando `Tag` e `Block Index` ottengo l'indirizzo completo del blocco contenuto in memoria
  - Questo perché ogni blocco ha dimensione 1 byte
- Il campo `Tag` va aggiornato nel momento in cui si copia un blocco dalla memoria principale alla cache



Cache Direct-Mapped

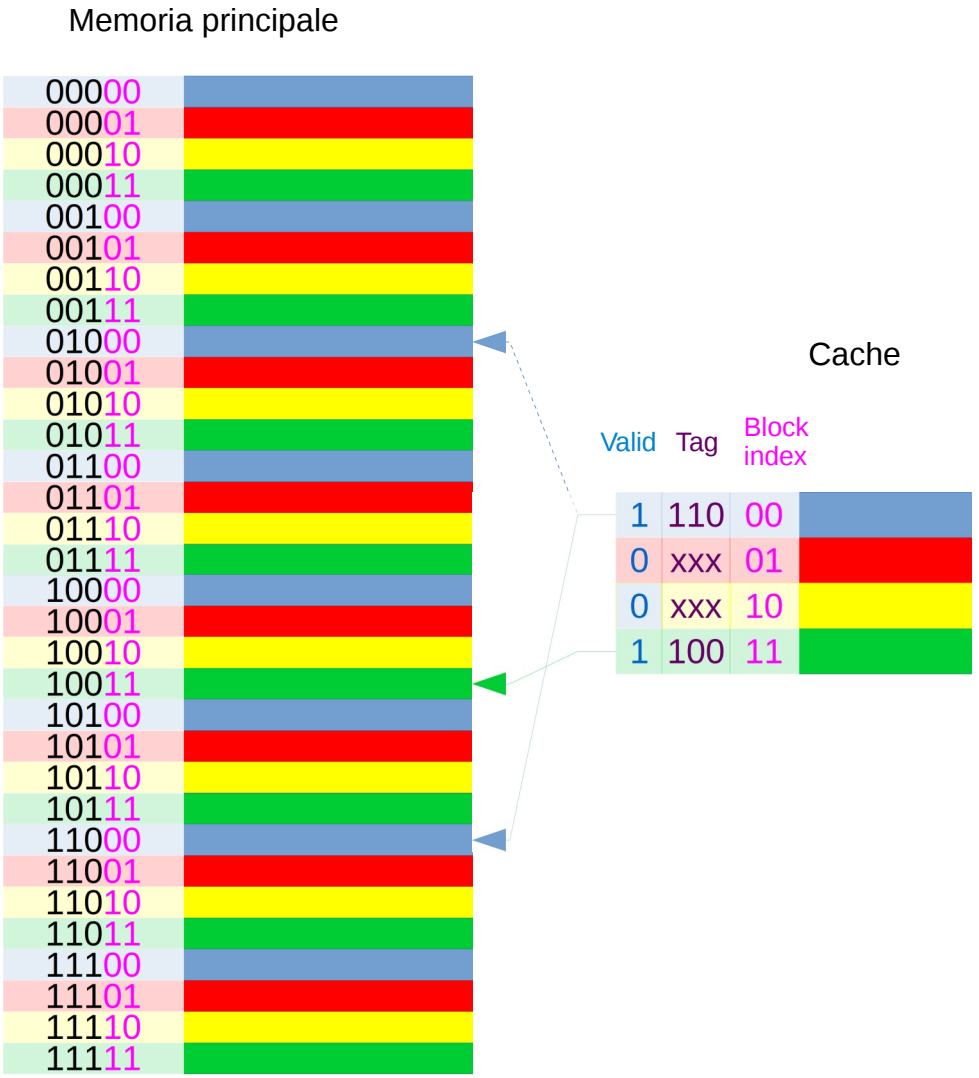
- È necessario avere controllo sul processo di riempimento e svuotamento della cache
  - All'avvio, i dati in cache non saranno validi
  - Posso voler forzare il caricamento o scaricamento di uno specifico indirizzo
- Necessario bit di validità
  - Se 1 il blocco è valido e corrisponde all'indirizzo dato da `Tag` e `Block Index`
    → **cache hit**
  - Se 0 il blocco non è valido, quindi un'operazione di lettura causerà un caricamento dalla memoria principale
    → **cache miss**



Cache Direct-Mapped

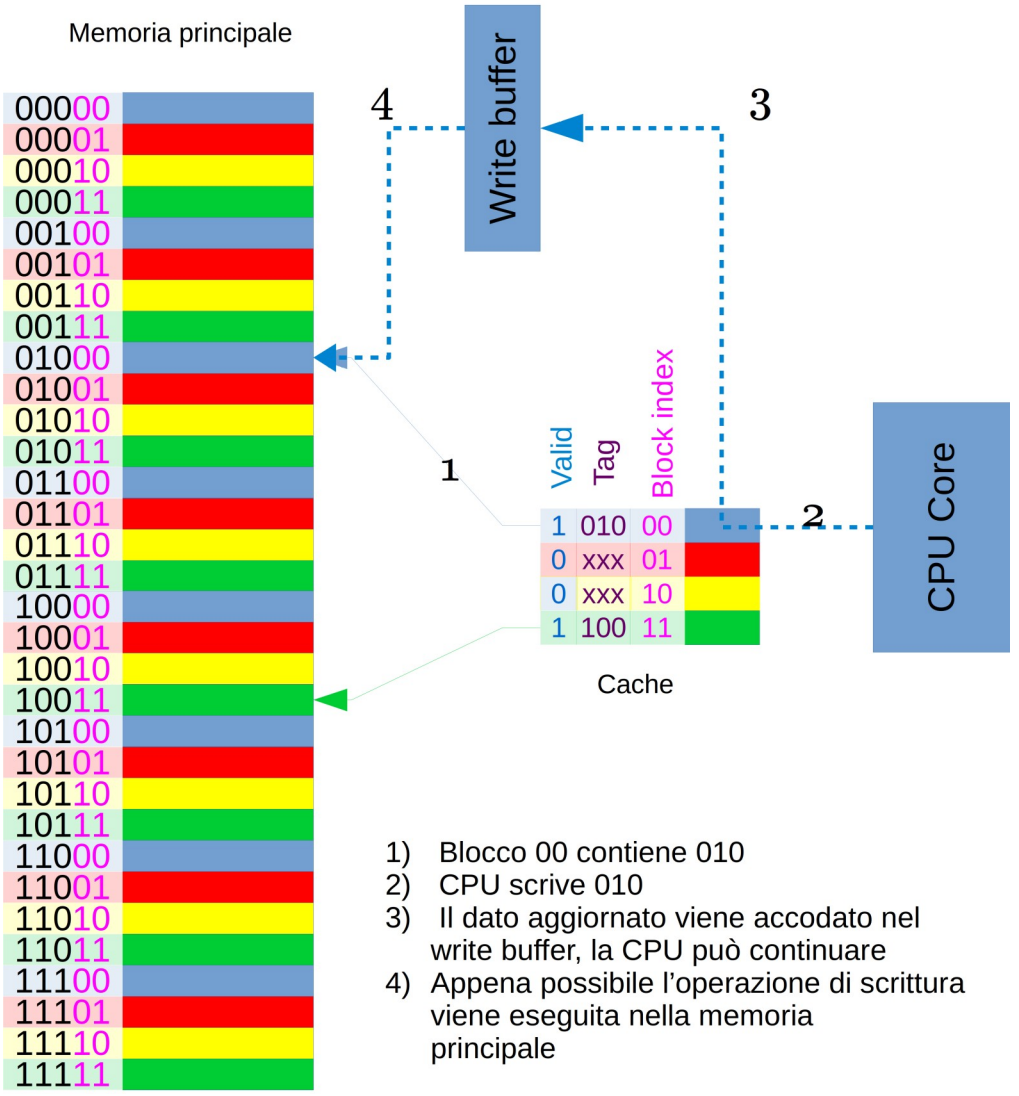
- Cosa succede se la CPU richiede un indirizzo il cui blocco corrispondente è già usato?
  - devo ricaricare un blocco da memoria principale e associarlo al nuovo indirizzo
  - Sovrascrivo il dato preesistente, aggiornando il campo `Tag`

    Sufficiente per memorie in sola lettura (es. memoria istruzioni)



