Gerarchia di Memoria e Cache

Andrea Bartolini – a.bartolini@unibo.it

Derivate da slides Onur Mutlu, Tulio Salmon Cinotti

Nozioni di base e funzionamento della cache

Cache

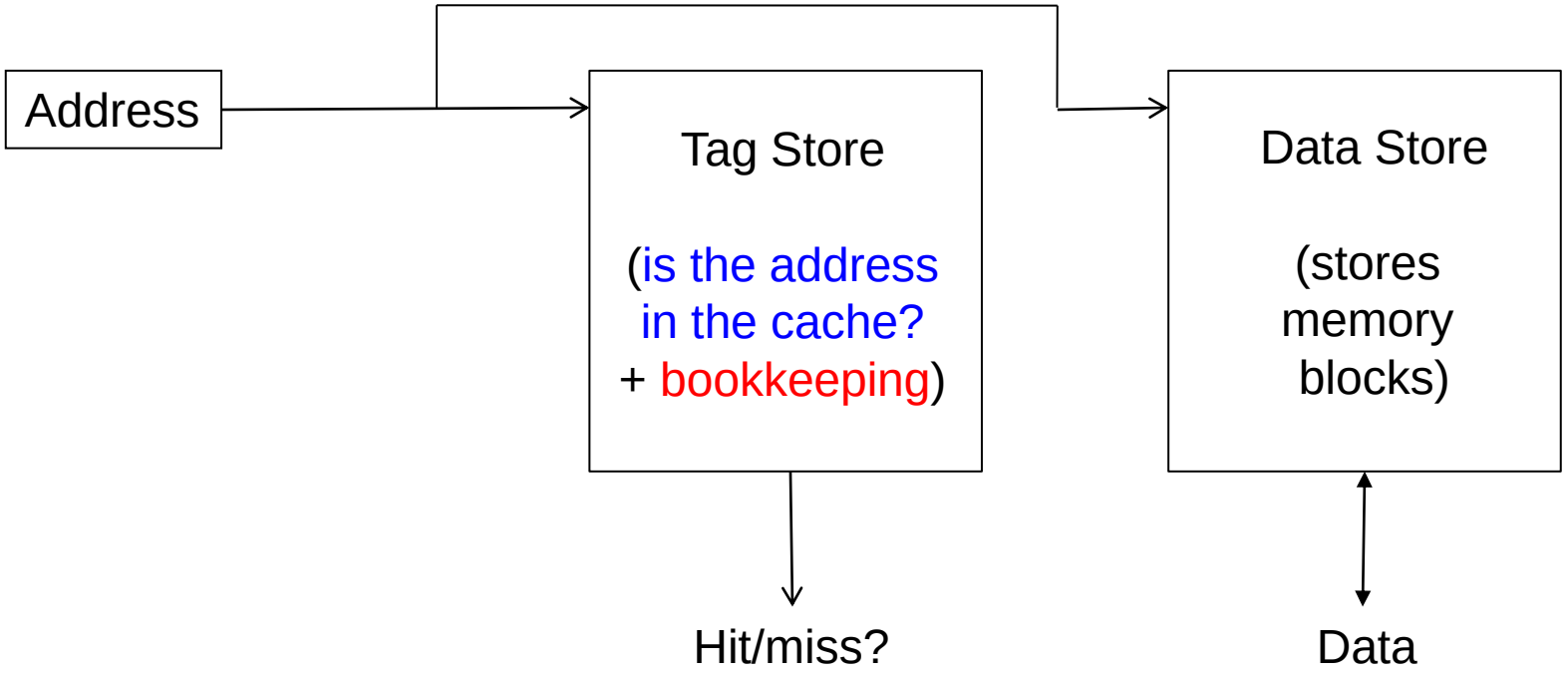
- Genericamente, qualsiasi struttura che «ricorda» i risultati utilizzati di frequente per evitare di ripetere le operazioni a lunga latenza necessarie per riprodurre i risultati da zero, ad esempio una cache Web

- 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

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?

Cache Abstraction and Metrics



- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)
  = (hit-rate * hit-latency) + (miss-rate * miss-latency)

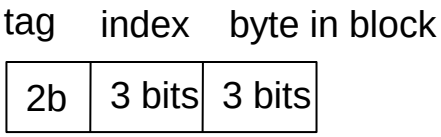
* miss-latency = include anche la hit-latency.

A Basic Hardware Cache Design

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

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 inidicizzare all'interno dei campi di tag e data



Es: 8-bit address

- Accesso alla Cache :

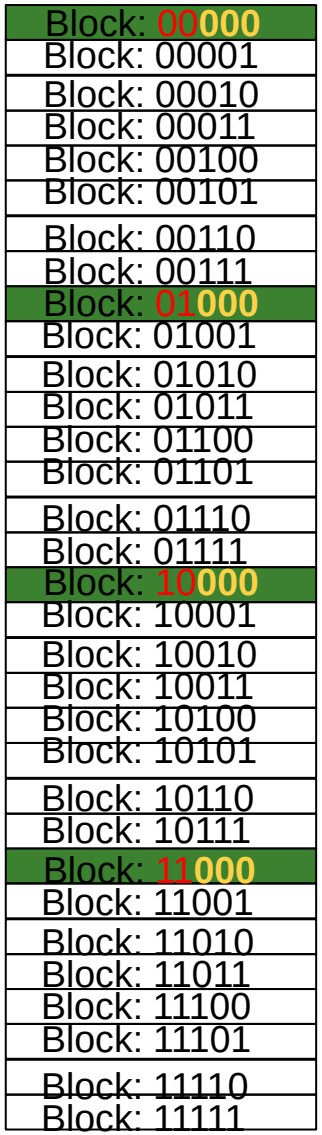
  1) Accedo ai campi di tag e dati con bit di indice dell'indirizzo

  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.

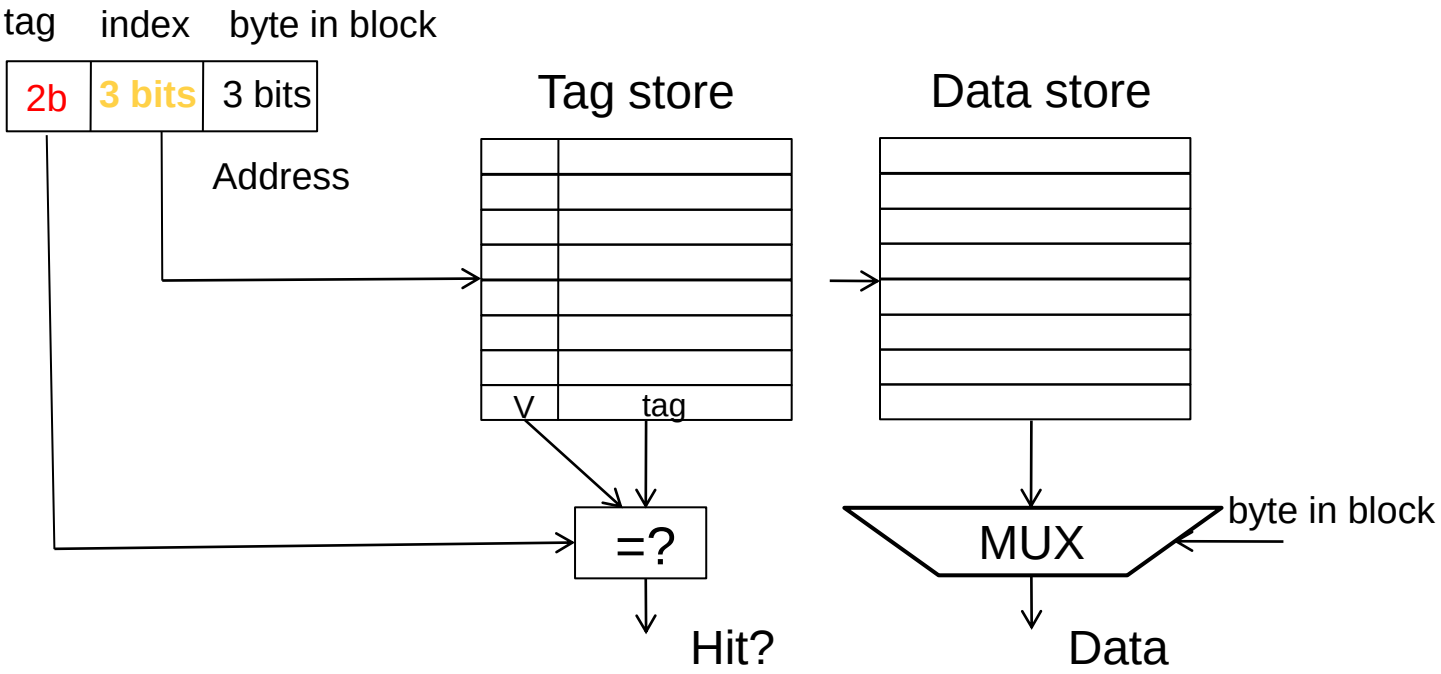
- Se un blocco si trova nella cache (cache hit), il tag memorizzato deve essere valido e corrispondere al tag del blocco per il quale ho fatto l'accesso.