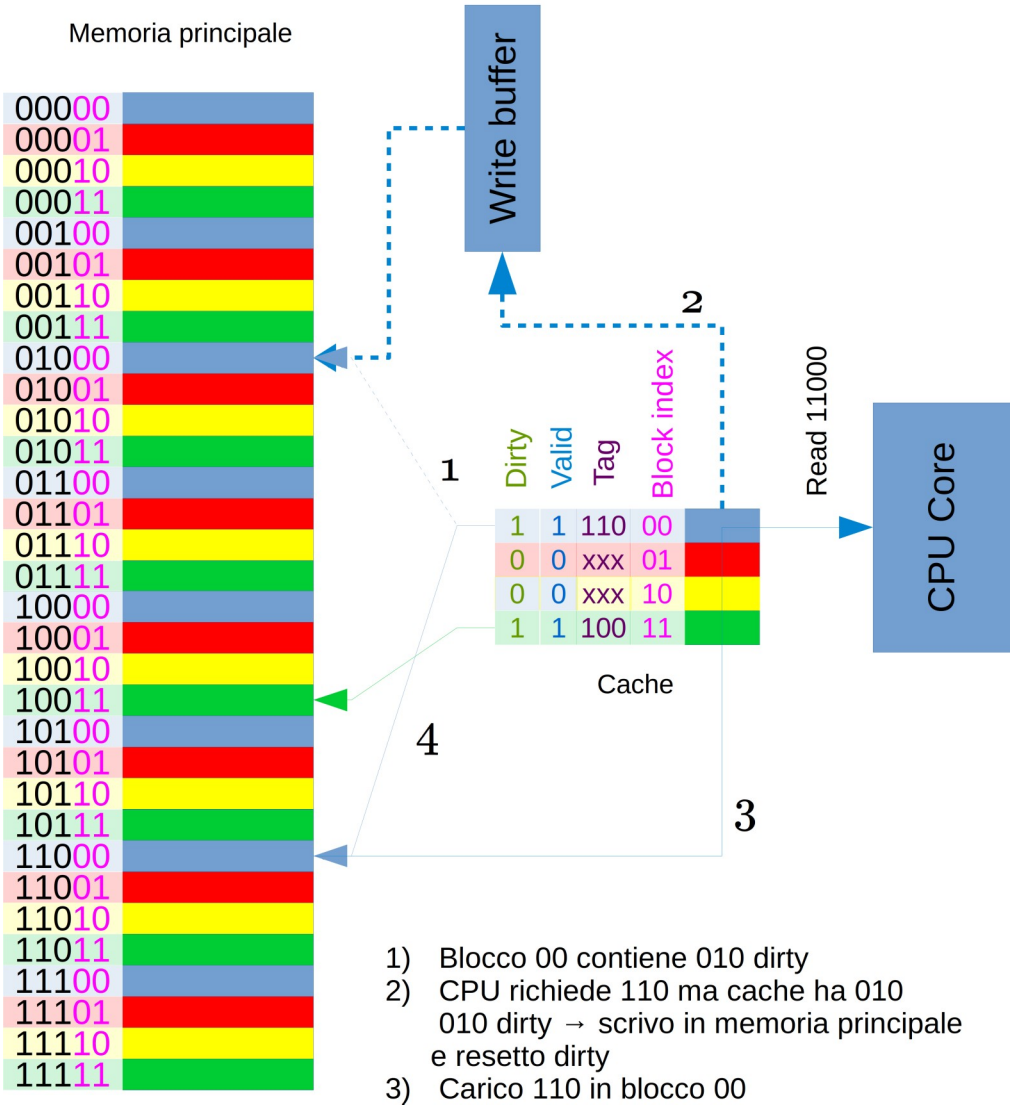
Cache Direct-Mapped

- Cosa succede se devo modificare un blocco?
  - Se il blocco è valido (write hit) modifico la cache
- Politica **write through**
  - Ogni operazione di scrittura nella cache aggiorna anche la memoria principale
  - Mantiene coerenza della memoria
  - Calo prestazioni, ogni scrittura richiede accesso a memoria principale
  - Soluzione: uso un write buffer
    - Coda per gestire le operazini di scrittura e permettere alla CPU di continuare
    - La CPU deve attendere la scrittura solo se il buffer è pieno



Cache Direct-Mapped

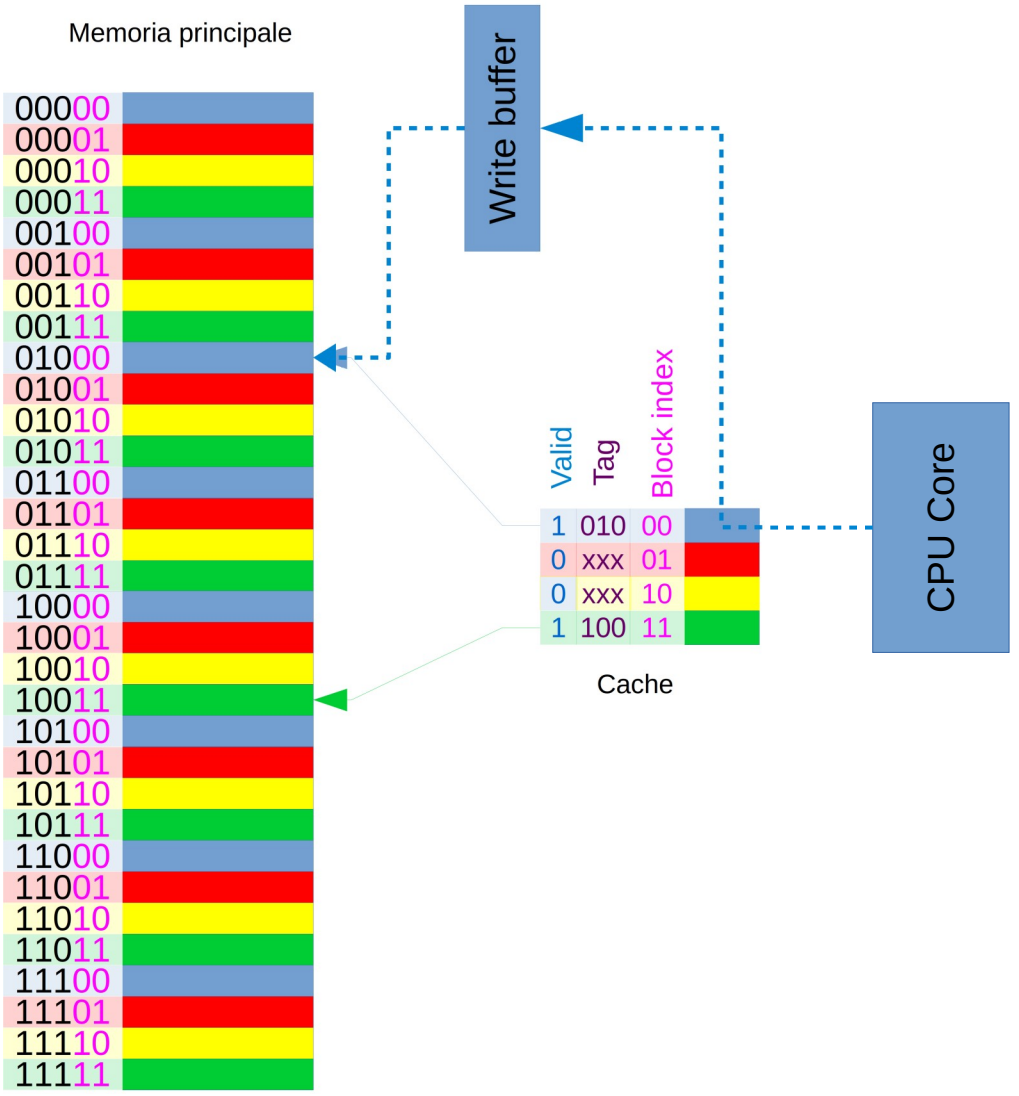
- Cosa succede se devo modificare un blocco?
  - Se il blocco è valido (write hit) modifico la cache
- Politica **write-back**
  - Aggiungo un bit “dirty” alla cache per segnalare l’incoerenza del blocco
  - Aggiorno memoria principale solo quando devo sostituire il blocco
  - Prestazioni maggiori, a patto di poter tollerare temporanea incoerenza della memoria
  - Anche qui posso sfruttare un write buffer



1) Blocco 00 contiene 010 dirty
2) CPU richiede 110 ma cache ha 010
   010 dirty → scrivo in memoria principale e resetto dirty
3) Carico 110 in blocco 00
4) Blocco 00 valido contiene 110

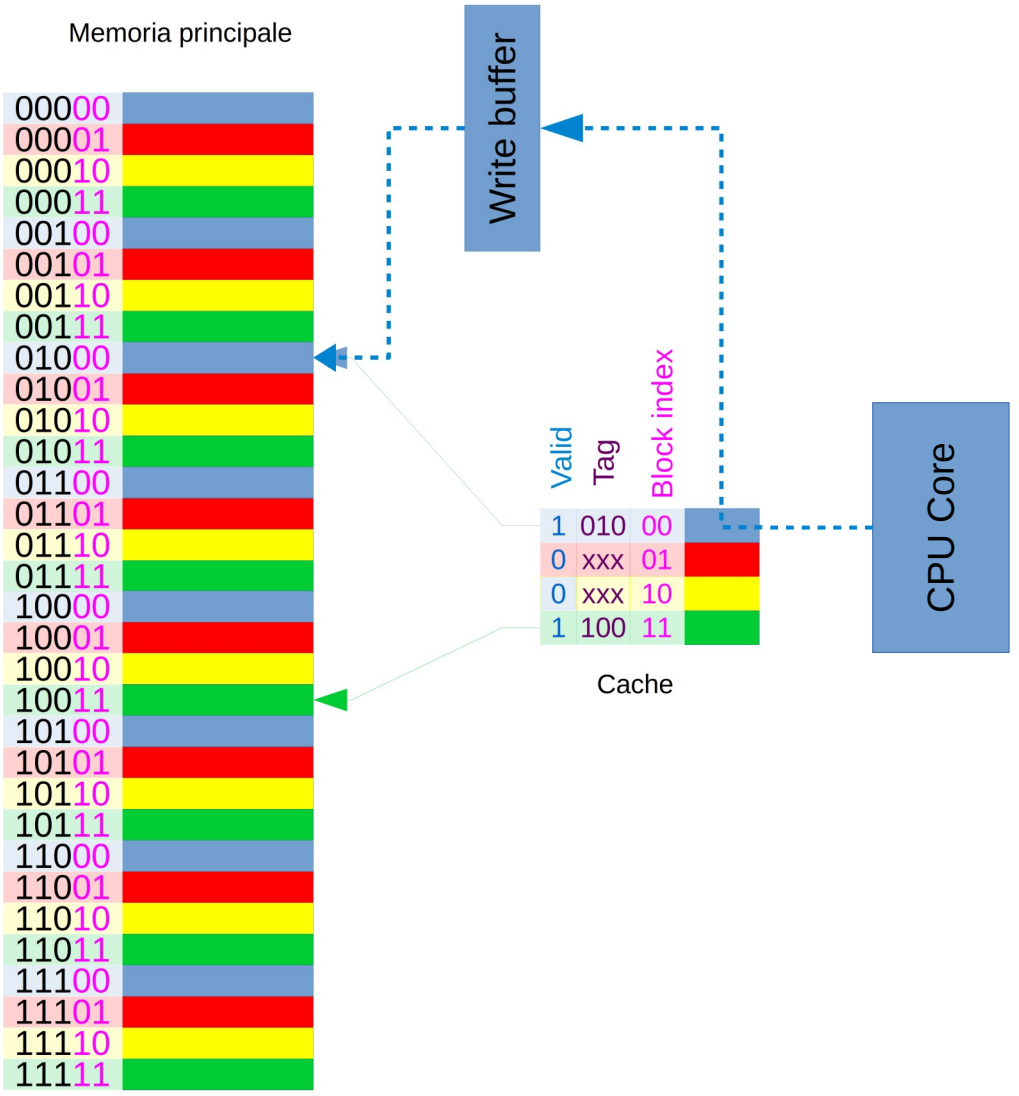
Cache Direct-Mapped

- Cosa succede se devo modificare un blocco?
  - Se il blocco non è valido (write miss)?
- Politica **write allocate**
  - Carico il blocco in cache prima di modificarlo
  - Cache più aggiornata (località temporale)
  - Due accessi in memoria
  - Solitamente usato con write-back