Direct-Mapped Cache: Placement and Access



Main memory

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



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

Direct-Mapped Caches

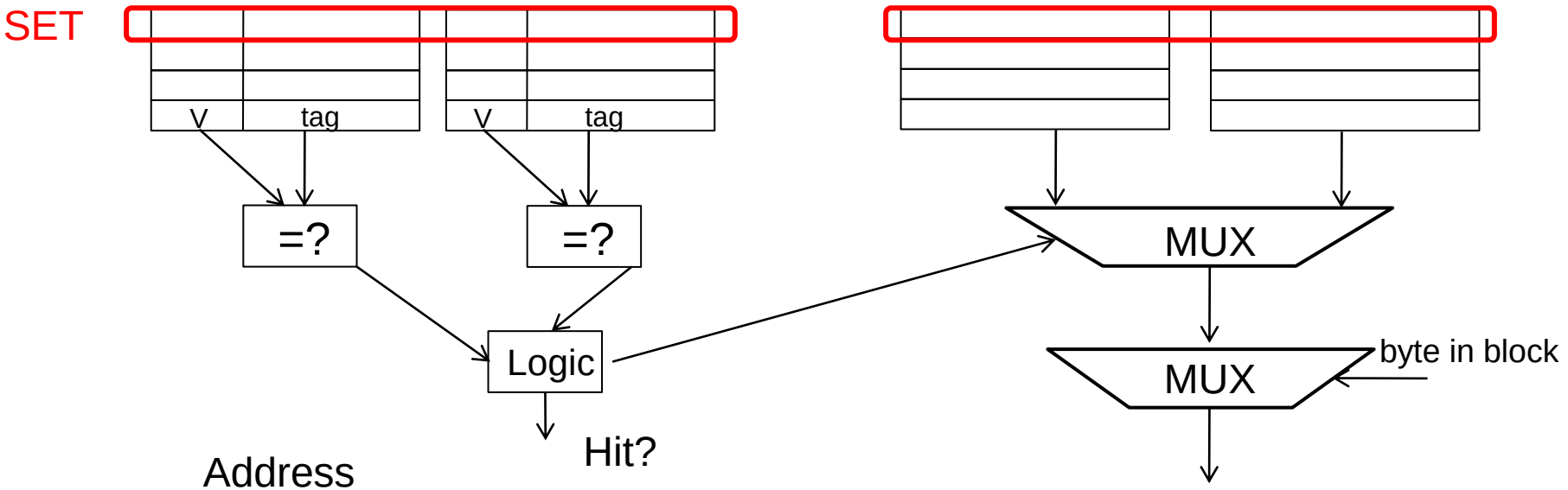
- Direct-mapped cache: Due blocchi in memoria mappati allo stesso indice della cache (con gli stessi bit di indice) non possono essere presenti contemporaneamente nella cache
- One index → one entry

- 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
- Tag store Data store



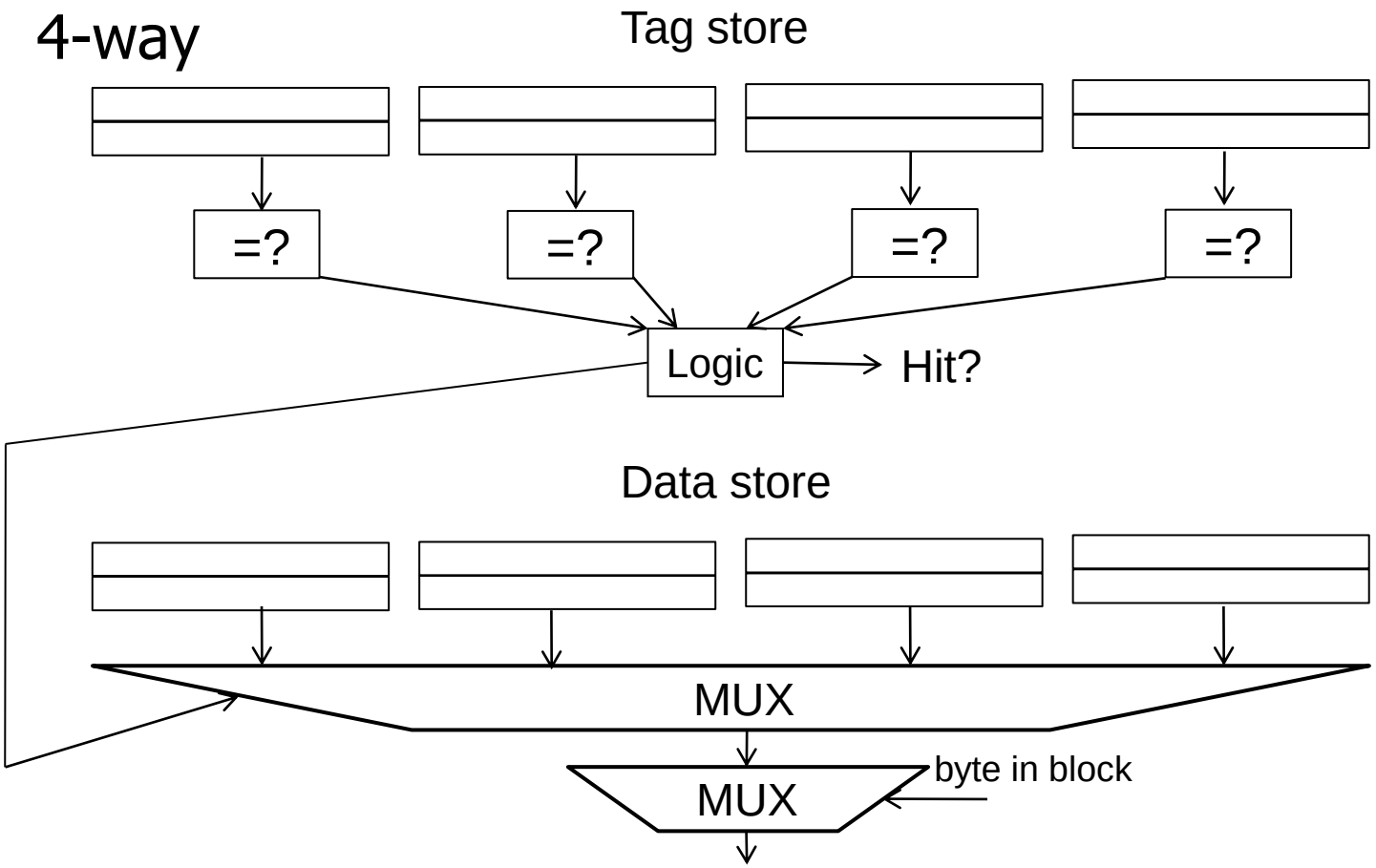
Address

tag	index	byte in block
3b	2 bits	3 bits

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

Higher Associativity

- 4-way

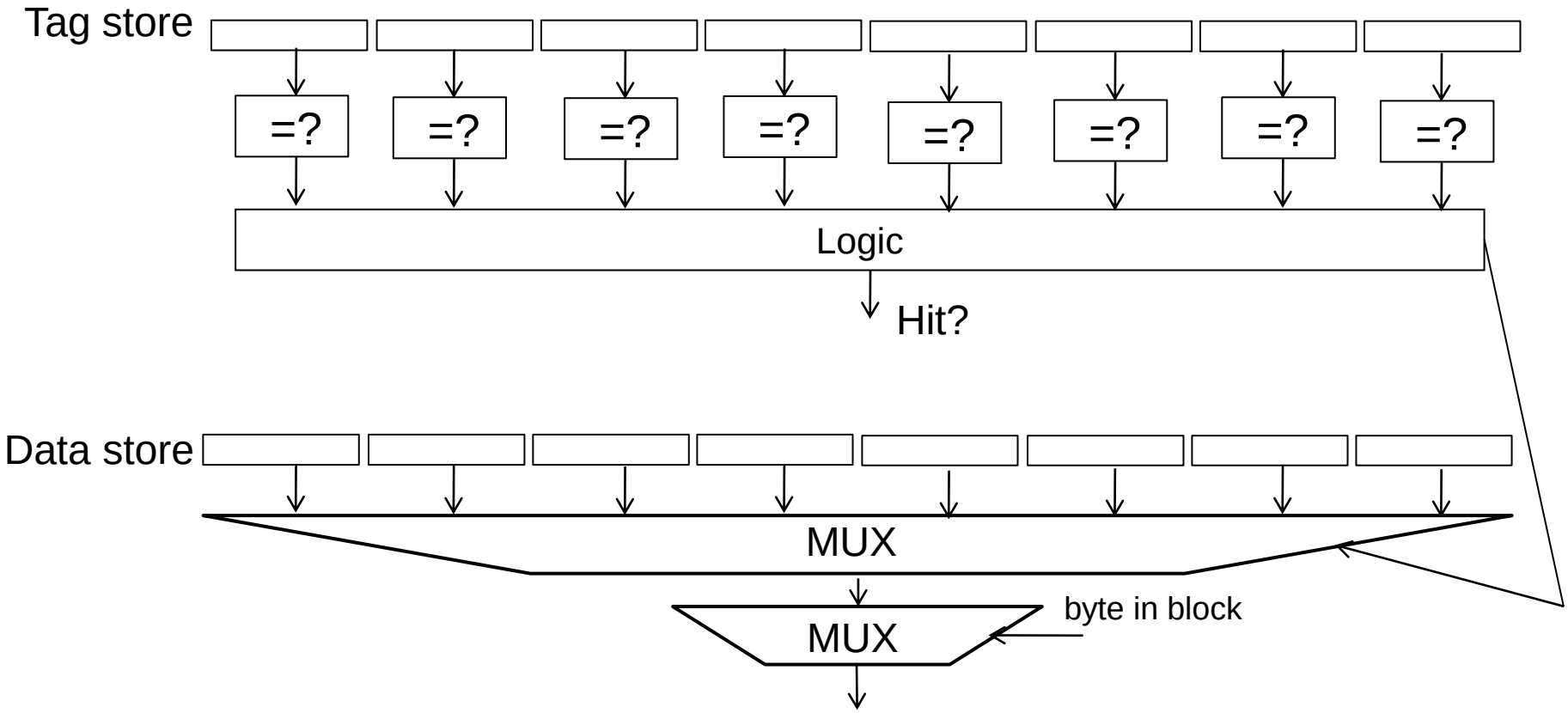


+ Minor probabilità di miss dovuta a conflitti

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

Full Associativity

- Fully associative cache
  - Un blocco può essere posizionato in qualunque posizione della cache



Associativity (and Tradeoffs)

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

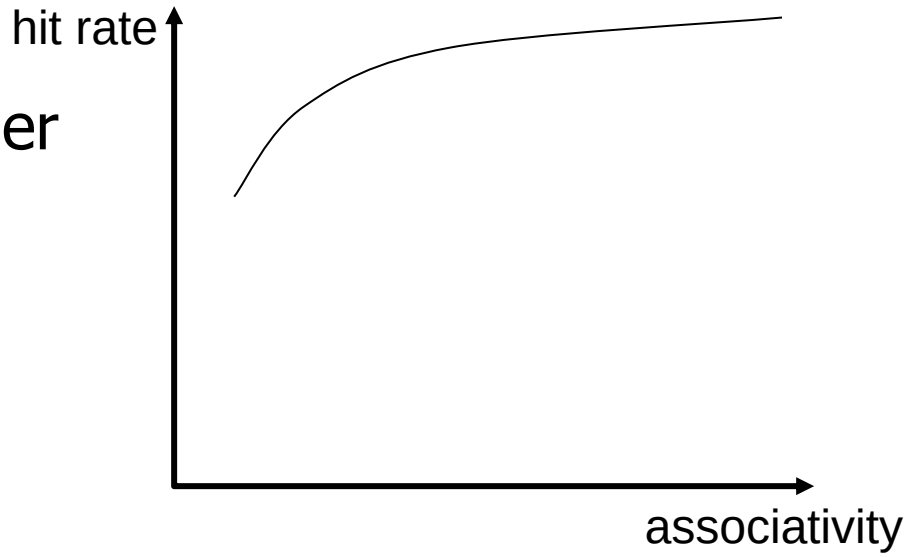
- Higher associativity

  ++ Higher hit rate

  -- Slower cache access time (hit latency and data access latency)

  -- More expensive hardware (more comparators)

- Diminishing returns from higher associativity



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

- 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
- Tag
- Replacement policy bits

- Dirty bit?
  - Write back vs. write through caches

Handling Writes (I)

- Quando scrivere i dati modificati nel livello successivo della cache?
  - Write through: Nel momento in cui avviene la scrittura
  - Write back: Quando il blocco viene rimosso
- Write-back

  + 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)

- Allochiamo 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

- Write Around - No-allocate

  \+ Risparmia spazio nella cache se la località delle scritture è bassa (potenzialmente miglior cache hit rate)

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:

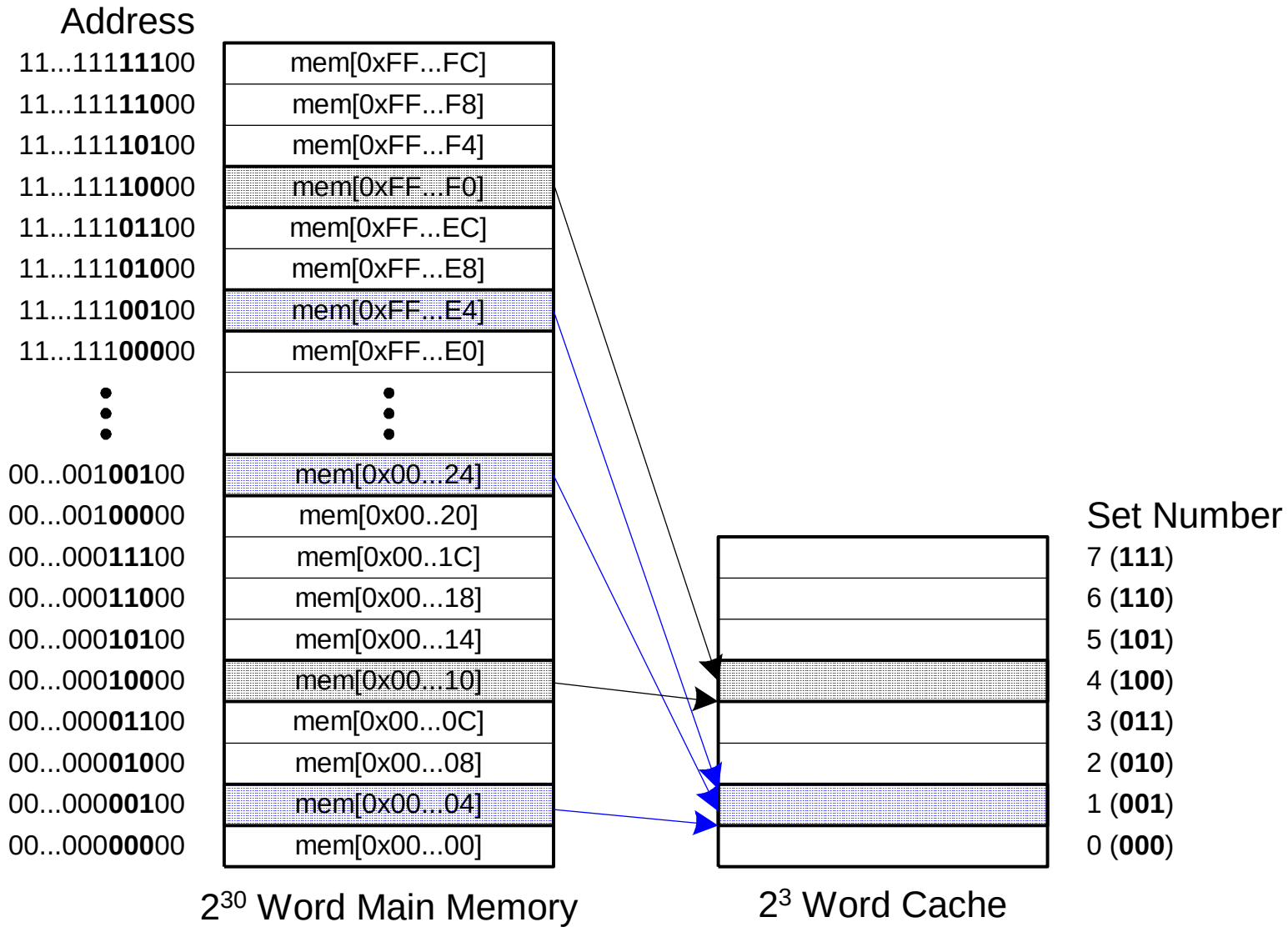
Cache Terminology

- Capacità (*C*):
  - Numero di bytes che la cache può immagazzinare
- Dimensione Blocco/Linea (*b*):
  - 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
- Numbero di sets (*S = B/M*):
  - ogni indirizzo di memoria viene mappato esattamente a un set nella cache

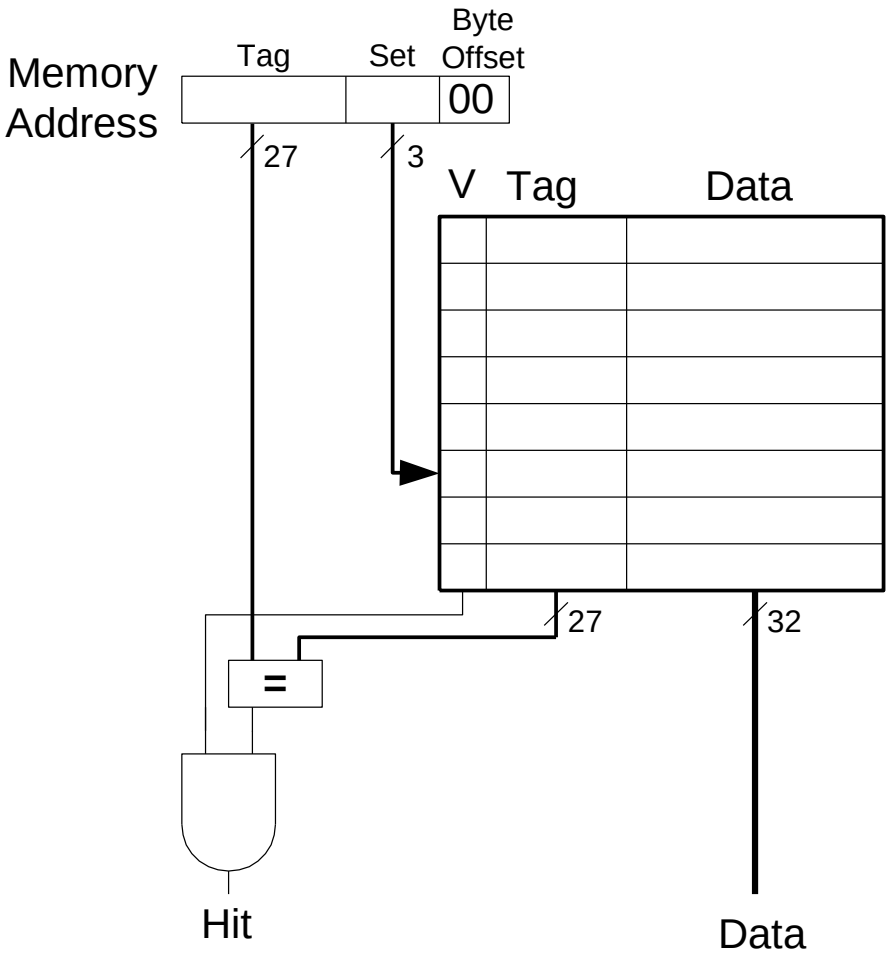
How is data found?