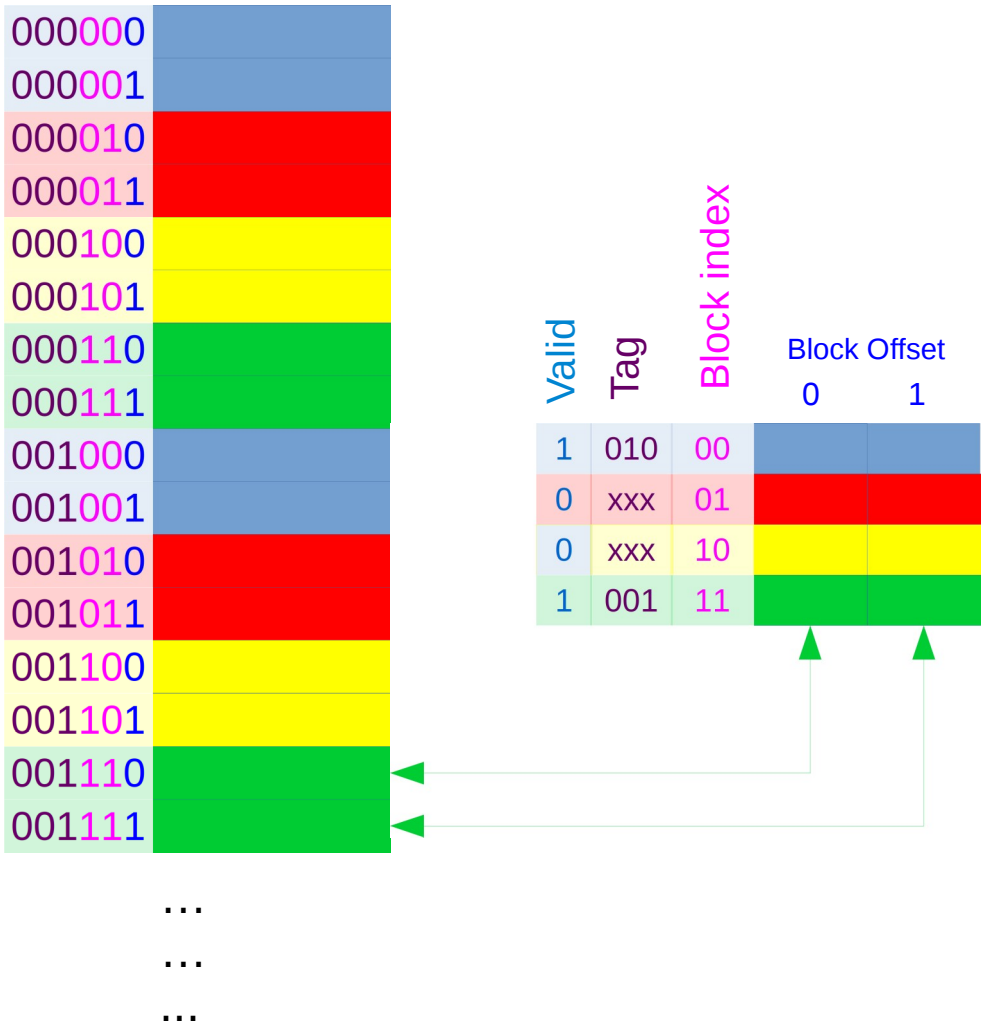
Cache Direct-Mapped

- Cosa succede se devo modificare un blocco?
  - Se il blocco non è valido (write miss)?
- Politica **no write allocate**
  - Evito di caricare il blocco in cache, propago direttamente la scrittura al write buffer
  - Un solo accesso in memoria, prestazioni maggiori
  - Solitamente usato con write-through



Cache Direct-Mapped

- Località spaziale → necessito blocchi di dimensione maggiore $b=2^q$
- Tag e Block Index si riferiscono a blocchi allineati a q bit
- In memoria principale non ho più accessi in memoria a 1 byte ma sempre a blocchi di $b=2^q$
- Tag e Block index corrispondono sempre a parte dell'indirizzo del blocco
- I q bit meno significativi corrispondono all'offset di un indirizzo all'interno del blocco



Esempio direct mapped

- Cache da 4 KB:
  - Indirizzi a 32 bit
  - 4 byte/blocco

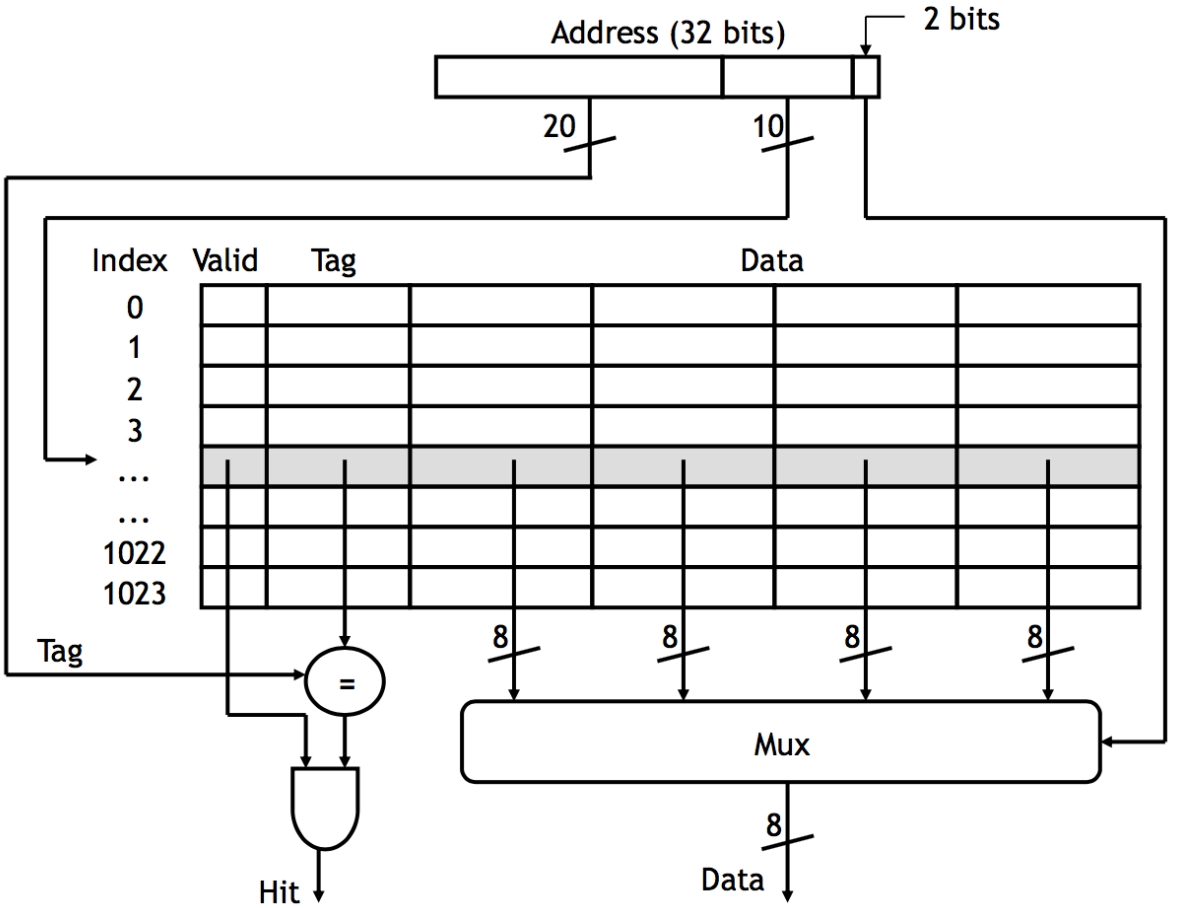
- Calcolo numero blocchi

  `4096/4 = 1024`
- Calcolo dimensione index:

  $\log_2 1024 = 10$ bit
- Calcolo allineamento blocchi

  $\log_2 4 = 2$ bit
- Calcolo dimensione tag

  `32 - 10 - 2 = 20 bit`

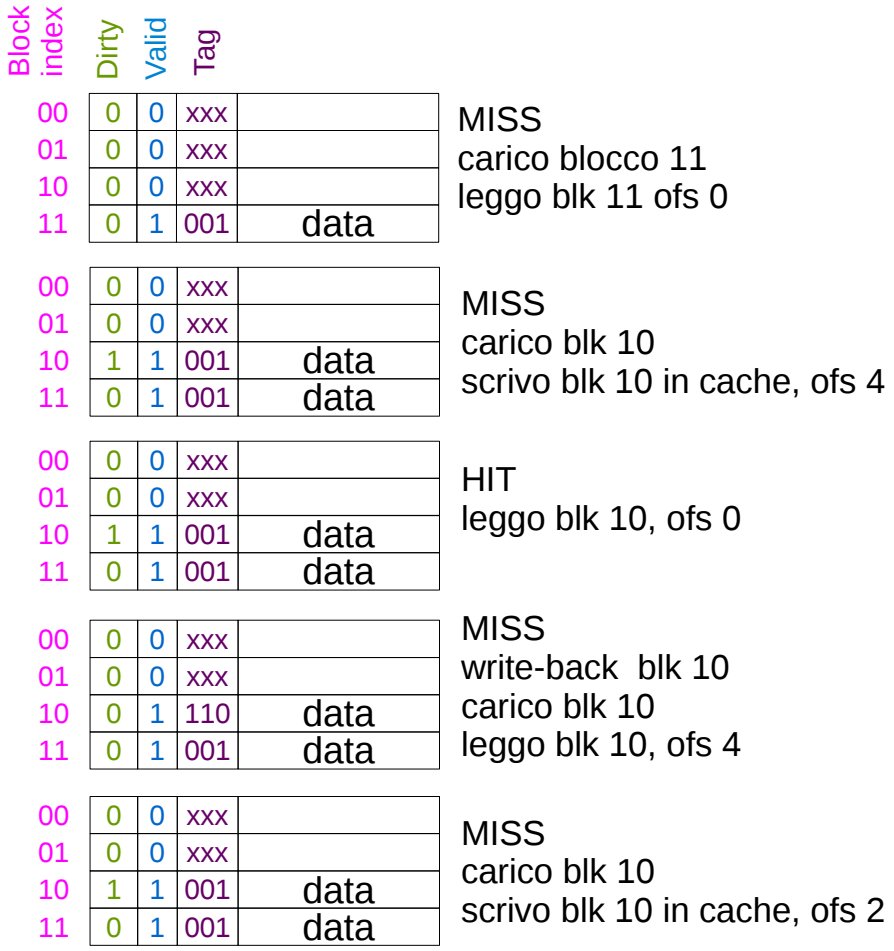


Esempio direct mapped

- Cache direct-mapped 32 32yte, 8 byte/blk, write-back, write-allocate
- Memoria principale 256 Byte

  `32/8=4 blocchi`
  `log2 4=2 bit index`
  `log2 8=3 bit offset`
  `8-3-2=3 bit tag`

- Operazioni:
  1) R 0x38 = 0b 00111000
  2) W 0x34 = 0b 00110100
  3) R 0x30 = 0b 00110000
  4) R 0xD4 = 0b 11010100
  5) W 0x32 = 0b 00110010



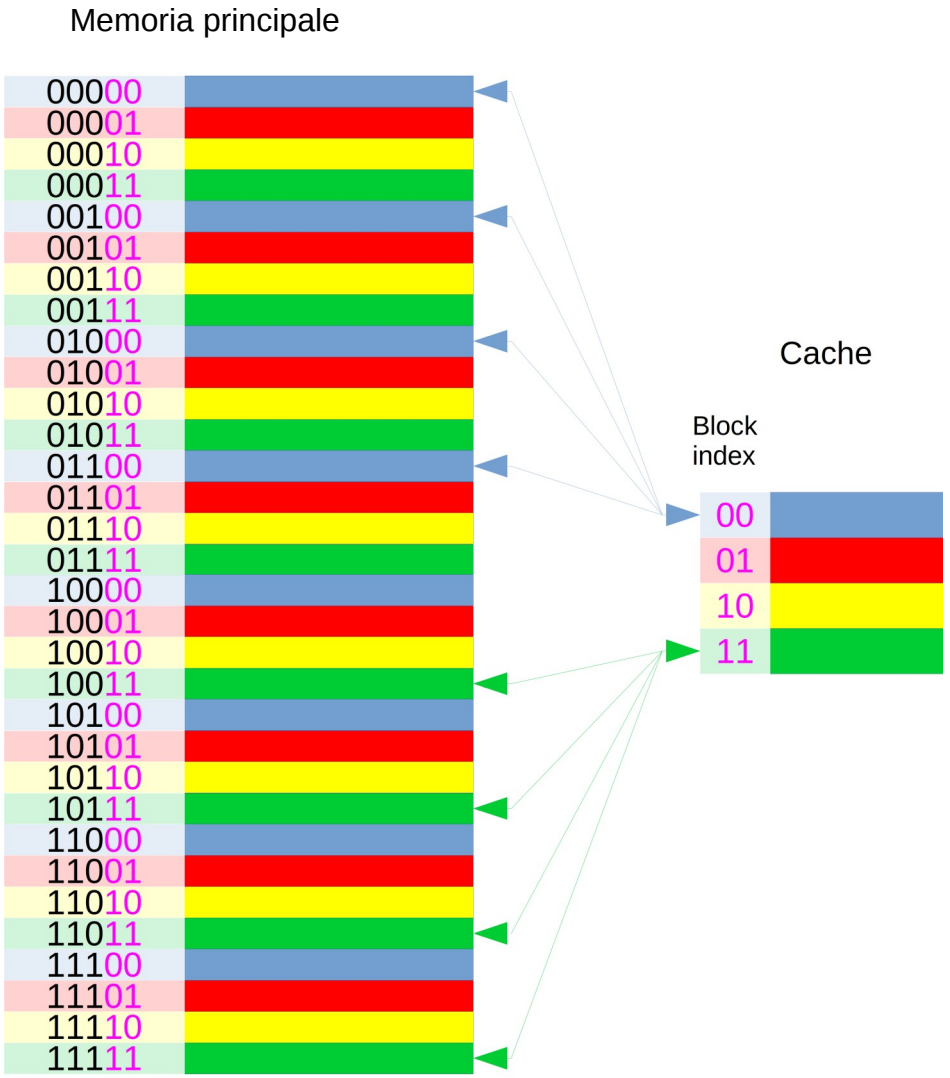
Hit rate = 1/5 Miss rate = 4/5

Considerazioni direct mapping

- Vantaggi:
  - Semplice da implementare
  - Relativamente poco costoso
- Svantaggi:
  - Ogni indirizzo di memoria corrisponde ad un solo blocco di cache.
  - Prestazioni degradano in caso di conflitto, se ho accessi frequenti a due indirizzi mappati sullo stesso blocco

  Nell'esempio precedente, se un programma lavora solo a byte allineati a 32 bit deve ricaricare in cache il dato ogni volta
  - Ogni volta accesso a memoria principale, cache completamente inefficiente! Es:

```
# $sp = 0b00000 (blocco blu)
lb   $t0, 0($sp)   # accedo 0b00000
lb   $t1, 4($sp)   # accedo 0b00100
lb   $t2, 8($sp)   # accedo 0b01000
lb   $t3, 12($sp)  # accedo 0b01100
```



Cache associative

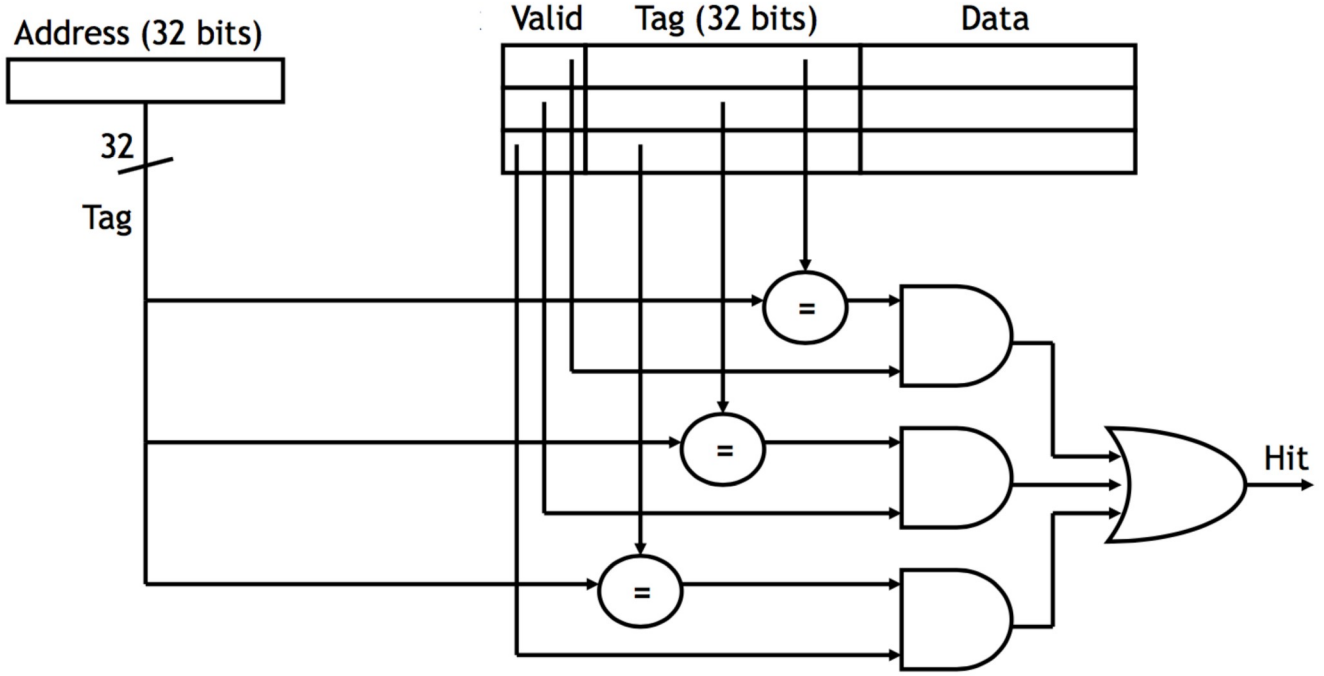
- Cache **fully-associative**
  - Posso associare un qualunque indirizzo a un qualunque blocco
  - Ricerca del blocco corrispondente ad un certo indirizzo (cache hit) complessa e costosa, richiede un comparatore per blocco
- Cache **n-way associative**
  - Compromesso tra direct-mapping e fully-associative
  - Ad ogni indirizzo può corrispondere uno tra *n* blocchi
  - Riduco conflitti mantenendo bassa complessità di ricerca
- Si possono implementare **politiche di sostituzione** più efficienti