- Cache organizzata in *S* sets
- Ogni indirizzo si mappa esattamente ad un solo set
- 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)
  - Block size (*b* = 1 word)
  - So, number of blocks (*B* = 8)

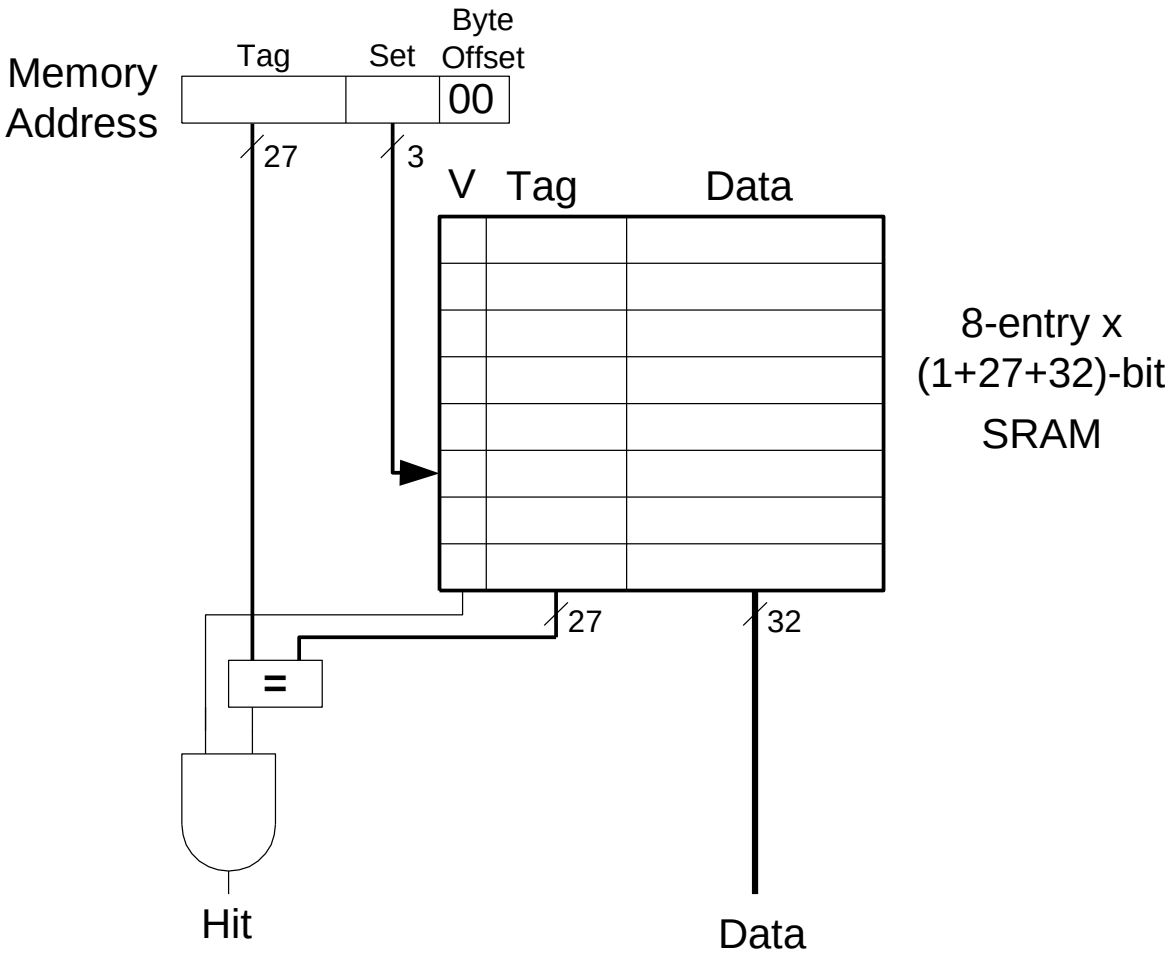
Direct Mapped Cache



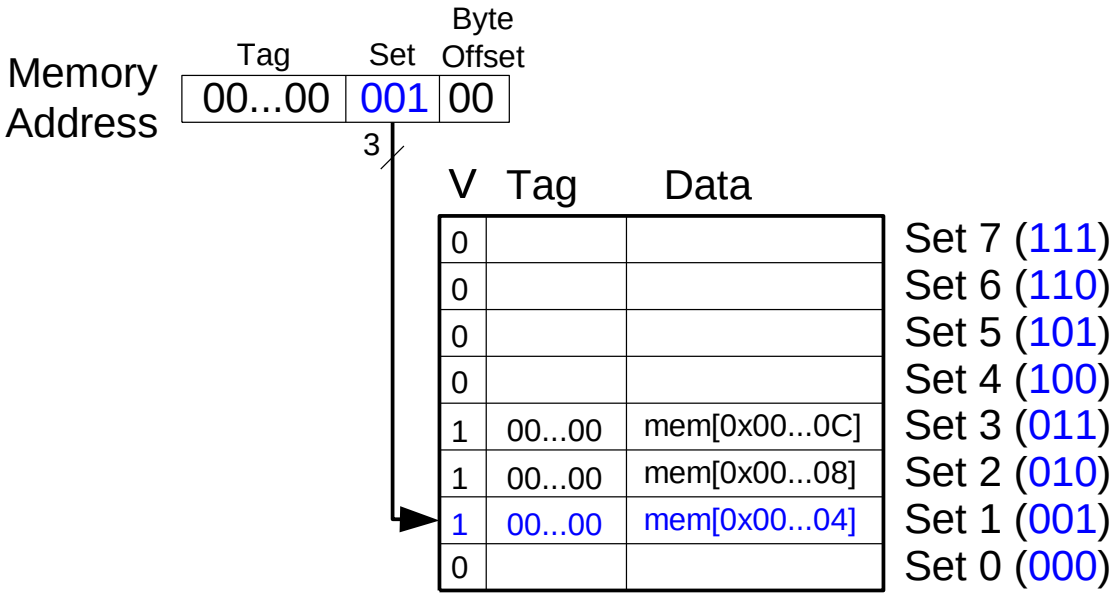
Direct Mapped Cache Hardware



Direct Mapped Cache Hardware



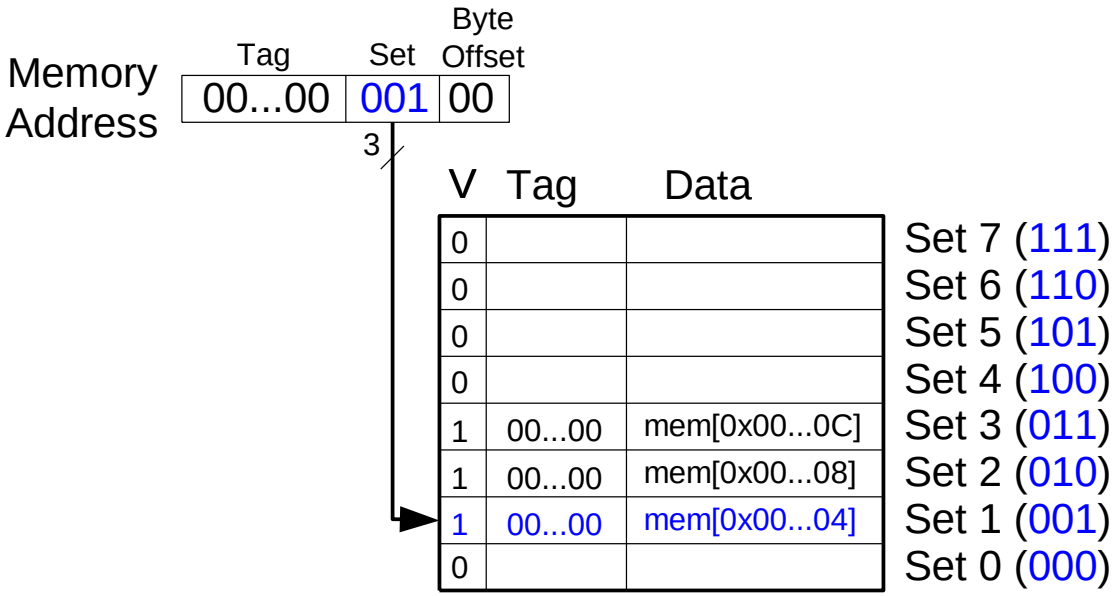
Direct Mapped Cache Performance




```
# 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:
```

Miss Rate = 3/12

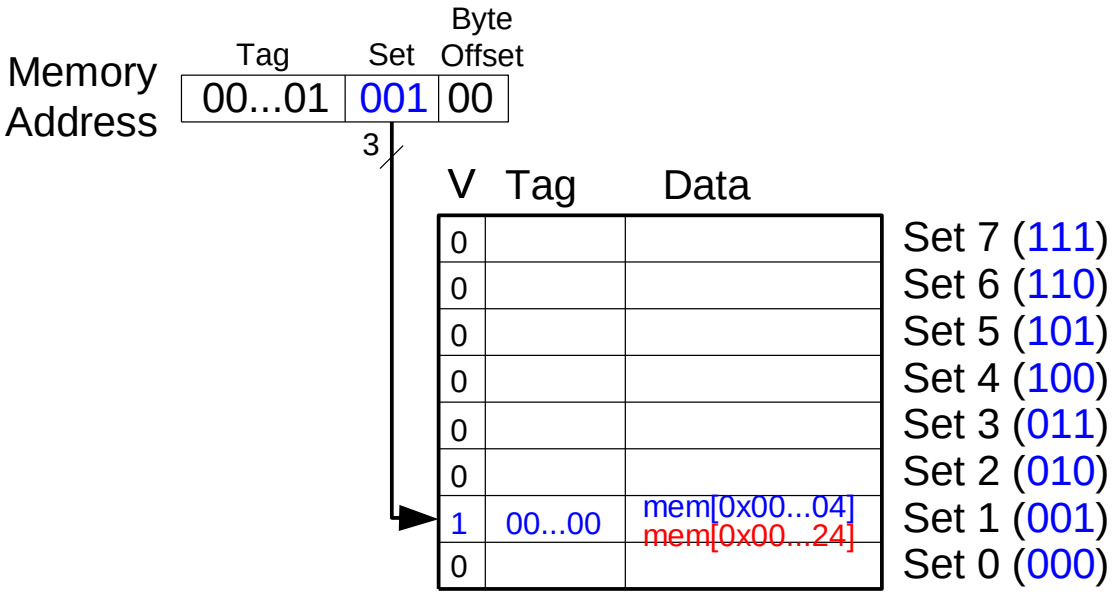
=

25%

Temporal Locality

Compulsory Misses

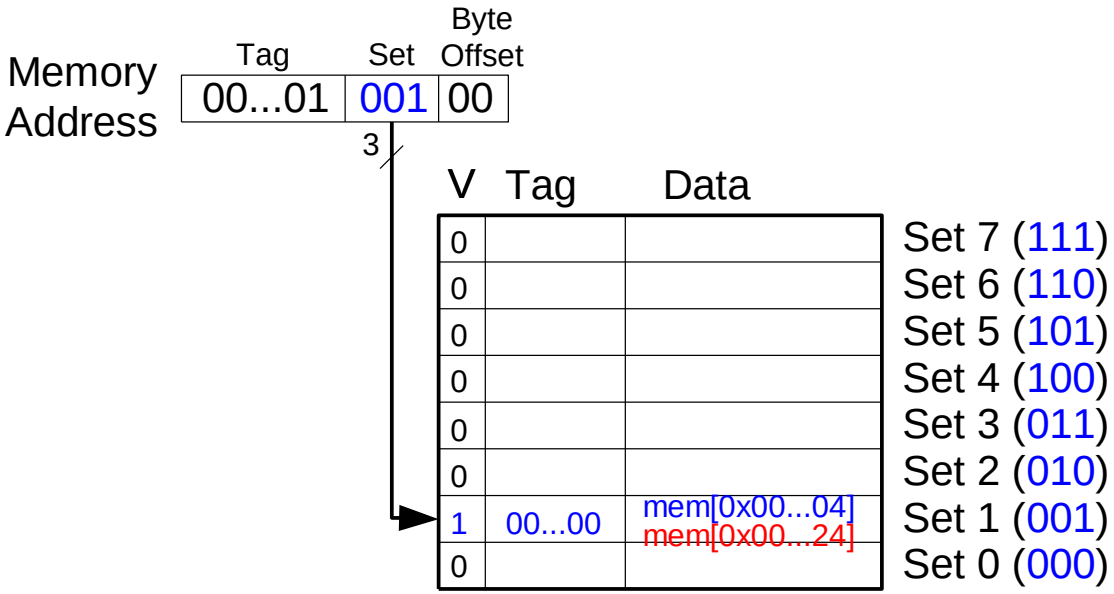
Direct Mapped Cache: Conflict




```
# 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 =

Direct Mapped Cache: Conflict

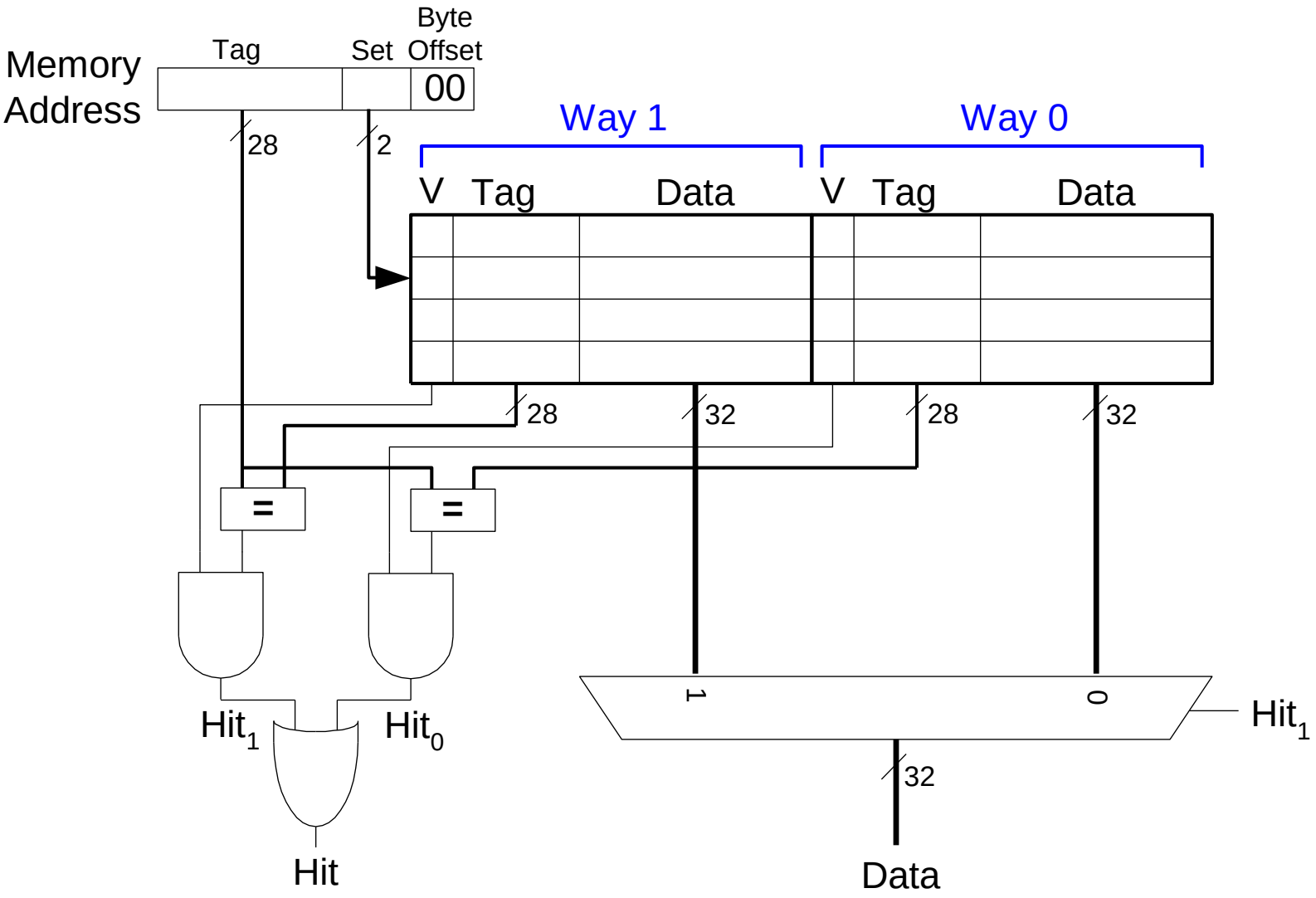



```
# RV assembly code
        addi  x1, x0, 4
loop:   beq   x0, x1, done
        addi  x1, x1, -1
        lb    x2, 0x4(x1)
        lb    x3, 0x24(x1)
        beq   x0, x0, loop
done:
```

Miss Rate = 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		
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]	Set 1
0			0			Set 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:
```