Cache fully-associative

- Quando un dato viene caricato dalla memoria può essere messo in qualsiasi blocco non utilizzato della cache
  - in questo modo non ci sarà mai conflitto tra due o più indirizzi di memoria che fanno riferimento allo stesso singolo blocco della cache
- Politica **LRU** (Least Recently Used)
  - Se tutti i blocchi sono già in uso possiamo eliminare il meno recente, supponendo che se non è stato utilizzato da un periodo di tempo non sarà riutilizzato a breve.
- Ricerca del blocco LRU molto costosa (in termini di hw), ogni volta la cache deve controllare *tutti* i blocchi!
  - Esistono algoritmi **pseudo-LRU** a complessità minore
- Alternativa: sostituzione casuale (politica **Random**)

Cache fully-associative

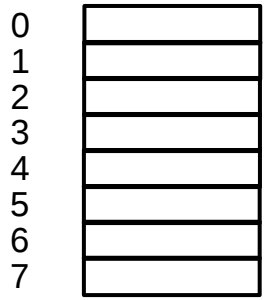
- Cache hit con politica di sostituzione LRU:
  - non essendoci un campo index, l'intero indirizzo deve essere usato come tag, aumentando la dimensione della cache
  - il dato può essere ovunque nella cache, quindi è necessario controllare il tag di ogni blocco (costoso in termini di hw)
    - es. 32 bit indirizzi, cache 64 KB, 1 byte/blocco → 65536 blocchi → 65536 comparatori a 32 bit!



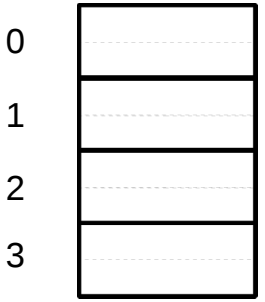
Cache n-way associative

- Set-associativity
  - I blocchi della cache sono raggruppati in **insiemi** (set)
  - Comportamento misto tra direct-mapping e fully-associative:
    - Ogni indirizzo può essere associato ad un solo set
    - Qualunque blocco all'interno del set può contenere il dato relativo.
- Cache n-way associative → Ogni set ha n blocchi

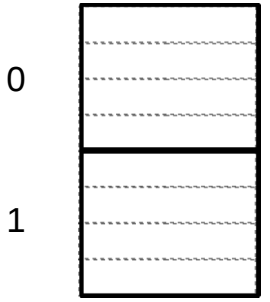
1-way set associative
8 sets, 1 blocco/set
DIRECT-MAPPED



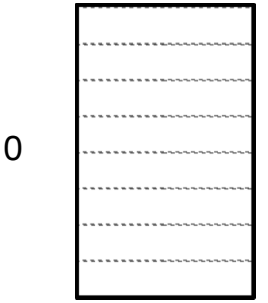
2-way set associative
4 sets, 2 blocchi/set



4-way set associative
2 sets, 4 blocchi/set



8-way set associative
1 sets, 8 blocchi/set
FULLY-ASSOCIATIVE



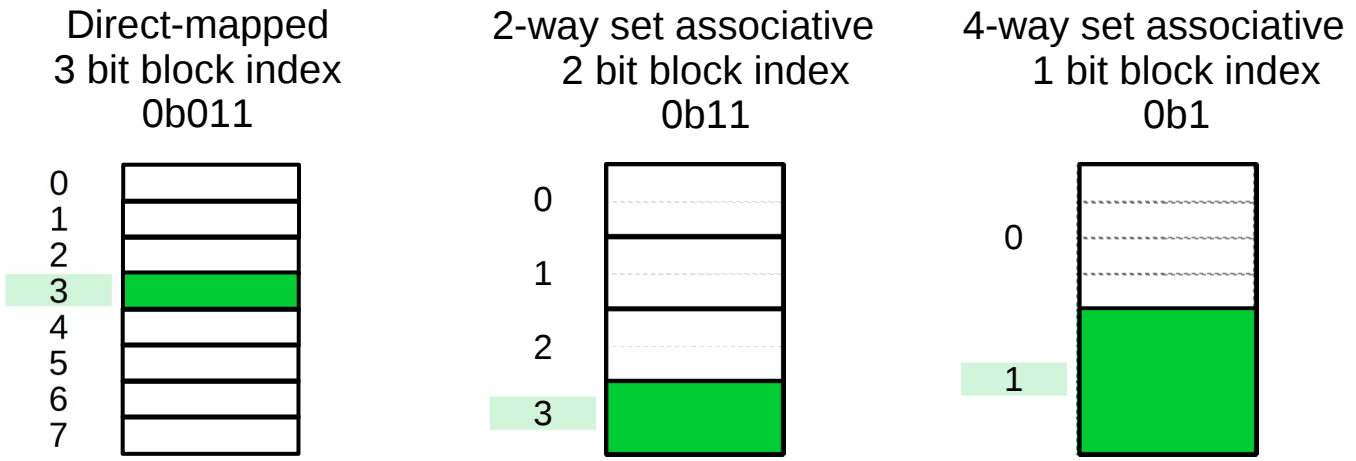
Cache n-way associative

- Trovare il set corrispondente ad un certo indirizzo è analogo a trovare il blocco nella cache direct-mapped
  - Identifico il numero del set direttamente con alcuni bit dell'indirizzo
- Es. cache a 8 blocchi, 16 byte/blocco

  Trovare il set corrispondente all'indirizzo

`0x1437 = 0b 000101000 0110111` ← Offset nel blocco di 16 byte

Tag — Block/Set index



Esempio cache n-way associative

- Cache da 32 KB:
  - Indirizzi a 32 bit
  - 32 byte/blocco
  - 16-way associative

- Calcolo numero blocchi

  `32K/32 = 32768/32 = 1024`

- Calcolo dimensione index:

  $\log_2 1024/16 = \log_2 64 = 6$ bit

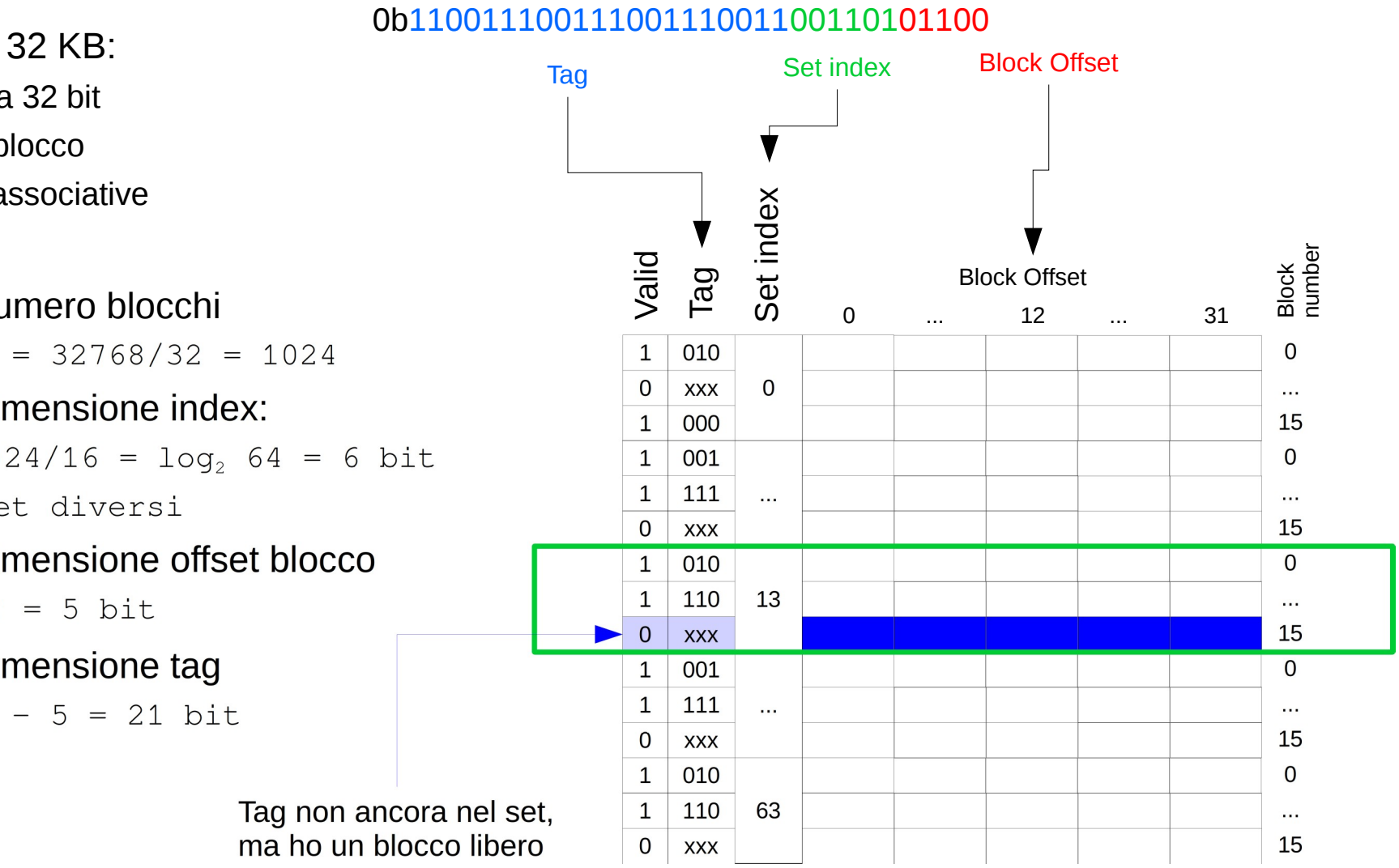
  → 64 set diversi

- Calcolo dimensione offset blocco

  $\log_2 32 = 5$ bit

- Calcolo dimensione tag

  `32 - 6 - 5 = 21 bit`

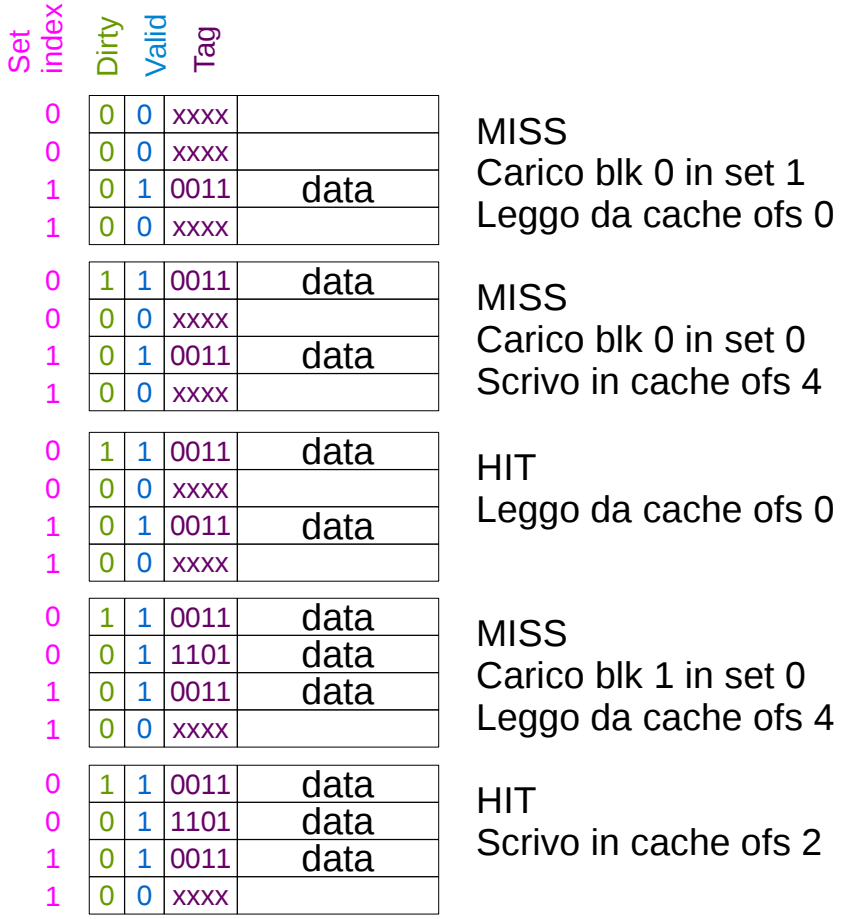


Esempio n-way associative

- Cache 32 byte 2-way associative, 8 byte/blk, write-back, write-allocate, politica random
- Memoria principale 256 Byte

  32/8 = 4 blocchi
  4/2 = 2 set
  $\log_2 2$ = 1 bit index
  $\log_2 8$ = 3 bit offset
  8−3−1 = 4 bit tag

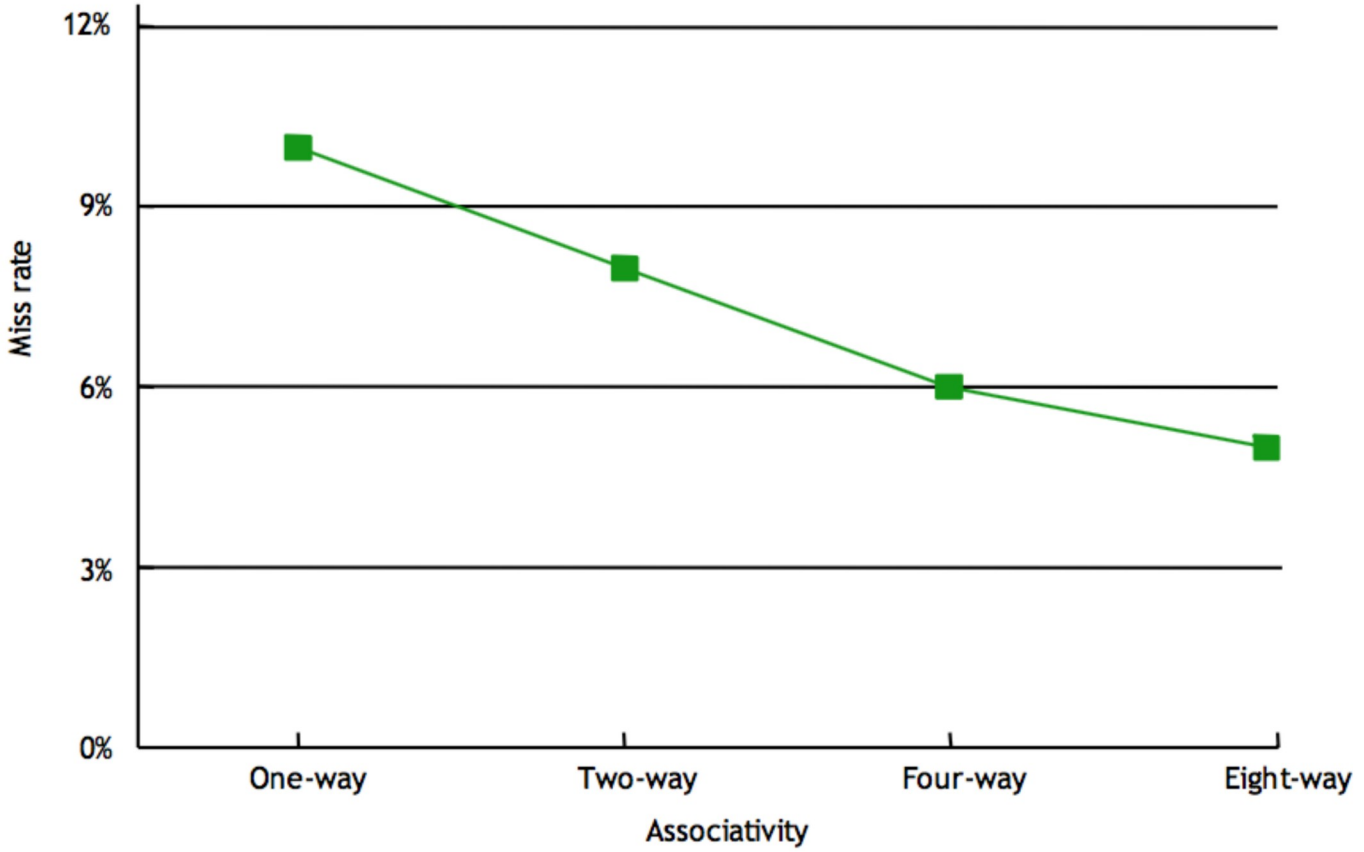
- Operazioni:
  1) R 0x38 = 0b 00111000
  2) W 0x34 = 0b 00110100
  3) R 0x30 = 0b 00110000
  4) R 0xD4 = 0b 11010100
  5) W 0x32 = 0b 00110010



Hit rate = 2/5 Miss rate = 3/5

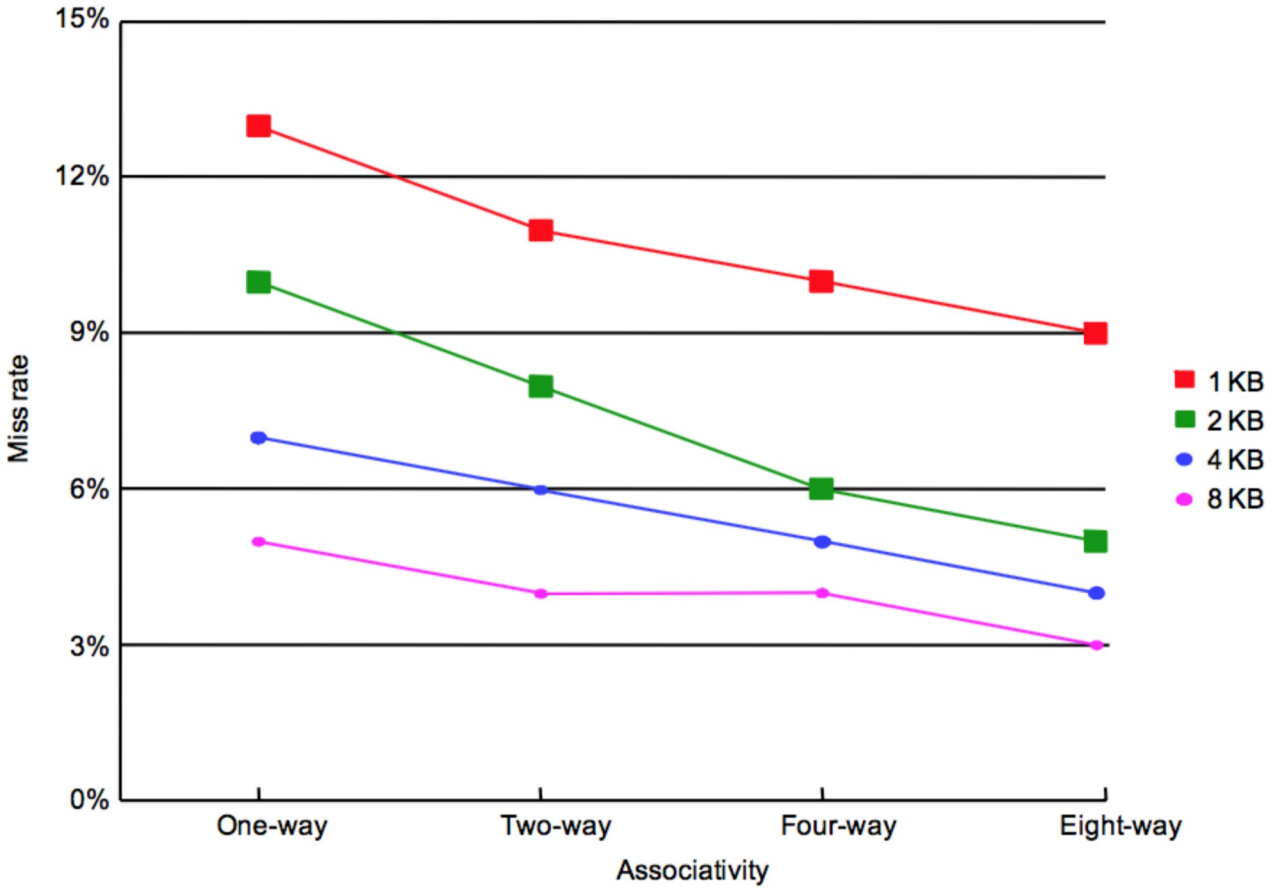
Associatività e miss-rate

- Un’alta associatività significa maggior costo hardware.
- Ma comporta anche un calo delle miss rate:
  - ogni set ha più blocchi → meno possibilità di conflitti