Miss Rate = 2/8

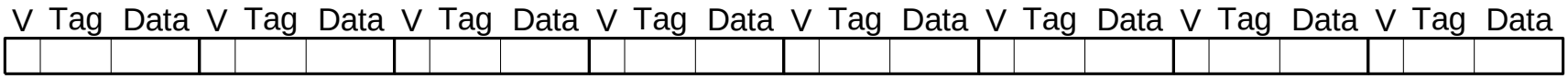
= 25%

Associativity reduces conflict misses

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

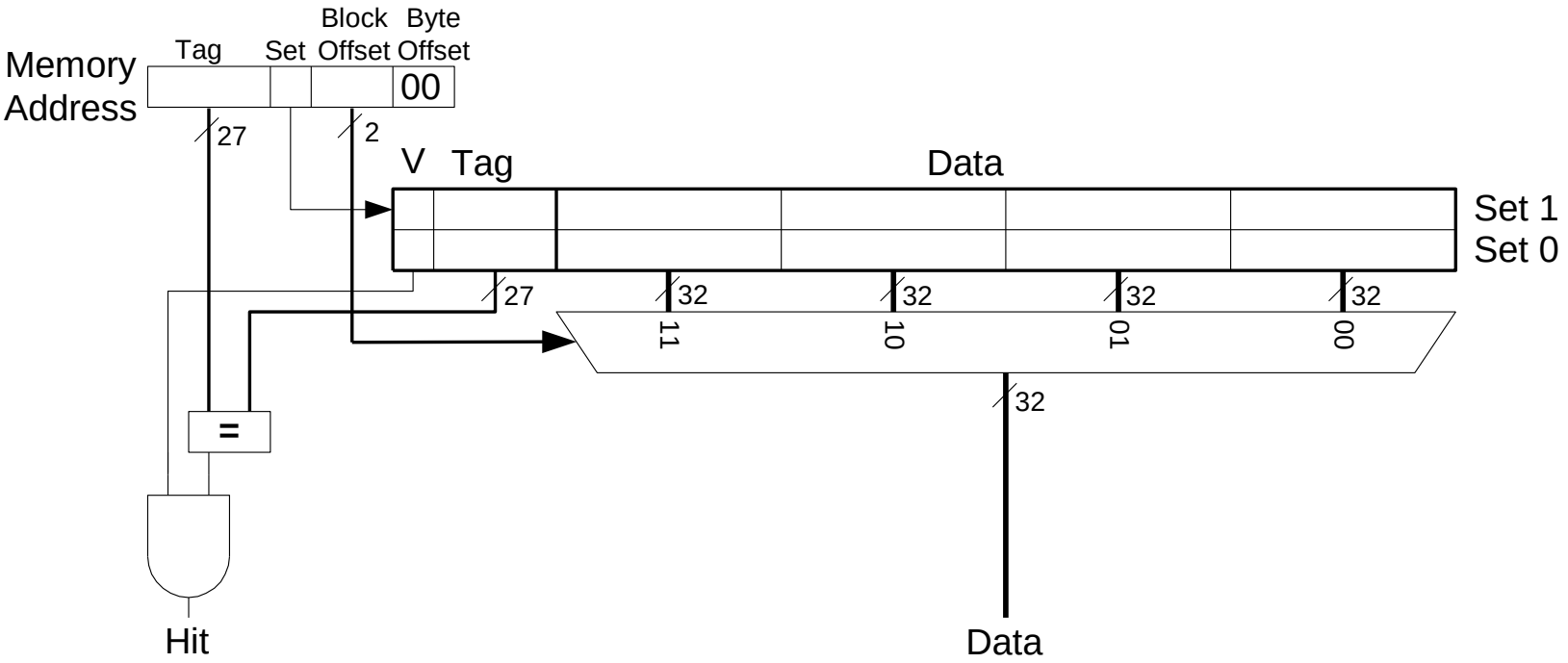
Fully Associative Cache

- No conflict misses
- Expensive to build



Spatial Locality?

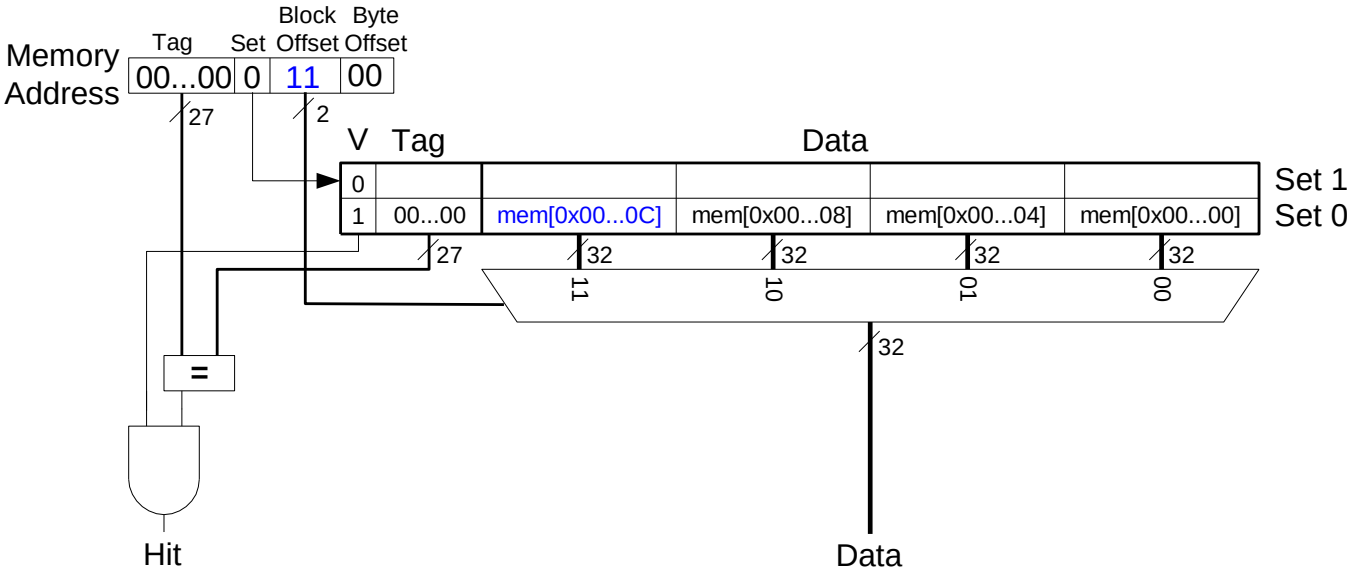
- Increase block size:
  - Block size, ***b* = 4** words
  - *C* = 8 words
  - Direct mapped (1 block per set)
  - Number of blocks, *B* = *C*/*b* = 8/4 = 2



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 =



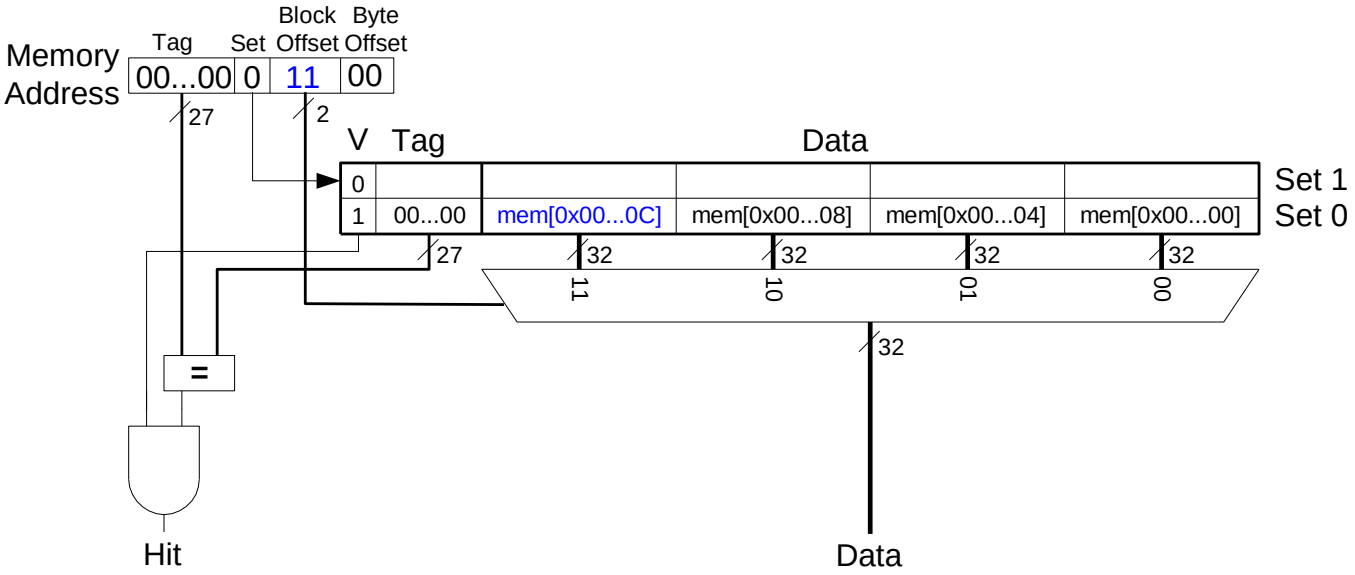
Direct Mapped Cache Performance

```
# 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:
```

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*

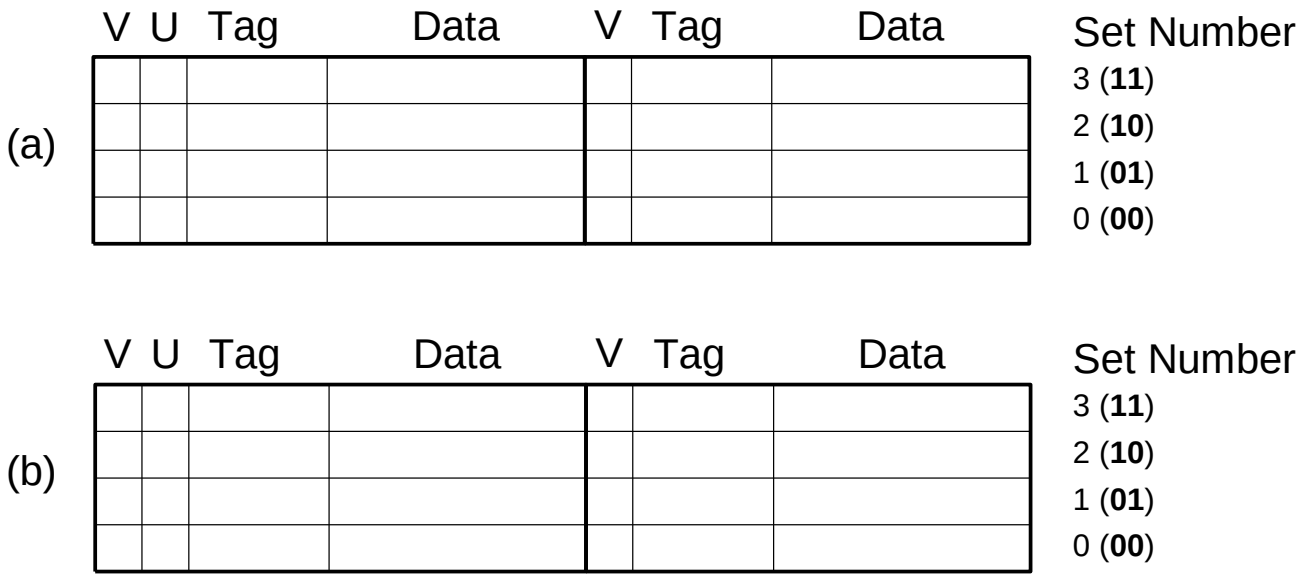
Organization	Number of Ways (N)	Number of Sets (S = B/N)
Direct Mapped	1	B
N-Way Set Associative	1 < N < B	B / N
Fully Associative	B	1

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.