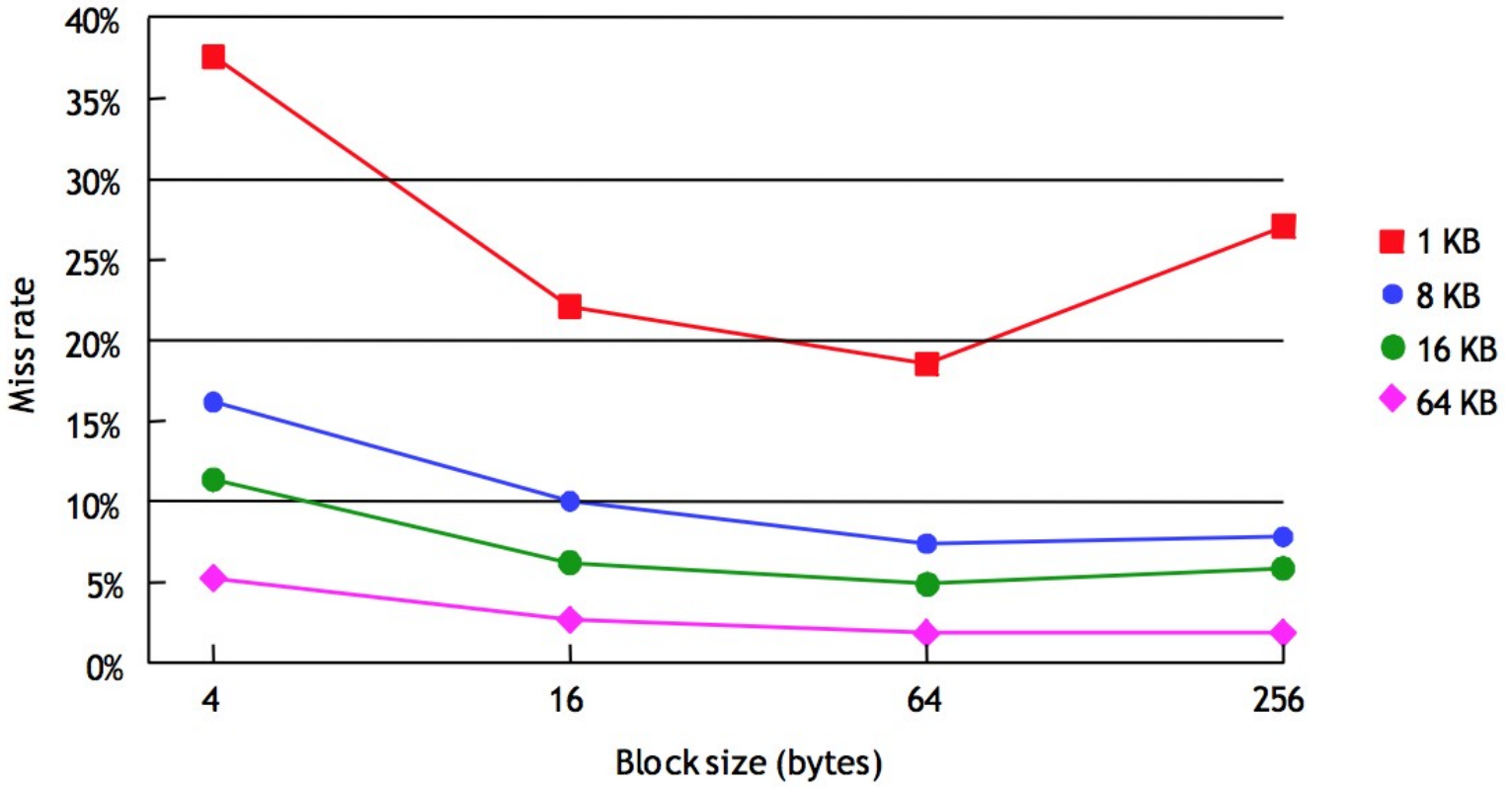
Dimensione cache e miss-rate

- Anche la dimensione della cache ha impatto sulle performance:
  - più grande è la cache, più bassa è la possibilità di avere conflitti

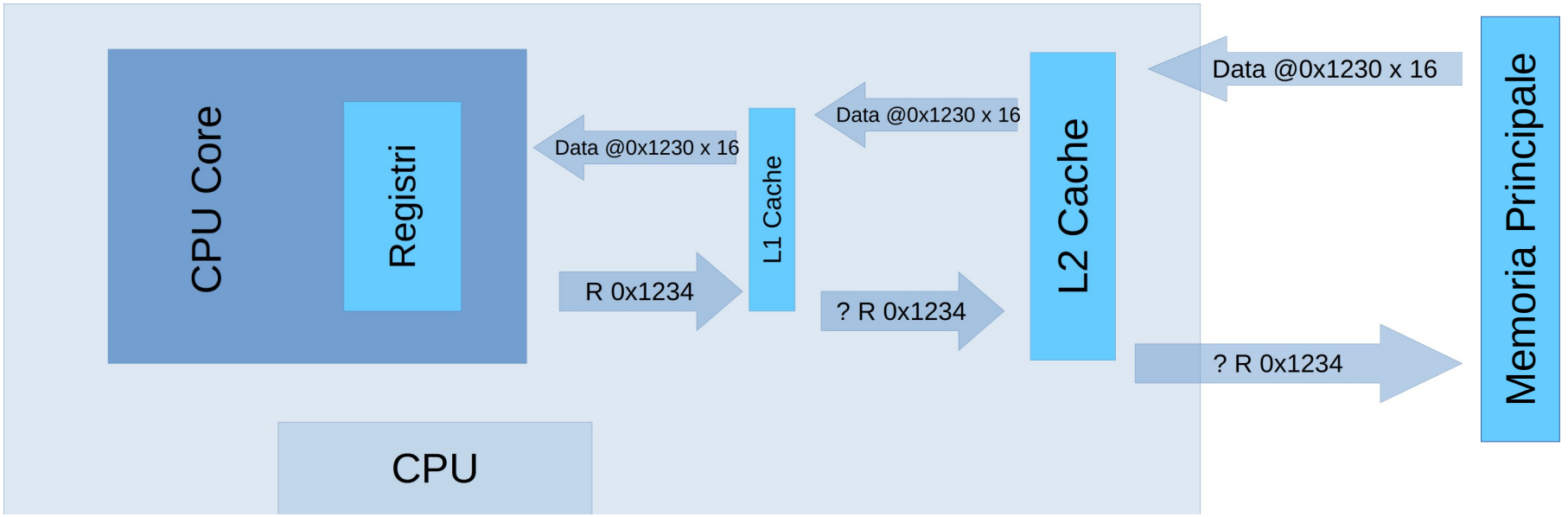


Dimensione blocco e miss-rate

- Anche la dimensione della blocco ha impatto sulle performance:
  - più grande è il blocco, più viene sfruttata la località spaziale