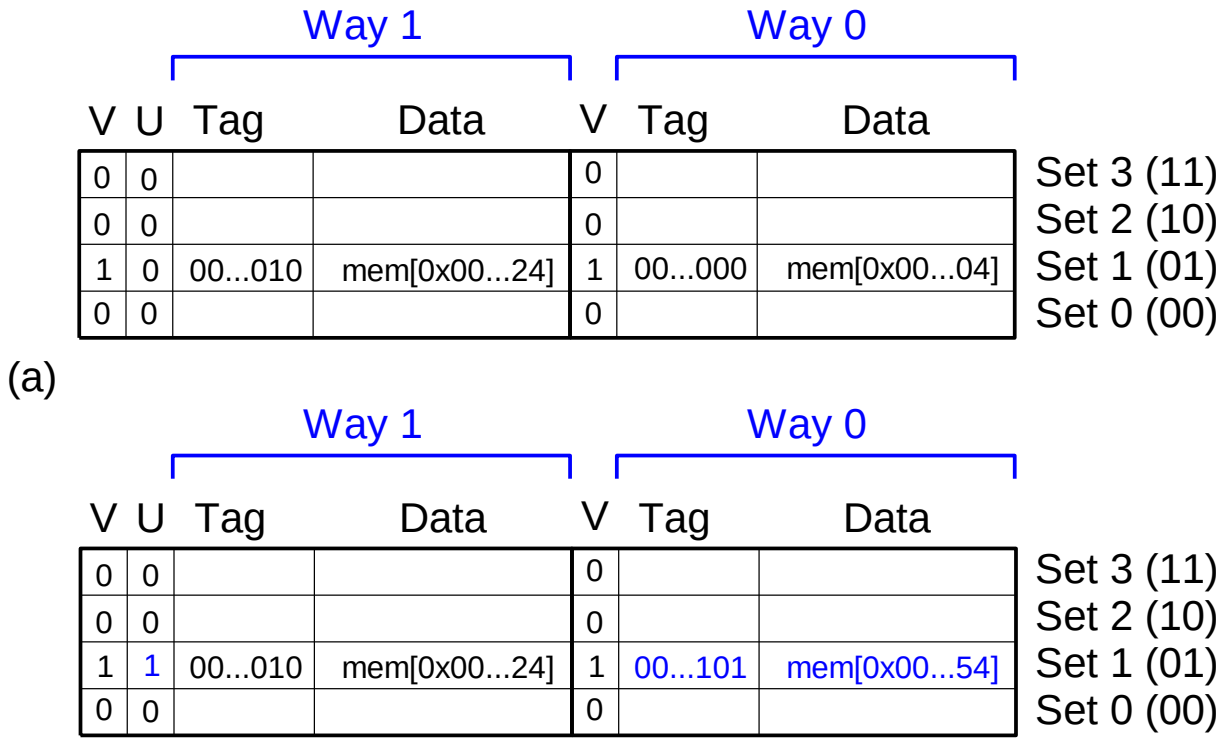
LRU Replacement

```
# R5 assembly

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



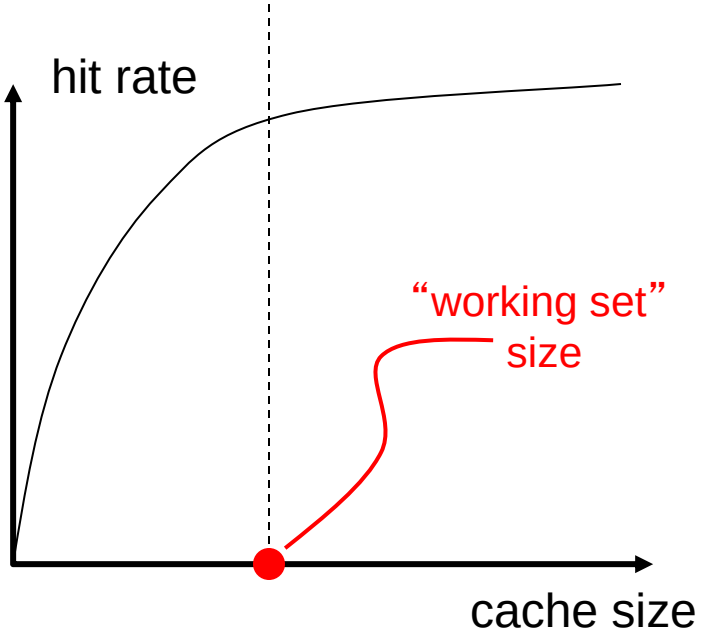
LRU Replacement

```
# R5 assembly

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



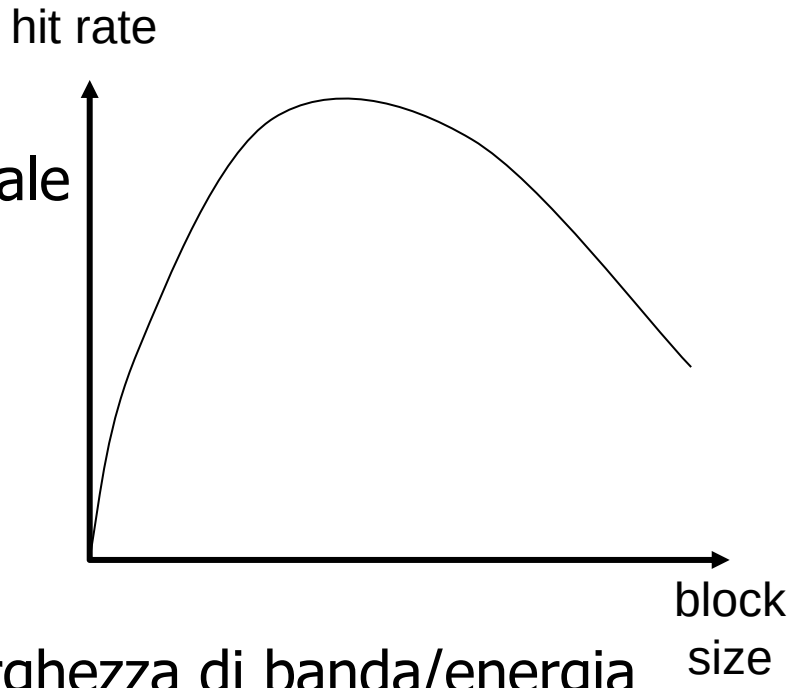
Cache Size

- 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

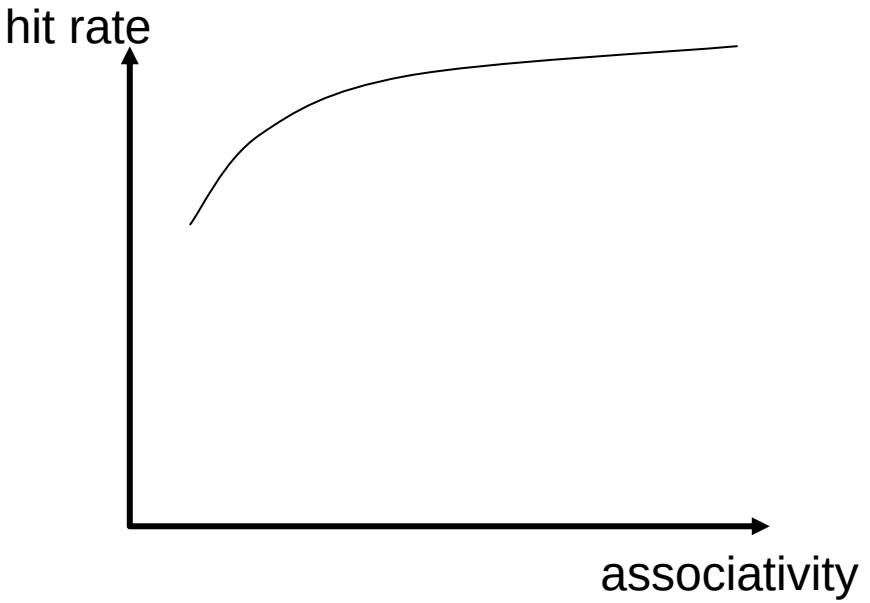
- Block size è la dimensione del dato associato a un address tag



- 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)?
- 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)
  - 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

- Miss di Capacità (Capacity miss)
  - 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

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
    - 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: int x[5000][100] = {{..},{..},..,{..}};
  - 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: int x[5000][100] = {{..},{..},..,{..}};
  - 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];
```

Restructuring Data Access Patterns (II)

- Blocking
  - 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)

- Es: Matrix Multiplication
  - *int Y[N][N] = {{..},{..},..,{..}};*
  - *int Z[N][N] = {{..},{..},..,{..}};*
  - *int X[N][N] = {{..},{..},..,{..}};*

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;
        };
```

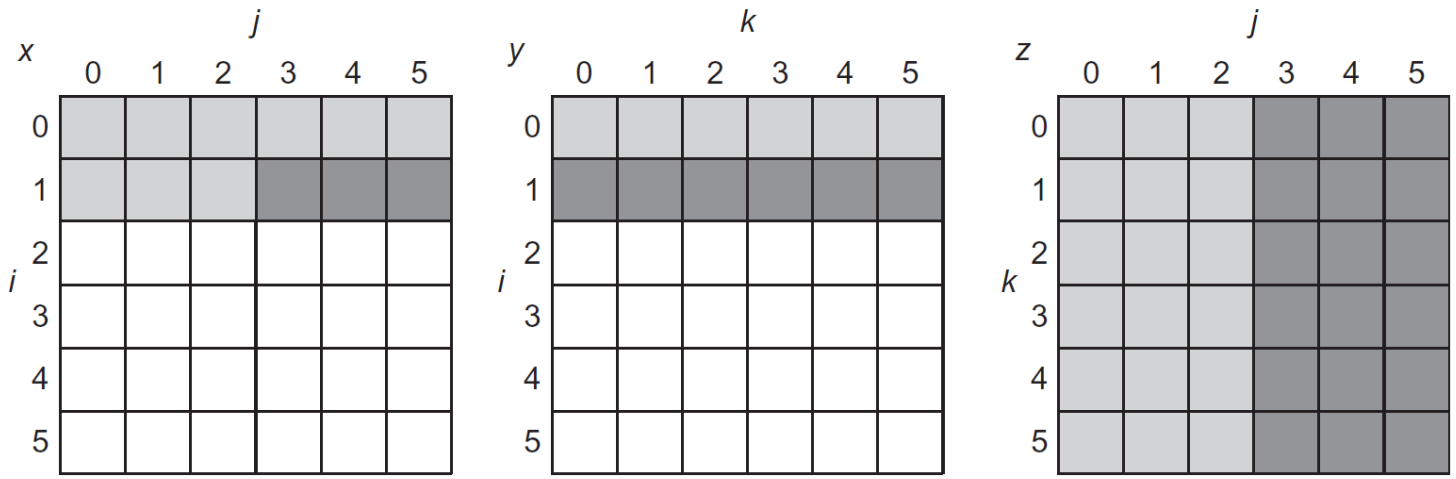


Figure 2.13 **A snapshot of the three arrays** x, y, **and** z **when** $N = 6$ **and** i = 1. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. The elements of y and z are read repeatedly to calculate 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)

I due cicli più interni leggono tutti gli elementi di Z (NxN), 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;
        };
```

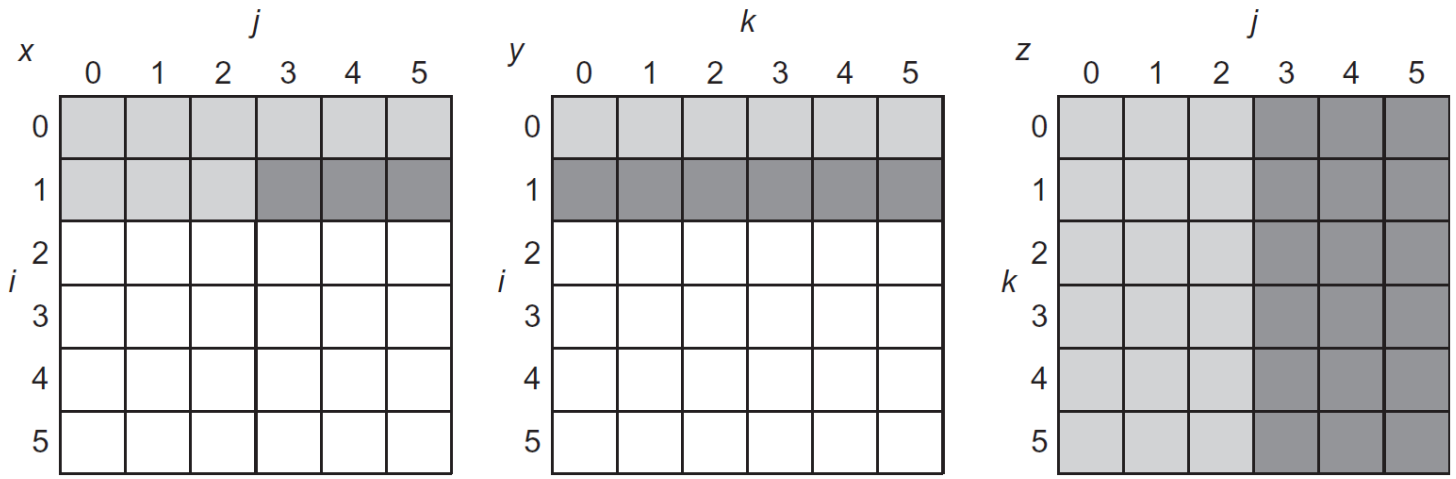


Figure 2.13 **A snapshot of the three arrays** x, y, **and** z **when** $N = 6$ **and** i = 1. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. The elements of y and z are read repeatedly to calculate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

} ;

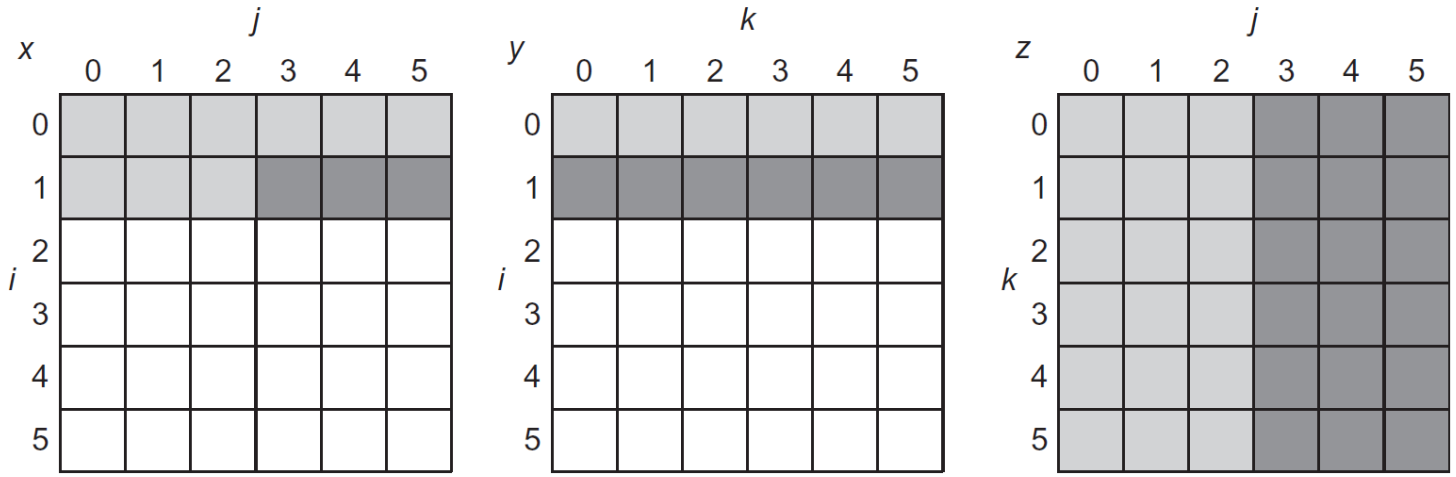


Figure 2.13 **A snapshot of the three arrays** `x`, `y`, **and** `z` **when** $N = 6$ **and** `i` $= 1$. The age of accesses to the array elements is indicated by shade: white means not yet touched, light means older accesses, and dark means newer accesses. The elements of `y` and `z` are read repeatedly 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
- *int Y[N][N] = {{..},{..},..,{..}};*
- *int Z[N][N] = {{..},{..},..,{..}};*
- *int X[N][N] = {{..},{..},..,{..}};*

```
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;
        };
```

```
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;
        };
```

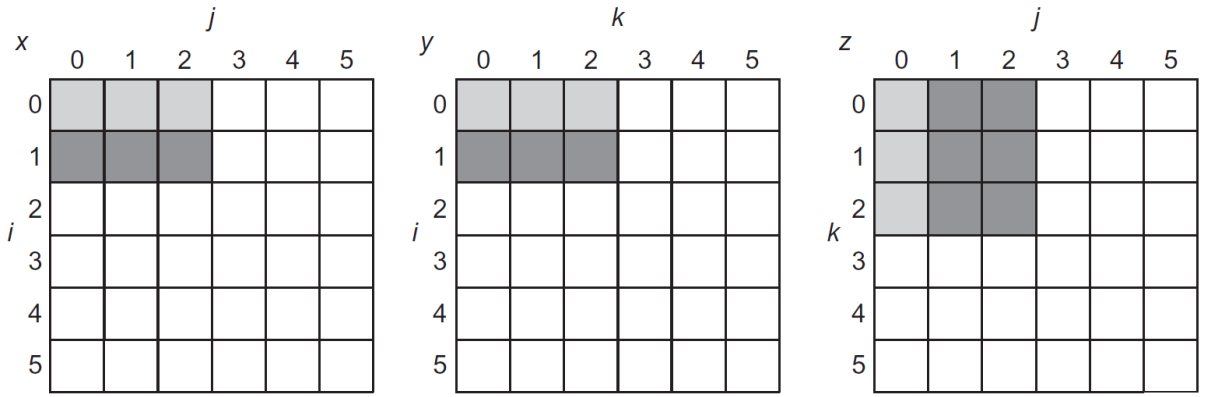


Figure 2.14 The age of accesses to the arrays x, y, and z when $B = 3$. Note that, in contrast to Figure 2.13, a smaller number of elements is accessed.

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;
}
```

- 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?