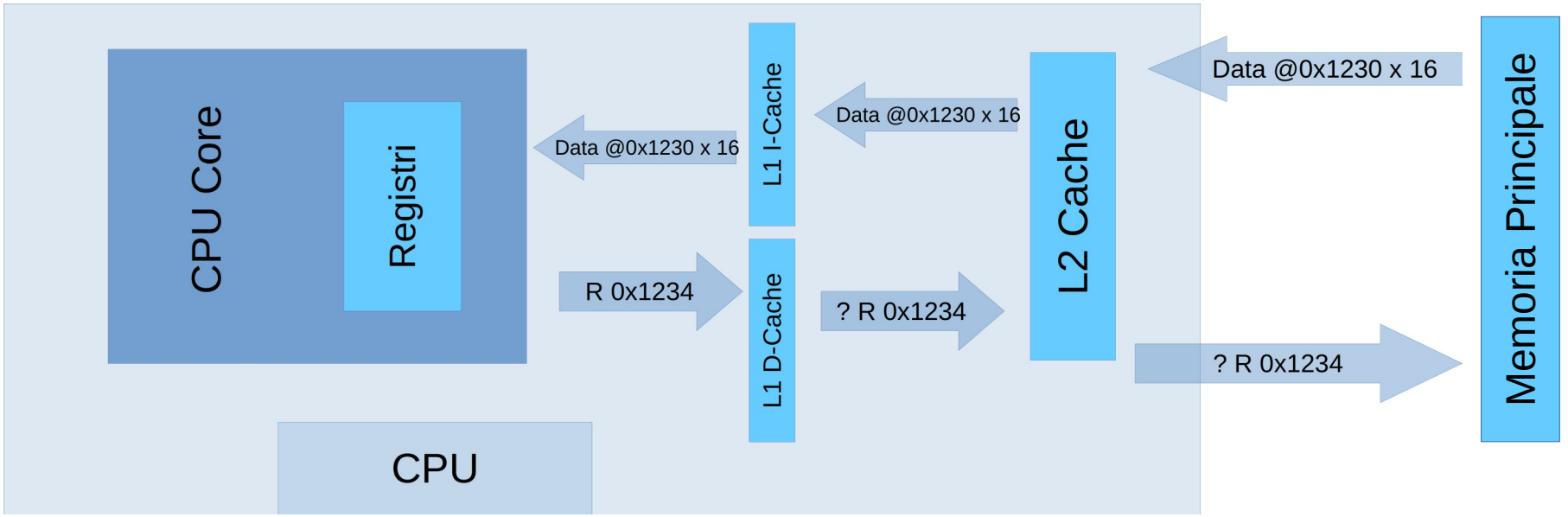


Cache Multilivello



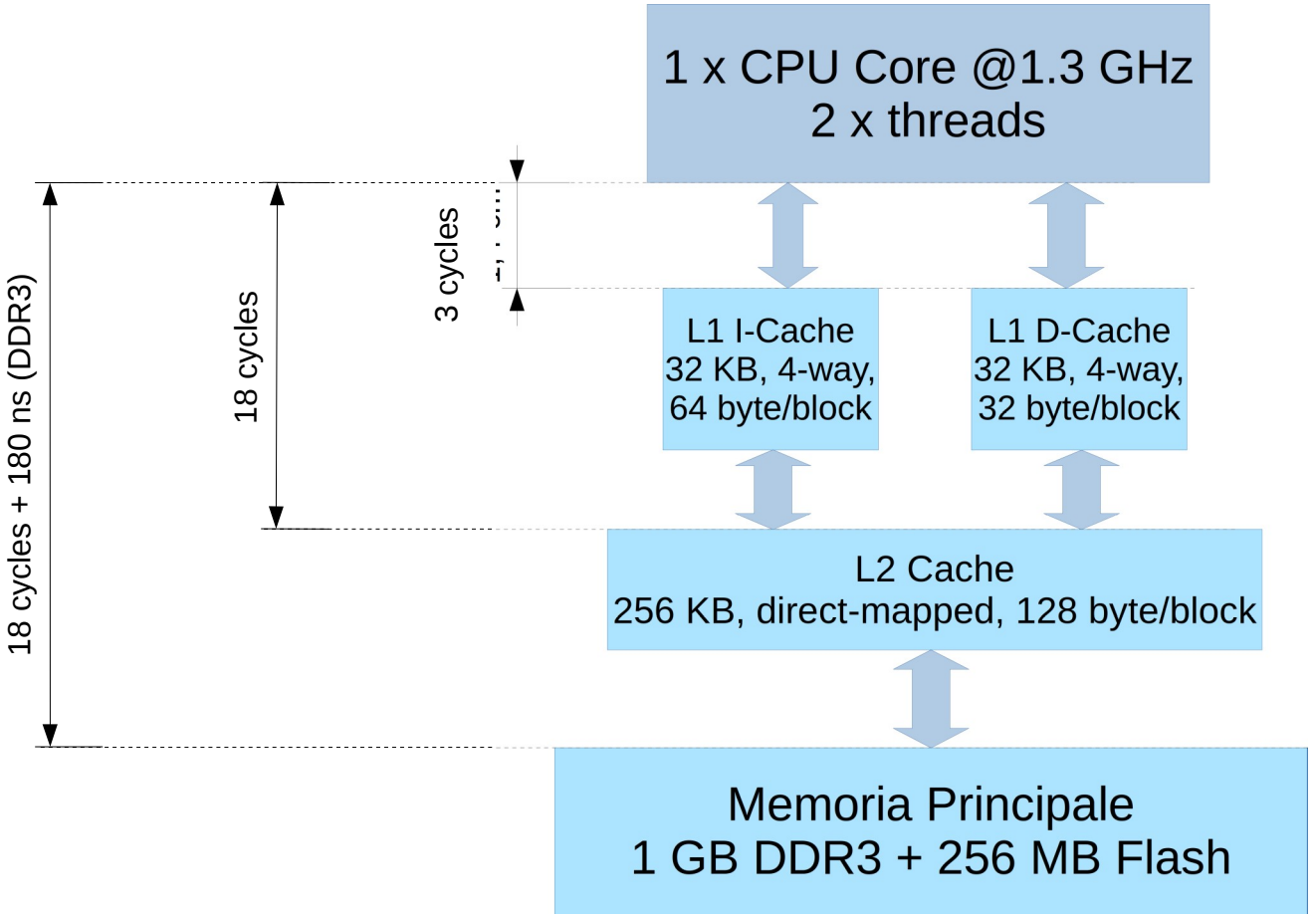
- Diminuire AMAT (tempo medio di accesso memoria):
  - Diminuire miss_rate → cache più grande
  - Diminuire miss_penalty → cache a più livelli
- A livelli più vicini alla memoria principale la cache avrà dimensioni maggiori e prestazioni inferiori
- A livelli più vicini alla CPU la cache avrà dimensioni minori e prestazioni maggiori

Cache dati e istruzioni

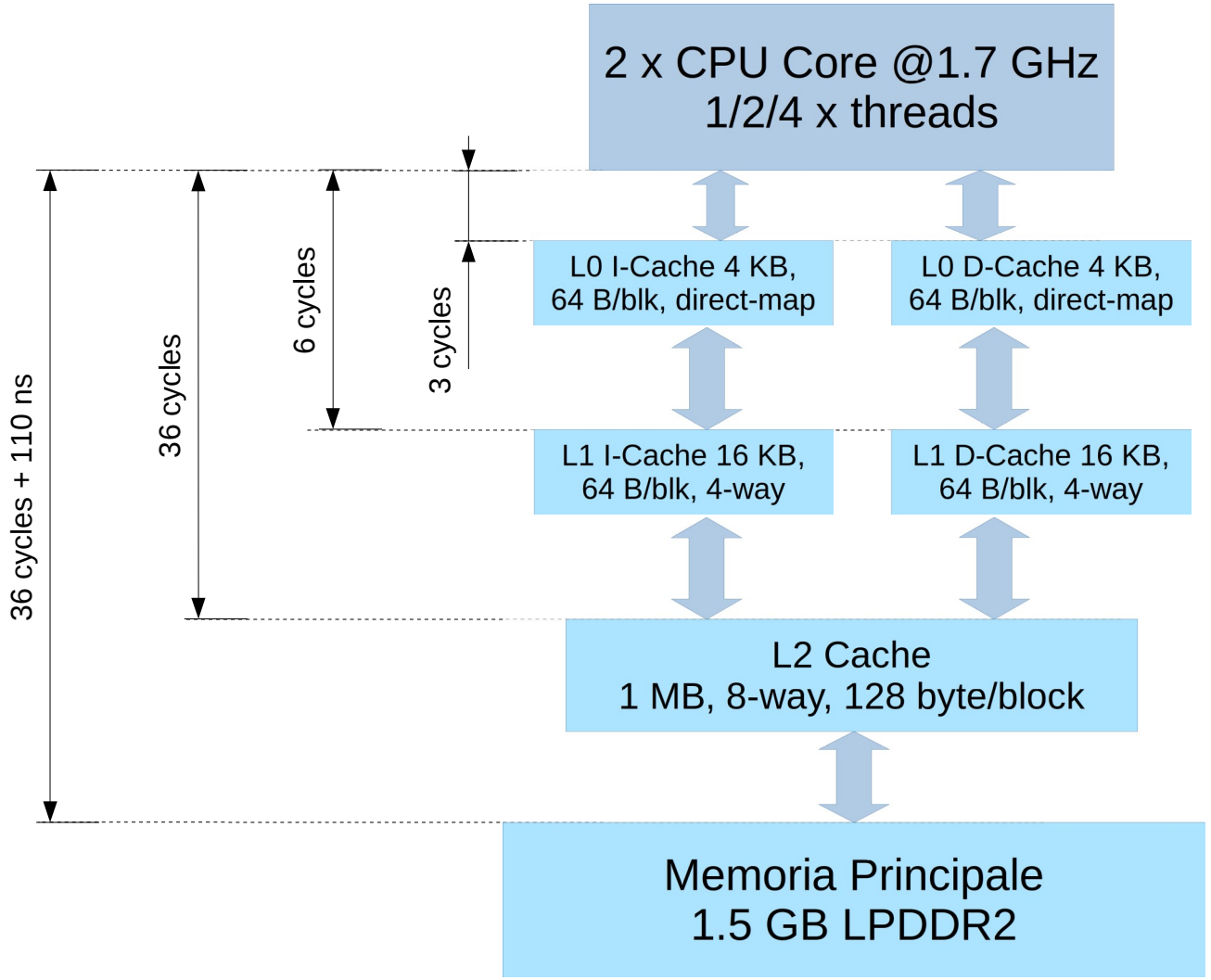


- È conveniente separare la cache dati dalla cache istruzioni
  - Architettura Harvard, posso eseguire 2 accessi in memoria senza conflitti (avendo hit in L1)
  - Vantaggioso soprattutto con pipeline (evito criticità strutturali)
  - Costoso: duplico logica di controllo
- Nel caso di cache multi-livello solitamente si separa solo il livello più vicino alla CPU (L1), mantenendo condivisi ulteriori livelli
  - L1 → hit time basso, L2 → hit rate alto

Broadcom BCM7356 (MIPS32)



Qualcomm Krait 300 (ARM32)
(Snapdragon 400)



Intel Broadwell (i7-6900K, x86-64)